LEARNING-BASED INDUCTIVE INVARIANT SYNTHESIS

BY

PRANAV GARG

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Doctoral Committee:

Dr. Sumit Gulwani, Microsoft Research, Redmond, WA
Associate Professor Darko Marinov
Dr. Kenneth McMillan, Microsoft Research, Redmond, WA
Associate Professor Madhusudan Parthasarathy, Chair
Associate Professor Mahesh Vishwanathan

ABSTRACT

The problem of synthesizing adequate inductive invariants to prove a program correct lies at the heart of automated program verification. We investigate, herein, learning approaches to synthesize inductive invariants of sequential programs towards automatically verifying them. To this end, we identify that prior learning approaches were unduly influenced by traditional machine learning models that learned concepts from positive and negative counterexamples. We argue that these models are not robust for invariant synthesis and, consequently, introduce ICE, a robust learning paradigm for synthesizing invariants that learns using positive, negative and implication counterexamples, and show that it admits honest teachers and strongly convergent mechanisms for invariant synthesis. We develop the first learning algorithms in this model with implication counterexamples for two domains, one for learning arbitrary Boolean combinations of numerical invariants over scalar variables and one for quantified invariants of linear data-structures including arrays and dynamic lists. We implement the ICE learners and an appropriate teacher, and show that the resulting invariant synthesis is robust, practical, and efficient.

In order to deductively verify shared-memory concurrent programs, we present a sequentialization result and show that synthesizing rely-guarantee annotations for them can be reduced to invariant synthesis for sequential programs. Further, for verifying asynchronous event-driven systems, we develop a new invariant synthesis technique that constructs almost-synchronous invariants over concrete system configurations. These invariants, for most systems, are finitely representable, and can be thereby constructed, including for the Windows USB driver that ships with Windows phone.

To my parents.

ACKNOWLEDGMENTS

I would like to thank Madhusudan Parthasarathy for his continuous support and encouragement through the past six years. Over this time, he has tried to inculcate in me his keenness to think about a research problem afresh, and his taste for rigor and simplicity, for which I will be forever grateful. I thank other members of my doctoral committee– Sumit Gulwani, Darko Marinov, Kenneth McMillan, and Mahesh Vishwanathan. They have all eagerly provided me with advice on both my research and career.

I would specially like to thank Christof Löding and Daniel Neider, with whom I have now been fruitfully collaborating for more than four years and all the work presented herein has been mostly done together with them. I still remember how when we started our collaboration the main premise of this thesis looked insurmountable. If we have made any headway towards solving the undertaken problem, a large portion of the credit goes to them and Madhu. Dan Roth has also been a fantastic collaborator, and I have gained immensely from his mentorship and teaching. I also thank others with whom I have worked over the course of my doctoral research, which includes Rishi Agarwal, Gogul Balakrishnan, Ankush Desai, Aarti Gupta, Indranil Gupta, Alex Gyori, Franjo Ivančić, Akash Lal, Gennaro Parlato, Edgar Pek, Xiaokang Qiu, Muntasir Raihan Rahman, Shambwaditya Saha, Francesco Sorrentino, Andrei Stefanescu and Josep Torrellas.

I spent six memorable months at MSR India under Akash Lal, who was the perfect host, and several ideas which form the bulk of my thesis germinated during that time. I also had a wonderful opportunity of spending a summer at NEC Laboratories in Princeton, NJ and my association with my mentor, Franjo Ivančić, continues to this day, for which I am very thankful.

The Computer Science department in the University of Illinois provided me with an environment which was very conducive to learning, and I specially enjoyed the courses taught by Vikram Adve, Madhusudan Parthasarathy, Dan

Roth, Grigore Roșu and Mahesh Vishwanathan. I would also like to thank my teachers from my undergraduate institute, IIT Kanpur– Piyush P. Kurur, Somenath Biswas and (Late) Sanjeev Kumar Aggarwal, who were the first ones to introduce me to research and sparked my interest in programming languages and formal methods.

The formal methods group at UIUC has been extremely vibrant and I learned a lot from the everyday interactions I had with my colleagues at the university– apart from my collaborators mentioned above, these include Sruthi Bandhakavi, Parasara Sridhar Duggirala, Rajesh Karmani, and Dileep Kini.

Finally, my journey during these past six years has seen a fair share of ups and downs, at both professional and personal levels. While my advisor, Madhu, has been full of support throughout the highs and the lows, the support of my friends and family has been invaluable towards the completion of this journey. I would like to dearly thank my parents and my siblings– Yamini and Pallavi, for always being there for me.

My stay at UIUC has been extremely pleasant and I would like to end by thanking my friends who made this stay memorable– Pooja Agarwal, Rishi Agarwal, Shikha Aggarwal, Shashank Agrawal, Mayank Baranwal, Ankita Bhutani, Piyush Deshpande, Rajesh Bhasin, Parasara Sridhar Duggirala, Abhishek Gupta, Dileep Kini, Vivek Kumar, Kapil Mathur, Shishir Pandya, Advitya Patyal, Deeksha Rastogi, Manila Sarangi, Prakalp Srivastava, and Ankur Taneja.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The problem of program verification– given a program P and a safety specification φ, does P satisfy φ– is a classic problem that is inherently related to the semantics of programs [Flo67, Hoa69]. Given a set of initial states of a program and a set of *bad* states, $\neg\varphi$, that do not meet the specification, the general methodology for proving the correctness of the program involves specifying an invariant for the program. An invariant is a set of program states that includes the initial states of the program, excludes the set of bad states and is *inductive*– which means that it is closed with respect to the transition relation of the program, and guarantees that any execution of a statement in the program from a state that belongs to the invariant region results in a state that also belongs to the invariant (see Figure 1.1). By using induction on the transition relation of the program, one can easily argue that an inductive invariant with the above properties is an over-approximation of the reachable states of the program, and since it does not intersect with the set of bad states, the program is correct. *The problem of synthesizing adequate inductive invariants to prove a program correct is at the heart of automated program verification.* Synthesizing invariants is in fact the *hardest* aspect of program verification—once adequate inductive invariants are synthesized, program verification reduces to checking validity of verification conditions[1] obtained from finite loop-free paths [Flo67, Hoa69, BCD+05], which is a logic problem that has been highly automated over the years with the advances in automated logic solvers.

The problem of synthesizing inductive invariants is undecidable. This result follows from the undecidability of constant propagation, to which the post correspondence problem (PCP) reduces to [Hec77]. Nevertheless, there are known several procedures for synthesizing inductive program invariants

in the context of deductive verification

[1] simply put, verification conditions check if the invariants specified are adequate and inductive.

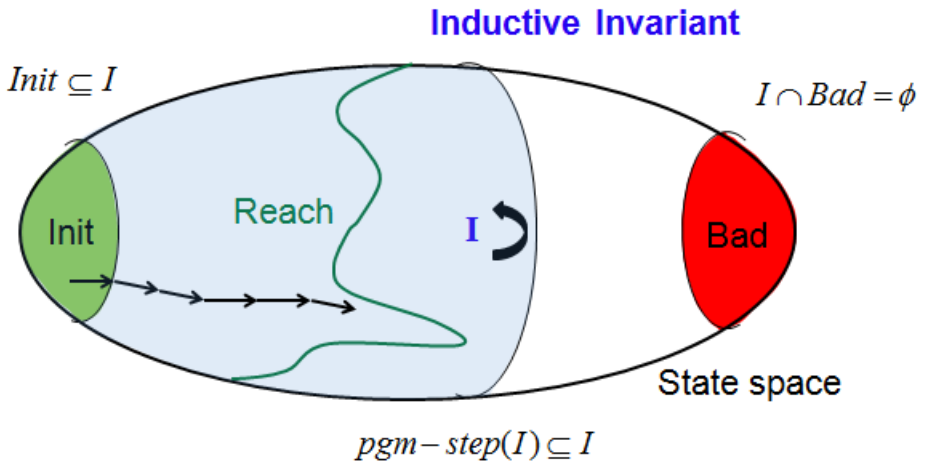


Figure 1.1: If we view a program as a transition system with a set of initial states *Init* and bad states *Bad*, a program execution is a transition sequence. If the program is correct, then there is no transition sequence from an initial state to a bad state. *Reach* is the set of reachable states and I is an adequate inductive invariant that verifies the program.

including (symbolic) model checking [McM92], abstract interpretation [CC77], predicate abstraction [BR02, HJMS02], Craig's interpolation [McM03] and IC3 [Bra11]. All of these techniques are mostly tuned towards verifying a certain class of programs or verifying programs with respect to a certain class of properties, and due to the inherent hardness of the problem, come with certain limitations. As an example, model checking can successfully synthesize an invariant when the program has a finite state-space or the paths in the program are bounded. However, for programs over an infinite domain, for eg. integers, with unbounded number of paths in the program, model checking is bound to fail. Similarly, invariants synthesized by abstract interpretation over a given abstract domain might be inadequate due to widening [CC77], and predicate abstraction, which has been very successful in synthesizing invariants over type-state predicates [BR02], is mostly inadequate to synthesize invariants over numerical domains. As opposed to the above *white-box* techniques, where the synthesizer of the invariant is acutely aware of the precise program and the property that is being proved, this thesis explores a completely different way for synthesizing inductive invariants– using *learning*.

As compared to the techniques mentioned above, ours is a black-box

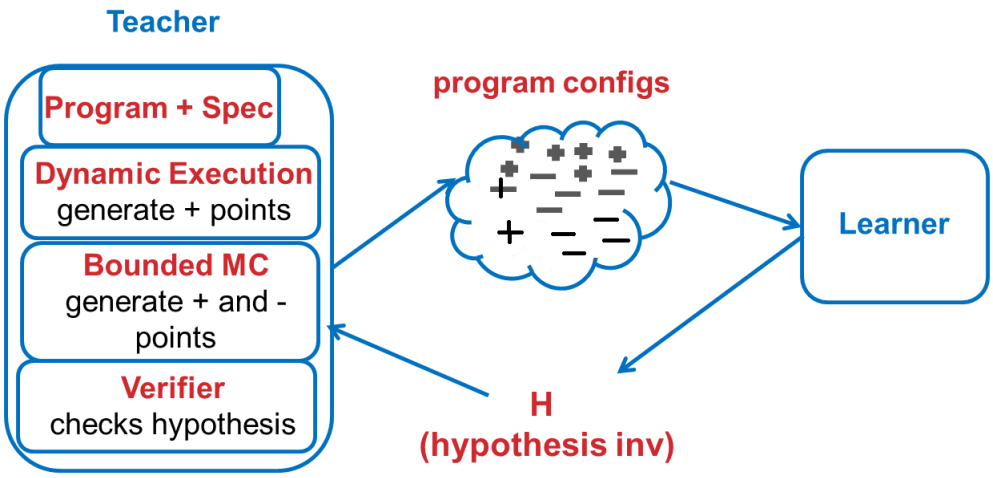


Figure 1.2: Black-box learning of invariants

approach where the invariant synthesizer is largely agnostic to the structure of the program and the property, but works with a partial view of the requirements of the invariant. Given a finite set (or a sample) of program configurations consisting of those that are reachable along bounded executions of the program and those that lead to violations of the specification, the invariant synthesizer is a learning algorithm that learns a classifier that separates the former set of configurations from the latter. The inductive bias of the learning algorithm forces generalization to unseen program configurations.

In this data-driven approach, we split the synthesizer of invariants into two parts (see Figure 1.2). One component is a *teacher*, which is essentially a program verifier that can verify the program using a conjectured invariant and generates counterexamples; it may also have other ways of generating configurations that must or must not be in the invariant (e.g., dynamic execution engines, bounded model-checking engines, etc.). The other component is a *learner*, which learns from counterexamples given by the teacher to synthesize the invariant. In each round, the learner proposes an invariant hypothesis H, and the teacher checks if the hypothesis is adequate to verify the program against the specification; if not, it comes up with concrete program configurations that need to be added or removed from the invariant (denoted by $+$ and $-$ in Figure 1.2). The learner, who comes up with the invariant H is completely agnostic of the program and property being verified, and aims to build a simple formula that is consistent with the sample. Hence, when learn-

ing an invariant, the teacher and learner talk to each other in rounds, where in each round the teacher comes up with additional constraints involving new data-points and the learner replies with some set satisfying the constraints, until the teacher finds the set to be an adequate inductive invariant. The above learning approach for synthesizing invariants has several advantages; it was explored before in other contexts [CGP03, AMN05, ACMN05, ECGN00], and has been explored with renewed interest in this thesis and concurrently by other researchers in [SNA12, SGH+13b, SGH+13a, SA14].

Advantages of Learning: There are many advantages the learning approach has over traditional white-box approaches. First, a synthesizer of invariants that works cognizant of the program and property is very hard to build, simply *due* to the fact that it has to deal with the complex logic of the program. When a program manipulates complex data-structures, pointers, objects, etc. with a complex memory model and semantics, building a set that is guaranteed to be an inductive invariant gets extremely complex. However, the invariant for a loop in such a program may be much simpler, and hence a black-box technique that uses a "guess and check" approach guided by a finite set of configurations is much more light-weight and has better chances of finding the invariant. (See [TLLR13] where a similar argument is made for black-box generation of the abstract post in an abstract interpretation setting). Second, learning, which typically concentrates on finding the *simplest* concept that satisfies the constraints, implicitly provides a tactic for generalization, while white-box techniques (like interpolation) need to build in tactics to generalize. Finally, the black-box approach allows us to seamlessly integrate highly scalable machine-learning techniques into the verification framework [Mit97a, KV94]. Machine learning algorithms can be trained to learn invariants that belong to various classes of Boolean functions, such as k-CNF/k-DNF, Linear Threshold Functions, Decisions Trees, etc., and we develop such machine learning algorithms in this thesis.

However, standard machine learning algorithms for classification are mostly trained on given sets of positive and negative examples. We show, in this thesis, that such a learning model, shown in Figure 1.2, is not robust for synthesizing invariants. Consequently, we introduce a new learning model for robust invariant synthesis called *ICE*, which stands for *learning invariants using Implication Counter-Examples*, and develop several learning algorithms

in the ICE model and deploy them in the verification setup to prove programs correct. Before we delve deeper into our learning model in Chapter 2, let us next introduce some preliminary concepts (Section 1.1), followed by our main contributions (Section 1.2) and related research (Section 1.3).

1.1 Background

This thesis assumes the familiarity of the reader with Floyd-Hoare logic [Flo67, Hoa69] for proving the correctness of programs (readers can refer to the survey paper [Apt81] or the book [BM07] for a comprehensive exposition of this topic). The central feature of Hoare logic is the *Hoare triple*. A valid Hoare triple $\{P\}\ C\ \{Q\}$, where C is a command and P, Q are assertions, implies that from any program configuration that satisfies assertion P, if the program executes C then, on termination, the resulting configuration will satisfy Q. These assertions are formulas in predicate logic, which provides a finite representation for a possibly infinite set of program configurations. Hoare logic provides a set of proof rules for checking the validity of such Hoare triples. The methodology for verifying programs using Hoare logic involves annotating assertions at various locations in the program such that the corresponding Hoare triples are valid. Given these logical annotations, the validity of the corresponding Hoare triples is reduced to checking the validity of a set of predicate logic formulas called *verification conditions*. This reduction is achieved by predicate transformers, *weakest precondition* or *strongest postcondition*, which provide strategies to build valid deductions in Hoare logic.

Given a statement S, the weakest precondition of a formula R with respect to S– $wp(S, R)$, is the weakest precondition on the initial state ensuring that an execution of S on termination results in a final state satisfying R. Conversely, the strongest postcondition of a formula R with respect to S– $sp(S, R)$, is the strongest postcondition satisfied by the final state of any terminating execution of S, for any initial state satisfying R. These predicate transformers can be constructed for various statements in the language, such as an assignment or an assume statement, and also for any arbitrary sequence of such statements. As a result, it is not required to annotate logical assertions at every location in the program but only annotate pre/post-conditions for

(recursive) functions and invariants for loops in the program.

Given user annotations P and Q enclosing a straight-line sequence of statements S, the Hoare triple $\{P\}S\{Q\}$ is valid iff the verification condition $P \Rightarrow wp(S, Q)$ is a valid formula. This is equivalent to checking if the verification condition $sp(S, P) \Rightarrow Q$ is valid. Once program annotations have been provided by the user, verification conditions such as these can be automatically generated and checked for validity using (mostly automated) constraint solvers. Coming up with these annotations is, however, mostly manual and forms the main bottleneck to automatic software verification. Automatically synthesizing these annotations is the central problem we address in this thesis.

1.2 Contributions of the thesis

ICE learning model for synthesizing invariants. We propose a robust learning model for synthesizing inductive invariants called ICE learning. Learning has been explored before in the context of synthesizing program invariants [SNA12, SGH+13b, SGH+13a, CW12, KJD+10] and, also, in other verification contexts [CGP03, AMN05, ACMN05, ECGN00]. We show that these prior-known learning approaches are not robust for invariant synthesis. Our work in this thesis (Chapter 2) is the first that develops a robust learning model that explicitly incorporates the search for *inductive* invariants in black-box invariant generation.

ICE algorithms for learning numerical invariants. We develop *two* new ICE learning algorithms for learning numerical program invariants. In Chapter 3, we adapt the template-based synthesis techniques that use constraint solvers [CSS03, GSV08, GMR09, GR09] to build a black-box ICE learning algorithm. Our algorithm iterates over all possible template formulas in increasing complexity, till it finds, using a constraint solver, an adequate invariant in the appropriate template. The resulting learning algorithm is, thereby, *strongly convergent* despite the concept class of invariants being infinite. Our second algorithm for learning numerical invariants (Chapter 4) adapts decision-tree learning to the ICE framework. This work breaks new ground in adapting machine learning techniques to invariant synthesis, giving

the first efficient robust ICE machine-learning algorithms for synthesizing invariants. We build both these learning algorithms over Boogie [BCD+05] and show that they are extremely effective in synthesizing adequate inductive invariants over scalar program variables. Our learning algorithms outperform existing white-box invariant synthesis techniques including interpolation [BK11, McM06a], template-based invariant generation [GR09], abstract interpretation [CC77, TLLR13], and existing black-box learning techniques including learning algorithms based on computational geometry [SGH+13b] and randomized search [SA14].

Synthesizing universally quantified invariants of linear data-structures. We present a novel automata-based representation of quantified properties of linear data-structures, called *quantified data automata* (QDA), and develop learning algorithms for them (Chapter 5). In particular, we first develop a passive learning algorithm, based on Angluin's L^* algorithm [Ang87a], for learning QDAs from concrete data structure configurations that manifest along dynamic test runs. We also develop an RPNI-based [OG92] learning algorithm in the ICE model for robustly learning QDAs. We, further, identify a subclass of QDAs, called elastic QDAs, that can be converted to formulas of decidable logics over arrays and lists, and adapt the above learning algorithms to learning elastic QDAs. Using a wide variety of programs manipulating arrays and lists, we show that we can effectively learn adequate quantified invariants in these settings using these learning algorithms. Elastic QDAs, it turns out, also form an abstract domain for heap analysis of programs manipulating acyclic heaps with a single pointer field, including data-structures such singly-linked lists (Chapter 6). We develop an abstract transformer and a widening operator over the domain of elastic QDAs, and show that our abstract interpretation is able to prove the naturally required universally-quantified properties of a large suite of list-manipulating programs.

Verification and invariant synthesis of concurrent programs. In Chapter 7, we address the problem of verification of shared-memory concurrent programs. We develop a sound *sequentialization* for concurrent programs, thereby showing that the problem of compositionally verifying concurrent program reduces to sequential verification. The sequentialization we develop

also provides a way to synthesize compositional rely-guarantee proofs of correctness of concurrent programs by reducing the problem to invariant synthesis of the corresponding recursive sequential program. This brings learning-based invariant synthesis techniques, we have investigated in this thesis, into play for automatically proving concurrent programs correct. In fact, we use the sequentialization followed by an automatic predicate-abstraction tool [BR02] to automatically verify concurrent programs.

Finally, in Chapter 8, we investigate the problem of provably verifying that an asynchronous message-passing system satisfies its local assertions. We present a novel reduction scheme for asynchronous event-driven programs that synthesizes *almost-synchronous invariants*— invariants consisting of global states where message buffers are close to empty. The reduction finds almost-synchronous invariants and simultaneously argues that they cover all local states of the system. We implement our reduction strategy, which is sound and complete, and show that it is more effective in proving programs correct as well as more efficient in finding bugs in several programs, compared to current search strategies which almost always diverge. The high point of our experiments is that our technique can prove the Windows Phone USB Driver written in P [DGJ+13] correct for the responsiveness property, which was hitherto not provable using state-of-the-art model-checkers.

1.3 Related Research

White-box techniques for synthesizing invariants. Invariant synthesis is one of the fundamental problems in automated program verification, and there has been a lot of related research towards automatically synthesizing program invariants. As opposed to the black-box learning approach investigated in this thesis, where the invariant synthesizer is largely agnostic to the structure of the program and the property but works with a partial view of the requirements of the invariant, white-box invariant synthesizers are acutely aware of the precise program and the property that is being verified. Prominent examples for white-box techniques for synthesizing invariants include abstract interpretation [CC77], predicate abstraction [BR02], interpolation [McM03, McM06a], and IC3 [Bra11].

In abstract interpretation [CC77], invariant generation is achieved by

computing fixed-points on abstract lattices of finite height, or using *widening* and *narrowing* techniques for lattices of infinite height (see [Kar76] and [CH78] for abstract interpretation over affine equalities and inequalities and [Min01] for abstract interpretation over octagons). Abstract interpretation has been predominately used for synthesizing invariants over convex domains [Min01, CH78], but there has also been research into non-convex domains, such as by considering powerset extensions of abstract domains in [FR99a,FR99b,SISG06a,SISG06b], and even non-lattice domains [GNS+13]. However, often for an abstract domain, the most precise abstract transformer does not exist or is not computable [RSY04, TLLR13]. This is often true when the abstract domain elements are very complex, for eg., involve non-convex concepts or abstractions over heap structures such as the domain elements involving QDAs we introduce in Chapter 6, or when the programming language is complicated [TLLR13]. Abstract interpretation in such a case is bound to be imprecise. Even otherwise, when the most precise abstract transformer is computable, abstract interpretation often synthesizes imprecise invariants due to widenings it performs when the domain has an infinite height.

As opposed to abstract interpretation, abstraction-refinement approaches based on guidance by counterexamples strive to be property driven. Predicate abstraction [BR02, HJMS02], for instance, tunes the abstract lattice according to the property being verified, and the Boolean program model-checker computes the reachable set of predicate-states over the chosen lattice, and hence essentially synthesizes a program invariant.

In recent years, techniques based on *Craig's interpolation* [McM03, JM06, McM06a] have emerged as a new method for invariant synthesis, and use the unsatisfiable core of the proof that a bounded run does not violate the post-condition to infer a generalization that is likely to be an invariant. Interpolation techniques, which are inherently white-box as well, are known for several theories, including linear arithmetic [KLR10], uninterpreted function theories [RS10, BZM08], and even quantified properties over arrays and lists [JM07b, McM08, ABG+12, SPW09]. These methods use different heuristics like term abstraction [ABG+12], finding interpolants over a restricted language [JM07b, McM08] and use of existential ghost variables [SPW09] to ensure that the interpolant converges on an invariant from a *finite* set of spurious counter-examples. As opposed to using various heuristics we mention above to force generalization to an inductive invariant, learning

approaches, we investigate in this thesis, rely on the *inductive bias* of the learners to generalize. Complementary to interpolation, IC3 [Bra11] has also been proposed recently as an alternate approach for generalizing results of symbolic bounded model-checking to synthesize program invariants.

Template based approaches to synthesizing numerical invariants using constraint solvers have also been explored in the past in white-box settings [CSS03, GSV08, GR09, GMR09]. These approaches directly encode the adequacy of the invariant (encoding the entire program's body) into a constraint, and use Farkas' lemma to reduce the problem to satisfiability of a quantifier-free non-linear arithmetic formulas. We adapt this approach, somewhat, to develop an ICE learning algorithm for learning numerical invariants, presented in Chapter 3. Our approach splits the task of invariant synthesis between a white-box teacher and a black-box learner, communicating only through implication constraints on concrete data-points. This greatly reduces the complexity of the problem, as opposed to template-based white-box approaches, leading to simple teacher and a learner that solves the satisfiability problem for only linear arithmetic constraints, which is decidable.

Heap analysis using abstract interpretation For invariants expressing properties on the dynamic heap, *shape analysis* techniques are the most well known [SRW02], where locations are classified using *unary* predicates (some dictated by the program and some given as instrumentation predicates by the user), and abstractions summarize all nodes with the same predicates into a single node. These user-provided instrumentation predicates are often too particular to the program being verified. Coming up with them is, therefore, not an easy task, and often is as hard as manually specifying the invariant. For singly-linked lists, [MYRS05] introduces a family of abstractions based on a set of instrumentation predicates which track uninterrupted list segments. While this alleviates the burden of the user to some extent, the instrumentation predicates in these abstractions only handle structural properties of lists and not the more-complex quantified data properties that are the subject of our investigation in Chapter 5-6. Several separation logic based shape analysis techniques have also been developed over the years [DOY06, GVA07, BCC+07, BCI11]. But like [MYRS05], they too mostly handle properties concerning the shape of heap structures.

In recent work [MRS10, CR08, BDES12, GMT08, CCL11, HP08], several

abstract domains have been explored for analyzing heaps which combine the shape and the data constraints. Though some of these domains [MRS10,CR08] can handle heap structures more complex than singly-linked lists, most of these domains require the user to provide a set of data predicates [GMT08], a set of structural guard patterns [BDES12], or predicates over both the structure and the data constraints [MRS10, CR08]. In contrast, the abstract domain we develop in Chapter 6, based on quantified data automata, is strictly more expressive than the abstract domains in [CCL11, HP08]– the data automata we introduce allows expressing arbitrary n-ary quantified relations between data elements, such as sortedness of a list that [CCL11], for example, cannot express. Also, these n-ary quantified relations are automatically synthesized in our abstract analysis and are not required to be provided by the user. The only assistance our technique requires from the user is specifying a numerical domain over data formulas and the number of universally quantified variables.

Automata based abstract interpretation has been explored in the past [HHR$^+$11] for inferring shape properties about the heap. However, in our work (Chapter 6) we are interested in strictly-richer universally quantified properties on the data stored in the heap. [Av11] introduces a streaming transducer model for algorithmic verification of single-pass list-processing programs. However the transducer model severely constrains the class of programs it can handle; for example, [Av11] disallows repeated or nested list traversals which are required in sorting routines, etc.

[PW10] proposes a counter-example driven shape analysis where the user is required to provide certain quantified predicates for the analysis. Given these predicates, [PW10] uses a CEGAR-loop for incrementally improving the precision of the abstract transformer and also discovering new predicates on the heap objects that are part of the invariant.

Decidable heap logics. Quantified data automata model we introduce in Chapter 5 for representing quantified invariants over lists and arrays is inspired by decidable heap logics– the decidable fragment of Strand [MPQ11] and the array property fragment [BMS06], and can track invariants with guard constraints of the form $y \leq t$ or $t \leq y$ for universal variables y and some term t. These structural constraints on the guard are very similar to array partitions in [GRS05, HP08, CCL11]. However, our automata model is strictly more general. For instance, none of these related works can handle sortedness

of arrays which requires quantification over more than one variable.

Using learning to synthesize program invariants. We primarily investigate in this thesis learning techniques to synthesize inductive program invariants. A prominent early example of using black-box learning for synthesizing invariants is Daikon [ECGN00], which uses conjunctive Boolean learning to synthesize *likely* invariants from program configurations recorded along dynamic test runs. A similar line of work has been explored more recently in [NKWF12] for synthesizing likely polynomial and array invariants. Our work in Chapter 5 [GLMN13] also synthesizes likely invariants, but for rich quantified data properties over linear data-structures such as linked-lists and arrays. However, it is important to note that the invariants synthesized in this manner might not be *true* inductive invariants of the program. This shortcoming is gotten rid of in Houdini [FL01], which, like Daikon also uses conjunctive Boolean learning to learn conjunctive invariants over templates of atomic formulas but, has a constraint-solver based teacher to iteratively guide the conjunctive learner by providing counterexamples till the concept learned is inductive and adequate. A similar algorithm is used in liquid types [RKJ08] for inferring refinement types for program expressions to statically verify programs memory-safe.

Following these early works [ECGN00, FL01], learning was explicitly introduced in the context of verification by Cobleigh et al. [CGP03], which was then followed by several algorithms based on Angluin's L^* algorithm [Ang87a] for learning regular languages applied to finding rely-guarantee contracts [AMN05], learning stateful interfaces for programs [ACMN05] and verifying CTL properties [VV07].

Recently, there has been a renewed interest in the application of learning to program verification, in particular to synthesize invariants by using scalable machine learning techniques [Mit97a, KV94] to find classifiers that separate good states that the program can reach (positive counterexamples) from bad states the program is forbidden from reaching (negative counterexamples). This includes the work we present in this thesis [GLMN13, GLMN14, GNMR15] and also concurrent work by other researchers [CW12, KJD+10, SNA12, SGH+13b, SGH+13a, SA14, KPW15]. However, it turns out that merely learning from positive and negative counterexamples for synthesizing invariants is inherently not robust. We therefore

introduce ICE learning (see Chapter 2) [GLMN14], which extends this classical learning setting with learning using implication counterexamples and results in a robust model for invariant synthesis. Implication counterexamples were also identified by Sharma et al in [SGH+13b], but the learners proposed by them did not handle them in any way. Subsequent to introducing ICE learning we have developed ICE learners for learning invariants over octogonal domains (Chapter 3-4) [GLMN14, GNMR15] and universally quantified invariants over linear data structures [GLMN14]. ICE learners have also been, consequently, developed by other researchers for Regular Model Checking [Nei14], and invariant synthesis using randomized search [SA14]. In fact, as we show in Chapter 2 [GLMN14], the Houdini algorithm [FL01] along with its generalization by Thakur et al. [TLLR13] and [GKT13] to arbitrary abstract domains like intervals, octagons, polyhedrons, linear equalities, etc. can also be seen as ICE learning algorithms. In the context of black-box invariant synthesis, counterexamples to inductiveness of an invariant have been handled in the past in [RSY04, YBS06, vE98, FL01], but only in the context of lattice domains where the learned concepts grow monotonically and implications essentially yield positive examples (see Chapter 2). The ICE learning model, we propose, uniformly incorporates the search for inductive sets in black-box invariant generation.

One primary difference in our work, compared to white-box approaches for invariant synthesis we have described before, is that our *black-box approach* does not look at the code of the program, but synthesizes an invariant from a snapshot of examples and counter-examples that characterize the invariant. This has both advantages and disadvantages compared to white-box techniques. The main disadvantage is that information regarding what the program actually does is lost in invariant synthesis. However, this is the basis for its advantage as well– by *not* looking at the code, the learning algorithm promises to learn the sets with the simplest representations, and can also be much more flexible. For instance, even when the code of the program is complex, for example having non-linear arithmetic or complex heap manipulations that preclude logical reasoning, black-box learning gives ways to learn simple invariants for them.

A problem that is very related to invariant synthesis is that of synthesizing programs such that they satisfy a given specification. In *program* or *expression* synthesis, a popular approach is to use counterexample guided inductive syn-

thesis (CEGIS) [ABJ+13, SL08, SLTB+06], which is also a black-box learning paradigm like ICE learning, and is gaining traction aided by synthesizers based on explicit enumeration, symbolic constraint-solving and stochastic search. In addition, learning has also been used as a technique for solving various problems in software engineering including for model extraction and testing of software [ACH+10, CNS13].

Machine learning theory. One of the advantages of the learning approach to synthesizing invariants, we investigate in this thesis, is the opportunity to build highly scalable program analyses based on machine learning algorithms. Machine learning algorithms (see [Mit97b] for an overview) are often used in practical learning scenarios due to their high scalability. Amongst the most well-known classification algorithms are the winnow algorithm [Lit87], perceptron [Ros58], and support vector machines [CV95] for learning linear classifiers, and decision tree algorithms, such as ID3 [Qui86] and C4.5, C5.0 [Qui93] for learning arbitrary Boolean functions. Since the concept class of invariants we learn in Chapter 4 are arbitrary Boolean functions, we base our learner in Chapter 4 on decision tree learning algorithms [Qui86, Qui93]. Apart from our work, recent work has investigated abstract domains based on decision trees for proving conditional termination [UM14] and learning program invariants using decision tree learning [KPW15].

Turning to general learning theory, the field of algorithmic learning theory is a well-established field [KV94, Mit97a]. The PAC learning model of learning approximating concepts with high probability using samples drawn randomly from a distribution is a well-studied model. In the absence of any underlying distribution, the classical results in learning theory are for learning using *queries* where the learner is allowed to ask the teacher various different kinds of questions (see [Ang87b] for an overview). A celebrated result in this realm is the polynomial-time learning of regular languages by Angluin which uses membership and equivalence queries [Ang87a]. Gold [Gol78] showed that the problem of finding the smallest automaton consistent with a set of accepted and rejected strings is NP-complete. The RPNI algorithm [OG92] is a passive learning algorithm for regular languages that learns from positive and negative examples, works in polynomial time, but does not guarantee learning the smallest automaton. The ICE learning algorithm we develop in Chapter 5 for learning quantified properties of linear data structures

is based on RPNI, but extended to the ICE setting. Angluin's learning using only equivalence queries [Ang87b, Ang90] is very closely tied to the mistake-bound online learning model proposed by Littlestone in [Lit87], in which a learner iteratively learns from incorrectly classified data and needs to converge to a correct classifier within a (polynomially) bounded number of wrong conjectures. Both these learning models are closely related to the iterative learning we propose in ICE. However, though, machine learning community has developed several algorithms for learning classifiers in the online learning model, such as the winnow [Lit87] and perceptron [Ros58], we are not aware of any prior machine learning algorithms in the online learning framework designed to learn from positive and negative data as well as implications.

CHAPTER 2

THE ICE LEARNING MODEL FOR ROBUST INVARIANT SYNTHESIS

In this chapter we develop a new learning model called *ICE* for robustly synthesizing inductive program invariants. To do so, let us first take a closer look at the "guess-and-check" learning approach for invariant synthesis as suggested in Chapter 1. Recall that in this approach (ref. Figure 1.2) the learner synthesizes suggestions for the invariants in each round. The teacher is completely aware of the program and the property being verified, and (a) checks if the invariant hypothesis H conjectured by the learner is indeed an inductive invariant and is adequate in proving the property of the program (typically using a constraint solver), and (b) if the invariant is not adequate, comes up with concrete program configurations that need to be added or removed from the invariant (denoted by $+$ and $-$ in Figure 1.2). The learner, who conjectures the invariant hypothesis H is completely agnostic of the program and the property being verified, and aims at learning a simple formula that is consistent with the sample of labeled program configurations. When learning an invariant, the teacher and the learner talk to each other in rounds, where in each round the teacher comes up with additional constraints involving new data-points and the learner replies with a new invariant hypothesis satisfying the constraints, until the teacher finds that the invariant conjectured is an adequate inductive invariant.

2.1 Non-robustness of Traditional Learning Models for Synthesizing Invariants

The learning approach described above is unduly influenced by computational learning theory, automata learning, and machine learning techniques, which have traditionally offered learning from positive and negative examples. As we show below, learning from labeled (positive and negative) examples as

above does not form a robust learning framework for synthesizing invariants. To see why, consider the following simple program—

$$\mathit{pre};\ S;\ \textbf{while}\ (b)\ \ \textbf{do}\ \ L;\ \textbf{od}\ \ S';\ \mathit{post}$$

with a single loop body for which we want to synthesize an invariant that proves that when the *pre*condition to the program holds, the *post*condition holds upon exit. Let us assume that the learner has conjectured H as a hypothesis invariant. In order to check if H is an adequate invariant, the teacher checks the following properties:

(a) whether the strongest-post of the *pre*condition across S implies H; if not the teacher finds a concrete program configuration p that should belong to the invariant and returns this as a *positive* example to the learner.

(b) whether the strongest-post of $(H \wedge \neg b)$ across S' implies the *post*condition; if not, the teacher returns a concrete program configuration p in H that should not belong to the invariant as a *negative* example.

(c) whether H is inductive; i.e., whether the strongest-post of $(H \wedge b)$ across the loop body L implies H; if not, the teacher finds two concrete program configuration p and p', such that on executing the loop body L from configuration p the program reaches configuration p', where $p \in H$ and $p' \notin H$.

Since the teacher does not *know* the precise invariant (there are after all many), in the last case above, the teacher has no way of knowing whether p should be excluded from H or whether p' should be included in H. In the learning model that requires the teacher to return a positive or negative counterexample to the learner in every round, the teacher gets stuck. In many learning algorithms in the program verification literature [CGP03, AMN05, ACMN05, GLMN13], the teacher cheats: she arbitrarily chooses to exclude p from H or include p' in H hoping that it will result in an inductive invariant. However, this arbitrary choice makes the entire framework non-robust, causing divergence, preventing the learner from learning the simplest concepts, and introducing arbitrary bias that is very hard to control. Another way to make progress, in this case, is to return to the learner the membership of p and p' with respect to the reachable set of configurations of the program. The teachers in the learning algorithms in [GLMN13, SGH$^+$13b], in fact, answer membership queries for p and p' by checking for their reachability along bounded executions. While answering these membership queries is sometimes possible in this way,

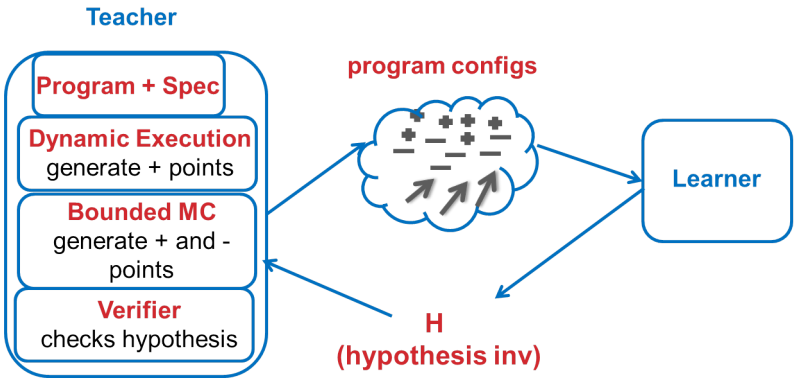


Figure 2.1: A schematic of the ICE learning model for synthesizing invariants.

in general the reachability problem is undecidable. Also the teacher by answering membership queries with respect to the reachable set guides the learner towards learning this set which is often more complex to represent than the simplest inductive invariants.

When faced with non-inductiveness of the current invariant hypothesis H in terms of a pair (p, p'), all that the teacher knows is that for any inductive invariant I of the loop in the program, if p belongs to I then configuration p' should also belong to I. In the new learning model that we formally describe in this chapter, the teacher simply communicates this pair of configurations (p, p') to the learner and demands that the learner learns hypotheses H such that if p is included in H, then so is p'. Consequently, we propose that we build learning algorithms that do not just learn from (positively or negatively) labeled counterexamples, as most traditional learning algorithms do, but instead also learn from unlabeled counterexamples for which the learner predicts labels such that they are constrained by an *implication* relation (shown as an arrow in the sample in Figure 2.1). The learning model that we propose is called *ICE* that stands for *learning using Implication counterexamples*. In this learning model, the learner assigns (positive or negative) labels to the unlabeled counterexamples based on considerations of simplicity of the learned concept, generalization, etc., as opposed to the teacher making arbitrary choices for these labels as in various traditional learning algorithms in the verification literature [CGP03, AMN05, ACMN05, GLMN13]. ICE learning is a *robust* learning model for synthesizing invariants, in the sense that the teacher can always communicate to the learner precisely why a hypothesis is not an inductive invariant (even for programs with multiple

loops, nested loops, etc.). The teacher in ICE learning consequently is both *honest* (never gives an example classification that precludes an invariant) and makes *progress* (is always able to refute an invariant that is not adequate or inductive). This is in sharp contrast to learning only from positive and negative examples, where the teacher is forced to be dishonest (as it does not know an invariant) to make progress.

2.2 The ICE Learning Framework

When defining a (machine) learning problem, one usually specifies a domain D (like points in the real plane or finite words over an alphabet), and a class of concepts $\mathcal{C}$ (like rectangles in the plane or regular languages), which is a class of subsets of the domain. In classical learning frameworks (see [KV94]), the teacher provides a set of positive examples in D that are part of the target concept, and a set of negative examples in D that are not part of the target concept. Based on these, the learner must construct a hypothesis that approximates the target concept the teacher has in mind.

ICE learning: In our setting, the teacher does *not* have a precise target concept from $\mathcal{C}$ in mind, but is looking for an inductive set which meets certain additional constraints. Consequently, we extend this learning setting with a third type of information that can be provided by the teacher: *implications*. Formally, let D be some domain and $\mathcal{C} \subseteq 2^D$ be a class of subsets of D, called the concepts. The teacher knows a triple (P, N, R), where $P \subseteq D$ is an (infinite) set of positive examples, $N \subseteq D$ is an (infinite) set of negative examples, and $R \subseteq D \times D$ is a relation interpreted as an (infinite) set of implications. We call (P, N, R) the *target description*, and these sets are typically *infinite* and are obtained from the program, but the teacher has the ability to query these sets effectively.

The learner is given a finite part of this information (E, C, I) with $E \subseteq P$, $C \subseteq N$, and $I \subseteq R$. We refer to (E, C, I) as an ICE *sample*. The task of the ICE learner is to construct *some* hypothesis $H \in \mathcal{C}$ such that $P \subseteq H$, $N \cap H = \emptyset$, and for each pair $(x, y) \in R$, if $x \in H$, then $y \in H$. A hypothesis with these properties is called a *correct hypothesis*. Note that a target description (P, N, R) may have several correct hypotheses (while H

must include P, exclude N, and be R-closed, there can be several such sets).

Iterative ICE learning: The above ICE learning corresponds to a passive learning setting, in which the learner does not interact with the teacher. In general, the quality of the hypothesis will heavily depend on the amount of information contained in the sample. However, when the hypothesis is wrong, we would like the learner to gain information from the teacher using new samples. Since such a learning process proceeds in rounds, we refer to it as *iterative ICE learning*.

The iterative ICE learning happens in rounds, where in each round, the learner starts with some sample (E, C, I) (from previous rounds or an initialization) and constructs a hypothesis $H \in \mathcal{C}$ from this information, and asks the teacher whether this is correct. If the hypothesis is correct (i.e., if $P \subseteq H$, $H \cap N = \emptyset$, and for every $(x, y) \in R$, if $x \in H$, then $y \in H$ as well), then the teacher answers "correct" and the learning process terminates. Otherwise, the teacher returns either some element $d \in D$ with $d \in P \setminus H$ or $d \in H \cap N$, or an implication $(x, y) \in R$ with $x \in H$ and $y \notin H$. This new information is added to the sample of the learner.

The learning proceeds in rounds and when the learning terminates, the learner has learned *some* R-closed concept that includes P and excludes N.

2.2.1 ICE Learning for Synthesizing Invariants

Given an ICE learning algorithm for a concept class, we can build algorithms for synthesizing invariants by building the required (white-box) teacher. We can apply such learning for finding invariants in programs with multiple loops, nested loops, etc. The learning will simultaneously learn all these invariant annotations. The teacher can check the hypotheses by generating verification conditions for the hypothesized invariants and by using automatic theorem provers to check their validity.

Honesty and Progress: The two salient features of ICE learning is that it facilitates *progress* and *honesty*. The teacher can always make progress by adding a positive/negative/implication counterexample such that H (and any other previous hypothesis) does not satisfy it. Furthermore, while augmenting the sample, the teacher can answer honestly, not precluding *any* possible

adequate inductive invariant of the program. Honesty and progress are impossible to achieve when learning just from positive and negative examples (when the hypothesis is not inductive, there is no way to make progress without making a dishonest choice).

Convergence: The setting of iterative ICE learning naturally raises the question of convergence of the learner, that is, does the learner find a correct hypothesis in a finite number of rounds? We say that a learner *strongly converges*, if for every target description (P, N, R) it reaches a correct hypothesis (from the empty sample) after a finite number of rounds, no matter what information is provided by the teacher (of course, the teacher has to answer correctly according to the target description (P, N, R)).

Note that the definition above demands convergence for arbitrary triples (P, N, R), and allows the teacher in each round to provide *any* information that contradicts the current hypothesis, and is hence a very strong property.

Observe now that for a *finite* class $\mathcal{C}$ of concepts, a learner strongly converges if it never constructs the same hypothesis twice. This assumption on the learner is satisfied if it only produces hypotheses H that are consistent with the sample (E, C, I), that is, if $E \subseteq H$, $C \cap H = \emptyset$, and for each pair $(x, y) \in I$, if $x \in H$, then $y \in H$. Such a learner is called a *consistent learner*. Since the teacher always provides a witness for an incorrect hypothesis, the next hypothesis constructed by a consistent learner must be different from all the previous ones.

Lemma 2.2.1. *For a finite class $\mathcal{C}$ of concepts, every consistent learner strongly converges.*

Learning Floyd-Hoare style Proof Annotations: While the above is a general scheme for learning invariants, we are more interested in restricted forms of expressing inductive invariants that use Floyd-Hoare style proof annotations for sequential imperative programs [Flo67,Hoa69]. In Floyd-Hoare style verification for sequential programs with function calls, the annotation mechanism requires pre- and post-conditions for all functions/methods in the program, and loop invariants for every while-loop in the program. Note that these annotations define, in turn, a particular form of an invariant over the set of global configurations of the program, but are restricted in the

sense that they syntactically talk only about the local configuration in scope. Floyd-Hoare style reasoning hence has several sources of incompleteness such as the ability to talk only about the local state in scope, though this can be mitigated by using ghost-variables [Apt81]. Post-conditions, as a result, express properties not just about the state at the method exit, but relate it to the state at the entry to the method as well, and hence we will think about post-conditions encoding properties of pairs of configurations.

As we mentioned above, Floyd-Hoare style proof annotations are restricted to only express properties about variables in the local scope. In particular, post-condition annotations of a function/method relate the return value of the method to only the input arguments (and the variables that are defined in the global scope). Consequently, if there is a function with several method calls, each with formal parameters being a subset of the local variables of the function, the conjunction of the post-conditions of the methods together with the pre-condition of the function defines, logically, the set of global configurations that can be reached in the current scope of the function. This, as a result, couples multiple proof annotations together and several of them together contribute to the safety of a specification. Counterexamples to non-inductiveness of Floyd-Hoare style proof annotations, in general, are, therefore, *horn clauses* and not *implications*. When we mention invariant synthesis, henceforth, we assume that the user supplies adequate pre- and post-conditions for all methods in the program, and we use ICE learning to automatically synthesize loop invariants that are adequate to establish the correctness of pre/post-condition annotations for every method (counterexamples in this scenario are necessarily implications), and subsequently the correctness of the entire program. Even though we have not experimented with and evaluated learning algorithms that generalize to horn counterexamples (and can learn pre/post-condition annotations for methods as well), we must note that all ICE learning algorithms that we present in this thesis can be easily generalized to learn from horn counterexamples, as well.

Assuming specifications are given in the form of assertions, ICE learning can now be used to find the annotations of the program that prove the program correct. We first set up a way to represent sets of configurations and set up an iterative ICE framework to learn such sets of configurations. We can add a new variable l to configurations that denotes the annotation location, and learn a set of configurations which can then be partitioned using this variable

into the various places of annotation in the program. When confronted with a hypothesis H, the teacher will check whether the annotations defined by H prove the program correct (by generating verification conditions that are then validated using an automated theorem prover), and if not, find witnesses (again, using the theorem prover) that show why H does not define an inductive invariant proving the assertions, and appropriately modify the sample (E, C, I) in the next round.

Note that most of the verification conditions, when invalid, will result in an *implication* constraint to be added to the constraint set. However, the entire process of deriving an invariant is property-based (dependent on the specification) as the counterexamples stem for the inability of the invariant to prove the safety property at hand. Also, note that all normal program constructs for while-programs can be handled using this general scheme (multiple loops, nested loops, etc.); however *synthesizing new ghost variables*, updating them, and using them in invariants cannot be done in this framework, and in general, is out of scope of our current work.

2.3 ICE Learning over Lattice Domains

When the class of concepts forms a *lattice*, ICE learning algorithms can be easily constructed, and in fact methods for synthesizing invariants already exist in the literature that can be seen as ICE learning algorithms. In particular, the abstract Houdini algorithm [FL01, TLLR13] and the work reported in [GKT13] for invariant synthesis over numerical abstract domain lattices are ICE learning algorithms.

Consider an abstract domain that is a lattice. Given any sample (E, C, I), we can learn the *best* (smallest with respect to the underlying partial-order) abstract element that satisfies the constraints (E, C, I) as follows:

(1) Set E' to be the set of examples E.

(2) Compute $H-$ the least upper bound of the set of all $\alpha(e)$, for each $e \in E'$, where α is the abstraction operator [CC77]. H is, arguably, the *smallest* hypothesis that can be expressed as an element of the lattice domain and that contains E'. If H does not satisfy the implication constraints I, then we find pairs (e, e') in I where $e \in H$ and $e' \notin H$,

and add the element e' to E', and repeat Step 2 for the new set. If H does not satisfy the negative counterexamples C, then the learner exits stating that there is *no concept* in the lattice domain that meets the constraints of the ICE sample.

To see why the above is a consistent ICE learner that terminates, note that for any E', whenever the learner hypothesizes H, H is the smallest concept (with respect to the underlying partial-order) that contains examples in E'. It follows that for any implication pair (e, e') that is not satisfied by H, *any* concept that satisfies the samples and includes examples in E' must include e' as well. This is so because every concept that includes E' is a superset of H, and hence must include e. This justifies updating E' to include e', and repeating the passive learning algorithm. Note that E' is initialized to the set of positive examples E in Step 1 of the learning algorithm. This establishes the consistency of the base case of the recursive argument. Similarly, following the same reasoning we can show that if there is a counterexample $c \in C$ that belongs to H, then *no concept* exists in the lattice domain that satisfies the ICE sample, since all such concepts are supersets of H and will necessarily include c. In this case, the learning algorithm terminates declaring that there is no concept that satisfies the sample.

To argue termination of the above learning algorithm, note that for every iteration in Step 2, the learner, since it's given new examples that are not in the current hypothesis, is forced to produce strictly larger hypotheses. Since I is finite, this process converges in a finite number of steps. It is also easy to see that the above ICE learner is *consistent*; it produces hypotheses that satisfy all constraints. Consequently, it follows by Lemma 2.2.1 that if the class of concepts is finite, then *iterative ICE learning* using the above algorithm strongly converges.

We can, using this argument, establish polynomial-time (non-iterative) ICE learning algorithms for conjunctive formulas (in fact, this is what the classical Houdini algorithm does [KV94, FL01]), k-CNF formulas [KV94], and for abstract domains such as intervals, octagons, polyhedra, linear equalities, etc. as in [RSY04, YBS06]. However, note that the iterative extension of the above ICE algorithm may not converge (unless the abstract domain has a finite height). One can of course use a widening heuristic after some rounds to terminate [CC77], but then the iterative ICE algorithm will not be necessarily

strongly convergent. The iterative ICE algorithm with widening is, in fact, precisely the *abstract Houdini* algorithm proposed recently in [TLLR13], and is similar to another recent work in [GKT13], and are not *strongly convergent*.

2.4 ICE Learning for Universally Quantified Properties

The ICE learning framework we described in Section 2.2 works for learning quantifier-free concepts that are most naturally used to express invariants over program variables, etc. In this section we describe an extension of ICE learning to learning *universally quantified* concepts. We consider synthesis of universal properties of the form $\psi = \forall y_1, \ldots, y_k.\ \varphi(y_1, \ldots, y_k)$, where φ is a quantifier-free formula. We now describe how to extend the ICE learning framework so that we can learn the universally quantified concept ψ by using a learner for the quantifier-free property described by $\varphi(y_1, \ldots, y_k)$. These universally quantified properties naturally arise in programs over data-structures. A configuration of a program can be described by the heap structure (locations, the various field-pointers, etc.), and a finite set of pointer variables pointing into the heap. Since the heap is unbounded, typical invariants for programs manipulating heaps require universally quantified formulas. For example, a list structure is sorted if the data at *all* pairs y_1, y_2 of successive positions in the list is sorted in the correct order.

The formula ψ describes a set of program configurations with a certain property. In the plain learning setting, the class of all sets that can be described by such universal formulas would form the concept class for the learning problem (over the domain D of all program configurations). We now describe how this setting can be modified in a fairly generic way such that we can instead use a learner for the quantifier-free property (described by the formula $\varphi(y_1, \ldots, y_k)$). For this purpose, we add valuations of the quantified variables to the elements of the quantified domain, and modify the learning setting a bit to reflect the universal semantics of the variables $y_1, \ldots, y_k$.

Consider for each concrete program configuration c, the set S_c of *valuation configurations* of the form (c, val), where val is a valuation of the variables $y_1, \ldots, y_k$. For example, if the configurations are heaps, then the valuation maps each quantified variable y_i to a cell in the heap, akin to a scalar pointer variable. Then $c \models \psi$ if $(c, val) \models \varphi$ for all valuations val, and $c \not\models \psi$ if

$(c, val) \not\models \varphi$ for some valuation val.

This leads to the setting of *data-set based ICE learning*. In this setting, the target description is of the form $(\hat{P}, \hat{N}, \hat{R})$ where $\hat{P}, \hat{N} \subseteq 2^D$ and $\hat{R} \subseteq 2^D \times 2^D$. A hypothesis $H \subseteq D$ is correct if $P \subseteq H$ for each $P \in \hat{P}$, $N \not\subseteq H$ for each $N \in \hat{N}$, and for each pair $(X, Y) \in \hat{R}$, if $X \subseteq H$, then also $Y \subseteq H$. The sample is a finite part of the target description, that is, it is of the form $(\hat{E}, \hat{C}, \hat{I})$, where $\hat{E}, \hat{C} \subseteq 2^D$, and $\hat{I} \subseteq 2^D \times 2^D$. A hypothesis $H \subseteq D$ is consistent with the sample if for each $E \subseteq H$ for each $E \in \hat{E}$, $C \not\subseteq H$ for each $C \in \hat{C}$, and for each pair $(X, Y) \in I$, if $X \subseteq H$, then also $Y \subseteq H$.

An ICE learner for the data-set based setting corresponds to an ICE learner for universally quantified concepts in the original data-point based setting, described in Section 2.2, using the following connection. Given a standard target description (P, N, R) over D, we now consider the domain D_{val} extended with valuations of the quantified variables $y_1, \ldots, y_k$ as described above. Replacing each element c of the domain by the set $S_c \subseteq D_{val}$ transforms (P, N, R) into a set-based target description. Then a hypothesis H (described by a quantifier-free formula $\varphi(y_1, \ldots, y_k)$) is correct w.r.t. the set-based target description iff the hypothesis described by $\forall y_1, \ldots, y_k.\ \varphi(y_1, \ldots, y_k)$ is correct w.r.t. the original target description. Note that augmenting program configurations with valuations for universal variables is different from the concept of "Skolem constants" present in the literature [FQ02]. Unlike "Skolem constants", learning over data-sets allows us to learn from not only examples, but also from counterexamples and implications (where simple Skolem constants do not work). We describe in Chapter 5 a new learning algorithm in the data-set based ICE setting that learns quantified properties over linear data-structures like arrays and lists.

2.5 Remarks

The ICE learning framework presents a way for synthesizing inductive program invariants that is essentially *property-driven*! In other words, the ICE learning framework essentially requires the program to be annotated with a safety specification (or an assertion) to learn a non-trivial inductive invariant. This is very different from other invariant synthesis techniques such as abstract interpretation [CC77] and model checking, learning *likely* invariants using

Daikon [ECGN00], etc. ICE learning over lattice domains, exemplified by the Houdini [FL01] and abstract Houdini [TLLR13] algorithms, though is an instantiation of the ICE learning framework, differs from general ICE learning in this aspect, in that it learns non-trivial invariants even in the absence of any specification. Another point to note is that strongly convergent ICE learners guarantee that, for a *correct* program, the learner will synthesize an invariant for it within a finite number of rounds, and hence prove the program correct. However, there is no termination guarantee on disproving an incorrect (or buggy) program. In the event of analyzing an incorrect program, however, the ICE sample might become inconsistent (implication gives contradictory classifications), in which case we can terminate and output a corresponding error trace.

Finally, we must note that the ICE learning framework relies fundamentally on the availability of a teacher that can return counterexamples to the non-inductiveness of conjectures hypothesized by the learner. While this allows the learner to be completely black-box that learns concepts purely from a sample of counterexamples and brings with it several advantages, the presence of a teacher, which is an NP-oracle making an SMT query, in the iterative-ICE limits, to some extent, the performance of the ICE framework. Recent advances in SMT technology together with an intrinsic compositionality ensure that deductive verification scales to programs with up to hundred thousands lines of code [CDH$^+$09]. Also, ICE learning algorithms based on machine learning, such as the one we present in Chapter 4, are very efficient and scale to very large sample sizes. Still, efficient invariant synthesizers such as those based on abstract interpretation for simple concept classes like intervals, analyses like constant propagation, etc. are not undesirable. In the future, we see existing white-box program analysis techniques such as program slicing and abstract interpretation for simple concept classes being used together with and complementing black-box invariant synthesis using ICE learning.

In this thesis, we develop ICE learning algorithms for learning numerical invariants (Section 3 and Section 4) and an ICE algorithm for learning quantified invariants over linear data-structures such as arrays and lists (Section 5), and evaluate them for synthesizing inductive program invariants.

CHAPTER 3

ICE LEARNING USING CONSTRAINT SOLVING

In this chapter, we present a general approach to ICE learning that reduces the learning problem to a constraint satisfaction problem. We instantiate this approach by developing a concrete learning algorithm for synthesizing numerical program invariants. We describe the algorithm and compare it favorably to a host of other state-of-the-art invariant synthesizers.

Figure 3.2 gives a graphical visualization of the learning problem. At its core, the learning problem is to learn a concept within a predefined concept class that is consistent with a given sample of ICE counterexamples. Figure 3.2 considers a very simple case of a program with two integer variables x_1 and x_2 (the state-space of the program, therefore, consists of the 2-D plane assigning values to these variables). Given an ICE sample, the learner learns an invariant that includes all positive counterexamples, excludes all negative counterexamples and consistently labels the implication counterexamples (see Figure 3.2). Note that the end points of these implications can be labeled by the learner in any way, i.e., they can be included in the concept learned or excluded from the concept, as far as the implication constraint is satisfied. As we will see, the learning approach in this chapter can learn very expressive concept classes, such as when the concept class is not convex. Learning such concept classes is difficult, even in a supervised setting. Learning from a sample that has unlabeled implication counterexamples adds an additional combinatorial complexity to the learning problem. The idea that we follow is to offload the combinatorial complexity to the advances in constraint solving, whereby a constraint solver is used to directly guess an invariant/concept that is consistent with the sample. Note that while doing so, the solver also labels the implication counterexamples in a consistent manner.

An alternate way is to arbitrarily assign labels to end-points of implication counterexamples in a consistent manner, first, and then use supervised learners for learning hypotheses consistent with all the labeled examples in the sample.

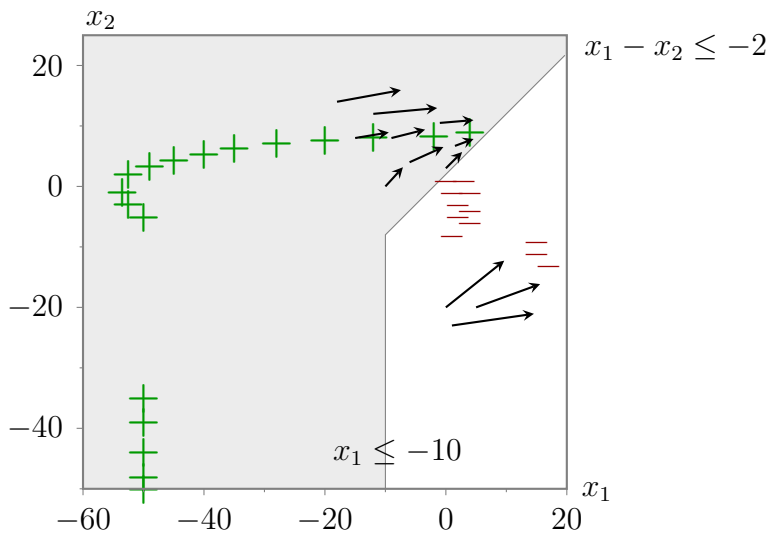


Figure 3.1: Graphical visualization of ICE learning. The program consists of two variables– x_1 and x_2; marked as $+$, $-$ and $\rightarrow$ are positive, negative and implication counterexamples that together comprise the ICE sample; the invariant region learned $x_1 \leq -10 \vee x_1 - x_2 \leq -2$ (colored as light blue) is a subset of the program state-space.

However, as mentioned in Chapter 2, we want the labels for implications to not be fixed beforehand, but be determined during the learning process such that the learned hypothesis generalizes well, has a small size, etc. The concept class we learn in this chapter is very expressive, which can precisely accept only the points that are labeled positively in the sample, and so on. Therefore, preventing *overfitting* of the learned hypothesis is an additional challenge. We solve this by organizing the concept class into a hierarchy of increasing complexity (wrt. size, values of constants in the domain elements, etc.) and explicitly forcing the constraint solver based learner to learn hypotheses of a small size (and, thereby also assigning, at the same time, labels to the implication end-points which are consistent to it). Next, we instantiate the general approach we describe above and develop an ICE learning algorithm for a concept class of numerical invariants.

3.1 Constraint Solver based ICE Learning Algorithm for Synthesizing Numerical Invariants

In this section, we describe a learning algorithm, based on the approach described above, for synthesizing invariants that are arbitrary Boolean combinations of numerical atomic formulas. Since we want the learning algorithm to generalize the sample (and not capture precisely the finite set of implication-closed positive examples), we would like it to learn a formula with the *simplest* Boolean structure. In order to do so, we iterate over templates over the Boolean structure of the formulas, and learn a formula in the given template.

Note that the domain is a join-semilattice (every pair of elements has a least upper bound) since formulas are closed under disjunction. Hence we can employ the generic abstract Houdini algorithm [TLLR13] to obtain a passive ICE learning algorithm. However, using this vanilla algorithm will return concepts that overfit the sample, learning only the precise set of positive and implication-closed set, and hence not generalize without a widening operation. Widening for disjunctive domains is not easy, as there are several ways to generalize disjunctive sets [BHZ04]. Furthermore, even with a widening, we will not get a *strongly convergent* iterative ICE algorithm that we desire (see experiments where abstract Houdini diverges even on conjunctive domains on some programs for this reason). The algorithm we build in this section will not only be strongly convergent but also will produce the *simplest* expressible invariant.

Let $Var = \{x_1, \cdots, x_n\}$ be the set of (integer) variables in the scope of the program. For simplicity, let us restrict atomic formulas in our concept class to octagonal constraints, over program configurations, of the general form:

$$s_1 v_1 + s_2 v_2 \leq c, \quad s_1, s_2 \in \{0, +1, -1\}, \quad v_1, v_2 \in Var, \ v_1 \neq v_2, \quad c \in \mathbb{Z}.$$

(We can handle more general atomic formulas as well; we stick to the above for simplicity and effectiveness.)

Our ICE-learning algorithm will work by iterating over more and more complex *templates* till it finds the simplest formula that satisfies the sample. A *template* fixes the Boolean structure of the desired invariants and also restricts the constants $c \in \mathbb{Z}$ appearing in the atomic formulas to lie within a finite range $[-M, +M]$, for some $M \in \mathbb{Z}^+$. Bounding the constants leads to strong convergence as we show below. For a given template $\bigvee_i \bigwedge_j \alpha^{ij}$,

the iterative ICE-learning algorithm we describe below learns an adequate invariant φ, of the form:

$$\varphi(x_1, \cdots, x_n) = \bigvee_i \bigwedge_j (\ s_1^{ij}\ v_1^{ij}\ +\ s_2^{ij}\ v_2^{ij}\ \leq\ c^{ij}\ ), \quad |\ c^{ij}\ |\ \leq\ M.$$

Given a sample (E, C, I), the learner iterates through templates, and for each template, tries to find concrete values for $s_k^{ij},\ v_k^{ij}$ $(k \in \{1, 2\})$ and c^{ij} such that the formula φ is consistent with the sample; i.e., for every data-point $p \in E$, $\varphi(p)$ holds; for $p \in C$, $\varphi(p)$ does not hold; and for every implication pair $(p, p') \in I$, $\varphi(p')$ holds if $\varphi(p)$ holds. Unfortunately, finding these values in the presence of implications is hard; classifying each implication pair (p, p') as both positive or p as negative tends to create an exponential search space that is hard to search efficiently. As we mentioned before, our ICE learner uses a constraint solver to search this exponential space in a reasonably efficient manner. It does so by checking the satisfiability of the formula Ψ (shown in Figure 3.3), over free variables s_k^{ij}, v_k^{ij} and c^{ij}, which precisely captures all the ICE constraints. In this formula, for every data-point p in the sample, b_p is a Boolean variable which tracks $\varphi(p)$; the Boolean variables b_p^{ij} represent the truth value of $(s_1^{ij}\ v_1^{ij}\ +\ s_2^{ij}\ v_2^{ij}\ \leq\ c^{ij})$ on data-point p (line 3 in Figure 3.3), r_{kp}^{ij} encode the value of $s_k^{ij} \cdot v_k^{ij}$ (line 5 in Figure 3.3); and d_{kp}^{ij} encode the value of v_k^{ij} (line 6 in Figure 3.3).

Note that Ψ falls in the theory of quantifier-free linear integer arithmetic, the satisfiability of which is decidable. A satisfying assignment for Ψ gives a *consistent* formula that the learner conjectures as an invariant. If Ψ is unsatisfiable, then there is no invariant for the current template consistent with the given sample. In this case we iterate by increasing the complexity of the template. For a given template, the class of formulas conforming to the template is finite. Our enumeration of templates *dovetails* between the Boolean structure and the range of constants in the template, thereby progressively increasing the complexity of the template. Consequently, the ICE learning algorithm always synthesizes a consistent hypothesis if there is one, and furthermore synthesizes a hypothesis of the simplest template.

A similar approach can be used for learning invariants over linear constraints, and even more general constraints if there is an effective solver for the resulting theory.

$$
\begin{aligned}
\Psi(s_k^{ij}, v_k^{ij}, c^{ij}) \equiv \quad & \left(\bigwedge_{p\in E} b_p\right) \wedge \left(\bigwedge_{p\in C} \neg b_p\right) \wedge \left(\bigwedge_{(p,p')\in I} b_p \Rightarrow b_{p'}\right) \wedge \\
& \left(\bigwedge_{p} \left(b_p \Leftrightarrow \bigvee_i \bigwedge_j b_p^{ij}\right)\right) \wedge \\
& \left(\bigwedge_{p,i,j} \left(b_p^{ij} \Leftrightarrow \Big(\sum_{k\in\{1,2\}} r_{kp}^{ij} \leq c^{ij} \Big)\right)\right) \wedge \\
& \left(\bigwedge_{i,j} \Big(-M \leq c^{ij} \leq M\Big)\right) \ \wedge \\
& \left(\bigwedge_{\substack{p,i,j \\ k\in\{1,2\}}} \left(\begin{array}{l} s_k^{ij} = 0 \Rightarrow r_{kp}^{ij} = 0 \\ s_k^{ij} = 1 \Rightarrow r_{kp}^{ij} = d_{kp}^{ij} \\ s_k^{ij} = -1 \Rightarrow r_{kp}^{ij} = -d_{kp}^{ij} \end{array}\right)\right) \wedge \\
& \left(\bigwedge_{\substack{p,i,j \\ k\in\{1,2\}}} \bigwedge_{l\in[1,n]} (v_k^{ij} = l \Rightarrow d_{kp}^{ij} = p(l))\right) \wedge \\
& \left(\bigwedge_{\substack{i,j \\ k\in\{1,2\}}} \Big(-1 \leq s_k^{ij} \leq 1\Big)\right) \wedge \left(\bigwedge_{\substack{i,j \\ k\in\{1,2\}}} (1 \leq v_k^{ij} \leq n)\right) \ \wedge \ \left(\bigwedge_{i,j} (v_1^{ij} \neq v_2^{ij})\right)
\end{aligned}
$$

Figure 3.2: The SMT formula Ψ that the learner uses to passively learn a hypothesis consistent with the ICE sample.

3.2 Convergence of the Learning Algorithm

Our iterative ICE algorithm conjectures a consistent hypothesis in each round, and hence ensures that we do not repeat hypotheses. Furthermore, the enumeration of templates using dovetailing ensures that all templates are eventually considered, and together with the fact that there are a finite number of formulas conforming to any template ensures strong convergence.

Theorem 3.2.1. *The above ICE learning algorithm always produces consistent conjectures and the corresponding iterative ICE algorithm strongly converges.*

Our learning algorithm is quite different from earlier white-box constraint based approaches to invariant synthesis [CSS03,GSV08,GMR09,GR09]. These

approaches directly encode the adequacy of the invariant (encoding the entire program's body) into a constraint, and use Farkas' lemma to reduce the problem to satisfiability of quantifier-free non-linear arithmetic formulas, which is harder and in general undecidable. We, on the other hand, split the task between a white-box teacher and a black-box learner, communicating only through ICE constraints on concrete data-points. This greatly reduces the complexity of the problem, leading to a simple teacher and a much simpler learner.

3.3 Experimental Results

We have implemented our learning algorithm as an invariant synthesis tool [1] in Boogie [BCD+05]. In our tool we enumerate templates in an increasing order of their complexity. For a given Boolean structure of the template B_i, we fix the range of constants M in the template to be the greater value out of i and the maximum integer in the program. If an adequate invariant is not found, we increase i and look at the next template for the invariants. If an adequate invariant is found for a given template B_i and M, we use binary search on M to find an invariant that has the same Boolean structure but the smallest constants. This enumeration of templates is complete and it ensures that we learn the *simplest* invariant. In our tool, ICE counterexamples discovered while learning an invariant belonging to a simpler template are not wasted but used in subsequent rounds. As already mentioned, our learner uses an incremental Z3 [dMB08] solver that adds a new constraint for every ICE counterexample discovered by the Boogie based teacher.

We evaluate our tool on SV-COMP benchmarks[2] and several other programs from the literature (see Table 3.1). We use SMACK [RE] to convert C programs to Boogie and use our tool to learn loop invariants for the resulting Boogie programs. We use inlining to infer invariants for programs with multiple procedures. In Table 3.1 we compare our tool to invariant synthesis using abstract Houdini [TLLR13] (called `absH` in Table 3.1), a black-box invariant learning algorithm based on computational geometry [SGH+13b] (called `Geometric`), Invgen [GR09] and the interpolation based Impact algo-

[1] Available at `http://www.cs.uiuc.edu/~madhu/cav14/`

[2] `https://svn.sosy-lab.org/software/sv-benchmarks/tags/svcomp13/loops/`

Program	Invariant	White Box		Black Box		
		InvGen [GR09]	CPA [BK11]	absH [TLLR13]	Geometric [SGH+13b]	ICE-CS
w1[GSV08]	$x \leq n$	0.1	×	0.1	0.2	**0.0**
w2[GSV08]	$x \leq n-1$	0.1	×	0.2	0.1	**0.0**
fig1[GSV08]	$x \leq -1 \vee y \geq 1$	×	4.5	×	×	**0.1**
fig3[GHK+06]	$lock = 1 \vee x \leq y-1$	0.1	1.4	×	0.1	**0.0**
fig6[GHK+06]	$true$	0.1	1.3	0.1	0.1	**0.0**
fig8[GHK+06]	$true$	0.0	1.4	0.0	0.0	**0.0**
fig9[GHK+06]	$x = 0 \wedge y \geq 0$	0.1	1.4	0.0	0.2	**0.0**
ex7 [IS]	$0 \leq i \wedge y \leq len$	×	1.6	0.2	0.4	**0.0**
ex14[IS]	$x \geq 1$	×	1.5	0.2	0.2	**0.0**
ex23[IS]	$0 \leq y \leq z \wedge z \leq c + 4572$	×	90.5	0.2	×	**14.2**
finf1	$x = 0$	0.1	1.5	0.1	0.4	**0.0**
finf2	$x = 0$	0.1	1.4	0.0	0.1	**0.0**
winf1	$x = 0$	0.0	1.4	0.0	0.0	**0.0**
winf2	$x = 0$	0.0	1.4	0.0	0.0	**0.0**
winf3	$x = 0$	×	1.4	0.3	0.1	**0.1**
term2	$true$	0.0	1.6	0.0	0.0	**0.0**
term3	$true$	0.0	1.4	0.0	0.0	**0.0**
sum1	$sn = i - 1 \wedge (sn = 0 \vee sn \leq n)$	×	×	×	×	**1.8**
sum3	$sn = x$	0.1	1.5	0.1	0.1	**0.0**
sum4	$sn = i - 1 \wedge sn \leq 8$	0.1	2.8	×	×	**2.6**
trex1	$z >= 1$	0.1	1.5	0.1	0.4	**0.0**
trex2	$true$	0.0	1.4	0.0	0.0	**0.0**
trex3	$0 \leq x1 \wedge 0 \leq x2 \wedge 0 \leq x3 \wedge d1 = 1 \wedge d2 = 1 \wedge d3 = 1$	0.5	×	×	×	**2.2**
trex4	$true$	0.0	1.4	0.0	0.0	**0.0**
tcs [JM06]	$i \leq j - 1 \vee i \geq j + 1 \vee x = y$	0.1	1.4	×	0.5	**1.4**
array	$j \leq 0 \vee m \leq a[0]$	×	×	×	0.2	**0.3**
lucmp	$n = 5$	×	77.0	0.0	0.1	**0.0**
matrix	$a[0][0] \leq m \vee j \leq 0;$ $a[0][0] \leq m \vee j + k \leq 0$	×	×	×	×	**5.8**
n.c11	$0 \leq len \leq 4$	0.1	2.2	×	0.2	**0.6**
cgr1[GSV08]	$x - y \leq 2$	0.1	1.5	0.1	0.2	**0.0**
cgr2[GSV08]	$N \leq 0 \vee (x \geq 0 \wedge 0 \leq m \leq N - 1)$	×	1.8	×	×	**7.3**
oct	$x + y \leq 2$	0.0	1.3	0.2	0.1	**0.2**
vmail	$i \geq 0$	×	1.4	0.1	0.3	**0.0**
vbsd	$pathlim \leq tmp$	×	1.6	0.5	×	**0.0**

Table 3.1: Results for ICE learning numerical invariants. All times reported are in seconds. × means that the corresponding verification tool could not find an adequate invariant.

rithm [McM06b] implemented in CPAchecker (called `CPA`) [BK11].

We implemented the octagonal domain in abstract Houdini for a comparison with our tool. As mentioned in Chapter 2.3, abstract Houdini is an ICE learning algorithm but is not strongly convergent. Unlike our tool, abstract Houdini is not able to learn disjunctive octagonal invariants. In addition, it is unable to prove programs like `trex3` and `n.c11` where a conjunctive invariant exists but abstract Houdini loses precision due to widening, and is unable to synthesize such an invariant.

InvGen [GR09] uses a white-box constraint-based approach to invariant synthesis. Unlike our tool that enumerates all templates, InvGen requires the user to specify a template for the invariants. Also, being white-box, it cannot handle programs with arrays and pointers, even if the required invariants are numerical formulas over scalar variables. Invgen is also incomplete (it reduces invariant synthesis to solving satisfiability of a non-linear constraint, which is undecidable) and is unable to prove several scalar programs like `fig1` and `cegar2`.

Finally, [SGH$^+$13b] is a computational geometry based learning algorithm for inferring numerical invariants. This learning algorithm is not an ICE algorithm– it conjectures an invariant from a sample consisting of purely positively and negatively labeled data-points that manifest along dynamic test runs, and we found it to be non-robust. From our experience, the inference procedure in [SGH$^+$13b] is very sensitive to the test harness used to obtain the set of safe/unsafe program configurations. For several programs, we could not learn an adequate invariant using [SGH$^+$13b] despite many attempts with different test harnesses.

The experiments show that our tool outperforms [TLLR13,SGH$^+$13b,GR09, BK11] on most programs, and learns an adequate invariant for all programs in reasonable time. Though we use the more complex but more robust framework of ICE learning that promises to learn the simplest invariants and is strongly convergent, it is generally faster than other learning algorithms like [SGH$^+$13b, SNA12] that learn invariants from just positive and negative examples, and lack any such promises.

CHAPTER 4

ICE LEARNING NUMERICAL INVARIANTS USING DECISION TREES

One of the biggest advantages of the black-box learning paradigm is the possible usage of *machine learning techniques* to synthesize invariants. The learner, being completely agnostic to the program (its programming language, semantics, etc.), can be seen as a machine learning algorithm that learns a Boolean classifier of configurations [Mit97b]. Machine learning algorithms can be trained to learn functions that belong to various classes of Boolean functions, such as k-CNF/k-DNF, Linear Threshold Functions, Decisions Trees, etc., and some algorithms have already been used in invariant generation [SNA12]. However, standard machine learning algorithms for classification are trained on given sets of positive and negative examples, but do not handle implications and hence do not help in building robust learning platforms for invariant generation.

In this section, we build the first true machine learning algorithms for robust invariant generation. We adapt and extend classical scalable machine learning algorithms for constructing decision trees (which can express any Boolean function) to ICE learning algorithms. We show that this results in efficient algorithms for synthesizing invariants.

Decision trees over a set of attributes provide a universal representation of a Boolean function defined over a fixed set of numerical and categorical attributes. Internal nodes in decision trees are decision variables that split over a value of a single attribute, and the leaves are labeled with classification labels (positive or negative in our setting).

Our starting point is the well-known decision tree learning algorithms of Quinlan [Qui86, Qui93, Mit97b] that work by constructing the tree top-down from positive and negative samples. These are efficient algorithms as they choose heuristically the best attribute that classifies the sample at each stage of the tree based on statistical measures, and do not backtrack nor look ahead. One of the well-known ways to pick these attributes is based on a notion

called *information gain*, which is in turn based on a statistical measure using Shannon's entropy function [Sha48, Qui93, Mit97b]. The inductive bias in these learning algorithms is roughly to compute the *smallest* decision tree that is consistent with the sample—a bias that again suits our setting well, as we would like to construct the smallest invariant formula amongst the large number of invariants that may exist. Machine learning algorithms, including the decision tree learning algorithm, often do not produce concepts that are fully consistent with the given sample—this is done on purpose to avoid over-fitting to the training set, under the assumption that, once learned, the hypothesis will be evaluated on new, previously unseen data. We remove these aspects from the decision tree algorithm (which includes, for example, pruning to produce smaller trees at the cost of inconsistency) as we aim at identifying a hypothesis that is *correct* rather than one that only *approximates* the target hypothesis.

We first present a generic top-down decision tree algorithm that works on samples with implications. This algorithm constructs a tree top-down, classifying end-points of implications in a way that reduces the sizes of trees and guarantees to always create a tree that is consistent with the sample (Section 4.2). Next, we develop several novel "information gain" measures that are used to determine the best attribute to split on the current collection of examples and implications, while learning decision trees (Section 4.3). Finally, we implement our ICE decision tree algorithms (i.e., the generic ICE learning algorithm with the various statistical measures for choosing attributes) and build teachers to work with these learners to guide them towards learning invariants. We perform extensive experiments that show that the decision tree learners we build are competitive, and superior in performance and convergence to existing black-box ICE learners, including one based on randomized search [SA14] as well as a constraint-solving based ICE learner presented in [GLMN14]. We use our tool to find invariants in around 35 programs, and the new learning techniques work extremely efficiently and, surprisingly, converge on all examples.

We believe that this work breaks new ground in adapting machine learning techniques to invariant synthesis, giving the first efficient robust ICE machine-learning algorithms for synthesizing invariants.

4.1 Background: Learning Decision Trees from Positive and Negative Examples

Our algorithm for learning invariants builds on the classical recursive algorithm to build *decision trees* proposed by Quinlan (we refer the interested reader to standard texts on learning for more information on decision tree learning [Mit97b]). The reader is encouraged to think of decision trees as Boolean combinations of formulae of the form $a_i \leq c$, where a_i is drawn from a fixed set of *numerical attributes* A (which assign a numerical value to each sample) and c is a constant, or of the form b_i, where b_i is drawn from a fixed set of *Boolean attributes* (which assign a truth value to each sample). When performing invariant learning, we will fix a set of attributes typically as certain arithmetic combinations of integer variables (for example, octagonal combinations of variables or certain linear combinations of variables with bounded co-efficients). Boolean attributes are useful for other non-numerical constraints (are x and y aliased, does x point to *nil*, etc.). Consequently, the learner would learn the thresholds (the values for c in $a_i \leq c$) and the Boolean combination of the resulting predicates, including arbitrary conjunctions of disjunctions as a proposal for the invariant. As an example, in Figure 3.2, x_1 and $x_1 - x_2$ are the relevant numerical attributes with constant thresholds -10 and -2, respectively, and the decision tree learner would learn these thresholds for these attributes, together with the Boolean form of the invariant, which is a disjunction of the two half-spaces: $x_1 \leq -10$ and $x_1 - x_2 \leq -2$.

Quinlan's algorithm, sketched in pseudo code as Algorithm 1, builds the tree top-down (without backtracking), choosing the best attribute at each stage using an information theoretic measure. It has been implemented by the ID3, C4.0, and C5.0 algorithms [Qui86, Qui93, Mit97b].

The crucial aspect of the extremely scalable decision tree learning algorithms is that they choose the attribute for the current sample in some heuristic manner, and never back-track (or look forward) to optimize the size of the decision tree. The prominent technique for choosing attributes is based on a statistical property, called *information gain*, to measure how well each attribute classifies the examples at any stage of the algorithm. This measure is typically defined using a notion called *Shannon entropy* [Sha48], which, intuitively, captures the impurity of a sample. The entropy of a sample S with p positive samples and n negative samples is a value between 0 and 1,

Algorithm 1: The basic inductive decision tree construction algorithm underlying ID3, C4.0, and C5.0

input : A sample $S = \langle S^+, S^- \rangle$ and *Attributes*

Return `ID3` ($\langle S^+, S^- \rangle$, *Attributes*).

Proc `ID3` *(Examples* $= \langle Pos, Neg \rangle$*, Attributes)*
  Create a root node of the tree.
  if *all examples are positive or all are negative* **then**
    Return the single node tree Root with label + or −, respectively.
  else
    Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples*.
    Label the root of the tree with this attribute (and threshold).
    Divide *Examples* into two sets: $Examples_a$ that satisfy the predicate defined by attribute (and threshold), and $Examples_{\neg a}$ that do not.
    Return tree with root and left tree `ID3` ($Examples_a$, *Attributes*) and right subtree `ID3` ($Examples_{\neg a}$, *Attributes*) ;
  end

defined to be

$$Entropy(S) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}.$$

Intuitively, if the sample contains only positive (or only negative) points (i.e., if $p = 0$ or $n = 0$), then its entropy is 0, while if the sample had roughly an equal number of positive and negative points, then its entropy is close to 1.

When evaluating an attribute a (and threshold) on a sample S, splitting it into S_a and $S_{\neg a}$ (points satisfying the attribute and points that do not), one computes the information gain of that attribute (w.r.t. the chosen threshold): the information gain is the difference between the entropy of S and the sum of the entropies of S_a and $S_{\neg a}$ weighted by the number of points in the respective samples. For numerical attributes, the thresholds also need to be synthesized; in the case of information gain, however, it turns out that the best threshold is always at some value occurring in the points in the sample [Qui93]. The algorithm chooses the attribute that results in the largest information gain.

The above heuristic for greedily picking the attribute that works best at each level has been shown to work very well in large and a wide variety of machine learning applications [Qui93, Mit97b]. When decision tree learning is used in machine learning contexts, there are other important aspects: (a) the learning is achieved using a small portion of the available sample, so that the tree learned can be evaluated for accuracy against the rest of the sample, and

(b) there is a *pruning* procedure that tries to generalize and reduce the size of the tree so that the tree does not overfit the sample. When using decision tree learning for synthesizing invariants, we prefer to use all the samples as we anyway place the passive learning algorithm in a larger context by building a teacher which is a verification oracle. Also, we completely avoid the pruning phase since pruning often produces trees that are (mildly) inconsistent with the sample; since we cannot tolerate any inconsistent trees, we prefer to avoid this (incorporating pruning in a meaningful and useful way in our setting is an interesting future direction).

In the context of invariant synthesis, we assume that all integer variables mentioned in the program occur explicitly as numerical attributes; hence, it turns out that *any* sample of mixed positive and negative points can be split (potentially using the same attribute repeatedly with different thresholds) and eventually separated into purely positive and purely negative points (in the worst case, each point is separated into its own leaf). Consequently, we are guaranteed to always obtain some decision tree that is consistent with any input sample.

4.2 A Generic Decision Tree Learning Algorithm in the Presence of Implications

In this section, we present the skeleton of our new decision tree learning algorithm for samples containing implication examples in addition to positive and negative examples. We present this algorithm at the level of Algorithm 1, excluding the details of *how* the best attribute at each stage is chosen. In Section 4.3, we articulate several different natural ways of choosing the best attribute, and evaluate them in experiments.

Our goal in this section is to build a top-down construction of a decision tree for an *ICE sample*, such that the tree is guaranteed to be consistent with respect to the sample; an ICE sample is a tuple $S = (S^+, S^-, S^{\Rightarrow})$ consisting of a finite set S^+ of positive points, a finite set S^- of negative points, and a finite set $S^{\Rightarrow}$ of pairs of points corresponding to the implications. The algorithm is an extension of the classical decision tree algorithm presented in Section 4.1, which preserves the property to be consistent with positive and negative samples. The main hurdle we need to cross is to construct a tree

Algorithm 2: The basic decision tree construction algorithm for ICE samples

input : An ICE sample $S = \langle S^+, S^-, S^{\Rightarrow} \rangle$ and *Attributes*

Initialize global *Impl* to $S^{\Rightarrow}$.

Initialize G, a partial valuation of end-points of implications in *Impl* , to be empty.

Let *Unclass* be the set of all end-points of implications in *Impl* .

Set G to be the transitive closure of the positive and negative classifications in S^+ and S^- with respect to *Impl*.

Return `DecTreeICE` ($\langle S^+, S^-, \mathit{Unclass} \rangle$, *Attributes*).

Proc `DecTreeICE` *(Examples* $= \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass} \rangle$*, Attributes)*

  Move all points from *Unclass* that are positively, respectively negatively, classified in G to *Pos*, respectively *Neg*.

  Create a root node of the tree.

  if $\mathit{Neg} = \emptyset$ **then**

    Mark all elements in *Unclass* as positive in G.

    Take the implication closure of G w.r.t. *Impl*.

    Return the single node tree Root, with label $+$.

  else if $\mathit{Pos} = \emptyset$ **then**

    Mark all elements in *Unclass* as negative in G.

    Take the implication closure of G w.r.t. *Impl*.

    Return the single node tree Root, with label $-$.

  else

    Select an attribute a (and a threshold c for a if a is numerical) that (heuristically) *best* classifies *Examples* and *Impl*.

    Label the root of the tree with this attribute a (and threshold c).

    Divide *Examples* into two sets: $\mathit{Examples}_a$ that satisfy the predicate defined by the attribute (and threshold) and $\mathit{Examples}_{\neg a}$ that do not.

    $T_1 = \mathit{DecTreeICE}(\mathit{Examples}_a, \mathit{Attributes})$ (may update G).

    $T_2 = \mathit{DecTreeICE}(\mathit{Examples}_{\neg a}, \mathit{Attributes})$ (may update G).

    Return tree with root, left tree T_1, and right tree T_2.

  end

consistent with the implications. Note that the pairs of points corresponding to implications do not have a classification, and it is the learner's task to come up with a classification in a manner consistent with the implication constraints. As part of the design, we would like the learner to not classify the points a priori in any way, but classify these points in a way that leads to a smaller concept (or tree).

Our algorithm, shown in pseudo code as Algorithm 2, works as follows. First, given an ICE sample $\langle S^+, S^-, S^{\Rightarrow} \rangle$ and a set of attributes, we store $S^{\Rightarrow}$ in a global variable *Impl* and create a set *Unclass* of unclassified points

as the end-points of the implication samples. We also create a global table G that holds the partial classification of all the unclassified points (initially empty). We then call our recursive decision tree constructor with the sample $\langle S^+, S^-, \mathit{Unclass}\rangle$.

Receiving a sample $\langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass}\rangle$ of positive, negative, and unclassified examples, our algorithm chooses the best attribute that divides this sample, say a, and recurses on the two resulting samples $\mathit{Examples}_a$ and $\mathit{Examples}_{\neg a}$. Unlike the classical learning algorithm, we do not recurse *independently* on the two sets—rather we recurse first on $\mathit{Examples}_a$, which will, while constructing the left subtree, make classification decisions on some of the unclassified points, which in turn will affect the construction of the right subtree for $\mathit{Examples}_{\neg a}$ (see the `else` clause in Algorithm 2). The new classifications that are decided by the algorithm are stored and communicated using the global variable G.

Whenever Algorithm 2 reaches a node where the current sample has only positive points and implication end-points that are either classified positively or unclassified yet, then the algorithm will, naturally, decide to mark *all* remaining unclassified points positive, and declare the current node to be a leaf of the tree (see first conditional in the algorithm). Moreover, it marks in G all the unclassified end-points of implications in *Unclass* as positive and propagates this constraint across implications (taking the implication closure of G with respect to the global set *Impl* of implications). For instance, if (x, y) is an implication pair, both x and y are yet unclassified, and the algorithm decides to classify x as positive, it propagates this constraint by making y also positive in G; similarly, if the algorithm decided to classify y as negative, then it would mark x also as negative in G. Deciding to classify x as negative or y as positive places no restrictions on the other point, of course.

We need to argue why Algorithm 2 always results in a terminating procedure that constructs a decision tree consistent with the sample. As a preparation, we introduce a property of the sample and the partial valuation for the implication end-points, called a *valid sample*.

In the following description, let us fix an ICE sample $S = \langle S^+, S^-, S^{\Rightarrow}\rangle$, and let G be a partial valuation of end-points of in $S^{\Rightarrow}$. By abuse of notation, we use $S \cup G$ to refer to the sample one obtains from classifying the end-points of implications in $S^{\Rightarrow}$ with the (partial) valuation of G.

Definition 4.2.1 (Valid sample)**.** *A sample* $S = \langle S^+, S^-, S^\Rightarrow \rangle$ *is* valid *if for every implication* $(x, y) \in S^\Rightarrow$

1. *it is not the case that* x *is classified positively and* y *negatively in* $S \cup G$*;*
2. *it is not the case that* x *is classified positively and* y *is unclassified in* $S \cup G$*; and*
3. *it is not the case that* y *is classified negatively and* x *is unclassified in* $S \cup G$*.*

A valid sample has the following property.

Lemma 4.2.2. *For any valid sample (with partially classified end-points of implications), extending it by classifying all unclassified points as positive will result in a consistent classification, and extending it by classifying all unclassified points as negative will also result in a consistent classification.*

Proof of Lemma 4.2.2. The above lemma is easy to prove: first, consider the extension of a valid sample by classifying all unclassified points as positive. Assume, for the sake of contradiction, that this valuation is inconsistent. Then, there exists an implication pair (x, y) such that x is classified as positive and y is classified as negative. Since such an implication pair could not have already existed in the valid sample (by definition), it must have been caused by the extension. Since we introduced only positive classifications, it must have been that x (and not y) is the only new classification. Hence the valid sample must have had the implication pair (x, y) with y classified as negative and x being unclassified, which contradicts Condition 3 of Definition 4.2.1. The proof of the extension with negative classifications follows from similar arguments. □

However, Algorithm 2 does not classify *all* implication end-points completely positive, or completely negative; recall that Algorithm 2 only changes the classification (from unknown to positive or negative, respectively) of unclassified implication end-points in the current leaf and those that need to be updated during the implication closure. It is not hard to verify that even such a partial assignment preserves consistency of an ICE sample.

Corollary 4.2.3. *Lemma 4.2.2 also holds if a subset of unclassified points are classified (completely positively or completely negatively) and the implication closure is taken.*

It is now straightforward to prove the correctness of the decision tree ICE learner.

Theorem 4.2.4. Algorithm 2, independent of how the attributes are chosen to split a sample, always terminates and produces a decision tree that is consistent with the input ICE sample.

Proof of Theorem 4.2.4. Theorem 4.2.4 follows from the fact that Algorithm 2 always maintains a valid sample:

1. Algorithm 2 receives an ICE sample and applies the implication closure, which results in a valid sample (or an inconsistency is detected and the learning stops as there does not exist a decision tree that classifies the sample correctly while satisfying the implication constraints).

2. When Algorithm 2 arrives at a leaf that has only *positive and unclassified* points, it classifies all these unclassified points to be positive and takes the implication closure. Assuming that the ICE sample was valid, the new sample is also valid due to Corollary 4.2.3. In the case that Algorithm 2 arrives at a leaf that has only *negative and unclassified* points, validity of the resulting sample follows using similar arguments.

The above argument shows that Algorithm 2 never ends up in an inconsistent sample, which proves its correctness. Moreover, we assume the presence of (numerical) attributes that can always split any sample subset with more than one example, which proves termination as well. □

The running time of Algorithm 2 depends, of course, on the size of the decision tree learned and the time taken to choose the best attribute in each recursive call, which is sub-quadratic in the size of the sample and is linear in the number of attributes. We explore several different ways to choose the best attribute at each stage in the next section, all of which are linear or quadratic on the sample.

4.3 Choosing Attributes in the Presence of Implications

Algorithm 2 ensures that the resulting decision tree is consistent with the given sample, irrespective of the exact mechanism used to determine the attributes

(a) Sample before splitting (b) Samples after splitting

Figure 4.1: The samples discussed in Example 4.3.1.

to split and their thresholds. As a consequence, the original split heuristic based on information gain (see Section 4.1), which is unaware of implications, might simply be employed. However, since our algorithm propagates the classification of data points once a leaf node has been reached, just ignoring implications can easily lead to seemingly good splits that later turn into bad ones. The following example illustrates such a situation.

Example 4.3.1. *Suppose that Algorithm 2 processes the sample shown in Figure 4.1a, which also depicts the (only) implication in the global set Impl.*

When using the original split procedure (i.e., using information gain while ignoring the implication and its corresponding unclassified data points), the learner splits the sample with respect to attribute x at threshold $c = 3$ since this split yields the highest information gain—the information gain is 1 since the entropy of the resulting two subsamples is 0. Using this split, the learner partitions the sample into Examples_a and $\text{Examples}_{\neg a}$ and recurses on Examples_a. Since Examples_a contains only unclassified and positively classified points, it turns all unclassified points in this sample positive and propagates this information along the implication. This results in the situation depicted in Figure 4.1b. Note that the learner now needs to split $\text{Examples}_{\neg a}$ since the unclassified data points in it are now classified positively.

On the other hand, if we consider the implications and their corresponding data points while deciding the split, it allows us to split the sample such that the antecedent and the consequent of the implication both belong to either Examples_a or $\text{Examples}_{\neg a}$ (e.g., on splitting with respect to x with threshold $c = 4$). Such a split has the advantage that no further splits are required and, often, results in a smaller tree. ⌟

In fact, experiments showed that a learner which ignores implications when choosing an attribute often learns relatively large decision trees or even diverges. Hence, we next propose two methods that take implications into

account while choosing the attribute to split.

Penalizing Cutting Implications: In order to better understand how to deal with implications, we analyzed classifiers learned by other ICE-learning algorithms for invariant synthesis, such as the randomized search of [SA14] and the constraint solver-based ICE learner of [GLMN14]. This analysis showed that the classifiers finally learned (and also those conjectured during the learning) almost always classify the antecedent and consequents of implications equally (either both positive or both negative).

The fact that successful ICE learners almost always classify antecedents and consequents of implications equally suggests that our decision tree learner should avoid to "cut" implications. Formally, we say that an implication $(p, q) \in \mathit{Impl}$ is *cut* by the samples S_a and $S_{\neg a}$ if $p \in S_a$ and $q \in S_{\neg a}$, or $p \in S_{\neg a}$ and $q \in S_a$;[1] in this case, we also say that the split of S into S_a and $S_{\neg a}$ cuts the implication.

A straightforward approach to discourage cutting implications builds on top of the original information gain and imposes a penalty for every implication that is cut. This idea gives rise to the *penalized information gain* that we define by:

$$\mathit{Gain}_{pen}(S, S_a, S_{\neg a}) \quad = \quad \mathit{Gain}(S, S_a, S_{\neg a}) \; - \; \mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl}) \quad (4.1)$$

where $S_a, S_{\neg a}$ is the split of the sample S, $\mathit{Gain}(S, S_a, S_{\neg a})$ is the original information gain based on Shannon's entropy, and $\mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl})$ is a total penalty function that we assume to be monotonically increasing with the number of implications cut (we make this precise shortly). Note that this new information gain does not prevent the cutting of implications (if required) but favors not to cut them.

However, not every cut implication poses a problem: implications whose antecedents are classified negatively and whose consequents are classified positively are safe to cut (as this helps creating more pure samples), and we do not want to penalize cutting those. Since we do not know the classifications of unclassified points when choosing an attribute, we penalize an implication depending on how "likely" it is an implication of this type (i.e., we assign no penalty if the sample containing the antecedent is predominantly negative and the one containing the consequent is predominantly positive). More

[1] Given a sample $S = \langle \mathit{Pos}, \mathit{Neg}, \mathit{Unclass} \rangle$, we write $x \in S$ as a shorthand notation for $x \in \mathit{Pos} \cup \mathit{Neg} \cup \mathit{Unclass}$.

precisely, given the samples S_a and $S_{\neg a}$, we define the penalty function $\mathit{Penalty}(S_a, S_{\neg a}, \mathit{Impl})$ by

$$\Big(\sum_{\substack{(x,y)\in \mathit{Impl}\\ x\in S_a, y\in S_{\neg a}}} 1 - f(S_a, S_{\neg a})\Big) + \Big(\sum_{\substack{(x,y)\in \mathit{Impl}\\ x\in S_{\neg a}, y\in S_a}} 1 - f(S_{\neg a}, S_a)\Big),$$

where for two samples $S_1 = \langle \mathit{Pos}_1, \mathit{Neg}_1, \mathit{Unclass}_1\rangle$, $S_2 = \langle \mathit{Pos}_2, \mathit{Neg}_2, \mathit{Unclass}_2\rangle$,

$$f(S_1, S_2) = \frac{|\mathit{Neg}_1|}{|\mathit{Pos}_1| + |\mathit{Neg}_1|} \cdot \frac{|\mathit{Pos}_2|}{|\mathit{Pos}_2| + |\mathit{Neg}_2|}$$

is the relative frequency of the negative points in S_1 and the positive points in S_2 (which can be interpreted as likelihood of an implication from S_1 to S_2 being safe).

Extending entropy to an ICE Sample: In the second variant of information gain that we develop for deciding the *best* attribute to split a given ICE sample, we do not change the definition of information gain (as a function of the entropy of the sample) but we extend Shannon's entropy to deal with implications in the sample using conditional probabilities. The entropy of a set of examples is a function of the discrete probability distribution of the classification of a point drawn randomly from the examples. In a classical sample that only has points labeled positive or negative, one could count the fraction of positive (or negative) points in the set to compute these probabilities. However, an estimation of these probabilities becomes non-trivial in the presence of unclassified points that can be classified as either positive or negative. Moreover, in an ICE sample, the classification of these points is not independent anymore as the classification for the points need to satisfy the implication constraints. Given a set of examples with implications and unclassified points, we will first estimate the probability distribution of the classification of a random point drawn from these examples, taking into account the implication constraints, and then use it for computing the entropy. We will use this new entropy to compute the information gain while choosing the attribute for the split.

Given $S = \langle \mathit{Pos, Neg, Unclass}\rangle$, and a set of implications Impl, let Impl_S be the set of implications projected onto S such that both the antecedent and consequent end-points in the implication are unclassified (i.e., $\mathit{Impl}_S = \{(x_1, x_2) \in \mathit{Impl} \mid x_1, x_2 \in \mathit{Unclass}\}$). For the purpose of entropy computation, we will assume that there is no point in the examples that is common to

more than one implication. This is a simplifying assumption, which also holds statistically if the space enclosing all the points is much larger than the number of points. Let $Unclass' \subseteq Unclass$ be the set of unclassified points in the sample that are not part of any implication in $Impl_S$ (for example, $x_1 \in Unclass'$ if $(x_1, x_2) \in Impl$ *and* $x_2 \in Pos$). Note that points in $Unclass'$ can be classified as either positive or negative by the learner, completely independent of the classification of any other point. This is, for instance, not true for points that are end-points of implications in $Impl_S$.

Let $Pr(x = c)$ be the probability of the event that a point $x \in S$ is classified as c, where c is either positive/+ or negative/−. Note that $Pr(x = +)$ is 1 when $x \in Pos$ and is 0 when $x \in Neg$. Let us define $Pr(S, c)$ to be the probability of the event that a point which is drawn randomly from S is classified as c. Then,

$$\begin{aligned} Pr(S,+) &= \frac{1}{|S|} \sum_{x \in S} Pr(x = +) \\ &= \frac{1}{|S|} \Big(\sum_{x \in Pos \cup Neg \cup Unclass'} Pr(x = +) \quad + \\ &\qquad \sum_{(x_1, x_2) \in Impl_S} Pr(x_1 = +) \quad + \quad Pr(x_2 = +) \Big) \end{aligned} \tag{4.2}$$

Recall that unclassified points $x_u \in Unclass'$ are statistically classified by the learner completely independent of other points in the sample; so, we assume that the probability that a point x_u is classified as positive (or negative) is in accordance with the distribution of points in the sample set S. In other words, we recursively assign $Pr(x_u = +) = Pr(S, +)$.

For points x_1 and x_2 that are involved in an implication $Impl_S$, we assume that the antecedents x_1 are classified independently and the classification of consquents x_2 is conditionally dependent on the classification of antecedents, such that the implication constraint is satisfied. As a result, we assign $Pr(x_1 = +) = Pr(S, +)$, for the same reason as described for x_u above. And for consequents x_2, using conditional probabilities we obtain, $Pr(x_2 = +) = Pr(x_2 = + \mid x_1 = +) \cdot Pr(x_1 = +) + Pr(x_2 = + \mid x_1 = -) \cdot Pr(x_1 = -)$. From the implication constraint between x_1 and x_2, we know that x_2 is guaranteed to be positive if x_1 is classified positive, i.e., $Pr(x_2 = + \mid x_1 = +) = 1$. However, when x_1 is classified negative, the consequent x_2 is allowed

to be classified as either positive or negative completely independently, and hence we assign $Pr(x_2 = + \mid x_1 = -) = Pr(S, +)$.

Plugging in these values for probabilities in Equation 4.2 and using $p = |Pos\ |, n = |Neg\ |, i = |Impl\ |, u' = |Unclass'\ |\ and\ |S\ | = p + n + 2i + u'$, $Pr(S, +)$ is the positive solution of the following quadratic equation:

$$ix^2 + (p + n - i)x - p = 0$$

As a sanity check, note that $Pr(S, +) = \frac{p}{p+n}$, if there are no implications in the sample set (i.e., $i = 0$). Also, $Pr(S, +) = 1$ if $n = 0$ and $Pr(S, +) = 0$ if $p = 0$ (i.e., when the set S has no negative or positive points). Once we have computed $Pr(S, +)$, the entropy of S can be computed in the standard way as

$$Entropy(S) = -Pr(S, +) \cdot \log_2\ Pr(S, +) - Pr(S, -) \cdot \log_2\ Pr(S, -)$$

where $Pr(S, -) = \Big(1 - Pr(S, +)\Big)$. Now, we plug this new entropy in the information gain and obtain a gain measure that explicitly takes implications into account.

4.4 Experiments and Evaluation

To assess the performance of our decision tree ICE learner, we implemented a prototype of Algorithm 2, with the two information gain measures, described in Section 4.3, as an invariant synthesis tool for Boogie [BCD+05] programs and compare it to other invariant generation algorithms. We conducted all of the following experiments on a Core i5 CPU with 6 GB of RAM running Windows 7 with a 10 minute timeout limit.

Learner: We implemented the learning algorithm on top of the freely available version of the C5.0 algorithm (Release 2.10) [Qui93]. Since we rely on learning without classification errors, we disable all of C5.0's special features, such as pruning, boosting, etc.

Furthermore, though our learning algorithm can work with any class of predicates (including non-linear predicates), we parameterize it with the class of octagonal predicates (of the form $\pm x \pm y \leq c$). More precisely, we add all numerical attributes of the form $\pm x \pm y$ for all combinations of variables x, y in the program (note that the learner learns the thresholds c as well as the Boolean combination of these predicates). This class of predicates is sufficient to express all invariants in our benchmark and, furthermore, all black-box learners that we compare with are also instantiated with this class

of predicates.

Teacher: We implemented a teacher in Boogie [BCD+05], which generates verification conditions for a given input program. The teacher prioritizes returning positive/negative counterexamples to implication counterexamples. Since loop invariants usually do not involve large constants, the teacher biases the learner towards trees with smaller thresholds by producing counterexamples that have small values. When searching for counterexamples, we iteratively bound the absolute values of the variables to 2, 5, 10, and ∞ till we find a counterexample.

Experimental Results: We evaluate the two configurations of the decision-tree based learner discussed in Section 4.3 in the context of invariant synthesis and compare them to various other invariant synthesis algorithms. The experimental results are tabulated in Table 4.1. We first tabulate times of CPAchecker [BK11], which is a white-box state-of-the-art verifier; we use the configuration that corresponds to the predicate abstraction and the interpolation [McM06a] based refinement. Note that CPAchecker does not restrict itself to finding Boolean combinations of octagonal constraints, while all the black-box learners do. Next we tabulate results for the black-box learners. The first is a randomized search based invariant synthesis tool [SA14]; this uses its own in-built teacher. The second is the constraint-solver based learner from Chapter 3 (called ICE-CS); this uses a teacher that does not bound counterexamples (as its in-built search bounds the constants in predicates iteratively anyway). The last two columns depict the new decision tree learners, from Section 4.3, which includes the learner that computes the information gain in a new way which accounts for implications (called ICE-DT-entropy) and the learner that penalizes splits that cut implications (called ICE-DT-penalty).

We report results for programs[2] taken from the literature [GLMN14, GR09] and from the SV-COMP benchmark suite. These programs have up to 50 lines of C code and often need complex invariants for their static verification. Some programs in Table 4.1 are the natural unbounded versions of programs that were artificially bounded in SV-COMP to simplify them.

Since randomized search based invariant synthesis is nondeterministic, we run it 10 times for every program and report the minimum and maximum

[2]Available at `http://web.engr.illinois.edu/~garg11/dtree/`

Table 4.1: Results comparing different invariant synthesis tools. χ_{TO} indicates that the tool times out (> 10 minutes) and χ indicates that the tool terminates without synthesizing an invariant; P, N, I are the number of positive, negative examples and implications in the final sample of the resp. learner; $\#R$ is the number of rounds, and T is the time in seconds.

Program	White-box	Black-box											
	CPAchecker	Randomized Search [SA14]			ICE-CS [GLMN14]			ICE-DT-entropy			ICE-DT-penalty		
	[BK11] (s)	Min.(s)	Max.(s)	Avg.(s) + TO	P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)
array	2.0	0.0	123.0	18.5 + 3/10 TO	4,7,11	14	0.5	15,16,67	95	6.3	4,6,16	25	1.4
array2	2.4	0.1	384.5	105.7 + 4/10 TO	4,7,5	7	0.3	2,1,2	5	0.2	2,1,2	5	0.2
afnp	χ_{TO}	0.1	0.7	0.3 + 0/10 TO	1,19,15	29	3.6	1,3,10	14	0.7	1,2,8	11	0.5
cegar1	1.9	0.0	0.1	0.1 + 0/10 TO	1,1,1	3	0.0	3,1,1	5	0.2	3,1,1	5	0.2
cegar2	2.2	1.2	305.6	82.1 + 3/10 TO	4,20,14	28	9.5	5,10,8	23	1.0	5,9,9	22	1.0
cggmp	2.0	-	-	- + **10/10 TO**	1,36,50	71	51.1	1,12,55	68	4.6	1,17,50	68	7.2
countud	χ				3,12,7	13	1.0	2,9,4	14	1.1	3,12,6	19	0.7
dec	15.4	0.0	0.0	0.0 + 0/10 TO	1,1,1	3	0.0	1,2,0	3	0.4	1,2,0	3	0.2
dtuc	χ_{TO}	4.9	190.4	62.8 + 2/10 TO	3,9,14	12	0.7	5,27,56	59	3.4	6,16,31	39	1.6
ex14	2.4	0.0	0.1	0.0 + 0/10 TO	2,5,1	7	0.0	1,1,0	2	0.1	1,1,0	2	0.1
ex14c	1.8	0.2	31.6	3.4 + 0/10 TO	2,2,1	4	0.0	1,1,0	2	0.0	1,1,0	2	0.1
ex23	5.4	0.1	127.5	21.8 + 1/10 TO	5,32,40	69	17.5	5,21,12	34	2.7	5,8,1	11	0.6
ex7	5.7	0.0	160.2	22.0 + 0/10 TO	1,2,1	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
fig1	1.9	0.5	51.2	11.1 + 4/10 TO	2,5,1	6	0.1	2,4,1	6	0.2	2,4,1	6	0.4
fig3	1.9	0.3	5.2	2.7 + 8/10 TO	2,4,2	6	0.1	4,5,0	6	0.3	4,5,0	6	0.2
fig9	1.9	0.0	0.1	0.0 + 0/10 TO	0,2,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
inc	15.4	0.0	0.0	0.0 + 0/10 TO	3,12,101	112	1.7	9,6,104	117	6.5	5,3,100	106	4.9
inc2	1.8	0.0	8.0	0.8 + 0/10 TO	3,4,3	8	0.1	3,3,1	7	0.3	4,3,3	10	0.4
matrixl1	3.3	-	-	- + **10/10 TO**	2,9,3	8	0.3	5,5,2	8	0.5	5,5,2	8	0.5
matrixl1c	3.0	-	-	- + **10/10 TO**	4,12,4	8	0.9	4,9,2	8	0.5	5,11,2	9	0.7
matrixl2	3.4	0.7	0.7	0.7 + 9/10 TO	8,19,13	27	22.9	9,10,7	24	1.3	9,10,5	22	1.2
matrixl2c	3.1	308.0	308.0	308.0 + 9/10 TO		χ		18,52,38	107	9.8	17,36,16	66	3.5
nc11	2.1	0.0	0.1	0.1 + 0/10 TO	5,15,7	18	0.7	3,6,5	13	0.6	2,4,4	9	0.4
nc11c	2.1	0.1	46.1	6.3 + 2/10 TO	4,6,3	10	0.4	3,3,3	8	0.4	3,3,3	8	0.3
sum1	1.9	270.2	270.2	270.2 + 9/10 TO	2,15,10	17	2.3	4,17,3	21	3.3	3,13,3	17	1.1
sum3	2.0	0.0	0.1	0.1 + 0/10 TO	1,3,1	4	0.1	1,4,1	6	0.3	1,4,1	6	0.3
sum4	2.2	4.7	26.8	11.4 + 0/10 TO	1,23,31	44	3.5	1,9,44	54	3.1	1,7,38	46	2.5
sum4c	2.0	3.1	420.2	171.2 + 6/10 TO	6,29,21	34	11.6	5,14,5	20	1.1	5,13,3	17	0.9
tacas	1.8	0.0	0.1	0.0 + 0/10 TO	7,8,5	14	1.7	10,7,4	19	0.9	10,7,4	20	0.9

Continued on next page

Continued from previous page

Program	White-box	Black-box											
	CPAchecker	Randomized Search [SA14]			ICE-CS [GLMN14]			ICE-DT-entropy			ICE-DT-penalty		
	[BK11] (s)	Min.(s)	Max.(s)	Avg.(s) + TO	P,N,I	#R	T(s)	P,N,I	#R	T(s)	P,N,I	#R	T(s)
trex1	1.9	0.0	90.6	9.1 + 0/10 TO	2,3,0	3	0.0	2,3,0	5	0.2	2,3,0	5	0.2
trex3	χ	-	-	- + **10/10 TO**	6,19,6	19	2.7	7,12,4	20	1.2	4,8,3	12	0.7
vsend	1.8	0.0	0.1	0.0 + 0/10 TO	1,1,0	2	0.0	1,1,0	2	0.1	1,1,0	2	0.1
w1	1.8	0.0	0.2	0.1 + 0/10 TO	1,3,3	5	0.0	2,1,1	4	0.4	2,2,1	5	0.2
w2	1.9	0.1	223.9	27.5 + 1/10 TO	2,4,1	4	0.0	2,2,1	5	0.3	1,2,1	4	0.2
formula22	2.0	1.7	347.9	172.8 + 6/10 TO	1,18,11	22	1.8	1,13,34	48	2.6	1,8,32	41	1.9
formula25	2.3	9.1	163.5	56.6 + 2/10 TO	1,46,30	49	14.0	1,71,6	78	3.7	1,62,4	67	3.0
formula27	2.2	-	-	- + **10/10 TO**		χ		1,142,13	156	9.5	1,104,15	120	5.8
multiply	χ	-	-	- + **10/10 TO**		χ		2,44,20	66	82.5	6,40,23	68	44.5
sqrt	χ	-	-	- + **10/10 TO**	3,26,26	32	9.2	3,32,14	47	4.3	5,43,20	64	3.2
add	χ	-	-	- + **10/10 TO**	1,11,0	12	0.1	1,11,1	13	0.9	1,11,1	13	0.5
square	χ	-	-	- + **10/10 TO**		χ		1,8,2	11	0.7	1,8,2	11	0.5
loops	$\chi_{\mathbf{TO}}$	96.1	284.1	159.2 + 4/10 TO	4,3,10	7	0.2	1,6,11	16	1.1	2,5,10	14	0.7
dillig01	1.9	4.8	56.4	16.7 + 0/10 TO	5,15,10	17	0.7	2,6,1	8	0.5	2,6,1	8	0.4
dillig03	$\chi_{\mathbf{TO}}$	0.4	6.3	4.0 + 4/10 TO	2,12,9	15	1.0	1,3,2	6	0.6	1,4,2	7	0.5
dillig05	$\chi_{\mathbf{TO}}$	6.4	172.3	87.5 + 4/10 TO	3,21,25	29	4.9	2,49,5	55	6.4	3,40,4	45	2.3
dillig07	1.9	0.2	16.6	4.1 + 0/10 TO	2,6,8	13	0.3	2,5,4	10	0.8	3,6,6	13	0.5
dillig12	$\chi_{\mathbf{TO}}$	-	-	- + **10/10 TO**		χ		1,8,192	136	18.8	1,5,75	52	3.7
dillig15	1.9	-	-	- + **10/10 TO**	3,8,16	22	2.9	3,5,11	18	1.4	3,3,12	17	0.5
dillig17	$\chi_{\mathbf{TO}}$	-	-	- + **10/10 TO**	3,15,53	34	12.7	2,6,22	22	1.6	2,6,22	22	0.9
dillig19	2.3	62.7	455.7	269.0 + 0/10 TO	4,12,18	20	8.6	6,4,14	21	1.6	7,5,11	22	0.8
dillig24	1.9	-	-	- + **10/10 TO**	6,7,28	17	1.4	0,6,9	13	1.0	0,6,9	13	0.7
dillig25	2.0	-	-	- + **10/10 TO**		χ		1,11,119	66	5.7	1,11,190	85	6.0
dillig28	$\chi_{\mathbf{TO}}$	115.3	228.5	193.6 + 2/10 TO	1,5,14	11	0.2	1,4,17	16	1.3	1,4,20	17	0.9
arrayinv1	3.8	-	-	- + **10/10 TO**		χ		5,38,156	196	31.9	5,30,79	111	6.7
arrayinv2	4.5	0.1	56.4	16.7 + 0/10 TO	4,22,33	43	20.9	$\chi_{\mathbf{TO}}$			4,17,24	43	1.8
Aggregate	**41 / 55 programs Time = 127s**	**40 / 55 programs Avg. Time = 2116.2s**			**48 / 55 programs Time = 215s**			**54 / 55 programs Time = 229s**			**55 / 55 programs Time = 119s**		

time, the average of the times (when it doesn't timeout) and the number of runs where it times out ($\geq$ 10 min.). For the constraint solver based learner and our decision tree learners we provide details about the composition of the final sample, in terms of the number of positive, negative and implication counter-examples that were required to learn an adequate invariant and the number of rounds to converge.

Being white-box, CPAchecker is not precise for programs with arrays and pointers; also for several programs over numerical variables that have complex disjunctive assertions, it reports an error even when these programs are safe. These are all reported as $\times$ in Table 4.1.

Invariant synthesis using randomized search [SA14], in our experience, is very volatile, as shown by the large variation in run times (see column showing minimum and maximum times), performing fast as well as timing out on the same program. It times out for *four* programs on all runs[3]. The tool also searches for numerical invariants that have constants that are mined from the program code (unlike our learners). Of the four programs, invariant synthesis for *cggmp* failed because it required certain constants that were not present in the program text. Further, there are *ten* programs for which randomized search fails more than half of the time. However, note that there are random walks where it finds the invariant very fast.

Our decision tree based learners are approximately *3-5* times faster than the constraint solver based learner ICE-CS even when restricted to programs on which ICE-CS does not time out. This is impressive given that ICE-CS in itself is quite a fast learner and, as reported in Chapter 3, outperforms various tools including Invgen [GR09], Houdini [TLLR13] and the Impact algorithm [McM06a] from CPAchecker [BK11].

Note that since our teacher returns implications only if it cannot find positive/negative examples, all programs that report a non-zero number of implications in the final sample ($\sim$ 80% of programs) *required* implication counterexamples to make progress.

Qualitatively, the most important observation regarding our learner is that *the decision tree learners converge and successfully find an adequate invariant for all programs.* Machine-learning algorithms are based on several heuristics, and typically do not guarantee convergence (note that we do not have any

[3]the tool gave an error on *f27* whose results are discounted

theoretical convergence for our algorithms, either). We were hence surprised to see convergence on all examples. Our conjecture is that the outer ICE learning guidance from the teacher effectively offsets the mistakes the learner makes, guiding it eventually to a correct invariant.

Robustness to small changes: To see how the learners would perform if there were a few other variables which were not involved in the invariant, we generated variations for 18 programs so that they have (*three*) extra variables that are havoc-ed inside the loop. This increases the search-space of the invariant synthesis problem for all the black-box learners. ICE-CS times out on *two* of these additional programs (*sum1, matrix2c*); it finishes but takes more time for some programs. Randomized search fails (no successful run) on *eight* additional programs. However, our decision tree learners continued to perform equally well for programs with these extra variables. The learning algorithms underlying our approach are particularly good in weeding out irrelevant attributes, and we believe this to be the reason for their superior performance.

In practical examples such as GPUVerify [BCD$^+$12], an invariant typically involves only a small set of variables (two or three), but candidates range over a large set of variables (some times up to hundreds of them). We believe it is important for black-box invariant generation algorithms to handle such scenarios. We have also successfully applied our decision tree learner to learn invariants for selected large GPUVerify programs (for instance *prefixSum* and *binomialOption*), and though these are conjunctive invariants, our learning algorithm worked well. A more careful study and a full-blown invariant generation based on our techniques for race-checking GPU programs is an interesting future research direction.

CHAPTER 5

LEARNING UNIVERSALLY QUANTIFIED INVARIANTS OVER LINEAR DATA STRUCTURES

Linear data structures, such as arrays and linked lists, are important unbounded data structures used in computer science. Properties of such linear data structures require *quantification* due to the unbounded nature of these structures. For instance, expressing that the data stored in a linked list is sorted requires quantification. Reasoning with such data structures requires expressing such properties, especially in program verification where such properties can encode pre/post conditions or invariants that help prove a program correct. In this section, our main motivation stems from program verification, in particular the problem of synthesizing loop invariants for programs that express properties of linear data structures. In particular, as we have argued in this thesis, our main motivation is to synthesize these invariants using learning.

In many fields of logic, *automata theory* plays an important role as a normal form for logic. For instance, monadic second order logic on labeled words and trees, both finite and infinite, is captured using finite-state automata on these structures [Tho97, GTW02]. A connection to automata theory is often useful as the simple structure of automata in terms of graphs gives a better arena than logic to study algorithmic problems on the associated logic. Decision procedures such as satisfiability on logic (and even model-checking of systems) can be translated to appropriate emptiness-checking algorithms on graphs [VW86]. Another important algorithmic procedure that automata yield are *learning algorithms*— automata-based learning algorithms give a means to learn the corresponding logical formulae in various learning models [Ang87a, Gol78].

In this section, we build learning algorithms for *quantified logical formulas describing sets of linear data structures*. Our aim is to build algorithms that can learn formulas of the kind "$\forall y_1, \ldots, y_k\ \varphi$", where φ is quantifier-free, and which capture properties of arrays and lists (the variables range over

indices for arrays, and locations for lists, and the formula can refer to the data stored at these positions and compare them using arithmetic, etc.). Due to the formal connection between logics and automata theory and the advantages we mentioned above, we model linear data structures as *data words*, where each position is decorated with a letter from a finite alphabet modeling the program's pointer variables that point to that cell in the list or index variables that index into the cell of the array, and with data modeling the data value stored in the cell, e.g., integers. Further, we seek automata models for expressing quantified properties of such data words.

Our main contribution is a novel representation (normal form) for quantified properties of linear data structures, called *quantified data automata* (QDA). In Section 5.3, we explore various properties of QDAs and the languages accepted by them. The class of quantified properties that can be expressed using QDAs is very powerful, and does not admit decidable satisfiability problems, in general. The validity of the corresponding logical formulas in the theory of arrays and lists is also undecidable, in general. In the context of program verification, even if we use QDAs to learn invariants, we will be unable to *verify* automatically whether the learned properties are adequate invariants for the program at hand. Even though SMT solvers support heuristics to deal with quantified theories (like e-matching [dMB07]), in our experiments, the verification conditions derived from invariants expressed as QDAs could not be handled by such SMT solvers. Section 5.4, therefore, introduces the subclass of QDAs, called elastic QDAs, that admit decidable validity problems. In Section 5.5 we describe how quantified properties of linear data structures can be modeled using QDAs/EQDAs, and present a translation from EQDAs to decidable fragments of arrays and lists. Finally, to complete the picture, we develop different learning algorithms for learning QDAs/EQDAs. First, in Section 5.6, we present an active learning algorithm that extends Angluin-style learning of finite automata to learn QDAs. We deploy this algorithm in a passive setting to learn quantified invariants of linear data-structures from a sample set of positively and negatively labeled program configurations that do or do not manifest along dynamic test runs. Subsequently, in Section 5.7, we develop a robust ICE learning algorithm for learning QDAs and apply this algorithm to robustly learn quantified invariants of linear data structures such as arrays and lists.

5.1 Overview

List and Array Invariants:

Consider a typical invariant in a sorting program over lists where the loop invariant is expressed as:

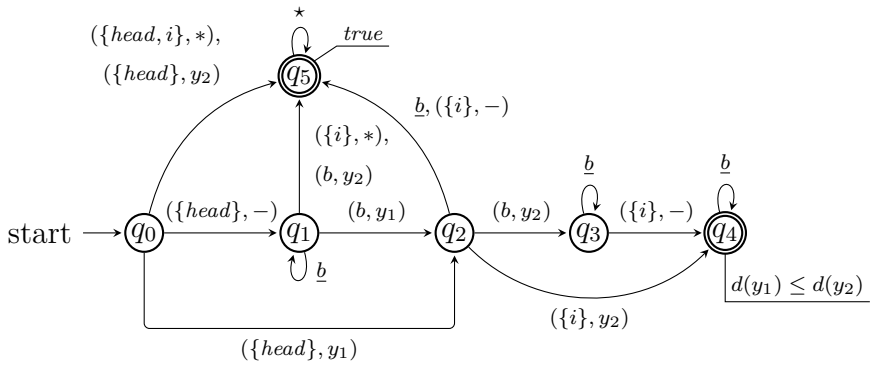
$$head \rightarrow^* i \ \ \wedge \ \ \forall y_1, y_2.((head \rightarrow^* y_1 \wedge succ(y_1, y_2) \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \tag{5.1}$$

This says that for all cells y_1 that occur somewhere in the list pointed to by *head* and where y_2 is the successor of y_1, and where y_1 and y_2 are before the cell pointed to by a scalar pointer variable i, the data value stored at y_1 is no larger than the data value stored at y_2. This formula is *not* in the decidable fragment of STRAND [MPQ11,MQ11] since the universally quantified variables are involved in a non-elastic relation *succ* (in the subformula $succ(y_1, y_2)$). Such an invariant for a program manipulating arrays can be expressed as:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_2 = y_1 + 1 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \tag{5.2}$$

Note that the above formula is not in the decidable array property fragment [BMS06].

Quantified Data Automata: The key idea we explore in our work is an automaton model for expressing such constraints called *quantified data automata* (QDA). The above two invariants are expressed by the following QDA:



The above automaton reads (deterministically) data words whose labels denote the positions pointed to by the scalar pointer variables *head* and i,

as well as valuations of the quantified variables y_1 and y_2. We use two *blank* symbols that indicate that no pointer variable ("b") or no variable from Y ("$-$") is read in the corresponding component; moreover, $\underline{b} = (b, -)$. Missing transitions go to a sink state labeled *false*. The above automaton accepts a data word w with a valuation v for the universally quantified variables y_1 and y_2 as follows: it stores the value of the data at y_1 and y_2 in two registers, and then checks whether the formula annotating the final state it reaches holds for these data values. The automaton accepts the data word w if for *all* possible valuations of y_1 and y_2, the automaton accepts the corresponding word with valuation. The above automaton hence accepts precisely those set of data words that satisfy the invariant formula.

Decidable Fragments and Elastic Quantified Data Automata: The emptiness problem for QDAs is undecidable; in other words, the logical formulas that QDAs express fall into undecidable theories of lists and arrays. A common restriction in the array property fragment as well as the syntactic decidable fragments of STRAND is that quantification is not permitted to be over elements that are only a *bounded* distance away. The restriction allows quantified variables to only be related through *elastic* relations (following the terminology in STRAND [MPQ11, MQ11]).

For instance, a formula equivalent to the formula in Eq. 5.1 but expressed in the decidable fragment of STRAND over lists is:

$$head \rightarrow^* i \quad \wedge \quad \forall y_1, y_2.((head \rightarrow^* y_1 \wedge y_1 \rightarrow^* y_2 \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \tag{5.3}$$

This formula compares data at y_1 and y_2 whenever y_2 occurs sometime after y_1, and this makes the formula fall in a decidable class. Similarly, a formula equivalent to the formula Eq. 5.2 in the decidable array property fragment is:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_1 \leq y_2 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \tag{5.4}$$

The above two formulas are captured by a QDA that is the same as shown before, except that the $\underline{b}$-transition from q_2 to q_5 is replaced by a $\underline{b}$-loop on q_2.

We identify a restricted form of quantified data automata, called *elastic quantified data automata* (EQDA) in Section 5.4, which structurally captures the constraint that quantified variables can be related only using elastic

relations (like $\rightarrow^*$ and $\leq$). Furthermore, we show in Section 5.5 that EQDAs can be converted to formulas in the decidable fragment of STRAND and the array property fragment, and hence expresses invariants that are amenable to decidable analysis across loop bodies.

It is important to note that QDAs are not necessarily a blown-up version of the formulas they correspond to. For a formula, the corresponding QDA can be exponential, but for a QDA the corresponding formula can be exponential as well (QDAs are like BDDs, where there is sharing of common suffixes of constraints, which is absent in a formula).

5.2 Quantified Data Automata Model to Express Invariants over Linear Data-Structures

We model lists (and finite sets of lists) and arrays that contain data over some data domain D as finite words, called *data words*, encoding the pointer variables and the data values.

Definition 5.2.1 (Data words)**.** *Let $PV = \{p_1, \ldots, p_r\}$ be a finite set of pointer variables, $\Sigma = 2^{PV}$, and D a data domain. A* data word *over PV and D is a word $u \in (\Sigma \times D)^*$ where every $p \in PV$ occurs exactly once in u (i.e., for each $u = a_1 \ldots a_n$ and $p \in PV$, there exists precisely one $j \in \{1, \ldots, n\}$ such that $a_j = (X, d)$ and $p \in X$).*

The empty set in the first component of a data word corresponds to a blank symbol indicating that no pointer variable occurs at this position. We also denote this blank symbol by the letter b.

Let $Y = \{y_1, \ldots, y_k\}$ be a nonempty, finite set of *universally quantified variables*. The automata we build accepts a data word if for all possible valuations of Y over the positions of the data word, the data stored at these positions satisfy certain properties. For this purpose, the automaton reads data words extended by valuations of the variables in Y, called *valuation words*. The variables are then quantified universally in the semantics of the automaton model (as explained later in this section).

Definition 5.2.2 (Valuation word)**.** *A* valuation word *is a word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ where v projected to its first and third component forms a data word and where each $y \in Y$ occurs exactly once in v.*

We use the symbol "$-$" to denote positions in valuation words where no universally quantified variable occurs. Note that the choice of the alphabet ensures that all universally quantified variables have to occur at different positions and is technically convenient.

A valuation word corresponds to a data word with a valuation of Y. This is formalized by the following definition.

Definition 5.2.3. *Given a valuation word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, the corresponding data word is the word* $\mathrm{dw}(v) \in (\Sigma \times D)^*$ *resulting from projecting v to its first and third components.*

Later, we will also consider a third type of words, called *symbolic words*. In contrast to data and valuation words, symbolic words only capture the structure of a list or array but do not contain data.

Definition 5.2.4 (Symbolic word)**.** *Let $\Sigma = 2^{PV}$ and $\Pi = \Sigma \times (Y \cup \{-\})$. A* symbolic word *is a word $w \in \Pi^*$ where each $p \in PV$ occurs exactly once in w and each $y \in Y$ occurs exactly once in w.*

We denote the symbol in Π representing neither a pointer nor a universally quantified variable by $\underline{b} = (b, -)$. The next definition establishes a connection between symbolic and valuation words.

Definition 5.2.5. *Given a valuation word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, the corresponding symbolic word is the word* $\mathrm{sw}(v) \in \Pi^*$ *resulting from projecting v to its first two components.*

To express the properties on the data, let us fix a set of constants, functions and relations over D. We assume that the quantifier-free first-order theory over this domain is decidable; we encourage the reader to keep in mind the theory of integers with constants (0, 1, etc.), addition, and the usual relations ($\leq$, $<$, etc.) as a standard example of such a domain.

Quantified data automata use a *finite* set F of formulas over the atoms $d(y_i)$, $i \in \{1, \ldots, n\}$, which we interpret as the data values of the cells pointed to by the variables $y_1, \ldots, y_n$. We assume that this set is organized in a (bounded semi-)lattice, which leads to the following definition.

Definition 5.2.6 (Formula lattice)**.** *A* formula lattice *$\mathcal{F} = (F, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ is a tuple consisting of a finite set F of formulas over the atoms $d(y_1), \ldots, d(y_n)$,*

a partial-order relation $\sqsubseteq$ over F, a least-upper bound operator $\sqcup$, and the formulas false *and* true*, which are required to be in F and correspond to the bottom and top elements of the lattice. Furthermore, we require that whenever $\alpha \sqsubseteq \beta$, then $\alpha \Rightarrow \beta$. Also, we require that the formulas in the lattice are pairwise inequivalent.*

One example of such a formula lattice over the data domain of integers can be obtained by taking a set of representatives of all possible inequivalent Boolean formulas over the atomic formulas involving no constants, defining $\alpha \sqsubseteq \beta$ if and only if $\alpha \Rightarrow \beta$, and taking the least-upper bound of two formulas as the disjunction of them. Such a lattice would be of size doubly exponential in the number of variables n, and consequently, in practice, we may want to use a different coarser lattice, such as the Cartesian formula lattice. The Cartesian formula lattice is formed over a set of atomic formulas and consists of conjunctions of literals (atoms or negations of atoms). The least-upper bound of two formulas is taken as the conjunction of those literals that occur in both formulas. For the ordering, we define $\alpha \sqsubseteq \beta$ if all literals appearing in β also appear in α. The size of a Cartesian lattice is exponential in the number of literals.

We are now ready to introduce the automaton model.

Definition 5.2.7 (Quantified data automata)**.** *Let PV be a finite set of program variables, Y a finite, nonempty set of universally quantified variables, D a data domain, and $\mathcal{F}$ a formula lattice over a finite set F of formulas. A* quantified data automaton (QDA) *is a tuple $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ where Q is a finite, nonempty set of states, $\Pi = \Sigma \times (Y \cup \{-\})$ is the input alphabet, $\delta \colon Q \times \Pi \dashrightarrow Q$ is the (partial) transition function, and $f \colon Q \to F$ is the* final-evaluation function*, which maps each state to a data formula.*

Intuitively, a QDA is a register automaton that reads the data word extended by a valuation that has a register for each $y \in Y$, which stores the data stored at the positions evaluated for Y, and checks whether the formula decorating the final state reached holds for these registers. It accepts a data word $u \in (\Sigma \times D)^*$ if it accepts *all possible* valuation words v extending u with a valuation over Y. We formalize this below.

A configuration of a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ is a pair (q, r) where $q \in Q$ and $r : Y \dashrightarrow D$ is a partial variable assignment. The initial configuration is (q_0, r_0) where the domain of r_0 is empty.

The run of $\mathcal{A}$ on a valuation word $v = (a_1, y_1, d_1) \dots (a_n, y_n, d_n) \in (\Sigma \times (Y \cup \{-\}) \times D)^*$ is a sequence $(q_0, r_0), \dots, (q_n, r_n)$ of configurations that satisfies $\delta(q_i, (a_i, y_i)) = q_{i+1}$ and

$$r_{i+1} = \begin{cases} r_i\{y_i \leftarrow d_i\} & \text{if } y_i \in Y; \\ r_i & \text{if } y_i = -; \end{cases}$$

where $i \in [0, n)$, the configuration (q_0, r_0) is the initial configuration, and $r_i\{y_i \leftarrow d_i\}$ corresponds to the mapping r_i in which the argument y_i is mapped to the value d_i. We use $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q_n, r_n)$ as a shorthand-notation.

The QDA $\mathcal{A}$ accepts a valuation word v if $\mathcal{A}\colon (q_0, r_0) \xrightarrow{v} (q, r)$ with $r \models f(q)$; that is, after reading the valuation word, the data stored in the registers satisfies the formula annotating the state finally reached. The language $L_{val}(\mathcal{A})$ is the set of valuation words accepted by $\mathcal{A}$.

The QDA $\mathcal{A}$ accepts a data word $u \in (\Sigma \times D)^*$ if $\mathcal{A}$ accepts all valuation words v with $\mathrm{dw}(v) = u$. The language $L_{dat}(\mathcal{A})$ is the set of data words accepted by $\mathcal{A}$.

To ease working with QDAs and to obtain the intended semantics, we assume throughout this chapter that each QDA satisfies two further constraints:

- Each QDA verifies that its input satisfies the constraints on the number of occurrences of variables from PV and Y. All inputs violating these constraints (i.e., all inputs that are not valuation words) either do not admit a run due to missing transitions or lead to a dedicated state labeled with the data formula *false*. This property implies that the states of an QDA are "typed" with the set of variables that have been read so far. As a consequence, cycles in the transition structure of an QDA can only be labeled with $\underline{b}$-symbols. Note that this assumption is no restriction because both the language of valuation words and the language of data words are defined in terms of words that satisfy the correct occurrence of variables from PV and Y.

- Each QDA verifies that the universally quantified variables occur in its input in the same fixed order, say $y_1 \prec \dots \prec y_k$. All valuation words violating this order lead to a dedicated state labeled with the data formula *true* (i.e., all such valuation words are accepted). The rationale behind this assumption is the following: since the variables $y \in Y$ are

universally quantified, it is sufficient to check a property with respect to a fixed order and a different order should not change the accepted language of data words.

Although this assumption is a restriction in general, each QDA can be transformed into one that accepts the same data language and respects the predetermined variable ordering if the formula lattice is closed under conjunction. The idea for such a construction is to use a subset construction that follows all paths that only differ in the order of Y. For each state in a set of states reached like that, one remembers in which order the variables in Y have occurred. At the final states, one uses the conjunction of all formulas in the set with the appropriate renaming of the variables in Y. Due to the universal semantics of QDAs, this captures a QDA that accepts the same data language as original automaton. Since most natural formula lattices, such as the full lattice and the Cartesian lattice (which we use in this chapter), are closed under conjunction, we can without loss of generality assume that each QDA respects a fixed ordering of the universally quantified variables.

5.3 Properties of Quantified Data Automata

In this section, we study properties of QDAs, such as viewing QDAs as Moore machines, whether QDAs allow for canonical representations, their closure under Boolean operations, and decidability results for them.

5.3.1 Viewing QDAs as Moore Machines

Moore machines are extensions of deterministic finite automata that are equipped with output at their states and define a mapping rather than accept a language. On a syntactical level, QDAs can be viewed as such machines, where the output corresponds to the formulas at the final states of the QDA. Taking this view of QDAs allows us to derive some results by using the theory of Moore machines. Formally, Moore machines are defined as follows.

Definition 5.3.1 (Moore machine)**.** *A* Moore machine *is a tuple* $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta, \lambda)$ *where* Q *is a nonempty, finite set of states,* Σ *is the input*

alphabet, Γ *is the output alphabet,* $q_0 \in Q$ *is the initial state,* $\delta\colon Q \times \Sigma \to Q$ *is the transition function, and* $\lambda\colon Q \to \Gamma$ *is the output function that assigns an output-symbol to each state.*

The run of a Moore machine $\mathcal{M}$ on a word $u = a_1 \dots a_n$ is a sequence $q_0, \dots, q_n$ of states that satisfies $\delta(q_i, a_{i+1}) = q_{i+1}$ for all $i \in [0, n)$; as in the case of QDAs, we use the shorthand-notation $\mathcal{M}\colon q_0 \xrightarrow{u} q_n$ to denote the run of $\mathcal{M}$ on u. Each Moore machine defines a total function $f_{\mathcal{M}}$ that maps an input-word $u \in \Sigma^*$ to the output of the state that $\mathcal{M}$ reaches after reading u; more precisely, we define $f_{\mathcal{M}}(u) = \lambda(q)$ where $\mathcal{M}\colon q_0 \xrightarrow{u} q$. Finally, we call a function $f\colon \Sigma^* \to \Gamma$ *Moore machine computable* if there exists a Moore machine $\mathcal{M}$ such that $f = f_{\mathcal{M}}$.

Let us now describe how one can view QDAs as Moore machines. Recall that QDAs define two kind of languages, a language of data words and a language of valuation words. On the level of valuation words, we can understand a QDA as an automaton that reads the structural part of a valuation word (i.e., a symbolic word) and outputs a data formula capturing the data. To make this intuition more precise, let us introduce another type of words, which we call *formula words.*

Definition 5.3.2 (Formula words)**.** *Let* PV *be a finite set of pointer variables,* Y *a finite set of universally quantified variables, and* $\mathcal{F}$ *a lattice over a set* F *of formulas. A* formula word *is a finite word* $(w, \varphi) \in (\Pi^* \times F)$ *where, as before,* $\Pi = \Sigma \times (Y \cup \{-\})$*, and each* $p \in PV$ *and each* $y \in Y$ *occurs exactly once in* w*.*

Note that a formula word does not contain elements of the data domain—it simply consists of the symbolic word that depicts the pointers into the list (modeled using Σ), a valuation for the quantified variables (modeled using $Y \cup \{-\}$), as well as a formula over lattice $\mathcal{F}$ over the data domain. For example, $((\{h\}, y_1)(b, -)(b, y_2)(\{t\}, -), d(y_1) \leq d(y_2))$ is a formula word, where h points to the first element, t to the last element, y_1 points to the first element, and y_2 to the third element; and the data formula is $d(y_1) \leq d(y_2)$.

We can now view a QDA as an acceptor of formula words.

Definition 5.3.3. *A QDA* $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ *over the set* F *of data formulas accepts a formula word* $(w, \varphi) \in \Pi^* \times F$ *if* $\mathcal{A}$ *reaches a state* $q \in Q$ *on reading the symbolic word* w *and* $f(q) = \varphi$*. Given a QDA* $\mathcal{A}$*, we define the language*

$L_f(\mathcal{A}) \subseteq \Pi^* \times F$ *of formula words accepted by* $\mathcal{A}$ *in the usual way. Moreover, we call a language* $L_{for} \subseteq \Pi^* \times F$ *of formula words* QDA-acceptable *if there exists a QDA* $\mathcal{A}$ *with* $L_f(\mathcal{A}) = L_{for}$.

Note that not every language of formula words is QDA-acceptable; for instance, consider the language

$$L^\star_{for} = \{(\underline{b}^i(h, y)\underline{b}^i,\ true) \mid i \geq 1\}.$$

A standard pumping argument shows that $L^\star_{for}$ cannot be accepted by a QDA since the number of blanks at the beginning and at the end of a word have to match. Furthermore, words whose symbolic component is not of the form $\underline{b}^i(h, y)\underline{b}^i$ are not present in $L^\star_{for}$ but a QDA necessarily assigns a unique formula to every symbolic word. In fact, every QDA-acceptable language L_{for} of formula words has to fulfill the following constraints:

- For every symbolic word $w \in \Pi^*$, there exists a formula φ such that $(w, \varphi) \in L_{for}$.
- If $(w, \varphi) \in L_{for}$ and $(w, \varphi') \in L_{for}$, then $\varphi = \varphi'$.
- There are only finitely many different formulas occurring in formula words in L_{for}.

These constraints allow us to treat QDAs as Moore machines that read symbolic words and output data formulas. In fact, we make the following observation.

Observation 5.3.1. *A QDA-acceptable language* $L_{for} \subseteq \Pi^* \times F$ *is an alternative representation of a Moore machine-computable mapping* $f\colon \Pi^* \to F$ *(in the sense that* $(w, \varphi) \in L_{for}$ *if and only if* $f(w) = \varphi$*).*

One easily deduces that two QDAs $\mathcal{A}$ and $\mathcal{A}'$ (over the same lattice of formulas) that accept the same set of valuation words also define the same set of formula words (assuming that all the formulas in the lattice are pairwise non-equivalent). Thus, we can easily reduce the problem of actively learning QDAs to the problem of actively learning Moore machines, as we show in Section 5.6.

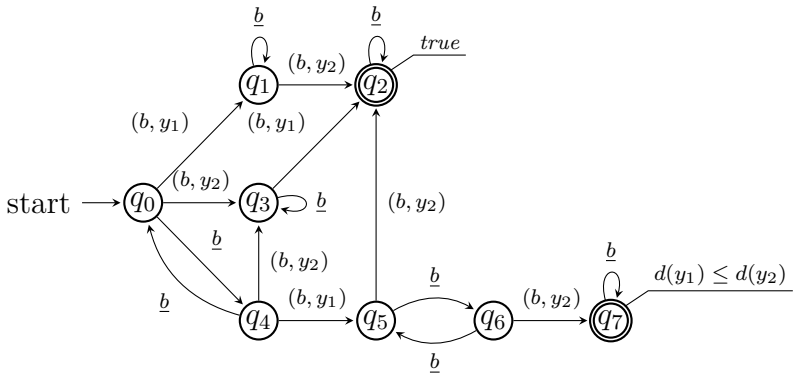


Figure 5.1: A QDA expressing the property over lists that the data on even positions is sorted. Missing transitions lead to a sink-state labeled with *false*, which is not shown for the sake of readability. All states depicted as a single circle are implicitly labeled with the formula *false*.

5.3.2 Canonical QDAs

Recall that QDAs define two kinds of languages, namely a language of data words and a language of valuation words. We begin by observing that we cannot hope for unique minimal QDA on the level of data words.

To see why, consider the QDA $\mathcal{A}$ in Figure 5.1 over $PV = \emptyset$ and $Y = \{y_1, y_2\}$. It accepts all valuation words in which

- $d(y_1) \leq d(y_2)$ if y_1 occurs before y_2 and y_1, y_2 are both on even positions; or
- $y_2 < y_1$; or
- at least one of y_1 and y_2 does not occur at an even position.

Hence, $\mathcal{A}$ accepts the language of data words that consist of all data words such that the data on even positions is sorted. Since each QDA has to ensure that each variable occurs exactly once, the number of states of $\mathcal{A}$ is minimal for defining this language of data words.

However, a QDA in which we replace the transition $\delta(q_6, \underline{b}) = q_5$ by the transition $\delta(q_6, \underline{b}) = q_1$ accepts the same language of data words. This new QDA checks the sortedness only for all y_1, y_2 with $y_2 = y_1 + 2$, which is sufficient. This shows that the transition structure of a state-minimal QDA for a given language of data words is not unique.

On the level of valuation words, on the other hand, there exists a minimal canonical QDA, which is formalized next. This is because the automaton

model is deterministic and, since all universally quantified variables are in different positions, the automaton cannot derive any relation on the data values during its run. Formally, we can state the following theorem.

Theorem 5.3.4. *For each QDA $\mathcal{A}$ there is a unique minimal QDA $\mathcal{A}_{min}$ that accepts the same set of valuation words.*

Proof. Consider a language L_{val} of valuation words that can be accepted by a QDA, and let $w \in \Pi^*$ be a symbolic word. Then there must be a formula ψ_w in the lattice that characterizes precisely the valuation words $v \in L_{val}$ that extend w with data (i.e., that satisfy $\mathrm{sw}(v) = w$). Since we assume that all the formulas in the lattice are pairwise non-equivalent, this formula is uniquely determined. This formula ψ_w is obtained by considering for each valuation word v with $\mathrm{sw}(v) = w$ the greatest-lower bound φ_v of all formulas in the lattice that are satisfied in v, and then taking the least-upper bound of all these φ_v.

In fact, the formula ψ_w is independent of the actual QDA. To prove this, take any QDA $\mathcal{A}$ that accepts L_{val}. Then w leads to some state q in $\mathcal{A}$ that outputs the formula $f(q)$, where f is the final-evaluation function in $\mathcal{A}$. If w leads to any other formula in another QDA $\mathcal{A}'$, then $\mathcal{A}'$ accepts a different language of valuation words.

Thus, a language of valuation words can be seen as a function that assigns to each symbolic word a uniquely determined formula, and a QDA can be viewed as a Moore machine that computes this function. For each such Moore machine, there exists a unique minimal one that computes the same function (see [Koh70]), hence the theorem. □ □

5.3.3 Boolean Operations on QDAs

Because of the universal semantics of QDAs, it is easy to see that the class of QDA-definable data languages is not closed under complement. Since the universal quantifier does not distribute over disjunctions, the class is also not closed under union.

Proposition 5.3.5. *There is a lattice $\mathcal{F}$ that is closed under all Boolean operations, such that the class of QDA-definable languages of data words over this lattice is not closed under complement and union.*

Proof. Take the data domain of the integers, and all Boolean formulas using the binary predicate $\leq$. The set of pointer variables is empty. We have already seen that the set L of data words in which the data is sorted in ascending order is QDA definable. The complement of this language is the set of data words in which there are two positions y_1 and y_2 such that $y_1 < y_2$ and $d(y_1) > d(y_2)$. Assume that there is a QDA $\mathcal{A}$ accepting this language. We assume here that the QDA uses only two variables y_1, y_2 but the argument can easily be extended to any number of variables. Consider the two data words $w_1 = (b,2)(b,1)(b,3)(b,4)$ and $w_2 = (b,1)(b,2)(b,4)(b,3)$. Both have to be accepted by $\mathcal{A}$. However, $\mathcal{A}$ then also accepts the data word $w = (b,1)(b,2)(b,3)(b,4)$ because for each valuation y_1, y_2 in w there is a valuation in w_1 or w_2 that cannot be distinguished from the valuation of w by $\mathcal{A}$ (i.e., the valuation word leads to the same state and satisfies the same data formulas); for instance, the valuation $(b, y_1, 1)(b, -, 2)(b, y_2, 3)(b, -, 4)$ in w cannot be distinguished from $(b, y_1, 2)(b, -, 1)(b, y_2, 3)(b, -, 4)$ in w_1. Thus, all valuations of w are accepted but w is in L and not in its complement.

For the non-closure under union consider the set L from above, and the set L' of data words in which the data is sorted in descending order. An argument similar to the one from above shows that the union of these two languages is not QDA definable. □ □

Since universal quantification distributes over conjunction, we obtain a positive result for intersection of data languages.

Proposition 5.3.6. *Let $\mathcal{F}$ be a formula lattice. If $\mathcal{F}$ is closed under conjunction, then the class of QDA-definable languages of data words is closed under intersection.*

Proof. A standard product construction for $\mathcal{A}_1$ and $\mathcal{A}_2$ with $f(q_1, q_2) = f(q_1) \wedge f(q_2)$ results in a QDA for the desired language. □ □

As for the case of canonical QDAs, we now consider closure properties on the level of valuation words.

Proposition 5.3.7. *Let $\mathcal{F}$ be a formula lattice. The class of QDA-definable languages of valuation words is closed under*

1. *Complement if $\mathcal{F}$ is closed under negation;*

2. *Union if $\mathcal{F}$ is closed under disjunction; and*

3. *Intersection if $\mathcal{F}$ is closed under conjunction.*

Proof. For the complement, just take the negation of the final formulas. For union and intersection use a product construction and combine the formulas by disjunction for union, and conjunction for intersection. □ □

This shows that, on the level of valuation words, QDAs behave much more like standard automata, given that the lattice has the corresponding properties. For the case of union and intersection, we additionally obtain the following weaker version of the results if we do not assume the corresponding closure properties of the lattice.

Proposition 5.3.8. *Let $\mathcal{F}$ be a formula lattice (with least upper bound and greatest lower bound operators), and let $\mathcal{A}_1, \mathcal{A}_2$ be two QDAs. There exists a unique minimal QDA-definable language of valuation words containing $L_{val}(\mathcal{A}_1) \cup L_{val}(\mathcal{A}_2)$, and there is a unique maximal QDA-definable language of valuation words contained in $L_{val}(\mathcal{A}_1) \cap L_{val}(\mathcal{A}_2)$.*

Proof. As in Proposition 5.3.7, we use product constructions, now combining the final formulas using the least upper bound and greatest lower bound instead of disjunction and conjunction. □ □

5.3.4 Decidability Results

The expressive power of QDAs depends on the data domain and the formula lattice for testing properties of the data. The formula lattices used for expressing nontrivial properties of data words usually lead to the undecidability of the emptiness problem for QDAs. For instance, using the integers as data domain, and an appropriate signature, it is easy to reduce the halting problem for two-counter machines to the emptiness problem of QDAs. Using blocks of three successive positions, one encodes the line number, and the two counter values in the data. The formulas at the final states are used to check that the data encoding the configurations faithfully simulates the computation of the given two-counter machine (a data domain with linear arithmetic would suffice). With a bit more effort, this result can even be extended to formulas that only use Boolean combinations of equality tests.

In contrast, the universality problem, that is, whether a given QDA accepts all data words (with the appropriate restrictions on the labeling by pointer variables), is decidable, provided the quantifier-free fragment used to express the data formulas is decidable. This amounts to a simple check whether there is a symbolic word that does not admit a run, or leads to a final state with a formula which is not *true* (i.e., not a tautology). In this case, one can construct a valuation word that is not accepted by the QDA, and thus the corresponding data word is also rejected.

5.4 Elastic Quantified Data Automata

Our aim is to translate the QDAs that are synthesized into decidable logics such as the decidable fragment of STRAND or the array property fragment. A property shared by both logics is that they cannot test whether two universally quantified variables are bounded distance away. We capture this type of constraint by the subclass of elastic QDAs (EQDAs) that have been already informally described in Section 5.1.

Definition 5.4.1 (Elastic quantified data automata)**.** *A QDA* $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ *is called elastic if each transition on* $\underline{b}$ *is a self-loop (i.e., whenever* $\delta(q, \underline{b}) = q'$ *is defined, then* $q = q'$*).*

If a state in an EQDA does not have any outgoing $\underline{b}$-transition, it might seem that the EQDA could still test whether two universally quantified variables, say y_1 and y_2, are bounded distance away (which is the reason for the undecidability of the emptiness problem for QDAs). However, because of the universal semantics of the automaton model, such a test is not possible. This is discussed in more detail in the translation from EQDAs to logic formulas in Section 5.5, where we introduce the notion of irrelevant self-loop.

The learning algorithm that we use to synthesize QDAs does not construct EQDAs in general. However, we can show that every QDA *uniquely over-approximated* by a language of valuation words that can be accepted by an EQDA, as stated in the following theorem. This result crucially relies on the particular structure that elastic automata have, that forces a unique set of words to be added to the language in order to make it elastic. We will refer to the construction in Definition 5.4.2 as *elastification*.

To ease the following definition, we introduce a few auxiliary notations: Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, let $R_{\underline{b}}(q)$ be the set of state reachable from q via a (possibly empty) sequence of $\underline{b}$-transitions and $R_{\underline{b}}(S) = \bigcup_{q \in S} R_{\underline{b}}(q)$ for a set $S \subseteq Q$. Moreover, we lift the transition function of $\mathcal{A}$ to sets of states: for $S \subseteq Q$ and $a \in \Pi$, let $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$.

Definition 5.4.2 (Elastification)**.** *Given a QDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$, we define the EQDA $\mathcal{A}_{el} = (Q_{el}, \Pi, S_0, \delta_{el}, f_{el})$ by*

- $Q_{el} = \{S \mid S \subseteq Q\}$;
- $S_0 = R_{\underline{b}}(q_0)$;
- $f_{el}(S) = \bigsqcup_{q \in S} f(q)$; *and*
- $\delta_{el}(S, a) = \begin{cases} R_{\underline{b}}(\delta(S, a)) & \text{if } a \neq \underline{b}; \\ S & \text{if } a = \underline{b} \text{ and } \delta(q, \underline{b}) \text{ is defined for some } q \in S; \\ \text{undefined} & \text{otherwise.} \end{cases}$

Note that this construction is similar to the usual powerset construction except that we take the "$\underline{b}$-closure" after applying the transition function of $\mathcal{A}$. Moreover, $\mathcal{A}_{\text{el}}$ loops in a state S as soon as a $\underline{b}$-transition is defined for a state $q \in S$.

Theorem 5.4.3. *For every QDA $\mathcal{A}$ one can construct an EQDA $\mathcal{A}_{el}$ such that*

- $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{el})$; *and*
- *for every EQDA $\mathcal{B}$ such that $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, the inclusion $L_{val}(\mathcal{A}_{el}) \subseteq L_{val}(\mathcal{B})$ holds.*

Proof. We begin by observing that $\mathcal{A}_{\text{el}}$ is elastic by definition of δ_{el}. Moreover, a standard induction over the length of valuation words $v = a_1 \dots a_n \in (\Pi \times D)^*$ shows the following: if the run of $\mathcal{A}$ on v is

$$\mathcal{A}\colon q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n,$$

then the run of $\mathcal{A}_{\text{el}}$ on v is

$$\mathcal{A}_{\text{el}}\colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} S_n$$

such that $q_i \in S_i$ for all $i \in \{1, \ldots, n\}$. This implies $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{A}_{\text{el}})$ because the implication $f(q_n) \to f_{\text{el}}(S_n)$ holds by definition of f_{el}.

Let us now show that the language $L_{val}(\mathcal{A}_{\text{el}})$ is indeed the most precise elastic over-approximation of $L_{val}(\mathcal{A})$. To this end, let $\mathcal{B} = (Q_{\mathcal{B}}, \Pi, q_0^{\mathcal{B}}, \delta_{\mathcal{B}}, f_{\mathcal{B}})$ be an EQDA with $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. Additionally, let $v \in L_{val}(\mathcal{A}_{\text{el}})$. Thus, the task is to prove that $v \in L_{val}(\mathcal{B})$ holds, too.

Let S be the state reached by $\mathcal{A}_{\text{el}}$ on reading v and p be the state reached by $\mathcal{B}$ on reading v. We now show that $f(q)$ implies $f_{\mathcal{B}}(p)$ for every $q \in S$. Once we have established this, we obtain that $f_{\text{el}}(S)$ implies $f_{\mathcal{B}}(p)$ because $f_{\text{el}}(S)$ is the least formula in the formula lattice that is implied by all formulas $f(q)$ for $q \in S$. Since $v \in L_{val}(\mathcal{A}_{\text{el}})$, the valuation word v satisfies $f_{\text{el}}(S)$ and, hence, also $f_{\mathcal{B}}(p)$. Thus, $v \in L_{val}(\mathcal{B})$.

To prove that $f(q)$ implies $f_{\mathcal{B}}(p)$ for every $q \in S$, pick a state $q \in S$. Following the definition of δ_{el}, we now construct a valuation word $v' \in (D \times \Pi)^*$ that satisfies the following properties:

- $v' \in L_{val}(\mathcal{A})$.
- The run of $\mathcal{A}$ on v' leads to q.
- The run of $\mathcal{B}$ on v' leads to p.

In order to obtain v', we insert symbols of the form $(\underline{b}, d)$ into v. Since the data values at such positions do not occur together with variables, their actual value is unimportant.

For the construction, let $v = a_1 \cdots a_n$ and let

$$\mathcal{A}_{\text{el}} \colon S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} S_n$$

be the run of $\mathcal{A}_{\text{el}}$ on v (so $S = S_n$). Let $q \in S$ and let $q'_n := q$. Since $\delta_{\text{el}}(S_{n-1}, a_n) = S_n$ and $q'_n \in S_n$, there is some state $q'_{n-1} \in S_{n-1}$ and $q_n \in S_n$ such that $\delta(q'_{n-1}, a_n) = q_n$, and $\mathcal{A} : q_n \xrightarrow{\underline{b}^{i_n}} q'_n$ for some $i_n \geq 0$. We continue this construction: if $q'_j \in S_j$ is defined for $j \in \{1, \ldots, n\}$, we construct q'_{j-1}, q_j and i_j as above. For $j = 0$ we finally pick i_0 such that $\mathcal{A} : q_0 \xrightarrow{\underline{b}^{i_0}} q'_0$.

Let $v' = \underline{b}^{i_0} a_1 \underline{b}^{i_1} a_2 \underline{b}^{i_2} \cdots a_n \underline{b}^{i_n}$. By construction, the run of $\mathcal{A}$ on v' leads to $q = q'_n$:

$$\mathcal{A} \colon q_0 \xrightarrow{\underline{b}^{i_0}} q'_0 \xrightarrow{a_1} q_1 \xrightarrow{\underline{b}^{i_1}} q'_1 \ldots \xrightarrow{a_n} q_n \xrightarrow{\underline{b}^{i_n}} q'_n$$

Since v' is obtained from v by inserting $\underline{b}$, the word v' also satisfies the formula $f(q)$ and thus $v' \in L_{val}(\mathcal{A})$. It remains to show that the run of $\mathcal{B}$ on v' leads to p. Since $\mathcal{B}$ is elastic, the only possibility that v' does not lead to p in $\mathcal{B}$ is a missing $\underline{b}$-loop at a position at which we inserted a non-empty sequence of $\underline{b}$. However, since $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$, such a position cannot exist.

We conclude that $f(q)$ implies $f_{\mathcal{B}}(p)$ using the following argument: If $f(q)$ does not imply $f_{\mathcal{B}}(p)$, then there exists an assignment of data values to the variables $y_1, \ldots, y_k$ such that $f(q)$ is satisfied but $f_{\mathcal{B}}(p)$ is not. By changing the data values in v' accordingly, we can produce a valuation word that is accepted by $\mathcal{A}$ but not by $\mathcal{B}$. However, this contradicts the assumption $L_{val}(\mathcal{A}) \subseteq L_{val}(\mathcal{B})$. Thus, $f(q)$ implies $f_{\mathcal{B}}(p)$. □ □

5.5 Modeling Linear Data Structures as Words and Converting EQDAs to Decidable Logics

In this section, we sketch briefly how to model arrays and lists as data words, and describe how to convert EQDAs to quantified logical formulas in decidable logics.

5.5.1 Modeling Program Configurations as Data Words

We model program configurations consisting of scalar variables, pointer or index variables,[1] and one (or more) linear data structures—lists or arrays in our case—as data words over a finite set of variables. The resulting data word is over the same domain D as the data in the cells of the data structure.

To simplify our modeling, we replace each scalar variable with an auxiliary pointer variable that points to a cell containing the data of the scalar variable. More precisely, for each scalar variable, we introduce a new pointer variable and extend the data structure with a new cell, which is located before the actual data structure begins and contains the data of the scalar variable; the order in which scalar variables are represented in the data structure is arbitrary but needs to be fixed. To be able to access the data at these positions (recall that QDAs can only access the data at position pointed

[1] Index variables occur in the case of arrays and index into arrays.

to by universally quantified variables), we amend QDAs with a register for each such pointer variable and extend the set F of formulas over which the considered QDA works with the atom $d(x)$ for each scalar variable x.

Let c be a program configuration over a linear data structure and a finite set PV of pointer or index variables, and let $\Sigma = 2^{PV}$. We model c as the data word

$$u_c = (a_1, d_1) \dots (a_n, d_n) \in (\Sigma \times D)^*,$$

such that the i-th symbol of the data word corresponds to the i-th cell of the data structure. In particular, the symbol $a_i \subseteq PV$ contains all pointer or index variables referencing the i-th cell, and d_i is the data stored in that cell.

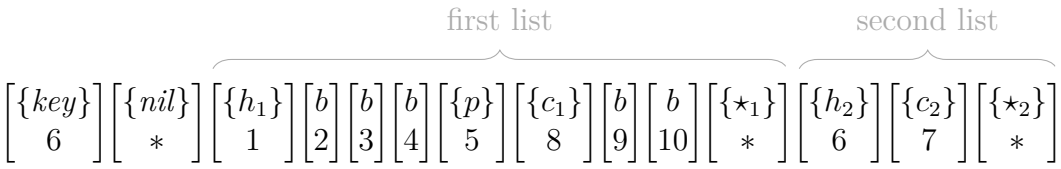
In the case of lists, some of the pointer variables might be *null* or point to unallocated memory, which cannot be referenced. We capture this situation in the data word by introducing an auxiliary pointer variable *nil* that points to a new cell at the beginning of the list. All pointer variables that are *null* or point to unallocated memory occur together with *nil*. The data value of the *nil* cell in the data word is not important and can be set to an arbitrary element of D.

Similarly, we introduce two new index variables *index_le_zero* and *index_geq_size* for arrays to capture index variables that are out-of-bounds (we assume that arrays are indexed starting at 0). The variable *index_le_zero* occurs together with all index variables that are less than zero, and *index_geq_size* occurs with those index variables that are either equal to or exceed the size of the array. Let the set *Aux* contain all auxiliary variables that may occur in our encoding.

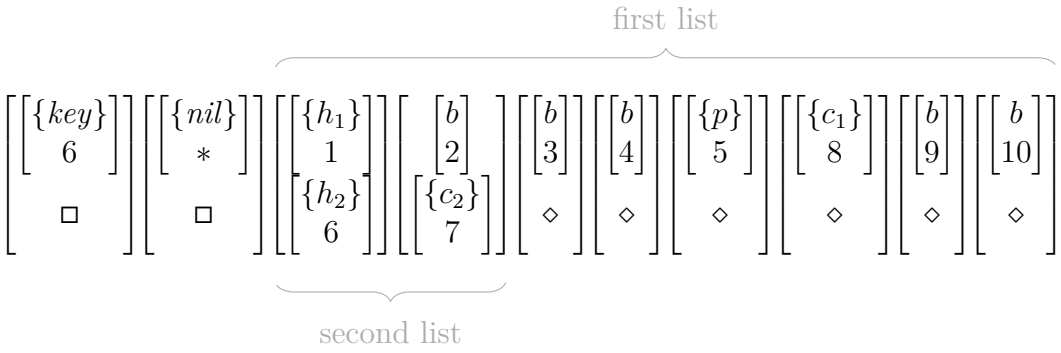
To model configurations of programs that manipulate more than one data structure, one can use one of the following two approaches: the first approach concatenates the data structures using a special pointer variable $\star_i$ to demarcate the end of the i-th data structure; the second approach models several data structures as one single combined data structure over an extended data domain by convolution of the original data words (i.e., by transforming a pair of words into a word over pairs); we refer the reader to standard textbooks (e.g., Khoussainov and Nerode [KN01]) for more details about convolution.

Let us illustrate the described translation with an example.

Example 5.5.1. *Consider a program that takes as input a scalar variable*



(a) Dataword modeling the concatenation of the two lists of Example 5.5.1.



(b) Dataword modeling the convolution of the two lists of Example 5.5.1. Both $\square$ and $\diamond$ are new auxiliary padding symbols (that must occur at the beginning and the end of the lists, respectively) used to equal the length of the lists.

Figure 5.2: Two datawords modeling the program configuration described in Example 5.5.1. A $*$-entry can be populated with an arbitrary data value.

key and a list l and partitions l into two separate lists: the first list contains all nodes whose data value is less than key and the second list contains all the remaining nodes of l. The program maintains pointer variables h_1 (corresponding to "head"), p, and c_1 (corresponding to a "current" pointer) to point into the first list and h_2 and c_2 to point into the second list. All nodes from h_1 through p in the first list are less than key. Similarly, all nodes in the second list (h_2 through c_2) are greater than or equal to key.

Let us consider a concrete scenario where l is a list with data values $1, 2, \ldots, 10$ in increasing order, and let key $= 6$. Moreover, consider the program configuration where the first seven nodes of the list have been processed and c_1 is pointing to the node with data value 8. Two data words corresponding to this program configuration—one using concatenation and one using convolution—are depicted in Figure 5.2. ⌟

5.5.2 Converting EQDAs to STRAND and the Array Property Fragment

We now describe a translation of an EQDA $\mathcal{A} = (Q, \Pi, q_0, \delta, f)$ into a formula $\varphi_{\mathcal{A}}$ (in the decidable syntactic fragment of STRAND, respectively in the Array Property Fragment) such that the data word language $L_{dat}(\mathcal{A})$ corresponds to the set of program configurations that model $\varphi_{\mathcal{A}}$. For brevity, we only consider the case of EQDAs working over a single list or array; for multiple lists or arrays, the translation is analogous.

Our translation is based on the notion of *simple paths* in EQDAs. A simple path is a sequence

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

of states connected by transitions starting in the initial state such that $\delta(q_i, a_{i+1}) = q_{i+1}$ is satisfies for all $i \in [n]$, no state occurs more than once, and all pointer and universally quantified variables occur exactly once; in particular, this implies $a_i \neq \underline{b}$. Note that there exist only finitely many simple paths in an EQDA because each state is allowed to occur at most once. We denote the set of all simple paths in the EQDA $\mathcal{A}$ by $P_{\mathcal{A}}$.

To simplify the translation, we assume without restricting the class of formulas represented by EQDAs that any EQDA $\mathcal{A}$ fulfills two structural properties:

1. Auxiliary variables, such as *nil* or scalar variables, which might have been introduced by the encoding of Section 5.5.1, occur in the beginning of any simple path in the exact same order. Although the exact order is unimportant, we fix one for the sake of simplicity: scalar variables occur first (in some fixed order), followed by *nil* in the case of lists, respectively *index_le _zero* and *index_geq _size* in the case of arrays.

2. Any simple path in $\mathcal{A}$ along which a universally quantified variable occurs together with auxiliary variables leads to a dedicated state labeled with the formula *true*. This means that the acceptance of a data word depends only on such valuations where no universally quantified variable occurs together with auxiliary variables. Since auxiliary variables were introduced for technical reasons only, valuation words in which a universally variable occurs together with auxiliary variables should, therefore, not influence the formula $\varphi_{\mathcal{A}}$.

EQDAs can check properties of the beginning and the end of a data structure, such as whether a pointer variable points to the head or tail of a list. In order to capture such properties, we use the constants 0 and *size* in the case of arrays, respectively *head* and *tail* in the case of lists, that point to the beginning and the end of the considered data structure. We assume that the *size* of an array is available as a variable in the scope of the program. For programs over list structures, the QDA only models the part of the list that can be accessed by traversing the *next* pointer from other pointer variables in the scope of the program. This implies that the head of the list is always available for reference in the EQDAs. Finally, we can express *tail* of the list in STRAND by the following formula: $\exists \mathit{tail}.\ (\mathit{succ}(\mathit{tail}, \mathit{nil}) \wedge \mathit{head} \rightarrow^* \mathit{tail})$.

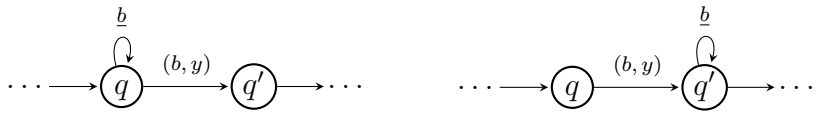
We are now ready to describe the actual translation. Roughly speaking, our translation considers each simple path of an EQDA individually, records the structural constraints of the variables along the path, and relates these constraints to the data formula of the final state of the path. By doing so, we construct a path formula φ_π for each simple path π in $\mathcal{A}$. The resulting formula $\varphi_\mathcal{A}$ is then the union of all such path formulas and an additional subformula that captures the valuation words not accepted by $\mathcal{A}$. Since there exists only finitely many simple path in $\mathcal{A}$, the resulting formula is finite.

Before we can enter the detailed definition of the formulas, we need another preprocessing of the path under consideration. Basically, we remove self-loops that on this path do not contribute to the acceptance of data words, which we call *irrelevant self-loops*. The precise reason for removing these loops becomes clear in Case 5 of the translation below.

For the formal definition of irrelevant self-loops, let π be a simple path in $\mathcal{A}$, and let q, q' be two states on π such that q' is the direct successor of q and the transition connecting q and q' is $\delta(q, (b, y)) = q'$. If q has a self-loop on $\underline{b}$ (i.e., $\delta(q, \underline{b}) = q$), then we define this self-loop inductively to be *irrelevant on* π if either q' has no self-loop on $\underline{b}$ or if this self-loop is irrelevant on π; the former situation is sketched in Figure 5.3a. Symmetrically, we define a self-loop on $\underline{b}$ at q' inductively as irrelevant on π if either q has no self-loop on $\underline{b}$ or this self-loop is irrelevant on π (see Figure 5.3b).

If a self-loop is irrelevant on π, it cannot contribute to the acceptance of a data word. To see why, consider two valuation words

$$v = v_1(b, y)\underline{b}v_2 \text{ and } v' = v_1\underline{b}(b, y)v_2$$



(a) An irrelevant self-loop in q.

(b) An irrelevant self-loop in q'.

Figure 5.3: Base cases of the inductive definition of irrelevant self-loops.

with $\mathrm{dw}(v) = \mathrm{dw}(v')$ (i.e., v and v' only differ in the valuation of the universally quantified variable y by one position). Moreover, assume that v is accepted along π using an irrelevant self-loop in q' on the $\underline{b}$ before v_2 (as in Figure 5.3b). In this situation, $\mathcal{A}$ reaches q after reading v_1 and hence rejects v' since q has no transition on $\underline{b}$. Thus, $\mathcal{A}$ rejects $\mathrm{dw}(v') = \mathrm{dw}(v)$. A similar argument applies to the other cases of the definition of irrelevant self-loop.

This reasoning shows that one can safely remove irrelevant loops from a path without changing the accepted language of data words. However, since being an irrelevant self-loop is a property depending on a path, it can happen that there are paths π, π' such that a self-loop in a state q is irrelevant on π but not on π'. We thus cannot remove irrelevant self-loops from $\mathcal{A}$ itself, but have to work on the level of paths.

For the actual translation into a formula, let $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$ be a simple path in $\mathcal{A}$ with $a_i \in \Sigma \times (Y \cup \{-\})$ and $a_i \neq \underline{b}$ for $i \in \{1, \dots, n\}$. The path formula corresponding to π is the implication

$$\varphi_\pi \coloneqq \psi_\pi \to \chi_\pi,$$

where the antecedent ψ_π (which we define shortly) serves as a guard that captures the relative positions of the variables along π and the consequent $\chi_\pi = f(q_n)$ is the data formula decorating the final state q_n of π (in the case of a translation into the Array Property Fragment, an overapproximation of $f(q)$ might be necessary).

We define the path guard ψ_π as follows: at each state q_i on the path, we construct *local constraints*, which describe how individual variables encoded in the incoming and outgoing transitions of q_i are related, and collect them in the set C_i; the path guard then is the conjunction

$$\psi_\pi \coloneqq \bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi.$$

For the construction of path guards, we use the following two notations: First, we use the notation $q_{i-1} \xrightarrow{a_i} q_i \in \pi$, respectively $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$, to denote parts of the simple path $\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$. Second, we use the input-symbol $a = (\sigma, y) \in \Sigma \times (Y \cup \{-\})$ and the set $(\Sigma \cup \{y\}) \setminus \{-\}$ of all variables (either pointer variables or universally quantified variables) occurring in a interchangeably; for instance, we write $x \in a$ to denote that the variable x occurs in a.

We divide the construction of path guards into two parts: The *auxiliary part* (i.e., Cases 1 and 2 below) covers the beginning of the path where pointer variables occur together with auxiliary variables, such as *nil*; recall that our encoding of Section 5.5.1 asserts that auxiliary variables occur always in the beginning of valuation words (and, correspondingly, in simple paths). The *data structure part* (i.e., Cases 3 to 6 below) deals with the remainder of the path, which is related to the actual data structure. The local constraints at state q_i are constructed according to the following (nonexclusive) case distinction:

Case 1: $q_{i-1} \xrightarrow{a_i} q_i \in \pi$ *and* $a_i \cap Aux \neq \emptyset$

Let $z \in a_i \cap Aux$ be the unique auxiliary variable.

- If z models a scalar variable, we set $C_i \leftarrow C_i \cup \{x = z\}$ for all $x \in a_i \setminus \{z\}$. (This case covers the second assumed structural property of EQDAs, described on Page 77. Note that x can only be a universally quantified variable and the state q_n of the simple path is labeled with the data formula *true*.)
- If $z = nil$, we set $C_i \leftarrow C_i \cup \{x = nil\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index\_le\_size$, we set $C_i \leftarrow C_i \cup \{x < 0\}$ for all $x \in a_i \setminus \{z\}$.
- If $z = index\_geq\_size$, we set $C_i \leftarrow C_i \cup \{x \geq size\}$ for all $x \in a_i \setminus \{z\}$.

Case 2: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$, $a_i \cap Aux \neq \emptyset$, *and* $a_{i+1} \cap Aux = \emptyset$

This case covers the boundary between the auxiliary and the data structure part of a simple path (i.e., processing the actual data structure starts at q_i). Here, we distinguish two cases:

- If $\delta(q_i, \underline{b})$ is undefined, we set $C_i \leftarrow C_i \cup \{x = 0\}$ for all $x \in a_{i+1}$ in the case of arrays, respectively $C_i \leftarrow C_i \cup \{x = \mathit{head}\}$ for all $x \in a_{i+1}$ in the case of lists.
- If $\delta(q_i, \underline{b}) = q_i$, we set $C_i \leftarrow C_i \cup \{0 \leq x\}$ for all $x \in a_{i+1}$ in the case of arrays, respectively $C_i \leftarrow C_i \cup \{\mathit{head} \rightarrow^* x\}$ for all $x \in a_{i+1}$ in the case of lists.

Cases 3 to 6 below only apply if no auxiliary variables occur in the incoming or outgoing transitions. Note that such situations indeed occur since we assume that Y contains at least one variable (which occurs on every simple path after all auxiliary variables).

Case 3: $q_{i-1} \xrightarrow{a_i} q_i \in \pi$

For all $x, x' \in a_i$ with $x \neq x'$, we set $C_i \leftarrow C_i \cup \{x = x'\}$.

Case 4: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ *and* $\delta(q_i, \underline{b}) = q_i$

Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. In the case of arrays, we consider two cases:

- If $x_1 \notin Y$ or $x_2 \notin Y$, then we set $C_i \leftarrow C_i \cup \{x_1 < x_2\}$.
- If $x_1 \in Y$, $x_2 \in Y$, and $(a_i \cup a_{i+1}) \cap \Sigma = \emptyset$ (i.e., only universally quantified variables occur), then the Array Property Fragment forbids two adjacent universally quantified variables to be related by the relation $<$; in this case, we set $C_i \leftarrow C_i \cup \{x_1 \leq x_2\}$ and $\chi_\pi \leftarrow \chi_\pi \vee (d(x_1) = d(x_2))$. At this point, the translation does not capture the exact semantics of the EQDA (we comment on this shortly). Note that $\leq$ is an elastic relation.

In the case of lists, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow^+ x_2\}$ where $\rightarrow^+$ is the transitive closure of the successor relation $\rightarrow$. Note that $\rightarrow^+$ is an elastic relation.

Case 5: $q_{i-1} \xrightarrow{a_i} q_i \xrightarrow{a_{i+1}} q_{i+1} \in \pi$ *and* $\delta(q_i, \underline{b})$ *is undefined*

Let $x_1 \in a_i$ and $x_2 \in a_{i+1}$. We distinguish two cases:

- Let $x_1 \notin Y$ or $x_2 \notin Y$. In the case of arrays, we set $C_i \leftarrow C_i \cup \{x_2 = x_1 + 1\}$. In the case of lists, we set $C_i \leftarrow C_i \cup \{x_1 \rightarrow x_2\}$.

- Let $x_1 \in Y$ and $x_2 \in Y$. Since both STRAND and the Array Property Fragment forbid expressing that two universally quantified variables are a fixed distance away, we express their relation indirectly: we identify a state q on the path π that is closest to q_i (the direction is not important) and has a transition containing a pointer variable $p \in PV$ (if $PV = \emptyset$, we use *head* or *tail*). Since $\mathcal{A}$ does not contain any irrelevant self-loops, the subpath from q_i to q has no self-loops. Thus, we can constrain the universally quantified variables at q_i to be a fixed distance away from the pointer variable p. For a translation into the Array Property Fragment, we achieve this using arithmetic on the pointer variables. For a translation into the decidable syntactic fragment of STRAND, we obtain the same effect by existentially quantifying monadic predicates $x_1, x_2, \cdots, x_d$ that track the distance d from the universally quantified variable y at q_i to the pointer variable p as follows: $succ(y, x_1) \wedge succ(x_1, x_2) \wedge \cdots \wedge succ(x_{d-1}, x_d) \wedge x_d = p$. Since the distance d between q and q_i is bounded, a finite number of such predicates suffices.

Case 6: $q_{n-1} \xrightarrow{a_n} q_n \in \pi$

In this case, q_n is the last state of π and $\delta(q_{n-1}, a_n) = q_n$ the last transition. We distinguish two cases:

- If $\delta(q_n, \underline{b})$ is undefined, we set $C_n \leftarrow C_n \cup \{x = size - 1\}$ for all $x \in a_n$ in the case of arrays, respectively $C_n \leftarrow C_n \cup \{x = tail\}$ for all $x \in a_n$ in the case of lists.
- If $\delta(q_n, \underline{b}) = q_n$, we set $C_n \leftarrow C_n \cup \{x < size\}$ for all $x \in a_n$ in the case of arrays, respectively $C_n \leftarrow C_n \cup \{x \rightarrow^* tail\}$ for all $x \in a_n$ in the case of lists.

Since the Array Property Fragment lacks the ability to check whether two universally quantified variables are different, Case 4 needs to introduce an overapproximation of the real constraints along a simple path if two universally quantified variables, say y and y', are adjacent at a state with a self-loop on $\underline{b}$ (i.e., the path guard is incorrectly satisfied even if $y = y'$ holds). In order

to compensate for this, we amend the formula χ_π by disjointly adding the constraint $d(y) = d(y')$, which ensures that the path formula is satisfied if $y = y'$ holds (since $y = y'$ implies $d(y) = d(y')$). This way, the path formula checks the structural and data constraints of the path if the valuation satisfies $y_1 < \cdots < y_k$, but also when universally quantified variables are equal (which cannot be checked by an EQDA due to fact that the input alphabet of EQDAs requires universally quantified variables to be at different position). Note that a path formula with such an approximation is imprecise in general.

The complete translation functions as follows: It collects the sets C_i along every simple path $\pi \in P_\mathcal{A}$ and constructs the formulas ψ_π and χ_π. For a translation into the decidable syntactic fragment of STRAND, it returns the formula

$$\varphi_\mathcal{A} \coloneqq \forall y_1 \colon \ldots \forall y_k \colon \Bigg[\underbrace{\left(\bigwedge_{\pi \in P_\mathcal{A}} \psi_\pi \to \chi_\pi \right)}_{\varphi_{sp}} \wedge \underbrace{\left(\left[(\mathit{head} \to^* y_1 \to^+ \cdots \to^+ y_k \to^* \mathit{tail}) \wedge \neg \left(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi \right) \right] \to \mathit{false} \right)}_{\varphi_{\neg sp}} \Bigg];$$

For a translation into the Array Property Fragment, it returns

$$\varphi_\mathcal{A} \coloneqq \forall y_1 \colon \ldots \forall y_k \colon \Bigg[\underbrace{\left(\bigwedge_{\pi \in P_\mathcal{A}} \psi_\pi \to \chi_\pi \right)}_{\varphi_{sp}} \wedge \underbrace{\left(\left[(0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}) \wedge \neg \left(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi \right) \right] \to \bigvee_{\substack{y, y' \in Y, \\ y \neq y'}} d(y) = d(y') \right)}_{\varphi_{\neg sp}} \Bigg].$$

The subformula φ_{sp} is the conjunction of all path formulas whereas the subformula $\varphi_{\neg sp}$ captures valuation words that have the right ordering of the universally quantified variables but do not admit a run of $\mathcal{A}$ (i.e., that are rejected by $\mathcal{A}$). As in the case of path formulas, the Array Property Fragment formula $\varphi_{\neg sp}$ only approximates the correct semantics of $\mathcal{A}$. Again, the disjunction constituting the consequent compensates for the necessary overapproximation in the antecedent ($y_1 \leq \cdots \leq y_k$ instead of $y_1 < \cdots < y_k$).

Since the decidable syntactic fragment of STRAND allows negating atomic formulas, $\varphi_{\mathcal{A}}$ is in this fragment. Though the Array Property Fragment also allows negation over atomic formulas that relate two pointer variables or a pointer variable and a universally quantified variable, negation of an atomic formula of the form $y \leq y'$ is not allowed [BMS06]. However, since we assume both a fixed variable ordering on Y along simple paths and that all other paths with a different ordering lead to the formula *true*, we can remove formulas of the form $\neg(y \leq y')$ from $\neg\big(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi\big)$; as before, considering a different ordering of the variables in Y is not necessary because these variables are universally quantified. After removing such subformulas, the formula $\varphi_{\mathcal{A}}$ falls into the Array Property Fragment.

When we apply our translation to an EQDA to obtain a formula in the syntactic decidable fragment of STRAND over lists, the obtained formula exactly characterizes the set of program configurations that correspond to the language of data words accepted by the given EQDA. However, due to the necessary abstractions introduced by our translation into the Array Property Fragment, the formula obtained from translating the EQDA over arrays might not characterize the semantics of the given EQDA exactly. However, we can at least assert that all data words accepted by this EQDA correspond to a program configuration satisfying the formula.

To make this intuition precise, let us introduce the following notations: Given a program configuration c, let (c) denote the natural translation of c into an interpretation for formulas in the Array Property Fragment, respectively in the decidable syntactic fragment of STRAND.[2] Moreover, let $(c, y_1, \ldots, y_k)$ denote the interpretation (c) in which the universally quantified variables are fixed to the values $y_1, \ldots, y_k$.

The following theorem now summarizes the main result of our translation.

Theorem 5.5.2. *Let $\mathcal{A}$ be an EQDA, c a program configuration, u_c the data word corresponding to c, and $\varphi_{\mathcal{A}}$ the formula obtained after the translation (either in the decidable syntactic fragment of* STRAND *or the Array Property Fragment).*

a) For a translation into the decidable syntactic fragment of STRAND*, the*

[2]We make sure that the type of an interpretation (i.e., whether it is for formulas in the Array Property Fragment or in the decidable syntactic fragment of STRAND) is always clear from the context.

equivalence

$$u_c \in L_{dat}(\mathcal{A}) \text{ if and only if } (c) \models \varphi_{\mathcal{A}}$$

holds.

b) For a translation into the Array Property Fragment, the implication

$$u_c \in L_{dat}(\mathcal{A}) \text{ implies } (c) \models \varphi_{\mathcal{A}}$$

holds.

The abstraction along simple paths with $y < y'$ introduced by our translation is the reason why Theorem 5.5.2 only holds in one direction for the Array Property Fragment. For this reason, we first prove Theorem 5.5.2 for the translation into the decidable syntactic fragment of STRAND; based on the insight gained in the proof, it becomes much easier to prove Theorem 5.5.2 for the translation into the Array Property Fragment.

Decidable syntactic fragment of Strand The pivotal fact on which Theorem 5.5.2 relies is that the path guard ψ_π exactly captures the structural constraints along π. The next lemma formalizes this intuition.

Lemma 5.5.3. *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the decidable syntactic fragment of* STRAND*. Moreover, let c be a program configuration, $y_1, \ldots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \ldots, y_k$. Then, the following equivalence holds:*

$$\text{the unique run of } \mathcal{A} \text{ on } v \text{ is along } \pi \text{ if and only if } (c, y_1, \ldots, y_k) \models \psi_\pi.$$

Proof. We split the proof into two parts: we first show the direction from left to right and subsequently the reverse direction. The direction from left to right is straightforward and simply exploits the fact we only add such local constraints to a path guard that are obviously satisfied along the given path. The direction from right to left, however, is more elaborate to prove.

From left to right Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} q_n$$

a simple path in $\mathcal{A}$ and assume that the unique run of $\mathcal{A}$ on v is along π. Since the path guard is the conjunction $\bigwedge_{i=1}^{n} \bigwedge_{\psi \in C_i} \psi$ of all local constraints along π, it is enough to prove that $(c, y_1, \dots, y_k)$ satisfies each individual local constraint. To this end, let ψ be a local constraint, say constructed at state q_i of π.

In order to show $(c, y_1, \dots, y_k) \models \psi$, we have to distinguish due to which case of the translation the constraint ψ has been constructed. However, since most cases are similar, we do not give a thorough proof here but exemplary consider Case 4.

If ψ has been introduced in Case 4, then $\psi \coloneqq x_1 \rightarrow^+ x_2$ with $x_1 \in a_i$ and $x_2 \in a_{i+1}$. Since the run of $\mathcal{A}$ on v is along π, we know that all variables $x \in a_i$ occur before the variables $x' \in a_{i+1}$. Thus, $(c, y_1, \dots, y_k)$ satisfies $x \rightarrow^+ x'$ for all such x, x'. This in turn means $(c, y_1, \dots, y_k) \models \psi$.

From right to left Let

$$\pi = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} q_m$$

be a simple path in $\mathcal{A}$ and $(c, y_1, \dots, y_k)$ a model of ψ_π. Towards a contradiction, assume that the run of $\mathcal{A}$ on v is along a different simple path, say

$$\pi' = q_0 \xrightarrow{a'_1} q'_1 \xrightarrow{a'_2} \dots \xrightarrow{a'_n} q'_n.$$

Then, there exists a position $i \in \mathbb{N}_+$ at which both paths diverge; that is, $a_j = a'_j$ and $q_j = q'_j$ for all $j \in [i]$, $a_i \neq a'_i$, and $q_i \neq q'_i$. Note that such a position always exists because the states of $\mathcal{A}$ are "typed" (i.e., $\mathcal{A}$ has to remember which variable it has already read). Figure 5.4 depicts such a situation.

We observe that all input symbols along the paths π and π' are different from $\underline{b}$ because $\mathcal{A}$ is elastic. Thus, if $a_i \neq a'_i$, then there exists a variable $x \in PV \cup Y$ that is missing in exactly one of a_i and a'_i (i.e., $x \in a_i$ if and only if $x \notin a'_i$). Without loss of generality, let us assume $x \in a_i$ and $x \notin a'_i$.

Since $a'_i \neq \underline{b}$, there also exists a variable $x' \in a'_i$ that is different from

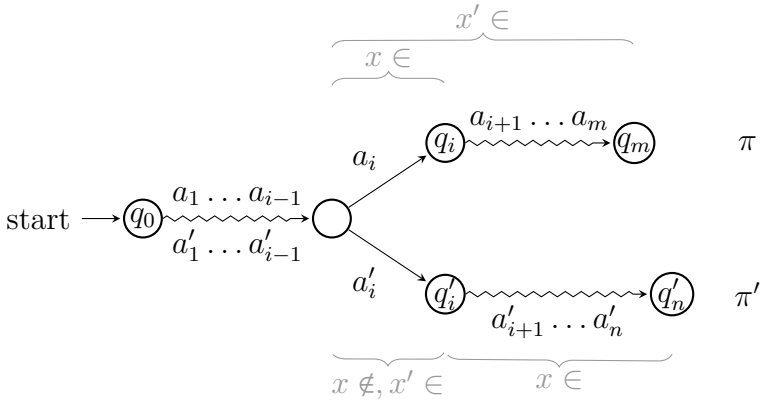


Figure 5.4: Two diverging simple paths π, π'.

x. Moreover, since π' is a simple path (which implies that all pointer and universally quantified variables occur exactly once), the variable x also occurs in π', but only in one of the inputs $a'_{i+1}, \dots, a'_n$; note that x' might or might not occur together with x on π.

We now distinguish two cases:

1. An auxiliary variable, say z, occurs in the input-symbol a_i on π; that is, q_i belongs to the auxiliary part of π. We first observe that x cannot be an auxiliary variable because we assume that auxiliary variables appear never together and always in the same, fixed order. Thus, the following two cases remain:

   (a) The variable x occurs on π' together with an auxiliary variable, say z', that is different from z. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models x = z'$. Consequently, $(c, y_1, \dots, y_k) \not\models x = z$ because $x = z \wedge x = z'$ is unsatisfiable if $z \neq z'$. However, the path guard ψ_π contains the local constraint $x = z$ (see Case 1). Thus, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.

   (b) The variable x does not occur together with an auxiliary variable on π'. Since we assume the run of $\mathcal{A}$ on v to be along π', this means $(c, y_1, \dots, y_k) \models \mathit{head} \rightarrow^* x$. Consequently, $(c, y_1, \dots, y_k) \not\models x = z$ because z is an auxiliary variable that occurs before head. However, the path guard ψ_π contains the local constraint $x = z$ (again, see Case 1). Thus, $(c, y_1, \dots, y_k) \not\models \psi_\pi$, which yields a contradiction.

2. The input-symbol a_i on π does not contain an auxiliary variable; that

is, q_i belongs to the data structure part of π. Since we assume the run of $\mathcal{A}$ on v to be along π', the variable x' points to a cell that is located before the cell pointed to by x. Hence, $(c, y_1, \ldots, y_k) \models x' \rightarrow^+ x$. Consequently, $(c, y_1, \ldots, y_k) \not\models x \rightarrow^* x'$ because $x \rightarrow^* x' \wedge x' \rightarrow^+ x$ is unsatisfiable. However, the path guard ψ_π implies $x \rightarrow^* x'$ (see Cases 4 and 5) although it might not contain this subformula explicitly. Thus, $(c, y_1, \ldots, y_k) \not\models \psi_\pi$, which yields the desired contradiction.

□ □

Using Lemma 5.5.3, we can now prove Part (a) of Theorem 5.5.2.

of Theorem 5.5.2(a). Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \cdots \prec y_k$ the predetermined order in which the universally quantified variables have to occur in the input of $\mathcal{A}$, and $\varphi_\mathcal{A}$ the formula in the decidable syntactic fragment of STRAND resulting from our translation. In addition, let c be a program configuration and u_c the data word modeling c.

We first show the direction from left to right (i.e., $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$) and subsequently the reverse direction (i.e., $(c) \models \varphi_\mathcal{A}$ implies $u_c \in L_{dat}(\mathcal{A})$).

From left to right Let $u_c \in L_{dat}(\mathcal{A})$. In order to prove that the interpretation (c) satisfies $\varphi_\mathcal{A} \coloneqq \forall y_1 \colon \ldots \forall y_k \colon (\varphi_{sp} \wedge \varphi_{\neg sp})$, we fix an arbitrary valuation $y_1, \ldots, y_k$ of Y and show

$$(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}.$$

In the case that $\mathit{head} \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* \mathit{tail}$ does not hold, we first observe that $(c, y_1, \ldots, y_k)$ does not satisfy any path guard because each path guard implies $\mathit{head} \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* \mathit{tail}$. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \ldots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In the case that $\mathit{head} \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* \mathit{tail}$ holds, let v be the valuation word resulting from extending u_c with the valuation $y_1, \ldots, y_k$ (which implies $\mathrm{dw}(v) = u_c$). We proceed the proof by first showing that $(c, y_1, \ldots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

1. Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. This particularly means that the unique run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \models f(q)$. By Lemma 5.5.3, we know $(c, y_1, \ldots, y_k) \models \psi_\pi$. Moreover, since $f(q) = \chi_\pi$ and $r \models f(q)$, we also know $(c, y_1, \ldots, y_k) \models \chi_\pi$ and, thus, $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$. On the other hand, Lemma 5.5.3 asserts that no other path guard is satisfied by $(c, y_1, \ldots, y_k)$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.

2. The fact that $(c, y_1, \ldots, y_k) \models \psi_\pi$ holds (see above) implies $(c, y_1, \ldots, y_k) \not\models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Hence, the antecedent of $\varphi_{\neg sp}$ is not satisfied and, therefore, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$.

Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_\mathcal{A}$.

From right to left Let u_c be a data word with $u_c \notin L_{dat}(\mathcal{A})$ and c the corresponding program configuration. We need to show that c does not satisfy $\varphi_\mathcal{A}$.

Since $u_c \notin L_{dat}(\mathcal{A})$, there exists a valuation $y_1, \ldots, y_k$ and a corresponding valuation word v (i.e., u_c extended by $y_1, \ldots, y_k$ results in v) such that $v \notin L_{val}(\mathcal{A})$. This valuation word is rejected either

1. due to a missing transition; or

2. due to the fact that the run of $\mathcal{A}$ on v ends in a configuration (q, r) with $r \not\models f(q)$.

In the first case, the run of $\mathcal{A}$ on v does not lead along a simple path. By Lemma 5.5.3, this implies $(c, y_1, \ldots, y_k) \not\models \psi_\pi$ for every $\pi \in P_\mathcal{A}$. Hence, $(c, y_1, \ldots, y_k) \models \neg(\bigvee_{\pi \in P_\mathcal{A}} \psi_\pi)$. Since we assume that $\mathcal{A}$ accepts all valuation words that violate the fixed order of the universally quantified variables or where at least one of these variables points to *nil*, we know that $(c, y_1, \ldots, y_k) \models \mathit{head} \rightarrow^* y_1 \rightarrow^+ \cdots \rightarrow^+ y_k \rightarrow^* \mathit{tail}$ holds. Thus, $(c, y_1, \ldots, y_k) \not\models \varphi_{\neg sp}$ and, consequently, $c \not\models \varphi_\mathcal{A}$.

In the second case, the run of $\mathcal{A}$ on v leads along a simple path, say π, ending in the configuration (q, r). By Lemma 5.5.3, this implies $(c, y_1, \ldots, y_k) \models \psi_\pi$. However, since $r \not\models f(q) = \chi_\pi$, we have $(c, y_1, \ldots, y_k) \not\models \chi_\pi$. Thus, $(c, y_1, \ldots, y_k) \not\models \varphi_{sp}$ (because $(c, y_1, \ldots, y_k) \not\models \psi_\pi \rightarrow \chi_\pi$) and, consequently, $c \not\models \varphi_\mathcal{A}$.

In total, $u_c \notin L_{dat}(\mathcal{A})$ implies $(c) \not\models \varphi_{\mathcal{A}}$ (i.e., $(c) \models \varphi_{\mathcal{A}}$ implies $u_c \in L_{dat}(\mathcal{A})$). □ □

Array Property Fragment The approximation in Case 4 of our translation is the reason why Theorem 5.5.2 holds only in one direction in the case of a translation into the Array Property Fragment. In order to prove this direction, we first show that the path guard ψ_π overapproximates the structural constraints of π. The next lemma formalizes this.

Lemma 5.5.4. *Let $\mathcal{A}$ be an EQDA over the finite set PV of pointer variables and the finite, nonempty set Y of universally quantified variables, π a simple path in $\mathcal{A}$, and ψ_π the corresponding path guard in the Array Property Fragment. Moreover, let c be a program configuration, $y_1, \ldots, y_k$ a valuation of Y, and v the valuation word modeling c and $y_1, \ldots, y_k$. Then, the following implication holds:*

$$\textit{if the unique run of } \mathcal{A} \textit{ on } v \textit{ is along } \pi\textit{, then } (c, y_1, \ldots, y_k) \models \psi_\pi.$$

Proof. One can prove Lemma 5.5.4 in the same way as Lemma 5.5.3 (see Page 85): again, we consider each local constraint ψ of a path guard individually and show $(c, y_1, \ldots, y_k) \models \psi$. In fact, we can reuse the proof of Lemma 5.5.3 except for a slightly different treatment of Case 4, which we sketch below.

Assume that ψ has been added at state q_i of the simple path $\pi = q_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} q_n$, and let $x_1 \in a_i$ and $x_2 \in a_{i+1}$.

If $x_1 \notin Y$ or $x_2 \notin Y$, then this situation matches Case 4 of the proof of Lemma 5.5.3 and immediately yields the desired result.

If $x_1 \in Y$, $x_2 \in Y$, and both variables do not occur together with a pointer variable, then the translation adds $\psi \coloneqq x_1 \leq x_2$ instead of the "correct" constraint $x_1 < x_2$. However, we know that all variables $x \in a_i$ occur before the variables $x' \in a_{i+1}$ because the run of $\mathcal{A}$ on v is along π. Thus, $(c, y_1, \ldots, y_k) \models x < x'$ for all such x, x', which implies $(c, y_1, \ldots, y_k) \models x_1 \leq x_2$ (i.e., $(c, y_1, \ldots, y_k) \models \psi$). □ □

We can now prove Part (b) of Theorem 5.5.2.

of Theorem 5.5.2(b). Let $\mathcal{A}$ be an EQDA over PV and Y, $y_1 \prec \cdots \prec y_k$ the predetermined order in which the universally quantified variables have to

occur in the input of $\mathcal{A}$, and $\varphi_{\mathcal{A}}$ the formula in the Array Property Fragment resulting from our translation. Moreover, let c be a program configuration and u_c the data word modeling c. Finally, assume $u_c \in L_{dat}(\mathcal{A})$.

We have to show that (c) is a model of $\varphi_{\mathcal{A}}$. This proof is similar to the direction from left to right of the proof of Theorem 5.5.2(a): we again fix an arbitrary valuation $y_1, \ldots, y_k$ of Y and show

$$(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}.$$

In the case that $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$ does not hold, we again observe that $(c, y_1, \ldots, y_k)$ does not satisfy any path guard because each path guard implies $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$ since the antecedent of each path formula is unsatisfied. Moreover, $(c, y_1, \ldots, y_k)$ does not satisfy the antecedent of $\varphi_{\neg sp}$ and, consequently, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In the case that $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$ holds, we distinguish two cases:

1. All universally quantified variables are different; that is, $y_i \neq y_j$ holds for all $i, j \in \{1, \ldots, k\}$ with $i \neq j$. In this case, let v be the valuation word resulting from extending u_c with the valuation $y_1, \ldots, y_k$. We proceed the proof by first showing that $(c, y_1, \ldots, y_k)$ satisfies φ_{sp} and subsequently that it satisfies $\varphi_{\neg sp}$.

   (a) Since $u_c \in L_{dat}(\mathcal{A})$, the valuation word v is also accepted by $\mathcal{A}$, say along the simple path π. By Lemma 5.5.4, we know that then $(c, y_1, \ldots, y_k) \models \psi_\pi$ holds. Since $v \in L_{val}(\mathcal{A})$, the registers satisfy the data formula of the final state of π. Thus, $(c, y_1, \ldots, y_k) \models \chi_\pi$ and, consequently, $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$.

   To complete this case, we argue that there exists no other path $\pi' \in P_{\mathcal{A}}$ with $\pi' \neq \pi$ and $(c, y_1, \ldots, y_k) \models \psi_{\pi'}$. Towards a contradiction, assume the contrary and let π' such a simple path. By using arguments similar to those in the direction from right to left of the proof of Lemma 5.5.3, one can show that this can only happen due to an overapproximation of the form $y_i \leq y_j$ (rather than $y_i < y_j$). This, in turn, implies that there exists $i, j \in \{1, \ldots, k\}$ with $i < j$ and $y_i = y_k$, which contradicts the assumption that all universally quantified variables are different.

In total, $(c, y_1, \ldots, y_k)$ satisfies the path formula of each simple path. Hence, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.

(b) Since $u_c \in L_{dat}(\mathcal{A})$, we know that there exists a simple path $\pi \in P_{\mathcal{A}}$ such that $(c, y_1, \ldots, y_k) \models \psi_\pi$ (see above). Thus, $(c, y_1, \ldots, y_k) \not\models \neg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi)$ because removing subformulas of the form $\neg(y \leq y')$ from a path guard potentially results in more interpretations satisfying it and, thus, less satisfying its negation. This implies $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$ since we assume $0 \leq y_1 \leq \cdots \leq y_k < \mathit{size}$.

Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

2. There exist $i, j \in \{1, \ldots, k\}$ such that $i < j$ and $y_i = y_j$. In this case, there might be a simple path $\pi \in P_{\mathcal{A}}$ such that $(c, y_1, \ldots, y_k) \models \psi_\pi$. Since universally quantified variables never occur together on a simple path (due to the choice of the input alphabet of QDAs), $(c, y_1, \ldots, y_k)$ can only satisfy ψ_π due to the overapproximation $y_i \leq y_j$ (rather than $y_i < y_j$) introduced by Case 4 of our translation. This means that the formula χ_π is constructed by taking the disjunction of the formulas $f(q)$ (assuming that q is the final state of π), $d(y_i) = d(y_j)$, and potentially other formulas of the form $d(y) = d(y')$ for $y, y' \in Y$. Thus, $(d(y_i) = d(y_j)) \rightarrow \chi_\pi$. Since $y_i = y_j$, we have $d(y_i) = d(y_j)$ and, hence, $(c, y_1, \ldots, y_k) \models \chi_\pi$. This, in turn, means $(c, y_1, \ldots, y_k) \models \psi_\pi \rightarrow \chi_\pi$. Since these arguments are true for all simple paths $\pi' \in P_{\mathcal{A}}$ for which $(c, y_1, \ldots, y_k) \models \psi_{\pi'}$ holds, $(c, y_1, \ldots, y_k) \models \varphi_{sp}$.

On the other hand, $(c, y_1, \ldots, y_k) \models \varphi_{\neg sp}$ because $(c, y_1, \ldots, y_k)$ satisfies the consequent of $\varphi_{\neg sp}$ due to the equality $y_i = y_j$. Thus, $(c, y_1, \ldots, y_k) \models \varphi_{sp} \wedge \varphi_{\neg sp}$.

In total, $u_c \in L_{dat}(\mathcal{A})$ implies $(c) \models \varphi_{\mathcal{A}}$. □ □

5.6 Learning Likely Invariants over Linear Data Structures by Passively Learning QDAs

Now that we have studied quantified data automata, their properties, and introduced a subclass called elastic QDAs that can be translated to decidable logics, we develop learning algorithms for QDAs and EQDAs that help us

learn these automata and, subsequently, learn quantified invariants over linear data structures. In this section, we develop a passive learning algorithm for QDAs that learns them from a labeled sample of program configurations that manifest along dynamic test runs. In the next section, Section 5.7, we develop an ICE learning algorithm for QDAs and deploy it together with a verification oracle to robustly learn quantified program invariants over linear data structures.

In an active black-box learning framework, we look upon the invariant as a set of configurations of the program, and allow the learner to query the teacher for membership and equivalence queries on this set. Furthermore, we fix a particular representation class for these sets, and demand that the learner learn the smallest (simplest) representation that describes the set. A learning algorithm that learns in time polynomial in the size of the simplest representation of the set is desirable. In passive black-box learning, the learner is given a sample of examples and counterexamples of configurations, and is asked to synthesize the simplest representation that includes the examples and excludes the counter-examples. Our motivation for doing passive learning is that we believe (and we validate this belief using experiments) that in many problems, a lighter-weight passive-learning algorithm which learns from a few randomly-chosen small data structures is sufficient to find the invariant. Note that passive learning algorithms, in general, often boil down to a guess-and-check algorithm of some kind, and often pay an exponential price in the size of the property learned. As opposed to developing purely passive algorithms for learning QDAs, we develop an active learning algorithm a la Angluin that allows the learner to query the teacher for membership and equivalence queries and the teacher answers on the set of configurations that belong to an invariant. We then employ the active learning algorithm in a passive learning setting where we show that by building an imprecise teacher that answers the questions of the active learner, we can build effective invariant generation algorithms that learn simply from a finite set of examples. Designing a passive learning algorithm using an active learning core allows us to build more interesting algorithms; in our algorithm, the inacurracy/guessing is confined to the way the teacher answers the learner's questions.

The passive learning algorithm works as follows. Assume that we have a finite set of configurations S, obtained from sampling the program (by perhaps just running the program on various random small inputs). We

are required to learn the simplest representation that captures the set S (in the form of a QDA). We now use an active learning algorithm for QDAs; membership questions are answered with respect to the set S (note that this is imprecise, as an invariant I must include S but need not be precisely S). When asked an equivalence query with a set I, we check whether $S \subseteq I$; if yes, we can check if the invariant is adequate using a constraint solver and the program.

It turns out that this is a good way to build a passive learning algorithm. First, enumerating random small data structures that get manifest at the header of a loop fixes for the most part the structure of the invariant, since the invariant is forced to be expressed as a QDA. Second, our active learning algorithm for QDAs promises never to ask long membership queries (queried words are guaranteed to be less than the diameter of the automaton), and often the teacher has the correct answers. Finally, note that the passive learning algorithm answers membership queries with respect to S; this is because we do not know the true invariant, and hence err on the side of keeping the invariant semantically small. This inaccuracy is common in most learning algorithms employed for verification (e.g, Boolean learning [KJD$^+$10], compositional verification [CGP03, AMN05], etc). This inaccuracy could lead to a non-optimal QDA being learnt, and is precisely why our algorithm need not work in time polynomial in the simplest representation of the concept (though it is polynomial in the invariant it finally learns).

The proof of the efficacy of the passive learning algorithm rests in the experimental evaluation. We implement the passive learning algorithm (which in turn requires an implementation of the active learning algorithm). By building a teacher using dynamic test runs of the program and by pitting this teacher against the learner, we learn invariant QDAs, and then over-approximate them using elastic QDAs (EQDAs). These EQDAs are then transformed into formulas over decidable theories of arrays and lists. Next, we describe the details about the active-learning algorithm and how we deploy it in a passive setting to learn quantified invariants from configurations realized along dynamic runs.

5.6.1 Angluin's Active Learning Setting

Angluin's active learning setting, which she has introduced in [Ang87a], is a framework in which the task is to "learn" a regular language $L \subseteq \Sigma^*$ over a fixed alphabet Σ—called *target language*—by actively querying an external source for information. The learning takes place between a learning algorithm—abbreviated *learner*—and the information source—called *teacher*. The teacher can answer two types of queries: membership and equivalence queries.

Membership query On a membership query, the learner provides a word $u \in \Sigma^*$ and the teacher replies "yes" if $u \in L$ and "no" if $u \notin L$.

Equivalence query On an equivalence query, the learner provides a regular language, usually given as a DFA $\mathcal{A}$, and the teacher checks whether $\mathcal{A}$ is equivalent to the target language. If this is the case, he returns "yes". If this is not the case, he returns a *counterexample* $u \in L(\mathcal{A}) \Leftrightarrow u \notin L$ as a witness that $L(\mathcal{A})$ and L are indeed different.

Given a teacher for a regular target language L, the learner's task is to find a DFA (usually of minimal size) that passes an equivalence query.

In [Ang87a], Angluin has not only introduced the active learning framework but also developed a learning algorithm that learns the unique minimal deterministic automaton that accepts the target language in polynomial time. This algorithm is based on the *Myhill-Nerode congruence* of the target language: given a language $L \subseteq \Sigma^*$, the Myhill-Nerode congruence is the equivalence relation $\sim_L$ over words defined by $u \sim_L v$ if and only if $uw \in L \Leftrightarrow vw \in L$ for all $w \in \Sigma^*$. Angluin's pivotal idea is to start with a coarse approximation of the Myhill-Nerode congruence and refine the approximation, using membership and equivalence queries, until the Myhill-Nerode congruence has been computed exactly; since the number of equivalence classes is finite for every regular language, this approach is guaranteed to terminate eventually.

Internally, Angluin's algorithm stores the data learned so far in a so-called *observation table* $O = (R, S, T)$; the set $R \subseteq \Sigma^*$ is a finite, prefix-closed set of *representatives* that serve to represent equivalence classes, the set $S \subseteq \Sigma^*$ is a finite set of *samples* that are used to distinguish representatives, and $T\colon (R \cup R \cdot \Sigma) \cdot S \to \{\text{"yes"}, \text{"no"}\}$ is a mapping that stores the actual table entries and is filled using membership queries.

Angluin's algorithm proceeds in rounds: In each round, the algorithm extends the observation table until it is *closed* and *consistent*, which roughly corresponds to the situation that the data stored in the table forms a congruence. Then, Angluin's algorithm derives a conjecture DFA from the table (similar to the construction of the minimal DFA from the Myhill-Nerode congruence) and submits this conjecture on an equivalence query. If the teacher replies "yes", the learning terminates; if the teacher returns a counterexample, on the other hand, Angluin's algorithm adds the counterexample along with all of its prefixes as new representatives to the table and proceeds with the next iteration.

We refer the reader to [Ang87a] for an in-depth presentation of Angluin's active learning setting and Angluin's algorithm. Here, we just want to summarize the main results.

Theorem 5.6.1 (Angluin [Ang87a])**.** *Given a teacher for a regular target language $L \subseteq \Sigma^*$, Angluin's algorithm learns the minimal DFA accepting L in time polynomial in the size n of this DFA and the length m of the longest counterexample returned by the teacher. It asks $\mathcal{O}(n)$ equivalence queries and $\mathcal{O}(mn^2)$ membership queries.*

5.6.2 Actively Learning QDAs as Moore Machines

For actively learning QDAs we take the view of QDAs as Moore machines, as described in Section 5.3.1. We first describe how to adapt Angluin's setting to Moore machines and then explain how to apply this to learning QDAs.

In the context of actively learning Moore machines, the target concept is a Moore machine computable function $f\colon \Sigma^* \to \Gamma$. Note that we obtain Angluin's original setting for learning regular languages by letting $\Gamma = \{0, 1\}$.

Given A Moore machine computable function $f\colon \Sigma^* \to \Gamma$, a teacher for f answers queries as follows.

Membership query On a membership query with a word $u \in \Sigma^*$, the teacher replies the classification $f(u)$.

Equivalence query On an equivalence query with a Moore machine $\mathcal{M}$, the teacher checks whether $f_\mathcal{M} = f$ is satisfied. Is this the case, he returns "yes". If this is not the case, he returns a counterexample $u \in \Sigma^*$ with $f_\mathcal{M}(u) \neq f(u)$.

Note that the learner and the teacher do not need to agree a priori on the output alphabet since the learner can obtain this knowledge through membership queries.

One can, in a straight forward manner, adapt Angluin's algorithm—in fact any observation table-based learning algorithms, such as Rivest and Schapire's algorithm [RS93]—to learn Moore machines. The idea is to lift the Myhill-Nerode congruence to Moore machine computable mappings $f\colon \Sigma^* \to \Gamma$ by defining

$$u \sim_f v \text{ if and only if } \forall w \in \Sigma^*\colon f(uw) = f(vw),$$

where $u, v \in \Sigma^*$. Then, it is indeed enough to adapt the mapping T of an observation table to $T\colon (R \cup R \cdot \Sigma) \cdot S \to \Gamma$ and the way conjectures are generated. For the latter, we do no longer produce a DFA as a conjecture but a Moore machine whose output is defined by the function value $f(u)$ of the representatives $u \in R$. Chen et al. [CFC+09] demonstrate this adaptation for the case $|\Gamma| = 3$.

In analogy to Angluin's algorithm (see Theorem 5.6.1), an algorithm adapted this way learns the unique minimal Moore machine for the target function in time polynomial in this minimal Moore machine and the length of the longest counterexample returned by the teacher. Thus, we obtain the following remark.

Remark 5.6.2. *Given a teacher for a Moore machine computable function that can answer membership and equivalence queries, the unique minimal Moore machine for this function can be learned in time polynomial in the size of this minimal Moore machine and the length of the longest counterexample returned by the teacher.*

We can now simply apply this setting to QDAs viewed as Moore machines. Reformulating the setting for this specific case, we assume that the teacher has access to a QDA-acceptable language $L_{for} \subseteq \Pi^* \times F$ of formula words and answers queries as follows.

Membership query. On a membership query, the learner provides a symbolic word $w \in \Pi^*$, and the teacher returns the unique formula $\varphi \in F$ with $(w, \varphi) \in L_{for}$. Note that such a formula word is guaranteed to exist since L_{for} is a QDA-acceptable language.

Equivalence query. On an equivalence query with a QDA $\mathcal{A}$, the teacher checks whether $L_f(\mathcal{A}) = L_{for}$ is satisfied. If this is the case, he returns "yes". If this is not the case, then there exists a formula word (w, φ) such that $(w, \varphi) \in L_f(\mathcal{A}) \Leftrightarrow (w, \varphi) \notin L_{for}$ (since both $L_f(\mathcal{A})$ and L_{for} contain a formula word of the form (w', φ') for every $w' \in \Pi^*$), and the teacher returns w as counterexample.

Such a teacher for QDAs answers queries in the same manner as a teacher for Moore machines, hence we have reduced the learning of QDAs to learning of Moore machines (using the correspondence from Observation 5.3.1 in Section 5.3.1). This allows us to adapt off-the-shelf learning algorithms, such as Angluin's or Rivest and Schapire's algorithm, and we immediately obtain the following result.

Theorem 5.6.3. *Given a teacher for a QDA-acceptable language of formula words that can answer membership and equivalence queries, the unique minimal QDA for this language can be learned in time polynomial in the size of this minimal QDA and the length of the longest counterexample returned by the teacher.*

5.6.3 Learning Quantified Invariants from Dynamic runs

We apply the active learning algorithm for QDAs, described above, in a passive learning framework in order to learn quantified invariants over lists and arrays from a finite set of samples S obtained from dynamic test runs. In this section, we present the implementation details and the experimental results of our evaluation.

Implementing the Teacher:

In an active learning algorithm, the learner can query the teacher for membership and equivalence queries. In order to build a passive learning algorithm from a sample S, we build a teacher, who will use S to answer the questions of the learner, ensuring that the learned set contains S.

The teacher knows S and wants the learner to construct a small automaton that includes S; however, the teacher does not have a particular language of data words in mind, and hence cannot answer questions precisely. We build a

teacher who answers queries as follows: On a membership query for a word w, the teacher checks whether w belongs to S and returns the corresponding data formula. The teacher has no knowledge about the membership for words which were not realized in test runs, and she rejects these. She also does not know whether the formula she computes on words that get manifest can be weaker; but she insists on that formula. By doing these, the teacher errs on the side of keeping the invariant semantically small. On an equivalence query, the teacher just checks that the set of samples S is contained in the conjectured invariant. If not, the teacher returns a counter-example from S.

Note that the passive learning algorithm hence guarantees that the automaton learned will be a superset of S, and the running time of the algorithm is guaranteed to be polynomial in the size of the learned automaton. We show the efficacy of this passive learning algorithm using experimental evidence later in this section.

Implementation of a Passive Learner of Invariants:

We first take a program and using a test suite, extract the set of concrete data structures that get manifest at loop-headers (for learning loop invariants) and at the beginning and end of functions (for learning pre/post-conditions). The test suite is generated by enumerating all possible arrays/lists of a small bounded length, and with data-values in them from a small bounded domain. We then convert the data structures into a set of formula words, as described below, to get the set S on which we perform passive learning.

We first fix the formula lattice $\mathcal{F}$ over data formulas to be the Cartesian lattice of atomic formulas over relations $\{=, <, \leq\}$. This is sufficient to capture the invariants of many interesting programs such as sorting routines, searching a list, in-place reversal of sorted lists, etc. Using lattice $\mathcal{F}$, for every program configuration which is realized in some test run, we generate a formula word for every valuation of the universal variables over the program structures. We represent these formula words as a mapping from the symbolic word, encoding the structure, to a data formula in the lattice $\mathcal{F}$. If different inputs realize the same structure but with different data formulas, we associate the symbolic word with the join of the two formulas.

Implementing the Learner:

We used the LIBALF library [BKK+10] as an implementation of the active learning algorithm [Ang87a]. We adapted its implementation to our setting by modeling QDAs as Moore machines. If the learned QDA is not elastic, we elastify it as described in Section 5.4. The result is then converted to a quantified formula over STRAND or the APF (see Section 5.5.2) and we check if the learned invariant is adequate and inductive. Due to the unavailability of an implementation of the STRAND decision procedure, we checked the inductiveness of the invariants learned over list data-structures by manually inspecting the learned QDAs. For programs over arrays, we checked the inductiveness of the learned invariants by manually generating the verification conditions and validating them using the Z3 solver [dMB08] . In the case of arrays, the APF formula that corresponds to a QDA and presented in Section 5.5 over-approximates the semantics of the EQDA. To obtain better results in the implementation, we used a more precise formula in which $\varphi_{\neg sp}$ is replaced by the formula $[(0 \leq y_1 < \ldots < y_k < size) \wedge \neg(\bigvee_{\pi \in P_{\mathcal{A}}} \psi_\pi)] \rightarrow false$. Although this formula does not fall in the APF, the constraint solver was able to handle it in our experiments.

Experimental Results:

We evaluate our approach on a suite of programs for learning invariants and preconditions. Our experimental results are tabulated in Table 5.1 [3]. For every program, we report the number of lines of C code, the number of test inputs and the time (T_{teach}) taken to build the teacher from the samples collected along these test runs. We next report the number of equivalence and membership queries answered by the teacher in the active learning algorithm, the size of the final elastic automata in terms of the number of states, whether the learned QDA required any elastification or not and, finally, the time (T_{learn}) taken to learn the QDA.

The first part of the table presents results for programs manipulating arrays like finding a key in an array, copying and comparing two arrays and simple sorting algorithms over arrays. The *inner* and *outer* suffix in insertion and

[3]The benchmark suite and the source code of our implementation is available at `http://www.cs.uiuc.edu/~madhu/cav13/`

Program	LOC	#Test inputs	T_{teach} (s)	# Eq.	# Mem.	# states	Elasti ified?	T_{learn} (s)
Learning Loop Invariants								
array-find	25	310	0.05	2	121	8	no	0.00
array-copy	25	7380	1.75	2	146	10	no	0.00
array-compare	25	7380	0.51	2	146	10	no	0.00
insertion-sort-outer	30	363	0.19	3	305	11	no	0.00
insertion-sort-innner	30	363	0.30	7	2893	23	yes	0.01
selection-sort-outer	40	363	0.18	3	306	11	no	0.01
selection-sort-inner	40	363	0.55	9	6638	40	yes	0.05
list-sorted-find	20	111	0.04	6	1683	15	yes	0.01
list-sorted-insert	30	111	0.04	3	1096	20	no	0.01
list-init	20	310	0.07	5	879	10	yes	0.01
list-max	25	363	0.08	7	1608	14	yes	0.00
list-sorted-merge	60	5004	10.50	7	5775	42	no	0.06
list-partition	70	16395	11.40	10	11807	38	yes	0.11
list-sorted-reverse	25	27	0.02	2	439	18	no	0.00
list-bubble-sort	40	363	0.19	3	447	12	no	0.01
list-fold-split	35	1815	0.21	2	287	14	no	0.00
list-quick-sort	100	363	0.03	1	37	5	no	0.00
list-init-complex	80	363	0.05	1	57	6	no	0.01
lookup_prev	25	111	0.04	3	1096	20	no	0.01
add_cachepage	40	716	0.19	2	500	14	no	0.01
Glib sort (merge)	55	363	0.04	1	37	5	no	0.00
Glib insert_sorted	50	111	0.04	2	530	15	no	0.01
devres	25	372	0.06	2	121	8	no	0.00
rm_pkey	30	372	0.06	2	121	8	no	0.00
GNU Coreutils sort	2500	1 File	0.00	17	4996	5	yes	0.07
Learning Method Preconditions								
list-sorted-find	20	111	0.01	1	37	5	no	0.00
list-init	20	310	0.02	1	26	4	no	0.00
list-sorted-merge	60	329	0.06	3	683	19	no	0.01

Table 5.1: Experimental Results.

selection sort corresponds to learning loop-invariants for the inner and the outer loops in those sorting algorithms. In the second part of the table, we present results for programs that manipulate lists and includes programs to find a key in a sorted list, insert a key in a sorted list such that the resulting list is sorted, initialize all nodes in a list with the value of a key, return the maximum data value in a list, merge two disjoint sorted lists such that the resulting list is also sorted, partition a list into two lists such that one list consists of elements that satisfy a given predicate and the other list consists of nodes that do not, and an in-place reversal of a sorted list where we check

whether the output list is reverse-sorted. The programs bubble-sort, fold-split and quick-sort are taken from [BDES12]. The program *list-init-complex* sorts an input array using heap-sort and then initializes a list with the contents of this sorted array. Since heap-sort is a complex algorithm that views an array as a binary tree, none of the current automatic white-box techniques for invariant synthesis can handle such complex programs. However, our learning approach being black-box, we are able to learn the correct invariant, which is that the list is sorted. Similarly, synthesizing post-condition annotations for recursive procedures like merge-sort and quick-sort is in general difficult for white-box techniques, like *interpolation*, which require a post-condition. Further more, many white-box tools based on *interpolation*, such as SAFARI [ABG+12], cannot handle list-structures, and also cannot handle array-based programs with *quantified* preconditions, which precludes verifying the array variants of programs like *list-sorted-find*, *list-sorted-insert*, etc., which we can handle.

In the third part of the table we present results for verifying methods or code fragments picked from real-world programs. The methods *lookup_prev* and *add_cachepage* are from the module cachePage in ExpressOS, which is a verified-for-security OS platform for mobile applications [MPX+13]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. Next, the method *sort* is a merge sort implementation and *insert_sorted* is a method for inserting a key into a sorted list. Both these methods are from the Glib library, which is a low-level C library that forms the basis of the GTK+ toolkit and the GNOME environment. The methods *devres* and *rm_pkey* are methods adapted from the Linux kernel and an Infiniband device driver, both mentioned in [KJD+10]. Finally, we learn the sortedness property (with respect to the method *compare* that compares two lines) of the method *sortlines* which lies at the heart of the GNU core utility to sort a file. The time taken by our technique to learn an invariant, being black-box, largely depends on the complexity of the property and not the size of the code, as is evident from the successful application of our technique to this large program. In this particular case, we ran the sort utility on an input text file which called the method sortlines multiple times with different array inputs; formula words obtained from these concrete array configurations, as described earlier in this section, form the sample S that the teacher uses to learn an invariant. We also used our learning approach for learning method preconditions, given a test suite; the results for those

experiments are presented in the fourth part of the table. Several methods in our collection of programs have the same method preconditions such as the input argument points to a list or that the input argument points to a sorted list; we only report results in the table for three methods that have different preconditions.

All experiments were completed on an Intel Core i5 CPU at 2.4GHz with 6GB of RAM. For all examples, our prototype implementation learns an adequate invariant really fast. Though the learned QDA might not be the smallest automaton representing the samples S (because of the inaccuracies of the teacher), in practice we find that they are reasonably small (fewer than 50 states). Moreover, we verified that the learned invariants were adequate for proving the programs correct by generating verification conditions and validating them using an SMT solver (these verified in less than 1s). It is possible that SMT solvers can sometimes even handle non-elastic invariants and VCs; however, in our experiments, the Z3 SMT solver we used was not able to handle such formulas without giving extra triggers, thus suggesting the necessity of the elastification of QDAs. It is important to note that the learned invariants might not always be inductive, even though this situation did not arise in our experiments. Exploring automated test generation techniques such as Klee [CDE08] to create a more exhaustive test suite that prevents this situation from arising is an interesting direction for future work. In the current setting, if the invariant learned is not inductive or is inadequate we try to learn from an improved test suite by running the program on more test inputs. ICE learning [GLMN14], as we described in Chapter 2, is an active learning model in which the teacher answers equivalence queries only and refutes the current invariant hypothesis, if it is not adequate or inductive, by adding a positive, negative or an implication counter-example. The ICE learning algorithms for QDAs are more robust that the one presented here and ensure that the invariants learned are adequate and inductive (covered in Section 5.7 [GLMN14]).

Learnt invariants are complex in some programs; for example, Figure 5.5 contains a graphical depiction of the invariant EQDA we learned for the program *list-sorted-find*. If we read the rightmost simple path in the EQDA from state q_0 to q_1 to state q_{14}, and then to q_3 and q_9, it handles the case when $head = cur \neq nil$ and $head \rightarrow^+ y_1$ and $y_1 \rightarrow^+ y_2$ and the EQDA asserts that the data at location pointed to by y_1 is less than or equal to the data at

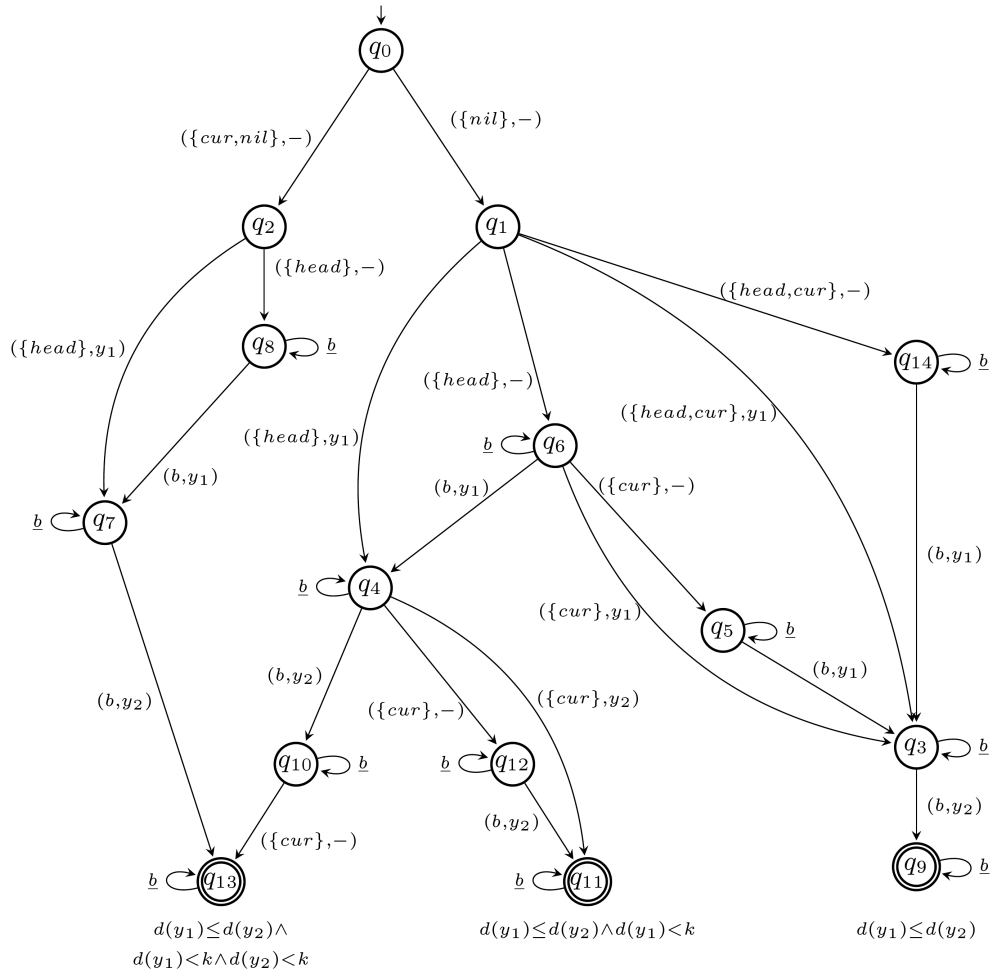


Figure 5.5: The learned EQDA that corresponds to the loop invariant of the program *list-sorted-find*.

y_2. In totality, the EQDA corresponds to the following formula:
$head \neq nil \wedge (\forall y_1 y_2.head \rightarrow^* y_1 \rightarrow^* y_2 \Rightarrow d(y_1) \leq d(y_2)) \wedge ((cur = nil \wedge \forall y_1.head \rightarrow^* y_1 \Rightarrow d(y_1) < k) \vee (head \rightarrow^* cur \wedge \forall y_1.head \rightarrow^* y_1 \rightarrow^+ cur \Rightarrow d(y_1) < k))$.

As opposed to the automata-based passive algorithm for learning likely invariants we have propose in this section, Daikon [ECGN00] is a very mature work, which also learns likely invariants over a set of atomic formulas, by enumerating all formulas and checking which ones satisfy the samples and where scalability is achieved in practice using several heuristics that reduce the enumeration space which is doubly-exponential. However, for quantified invariants over data-structures, such heuristics are not very effective, and Daikon often restricts learning only to formulas of very restricted syntax, like

formulas with a single atomic guard, etc. In our experiments, Daikon was, for instance, not able to learn an adequate loop invariant for the selection sort algorithm.

5.7 ICE Learning QDAs

In this section, we develop an ICE learning algorithm for learning QDAs and, together with a verification oracle as a teacher, deploy the learning algorithm for robustly synthesizing quantified invariants over linear data structures. Recall that neither active learning nor passive learning (covered in Section 5.6) are robust learning frameworks for synthesizing invariants, since there is no way for the teacher to ensure that the learned invariants are inductive. As opposed to these models– ICE learning, which proposes an active learning framework using positive, negative and implication counterexamples, with correctness queries only, is a robust model for synthesizing invariants (see Chapter 2).

A lower bound for ICE learning of EQDAs:

The goal of this section is to develop an iterative ICE learner for concepts represented by EQDAs. We start by a result showing that we cannot hope for a polynomial time iterative ICE learner, when the set of pointers and quantified variables is unbounded.

Theorem 5.7.1. *There is no polynomial time iterative ICE learner for EQDAs, when the alphabet size is unbounded.*

The theorem can be proven by adapting a lower-bound result by Angluin, from [Ang90], namely that there is no polynomial time learning algorithm for DFAs that only uses equivalence queries (no membership queries). The gist of the proof is as follows. One constructs a family $\mathcal{L}_n$ of languages, where each language in $\mathcal{L}_n$ is defined by a DFA of quadratic size in n, with the following property. For each learner that runs in polynomial time (meaning that it uses polynomially many rounds in the size of the target concept and each round only takes polynomial time in the size of the current sample), there is an n such that the teacher can answer the equivalence queries in such a way that

after the polynomial number of rounds available to the learner, there is more than one language left in $\mathcal{L}_n$ that is still consistent with the answers of the teacher. This means that the learner cannot, in general, identify each target concept from $\mathcal{L}_n$.

This proof can be adapted to EQDAs (using just the formulas *true* and *false*) because the DFAs used to define the classes $\mathcal{L}_n$ are acyclic. However, for each class $\mathcal{L}_n$ a different alphabet is needed because for a fixed alphabet there are only finitely many EQDAs.

This shows that there is no hope of obtaining an iterative ICE learner for EQDAs (or even QDAs) in the style of the well-known L^* algorithm of Angluin, which learns DFAs in polynomial time using equivalence and membership queries. Recall that membership queries cannot be answered honestly by an ICE teacher and thus we only have equivalence (or correctness) queries. Note that in Section 5.6 [GLMN13] a learning algorithm for (E)QDAs based on Angluin's L^* is developed. However, this algorithm uses a *dishonest teacher* that answers queries in an arbitrarily chosen way if it does not know the answer. The goal of this section is to develop a robust setting in which the teacher can answer all queries honestly.

Adapting the heuristic RPNI algorithm for EQDAs:

Though we cannot hope for a polynomial time iterative ICE learner, we develop a (non-iterative) ICE learner that constructs an EQDA from a given sample in polynomial time. In the iterative setting this yields a learner for which each round is polynomial, while the number of rounds is not polynomial, in general. For the case of learning DFAs from samples of positive and negative examples, such a heuristic is the passive RPNI algorithm [OG92]. It takes as input a sample (E, C), where E is a set of positive example words and C is a set of negative counterexample words, and constructs a DFA consistent with (E, C), that is, accepting all words in E and rejecting all words in C. Our ICE learning algorithm is adapted from the classical RPNI passive learning algorithm [OG92].

RPNI can be viewed as an instance of an abstract state merging algorithm that is sketched as Algorithm 3. In this general setting, the algorithm takes a finite collection $\mathcal{S}$ of data, called a *sample*, as input and produces a *Moore*

Algorithm 3: Generic State Merging algorithm.

Input: A sample $\mathcal{S}$
Output: A Moore machine $\mathcal{A}$ that passes `test`$(\mathcal{A})$

1 $\mathcal{A}_{\text{init}} = (Q, \Sigma, \Gamma, q_0, \delta, f) \leftarrow$ `init`$(\mathcal{S})$;
2 $(q_0, \ldots, q_n) \leftarrow$ `order`(Q);
3 $\sim_0 \leftarrow \{(q, q) \mid q \in Q\}$;
4 **for** $i = 1, \ldots, n$ **do**
5 **if** $q_i \not\sim_{i-1} q_j$ *for all* $j \in \{0, \ldots, i-1\}$ **then**
6 $j \leftarrow 0$;
7 **repeat**
8 Let $\sim$ be the smallest congruence that contains $\sim_{i-1}$ and the pair (q_i, q_j);
9 $j \leftarrow j + 1$;
10 **until** `test`$(\mathcal{A}_{init}/_{\sim})$;
11 $\sim_i \leftarrow \sim$;
12 **else**
13 $\sim_i \leftarrow \sim_{i-1}$;
14 **end**
15 **return** $\mathcal{A}_{\text{init}}/_{\sim_n}$;

machine (i.e., a DFA with output) that is consistent with the sample (we define this formally later). In the case of classical RPNI, $\mathcal{S} = (E, C)$ consists of two finite sets of positive and negative counterexample words, the resulting Moore machine is interpreted as a DFA, and we require that all words in E be accepted whereas all words in C be rejected by the DFA.

Algorithm 3 proceeds in two consecutive phases. In Phase 1 (Lines 1 and 2), it calls `init`$(\mathcal{S})$ to construct an initial Moore machine $\mathcal{A}_{\text{init}}$ from $\mathcal{S}$ that satisfies the sample (assuming that this is possible). Then, it picks a total order $q_0 < \ldots < q_n$ on the states of $\mathcal{A}_{\text{init}}$, which determines the order in which the states are to be merged in the subsequent phase. The actual state merging then takes place in Phase 2 (Lines 3 to 14). According to the given order, Algorithm 3 tries to merge each state q_i with a "smaller" state q_j (i.e., $j < i$) and calls `test` on the resulting Moore machine to check whether this machine still satisfies the sample; since a merge can cause nondeterminism, it might be necessary to merge further states in order to restore determinism. A merge is kept if the Moore machine passes `test`; otherwise the merge is discarded, guaranteeing that the final Moore machine still satisfies sample. Note that we represent merging of states by means of a congruence relation $\sim \subseteq Q \times Q$ over the states (i.e., $\sim$ is an equivalence relation that is compatible with the

transitions) and the actual merging operation as constructing the quotient Moore machine $\mathcal{A}_{\text{init}}/_{\sim}$ in the usual way. Note that in the case of DFAs, each merge increases the language and thus can be seen as a generalization step in the learning algorithm.

It is not hard to verify that RPNI indeed produces a consistent DFA: it starts with the prefix tree acceptor of E, which is clearly consistent with (E, C) because E and C are disjoint (otherwise there is no DFA consistent with (E, C)), and only keeps merges that do not violate the consistency. Moreover, the resulting DFA is never larger than the prefix acceptor for (E, C) because RPNI exclusively merges states. Note that RPNI does not construct the smallest DFA consistent with the sample because an early merge might prevent later merges which could have produced a smaller automaton.

We are now ready to describe our new ICE learning algorithm for EQDAs that extends the above Algorithm 3, handling both EQDAs and implication samples. Recall that in the data-set based ICE for learning quantified properties (Chapter 2), as in our setting, a sample is of the form $(\hat{E}, \hat{C}, \hat{I})$ where $\hat{E}, \hat{C}$ are sets of sets of valuation words and $\hat{I}$ contains pairs of sets of valuation words. From Section 5.3.1 we know that EQDAs can be viewed as Moore machines that read valuation words and output data formulas. Hence we can adapt the RPNI algorithm to learn EQDAs, as explained below.

For the initialization `init(`$\mathcal{S}$`)` we construct an EQDA whose language is the smallest (w.r.t. inclusion) EQDA-definable language that is consistent with the sample $\mathcal{S}$. To do this, we consider the set of all positive examples, i.e., the set $E := \bigcup \hat{E}$. This is a set of valuation words, from which we strip off the data part, obtaining a set E' of symbolic words only made up of pointers and universally quantified variables. We start with the prefix tree of E' using the prefixes of words in E' as states (as the original RPNI does). The final states are the words in E'. Each such word $w \in E'$ originates from a set of valuation words in E (all the extensions of w by data that result in a valuation word in E). If we denote this set by E_w, then we label the state corresponding to w with the least formula that is satisfied in all valuation words in E_w (recall that the formulas form a lattice). This defines the smallest QDA-definable set that contains all words in E. If this QDA is not consistent with the sample, then either there is no such QDA, or the QDA is not consistent with an implication, that is, for some $(X, Y) \in \hat{I}$ it accepts everything in X but not everything in Y. In this case, we add X and Y to $\hat{E}$ and restart the construction (since

every QDA consistent with the sample must accept all of X and all of Y).

To make this QDA $\mathcal{A}$ elastic, all states that are connected by a $\underline{b}$-transition are merged. This defines the smallest EQDA-definable set that contains all words accepted by $\mathcal{A}$ (see [GLMN13]). Hence, if this EQDA is not consistent with the sample, then either there is no such EQDA, or an implication $(X, Y) \in \hat{I}$ is violated, and we proceed as above by adding X and Y to $\hat{E}$ and restarting the computation. This adapted initialization results in an EQDA whose language is the *smallest EQDA-definable language that is consistent with the sample.*

Once Phase 1 is finished, our algorithm proceeds to Phase 2, in which it successively merges states of $\mathcal{A}_{\text{init}}$, to obtain an EQDA that remains consistent with the sample but has less states. When merging accepting states, the new formula at the combined state is obtained as the lub of the formulas of the original states. Note that merging states of an EQDA preserves the self-loop condition for $\underline{b}$-transitions. Finally, the `test` routine simply checks whether the merged EQDA is consistent with the sample.

It follows that the hypothesis constructed by this adapted version of RPNI is an EQDA that is consistent with the sample. Hence we have described a consistent learner. For a fixed set of pointer variables and universally quantified variables there are only a finite number of EQDAs. Therefore by Lemma 2.2.1 we conclude that the above learning is strongly convergent (though the number of rounds need not be polynomial).

Theorem 5.7.2. *The adaption of the RPNI algorithm for iterative set-based ICE learning of EQDAs strongly converges.*

Experimental Results

We implemented a prototype of the set-based ICE learning algorithm for learning quantified invariants over arrays and lists, that we just described above in this section. Now, we first describe the implementation details of the teacher, followed by the description of the implementation of the RPNI-based learner.

Given a conjectured hypothesis, the role of the teacher is to check whether the conjectured invariant is adequate or not. In our case, the learner conjectures an EQDA as a hypothesis. The teacher first converts the EQDA to

Program	White-Box	Black-Box		
	SAFARI (s)	R	|Q|	**ICE(s)**
copy	0.0	4	8	**0.7**
copy-lt-key	×	5	13	**1.2**
init	0.7	4	8	**0.6**
init-partial	×	8	12	**1.5**
compare	0.1	9	8	**1.3**
find	0.2	9	8	**1.2**
max	0.1	3	8	**0.4**
increment	×	5	8	**0.7**
sorted-find	×	8	17	**5.1**
sorted-insert	×	6	21	**2.0**
sorted-reverse	×	18	17	**9.4**
devres [KJD+10]	0.1	3	8	**0.7**
rm_pkey [KJD+10]	0.3	3	8	**0.7**

Table 5.2: RPNI-based ICE learning for quantified array invariants. R: # rounds of iterative-ICE; $|Q|$: # states in final EQDA. × means a timeout of 5 min.

a quantified formula in the array property fragment (APF) [BMS06] or the decidable STRAND fragment over lists [MPQ11], as described in Section 5.5.2. Then the teacher uses a constraint solver to check if the conjectured EQDA corresponds to an adequate invariant or not. If the answer is no, the teacher finds positive, negative and implication counterexamples over concrete data words that need to be added to the sample of the learner for the next iteration of iterative ICE. However, because of the quantified setting, the sample is defined over sets of valuation words and not data words. Therefore, for every data word, the teacher obtains a set of valuation words and then adds these sets, or pair of sets in the case of implications, to the sample.

The learner is an RPNI-based ICE learner which given a set-based sample $(\hat{E}, \hat{C}, \hat{I})$ conjectures an EQDA that is consistent with the sample. Let us first fix the formula lattice over data formulas to be the Cartesian lattice of atomic formulas over relations $\{=, <, \leq\}$. To check whether a valuation word v is rejected by an EQDA, the learner should just read v and check if its data values satisfy the data formula φ_v that the EQDA outputs on reading v. However, the learner actually implements this check in a slightly different manner. Given a valuation word v, the learner finds the smallest data-formula in the formula lattice which includes the data values in v, and rejects the word only if that formula is unsatisfiable in conjunction with

φ_v. With this criterion of rejecting valuation words, words which should be actually rejected by the EQDA might not be rejected under this new criterion. In terms of the RPNI-based learner which merges states only if the EQDA still rejects all $C \in \hat{C}$, the new rejection criterion leads to fewer states being merged. The new criterion is therefore more conservative and it ensures that the EQDA learned by the learner still remains consistent with the sample. Apart from this modification, the learner is implemented exactly as described in the previous subsection.

To start the learning process, the teacher in the beginning runs the program on a few random input lists/arrays and collects the concrete data words that manifest at the program locations for which we want to synthesize an invariant. Each such data word is converted to a set of valuation words and together they form the set of positive examples $\hat{E}$ in the sample with which the iterative ICE learning is initialized ($\hat{C}$ and $\hat{I}$ are empty to begin with).

We adapted the RPNI algorithm from the LIBALF library [BKK+10] to support the above described set-based ICE learning algorithm. We use Z3 [dMB08] (which supports APF) as the constraint solver in the teacher for checking the adequacy of the quantified array invariants. We evaluated the learning algorithm on several array-based programs (see Table 5.2). Since we did not have an implementation of the decision procedure for the decidable fragment of STRAND for lists, we could not build the teacher for lists and evaluate the learning algorithm over list-manipulating programs.

In Table 5.2 we report for each program the number of rounds taken by the iterative learning algorithm; the number of positive amd negative counterexamples and implications added to the sample of the learner, over all rounds; the number of states in the final EQDA conjectured and the total time taken to learn that EQDA. The program *sorted-find* finds the presence of a key in a sorted array; the program *sorted-insert* reads an array which is sorted from the second position onwards and inserts the first element of the array at its correct position such that the entire array becomes sorted; the program *devres* and *rm_pkey* are methods adapted from the Linux kernel and an Infiniband device driver, both mentioned in [KJD+10].

In section 5.6 we developed an L^* based learning algorithm for learning quantified invariants over arrays and lists. As compared to the ICE algorithm we have presented in this section, the algorithm from Section 5.6 does not use an honest teacher. The teacher does not know an invariant before hand and

therefore answers L* membership queries in an arbitrarily chosen way when it does not know the answer. Secondly, the L*-based algorithm learns from only positive configurations that manifest themselves in test runs. Hence, the invariants synthesized in Section 5.6 are only *likely* invariants (and might not actually be invariants). On the other hand, the learning algorithm we present here is based on the ICE paradigm and uses an honest teacher. The algorithm is robust and guarantees convergence to the actual invariant, regardless of the way the teacher answers equivalence queries. Though our learning algorithm is more complex than the L*-based algorithm, we show through experiments that our algorithm learns an adequate invariant in reasonable time, requiring only a few number of rounds and a small sample size.

We compare our results to Safari [ABG+12], a verification tool based on interpolation in array theories. Safari, in general, cannot handle list programs, and also array programs like *sorted-find* that have quantified pre-conditions. On the others, Safari diverges for some programs, and probably needs manually provided term abstractions to achieve convergence. The results show that our ICE learning algorithm for quantified invariants is effective, in addition to promising polynomial-per-round efficiency, promising invariants that fall in decidable theories, and promising strong convergence.

CHAPTER 6

SYNTHESIZING INVARIANTS FOR LISTS USING ABSTRACT INTERPRETATION

In the previous section we developed learning techniques, including a convergent ICE learning algorithm, for synthesizing quantified invariants over linear data structures such as arrays and lists. As opposed to learning, which has been the main focus in this thesis, in this section, we develop a completely different way of synthesizing quantified invariants for lists using abstract interpretation [CC77]. We propose a new approach to heap analysis through an abstract domain of automata, called *automatic shapes*. Automatic shapes are modeled after a particular version of *quantified data automata on skinny trees* (QSDAs), that allows to define universally quantified properties of programs manipulating acyclic heaps with a single pointer field, including data-structures such singly-linked lists. QSDAs are extensions of quantified data automata, we introduced in Chapter 5, to acyclic heaps. To ensure convergence of the abstract fixed-point computation, we introduce a subclass of QSDAs called elastic QSDAs, which forms an abstract domain. We evaluate our approach on several list manipulating programs and we show that the proposed domain is powerful enough to prove a large class of these programs correct.

Abstract analysis of the heap is hard because abstractions need to represent the heap which is of unbounded size, and must capture both the *structure* of the heap as well as the unbounded *data* stored in the heap. While several data-domains have been investigated for data stored in static variables, the analysis of unbounded structure and unbounded data that a heap contains has been less satisfactory. The primary abstraction that has been investigated is the rich work on *shape analysis* [SRW02]. However, unlike abstractions for data-domains (like intervals, octagons, polyhedra, etc.), shape analysis requires carefully chosen *instrumentation* predicates to be given by the user, and often are particular to the program that is being verified. Shape analysis techniques typically *merge* all nodes that satisfy the same unary predicate,

achieving finiteness of the abstract domain, and interpret the other predicates using a 3-valued (must, must not, may) abstraction. Moreover, these instrumentation predicates often require to be encoded in particular ways (for example, capturing binary predicates as particular kinds of unary predicates) so as to not lose precision.

For instance, consider a sorting algorithm that has an invariant of the form:

$$\forall x, y. \, (\ (x \rightarrow^*_{next} y \wedge y \rightarrow^*_{next} i) \Rightarrow d(x) \leq d(y)\ )$$

which says that the sub-list before pointer i is sorted. In order to achieve a shape-analysis algorithm that discovers this invariant (i.e., captures this invariant precisely during the analysis), we typically need instrumentation predicates such as $p(z) = z \rightarrow^*_{next} i$, $s(x) = \forall y.((x \rightarrow^*_{next} y \wedge y \rightarrow^*_{next} i) \Rightarrow d(x) \leq d(y))$, etc. The predicate $s(x)$ says that the element that is at x is less than or equal to the data stored in every cell between x and i. These instrumentation predicates are clearly too dependent on the precise program and property being verified.

In this chapter, we investigate an abstract domain for heaps that works *without user-defined instrumentation predicates* (except we require that the user fix an abstract domain for data, like octagons, for comparing data elements). Our abstract domain is modeled after a particular kind of automata, called *quantified data automata*, that define, logically, universally quantified properties of heap structures. In this work, we restrict our attention to acyclic heap structures that have only *one pointer field*; our analysis is hence one that can be used to analyze properties of heaps containing lists, with possible aliasing (merging) of them, especially at intermediate stages in the program. One-pointer acyclic heaps can be viewed as *skinny trees* (trees where the number of branching nodes is bounded).

Automata, in general, are classical ways to capture an infinite set of objects using finite means. A class of (regular) skinny trees can hence be represented using tree automata, capturing the structure of the heap. While similar ideas have been explored before in the literature [HHR+11], our main aim is to also represent properties of the *data* stored in the heap, building automata that can express universally quantified properties on lists, in particular those of the form

$$\bigwedge_i \forall \overline{x}. \, (Guard_i(\overline{p}, \overline{x}) \Rightarrow Data_i(d(\overline{p}), d(\overline{x})))$$

where $\overline{p}$ is the set of static pointer variables in the program. The $Guard_i$

formulas express structural constraints on the universally quantified variables and the pointer variables, while the $Data_i$ formulas express properties about the data stored at the nodes pointed to by these pointers. In this work, we investigate an abstract domain that can infer such quantified properties, parameterized by an abstract numerical domain $\mathcal{F}_d$ for the data formulas and by the number of quantified variables $\overline{x}$.

The salient aspect of the automatic shapes that we build is that (a) there is no requirement from the user to define instrumentation predicates for the structural *Guard* formulas; (b) since the abstraction will not be done by merging unary predicates and since the automata can define how data stored at *multiple* locations on the heap are related, there is no need for the user to define carefully crafted unary predicates that relate structure and data (e.g., the unary predicate $s(x)$ defined above that says that the location x is sorted with respect to all successive locations that come after x but before i). Despite this lack of help from the user, we show how our abstract domain can infer properties of a large number of list-manipulating programs adequately to prove interesting quantified properties.

The crux of our approach is to use a class of automata, called quantified data automata on skinny trees (QSDA), to express a class of single-pointer heap structures and the data contained in them. QSDAs read skinny trees with data along with *all* possible valuations of the quantified variables, and for each of them check whether the data stored in these locations (and the locations pointed to by pointer variables in the program) relate in particular ways defined by the abstract data-domain $\mathcal{F}_d$.

We show, for a simple heap-manipulating programming language, that we can define an abstract post operator over the abstract domain of QSDAs. This abstract post preserves the structural aspects of the heap *precisely* (as QSDAs can have an arbitrary number of states to capture the evolution of the program) and that it soundly abstracts the quantified data properties. The abstract post is nontrivial to define and show effective as it requires automata-theoretic operations that need to simultaneously preserve structure as well as data properties; this forms the hardest technical aspect of this work. We thus obtain an effective sound transfer function for QSDAs. However, it turns out that QSDAs are not complete lattices (infinite sets may not have least upper-bounds), and hence do not form an abstract domain to interpret programs. Furthermore, typically, in each iteration, the QSDAs obtained

would grow in the number of states, and it is not easy to find a fixed-point.

Traditionally, in order to handle loops and reach termination, abstract domains require some form of widening. Our notion of widening is founded on the principle that lengths of stretches of the heap that are neither pointed to by program variables nor by the quantified variables (in one particular instantiation of them) must be ignored. We would hence want the automaton to check the same properties of the instantiated heap no matter how long these stretches of locations are. This notion of abstraction is also suggested in the previous chapter work where we have shown that such abstractions lead to *decidability*; in other words, properties of such abstracted automata fall into decidable logical theories [MPQ11, GLMN13]. Assume that the programmer computes a QSDA as an invariant for the program at a particular point, where there is an assertion expressed as a quantified property p over lists (such as "the list pointed to by *head* is sorted"). In order to verify that the abstraction proves the assertion, we will have to check if the language of lists accepted by the QSDA is contained in the language of lists that satisfy the property p. However, this is in general *undecidable*. However, this inclusion problem is decidable if the automata abstracts the lengths of stretches as above. Our aim is hence to *over-approximate* the QSDA into a larger language accepted by a particular kind of data automata, called *elastic* QSDA (EQSDA) that ignores the stretches where variables do not point to, and where "merging" of the pointers do not occur [MPQ11, GLMN13].

This *elastification* will in fact serve as the basis for widening as well, as it turns out that there are only a *finite* number of elastic QSDAs that express structural properties, discounting the data-formulas. Consequently, we can combine the elastification procedure (which over-approximates a QSDA into an elastic QSDA) and widening over the numerical domain for the data in order to obtain widening procedures that can be used to accelerate the computation for loops. In fact, the domain of EQSDAs is an abstract domain and a complete lattice (where infinite sets also have least upper-bounds), and there is a natural abstract interpretation between sets of concrete heap configurations and EQSDAs, where the EQSDAs permit widening procedures. We show a unique elastification theorem that shows that for any QSDA, there is a unique elastic QSDA that over-approximates it. This allows us to utilize the abstract transfer function on QSDAs (which is more precise) on a linear block of statements, and then elastify them to

EQSDAs at join points to have computable fixed-points.

We also show that EQSDA properties over lists can be translated to a decidable fragment of the logic STRAND [MPQ11] over lists, and hence inclusion checking an elastic QSDA with respect to any assertion that is also written using the decidable sublogic of STRAND over lists is decidable. As mentioned above, the notion of QSDAs and elasticity are extensions of the work we presented in Chapter 5, where such notions were developed for *words* (as opposed to trees) and where the automata were used for *learning* invariants from examples and counterexamples.

We implement our abstract domain and transformers and show, using a suite of list-manipulating programs, that our abstract interpretation is able to prove the naturally required (universally-quantified) properties of these programs. While several earlier approaches (such as shape analysis) can tackle the correctness of these programs as well, our abstract analysis is able to do this *without* requiring program-specific help from the user (for example, in terms of instrumentation predicates in shape analysis [SRW02], and in terms of guard patterns in the work by Bouajjani et al [BDES11]).

6.1 Programs Manipulating Heap and Data

We consider sequential programs manipulating acyclic singly-linked data structures. A *heap structure* is composed of locations (also called nodes). Each location is endowed with a *pointer field* `next` that points to another location or it is undefined, and a *data field* called `data` that takes values from a potentially infinite domain $\mathbb{D}$ (i.e. the set of integers). For simplicity we assume a special location, called *dirty*, that models an un-allocated memory space. We assume that the `next` pointer field of *dirty* is undefined. Besides the heap structure, a program also has a finite number of *pointer variables* each pointing to a location in the heap structure, and a finite number of *data variables* over $\mathbb{D}$. In our programming language we do not have procedure calls, and we handle non-recursive procedures calls by inlining the code at call points. In the rest of the section we formally define the syntax and semantics of these programs.

$\langle prgm \rangle$ ::= pointer $p_1, \ldots, p_k$; data $d_1, \ldots, d_\ell$; $\langle pc\_stmt \rangle^+$

$\langle pc\_stmt \rangle$ ::= $pc : \langle stmt \rangle$;

$\langle stmt \rangle$ ::= $\langle ctrl\_stmt \rangle \mid \langle heap\_stmt \rangle$

$\langle ctrl\_stmt \rangle$ ::= d_i :=<data_expr> | `skip` | `assume`($\langle pred \rangle$)
| `if` $\langle pred \rangle$ `then` $\langle pc\_stmt \rangle^+$ `else` $\langle pc\_stmt \rangle^+$ `fi`
| `while` $\langle pred \rangle$ `do` $\langle pc\_stmt \rangle^+$ `od`

$\langle heap\_stmt \rangle$ ::= `new` $p_i \mid p_i :=$ `nil` $\mid p_i := p_j$
| $p_i := p_j \rightarrow$ `next` $\mid p_i \rightarrow$ `next` := `nil` $\mid p_i \rightarrow$ `next` $:= p_j$
| $p_i \rightarrow$ `data` $:= \langle data\_expr \rangle$

Figure 6.1: Simple programming language.

Syntax. The syntax of programs is defined by the grammar of Figure 6.1. A program starts with the declaration of pointer variables followed by a declaration of data variables. Data variables range over a potentially infinite data domain $\mathbb{D}$. We assume a language of data expressions built from data variables and terms of the form $p_i \rightarrow$ `data` using operations over $\mathbb{D}$. Predicates in our language are either data predicates built from predicates over $\mathbb{D}$ or structural predicates concerning the heap built from atoms of the form $p_i == p_j$, $p_i ==$ `nil`, $p_i \rightarrow$ `next` $== p_j$ and $p_i \rightarrow$ `next` $==$ `nil`, for some $i, j \in [1, k]$. Thereafter, there is a non-empty list of labelled statements of the form $pc : \langle stmt \rangle$ where pc is the *program counter* and $\langle stmt \rangle$ defines a language of either C-like statements or statements which modify the heap. We do not have an explicit statement to *free* locations of the heap: when a location is no longer reachable from any location pointed by a pointer variable we assume that it automatically disappears from the memory. For a program P, we denote with PC the set of all program counters of P statements. Figure 6.2(a) shows the code for program *sorted list-insert* which is a running example in this chapter. The program inserts a *key* into the sorted list pointed to by variable *head*.

Semantics. A *configuration* C of a program P with set of pointer variables PV and data variables DV is a tuple $\langle pc, H, pval, dval \rangle$ where

- $pc \in PC$ is the program counter of the next statement to be executed;
- H is a *heap configuration* represented by a tuple (Loc, `next`, `data`) where (1) Loc is a finite set of heap locations containing special elements called

nil and *dirty*, (2) $\texttt{next} : \mathit{Loc} \mapsto \mathit{Loc}$ is a partial map defining an edge relation among locations such that the graph $(\mathit{Loc}, \texttt{next})$ is acyclic, and (3) $\texttt{data} : \mathit{Loc} \mapsto \mathbb{D}$ is a map that associates each *non-nil* and *non-dirty* location of *Loc* with a data value in $\mathbb{D}$;

- $\mathit{pval} : \widehat{PV} \rightarrow \mathit{Loc}$, where $\widehat{PV} = PV \cup \{\texttt{nil}, \texttt{dirty}\}$, associates each pointer variable of P with a location in H. If $\mathit{pval}(p) = v$ we say that node v is *pointed* by variable p. Furthermore, each node in *Loc* is reachable from a node pointed by a variable in PV. There is no outgoing (`next`) edge from location *dirty* and there is a `next` edge from the location pointed by `nil` to *dirty* (henceforth we use use PV everywhere instead of the $\widehat{PV}$);

- $\mathit{dval} : DV \rightarrow \mathbb{D}$ is a valuation map for the data variables.

Figure 6.2(b) graphically shows a program configuration which is reachable at program counter 8 of the program in Figure 6.2(a) (as explained later we encode the data variable *key* as a pointer variable in the heap configuration). The *transition relation* of a program P, denoted $\xrightarrow{\mathit{stmt}}_P$ for each statement *stmt* of P, is defined as usual. The control-flow statements update the program counter, possibly depending on a predicate (condition). The assignment statements update the variable valuation or the heap structure other than moving to the next program counter. Let us define the concrete transformer $F^\natural = \lambda\mathcal{C}.\{\mathcal{C}' \mid \mathcal{C} \xrightarrow{\mathit{stmt}}_P \mathcal{C}'\}$. The concrete semantics of a program is given as the least fixed point of a set of equations of the form $\psi = F^\natural(\psi)$.

To simplify the presentation of our work, we assume that our programs do not have data variables. This restriction, indeed, does not reduce their expressiveness: we can always transform a program P into an *equivalent* program P' by translating each data variable d into a pointer variable that will now point to a fresh node in the heap structure, in which the value d is now encoded by $d \rightarrow \texttt{data}$. The node pointed by d is not pointed by any other pointer, further, $d \rightarrow \texttt{next}$ points to *dirty*. Obviously, wherever d is used in P will now be replaced by $d \rightarrow \texttt{data}$ in P'.

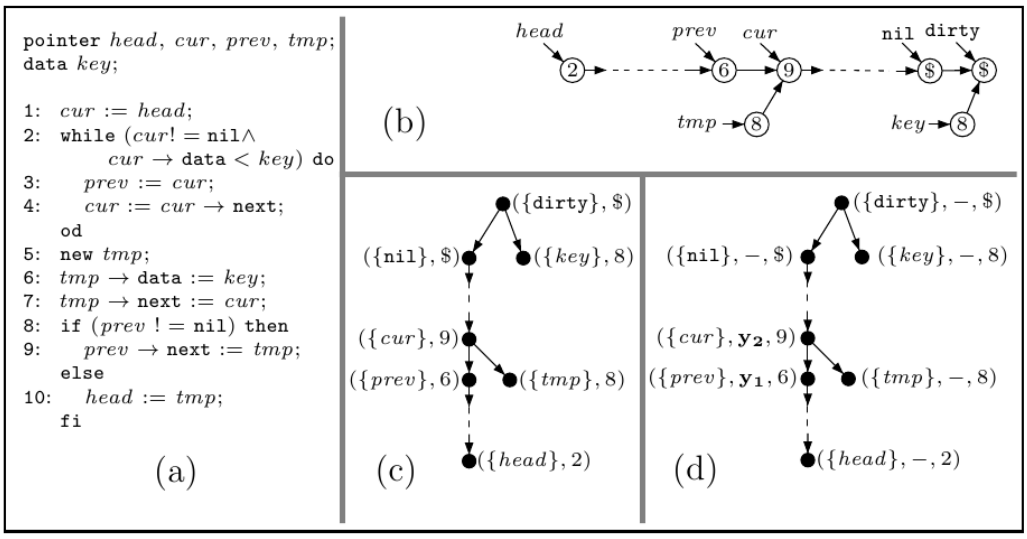


Figure 6.2: **(a)** *sorted list-insert* program P; **(b)** shows a P configuration at program counter 8; **(c)** is the heap skinny-tree associated to (b); **(d)** is a valuation tree of (c).

6.2 Quantified Skinny-Tree Data Automata

In this section we define *quantified skinny-tree data automata* (QSDAs, for short), an accepting mechanism of program configurations (represented as special labelled trees) on which we can express properties of the form $\bigwedge_i \forall y_1, \ldots, y_\ell.\ Guard_i \Rightarrow Data_i$, where variables y_i range over the set of locations of the heap, $Guard_i$ represent quantifier-free structural constraints among the pointer variables and the universally quantified variables y_i, and $Data_i$ (called *data formulas*) are quantifier-free formulas that refer to the data stored at the locations pointed either by the universal variables y_i or the pointer variables, and compare them using operators over the data domain. In the rest of this section, we first define *heap skinny-trees* which are a suitable labelled tree encodings for program configurations; we then define *valuation trees* which are heap skinny-trees by adding to the labels an instantiation of the universal variables. *Quantified skinny-tree data automata* is a mechanism designed to recognize valuation trees. The *language* of a QSDA is the set of all heap skinny-trees such that all valuation trees deriving from them are accepted by the QSDA. Intuitively, the heap skinny-trees in the language defined by the QSDA are all the program configurations that verify the formula $\bigwedge_i \forall y_1, \ldots, y_\ell . Guard_i \Rightarrow Data_i$.

Let T be a tree. A node u of T is *branching* whenever u has more than one

child. For a given natural number k, T is *k-skinny* if it contains at most k branching nodes.

Heap Skinny-Trees. Let PV be the set of pointer variables of a program P and $\Sigma = 2^{PV}$ (let us denote the empty set with a blank symbol b). We associate with each P configuration $C = \langle pc, H, pval, dval \rangle$ with $H = (Loc, \texttt{next}, \texttt{data})$, the $(\Sigma \times \mathbb{D})$-labelled graph $\mathcal{H} = (T, \lambda)$ whose nodes are those of Loc, and where (u, v) is an edge of T iff $\texttt{next}(v) = u$ (essentially we reverse all `next` edges). From the definition of program configurations, since all locations are required to be reachable from some program variable, it is easy to see that T is a k-skinny tree where $k = |PV|$. The labeling function $\lambda : Loc \to (\Sigma \times \mathbb{D})$ is defined as follows: for every $u \in Loc$, $\lambda(u) = (S, d)$ where S is the set of all pointer variables p such that $pval(p) = u$, and $d = \texttt{data}(u)$. We call $\mathcal{H}$ the *heap skinny-tree* of C.

Heap skinny-trees are formally defined as follows.

Definition 6.2.1 (Heap Skinny-Trees)**.** *A* heap skinny-tree *over a set of pointer variables PV and data domain $\mathbb{D}$, is a $(\Sigma \times \mathbb{D})$-labelled k-skinny tree (T, λ) with $\Sigma = 2^{PV}$ and $k = |PV|$, such that:*

- *for every leaf v of T, $\lambda(v) = (S, d)$ where $S \neq \emptyset$;*
- *for every $p \in PV \cup \{nil\}$, there is a unique node v of T such that $\lambda(v) = (S, d)$ with $p \in S$ and some $d \in \mathbb{D}$;*
- *for a node v of T such that $\lambda(v) = (S, d)$, if $\texttt{nil} \in S$ then v is one of the children of the root of T; if v is the root of T then $S = \{\texttt{dirty}\}$.* □

Figure 6.2(c) shows the heap skinny-tree corresponding to the program configuration of Figure 6.2(b). Note that though the program handles a singly linked list, in the intermediate operations we can get trees. However they are special trees with bounded branching. This example illustrates that program configurations of list manipulating programs naturally correspond to heap skinny-trees. It also motivates why we need to extend automata over words introduced in Chapter 5 to quantified data automata over skinny-trees. We now define valuation trees.

Valuation Trees. Let us fix a finite set of *universal* variables Y. A *valuation tree* over Y of a heap skinny-tree $\mathcal{H}$ is a $(\Sigma \times (Y \cup \{-\}) \times \mathbb{D})$-labelled tree obtained from $\mathcal{H}$ by adding an element from the set $Y \cup \{-\}$ to the label, in which every element in Y occurs exactly once in the tree. We use the symbol '$-$' at a node v if there is no variable from Y labelling v. A valuation tree corresponding to the heap skinny-tree of Figure 6.2(c) is shown in Figure 6.2(d).

Quantified skinny-tree data automata are a mechanism to accept skinny-trees. To express properties on the data present in the nodes of the skinny-trees, QSDAs are parameterized by a set of data formulas F over $\mathbb{D}$ which form a complete-lattice $\mathcal{F} = (F, \sqsubseteq_{\mathcal{F}}, \sqcup_{\mathcal{F}}, \sqcap_{\mathcal{F}}, \mathit{false}, \mathit{true})$ where $\sqsubseteq_{\mathcal{F}}$ is the partial-order on the data-formulas, $\sqcup_{\mathcal{F}}$ and $\sqcap_{\mathcal{F}}$ are the least upper bound and the greatest lower bound and *false* and *true* are formulas required to be in F and correspond to the bottom and the top elements of the lattice, respectively. Also, we assume that whenever $\alpha \sqsubseteq_{\mathcal{F}} \beta$ then $\alpha \Rightarrow \beta$. Furthermore, we assume that any pair of formulas in F are non-equivalent. For a logical domain as ours, this can be achieved by having a canonical representative for every set of equivalent formulas. Let us now formally define QSDAs.

Definition 6.2.2 (Quantified Skinny-Tree Data Automata)**.** *A* quantified skinny-tree data automaton *(*QSDA*) over a set of pointer variables PV (with $|PV| = k$), a data domain $\mathbb{D}$, a set of universal variables Y, and a formula lattice $\mathcal{F}$, is a tuple $\mathcal{A} = (Q, \Pi, \Delta, \mathcal{T}, f)$ where:*

- *Q is a finite set of states;*
- *$\Pi = \Sigma \times \widehat{Y}$ is the alphabet where $\Sigma = 2^{PV}$ and $\widehat{Y} = Y \cup \{-\}$;*
- *$\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ where, for every $i \in [1, k]$, $\Delta_i : (Q^i \times \Pi) \mapsto Q$ is a partial function and defines a (deterministic) transition relation;*
- *$\mathcal{T} : Q \to 2^{PV \cup Y}$ is the type associated with every state $q \in Q$;*
- *$f : Q \to \mathcal{F}$ is a final-evaluation.* □

A valuation tree (T, λ) over Y of a program P, where N is the set of nodes of T, is *recognized* by a QSDA $\mathcal{A}$ if there exists a node-labelling map $\rho : N \mapsto Q$ that associates each node of T with a state in Q such that for each node t of T with $\lambda(t) = (S, y, d)$ the following holds (here $\lambda'(t) = (S, y)$ is obtained by projecting out the data values from $\lambda(t)$):

- if t is a leaf then $\Delta_0(\lambda'(t)) = \rho(t)$ and $\mathcal{T}(\rho(t)) = S \cup \{y\} \setminus \{-\}$.
- if t is an internal node, with sequence of children $t_1, t_2, \ldots, t_i$ then
  - $\Delta_i\,(\,(\rho(t_1), \ldots, \rho(t_i)),\, \lambda'(t)\,) = \rho(t)$;
  - $\mathcal{T}(\rho(t)) = S \cup \{y\} \setminus \{-\} \cup \left(\bigcup_{j \in [1,i]} \mathcal{T}(\rho(t_j))\right)$.
- if t is the root then the formula $f(\rho(t))$, obtained by replacing all occurrences of terms $y \to \texttt{data}$ and $p \to \texttt{data}$ with their corresponding data values in the valuation tree, holds true.

A QSDA can be thought as a *register* automaton that reads a valuation tree in a bottom-up fashion and stores the data at the positions evaluated for Y and locations pointed by elements in PV, and checks whether the formula associated to the state at the root holds true by instantiating the data values in the formula with those stored in the registers. Furthermore, the role of map $\mathcal{T}$ is that of enforcing that each element in $PV \cup Y$ occurs exactly once in the valuation tree.

A QSDA $\mathcal{A}$ *accepts* a heap skinny-tree $\mathcal{H}$ if $\mathcal{A}$ recognizes all valuation trees of $\mathcal{H}$. The *language* accepted by $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of all heap skinny-trees $\mathcal{H}$ accepted by $\mathcal{A}$. A language $\mathcal{L}$ of heap skinny-trees is *regular* if there is a QSDA $\mathcal{A}$ such that $\mathcal{L} = L(\mathcal{A})$. Similarly, a language $\mathcal{L}$ of valuation trees is *regular* if there is a QSDA $\mathcal{A}$ such that $\mathcal{L} = L_{\text{v}}(\mathcal{A})$, where $L_{\text{v}}(\mathcal{A})$ is the set of all valuation trees recognized by $\mathcal{A}$.

QSDAs are a generalization of *quantified data automata* introduced in Chapter 5 which handle only lists, as opposed to QSDAs that handle skinny-trees. We now introduce various characterizations of QSDAs which are used later in the chapter.

Unique Minimal QSDA**.** In Chapter 5, we show that it is not possible to have a unique minimal (with respect to the number of states) quantified data automaton over words which accepts a given language over linear heap configurations. The proof gives a set of heap configurations over a linear heap-structure that is accepted by two different automata having the same number of states. Since QSDAs are a generalization of quantified data automata, the same counter-example language holds for QSDAs. However, under the assumption that all data formulas in $\mathcal{F}$ are pairwise non-equivalent, there

does exist a canonical automaton at the level of *valuation trees.* In Chapter 5, we prove the canonicity of quantified data automata, and their result extends to QSDAs in a straight forward manner.

Theorem 6.2.3. *For each* QSDA $\mathcal{A}$ *there is a unique minimal* QSDA $\mathcal{A}'$ *such that* $L_v(\mathcal{A}) = L_v(\mathcal{A}')$.

We give some intuition behind the proof of Theorem 6.2.3. First, we introduce a central concept called *symbolic trees.* A symbolic tree is a $(\Sigma \times (Y \cup \{-\}))$-labelled tree that records the positions of the universal variables and the pointer variables, but does not contain concrete data values (hence the word symbolic). A valuation tree can be viewed as a symbolic tree augmented with data values at every node in the tree. There exists a unique tree automaton over the alphabet Π that accepts a given regular language over symbolic trees. It can be shown that if the set of formulas in $\mathcal{F}$ are pairwise non-equivalent, then each state q in the tree automaton, at the root, can be decorated with a unique data formula $f(q)$ which extends the symbolic trees with data values such that the corresponding valuation trees are in the given language of valuation trees.

Hence, a language of valuation trees can be viewed as a function that maps each symbolic tree to a uniquely determined formula, and a QSDA can be viewed as a Moore machine (an automaton with output function on states) that computes this function. This helps us separate the structure of valuation trees (the height of the trees, the cells the pointer variables point to) from the data contained in the nodes of the trees. We formalize this notion by introducing *formula trees.*

Formula Trees. A *formula tree* over pointer variables PV, universal variables Y and a set of data formulas $\mathcal{F}$ is a tuple of a $\Sigma \times (Y \cup \{-\})$-labelled tree (or in other words a symbolic tree) and a data formula in $\mathcal{F}$. For a QSDA which captures a universally quantified property of the form $\bigwedge_i \forall y_1, \ldots, y_\ell. Guard_i \Rightarrow Data_i$, the symbolic tree component of the formula tree corresponds to guard formulas like $Guard_i$ which express structural constraints on the pointers pointing into the valuation tree. The data formula in the formula trees correspond to $Data_i$ which express the data values with which a symbolic tree (read $Guard_i$) can be extended so as to get a valuation tree accepted by the QSDA. In our running example, consider a QSDA with

a formula tree which has the same symbolic tree as the valuation tree in Figure 6.2(d) (but without the data values in the nodes) and a data-formula $\varphi = y_1 \rightarrow \texttt{data} \leq y_2 \rightarrow \texttt{data} \wedge y_1 \rightarrow \texttt{data} < key \wedge y_2 \rightarrow \texttt{data} \geq key$. This formula tree represents all valuation trees (including the one shown in Figure 6.2(d)) which extend the symbolic tree with data values which satisfy φ.

By introducing formula trees we explicitly take the view of a QSDA as an automaton that reads symbolic trees and outputs data formulas. We say a formula tree (t, φ) is accepted by a QSDA $\mathcal{A}$ if $\mathcal{A}$ reaches the state q after reading t and $f(q) = \varphi$. Given a QSDA $\mathcal{A}$, the language of valuation trees accepted by $\mathcal{A}$ gives an equivalent language of formula trees accepted by $\mathcal{A}$ and vice-versa. We denote the set of formula trees accepted by $\mathcal{A}$ as $L_f(\mathcal{A})$. A language over formula trees is called regular if there exists a QSDA accepting the same language.

Theorem 6.2.4. *For each* QSDA $\mathcal{A}$ *there is a unique minimal* QSDA $\mathcal{A}'$ *that accepts the same set of formula trees.*

6.3 A Partial Order over QSDAs

In the previous section we introduced quantified skinny-tree data automata as an automaton model for expressing universally quantified properties over heap skinny-trees. In this section, we first establish a partial order over the class of QSDAs and then show that QSDAs do not form a complete lattice with respect to this partial order. This motivates us to introduce a subclass of QSDAs called elastic QSDAs which we show, in Section 6.5, form a complete lattice and can be used to compute the semantics of programs. The partial order over EQSDAs with respect to which they form a lattice is the same as the partial order over QSDAs we introduce in this section.

Given a set of pointer variables PV and universal variables Y, let $\mathcal{Q}_{\mathcal{F}}$ be the class of all QSDAs over the lattice of data formulas $\mathcal{F}$. Clearly $\mathcal{Q}_{\mathcal{F}}$ is a partially-ordered set where the most natural partial order is the set-inclusion over the language of QSDAs. However, QSDAs are not closed under unions. Thus, a *least upper bound* for a pair of QSDAs does not exist with respect to this partial order. So we consider a new partial-order on QSDAs which allows us to define a least upper bound for every pair of QSDAs.

If we view a QSDA as a mapping from symbolic trees to formulas in $\mathcal{F}$, we can define a new partial-order relation on QSDAs as follows. We say $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if $L_f(\mathcal{A}_1) \subseteq L_f(\mathcal{A}_2)$, which means that for every symbolic tree t if $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$ then $\varphi_1 \sqsubseteq_{\mathcal{F}} \varphi_2$. Note that, whenever $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ implies that $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$. QSDAs, with respect to this partial order, form a lattice. Unfortunately, QSDAs do not form a complete lattice with respect to this above defined partial order (infinite sets of QSDAs may not have least upper-bounds). Consequently, we invent a subclass of QSDAs called elastic QSDAs (or EQSDAs) which we show form a complete lattice with respect to the above defined partial order. We also show that EQSDAs form an abstract domain by establishing an abstraction function and a concretization function between a set of heap skinny-trees and EQSDAs and showing that they form a Galois-connection. Even though QSDAs do not form a complete-lattice, we describe next a sound abstract transformer over QSDAs, a variant of which we use in Section 6.5 for abstracting the semantics of programs over EQSDAs.

6.4 Abstract Transformer over QSDAs

In this section we describe an abstract transformer over QSDAs which soundly over-approximates the concrete transformer over heap skinny-trees. We will later use a variant of this transformer when we compute the semantics of programs abstractly over EQSDAs.

We first show that it is not possible to capture the most-precise concrete transformer on QSDAs. A QSDA expresses universally quantified properties over heap trees, of the form $\forall y_1, \ldots, y_\ell.\psi$ where ψ is a quantifier-free formula over the pointer variables PV, the universal variables Y and the data values at the locations pointed to by these variables. Given a QSDA $\mathcal{A}$, the concrete transformer $F^\natural$ guesses a pre-state accepted by $\mathcal{A}$ (which involves existential quantification), and then constrains the post-state with respect to this guessed pre-state according to the semantics of the statement. For instance, consider the statement $p_i := p_j$. Given a QSDA accepting a universally quantified property $\forall y_1, \ldots, y_\ell.\psi$, its strongest post-condition with respect to this statement is the formula: $\exists p_i'.\forall y_1, \ldots, y_\ell.\psi[p_i/p_i'] \ \wedge \ p_i = p_j$. Note that, an interpretation of the existentially quantified variable p_i' in a model of this

formula gives the location node pointed to by variable p_i in the pre-state, such that the formula $\forall y_1, \ldots, y_\ell.\psi$ was satisfied by the pre-state. However it is not possible to express these precise post-conditions, which are usually of the form $\exists^*\forall^*\psi$, in our automaton model. So we over-approximate these precise post-conditions by a QSDA which semantically moves the existential quantifiers inside the universally quantified prefix – $\forall y_1 \ldots y_\ell.\exists p_i'.\psi[p_i/p_i']\ \wedge p_i = p_j$. The existential quantifier can now be eliminated using a combination of automata based quantifier elimination, for the structure, and the quantifier elimination procedures for the data-formula lattice $\mathcal{F}$. In the above example, intuitively, the abstract post-condition QSDA guesses a position for the pointer variable p_i for every valuation of the universal variables, such that the valuation tree augmented with this guessed position is accepted by the precondition QSDA. More generally, the abstract transformer computes the most precise post-condition over the language of valuation trees accepted by a QSDA, instead of computing the precise post-condition over the language of heap skinny-trees. In fact, we go beyond valuation trees to formula trees; the abstract transformer evolves the language of formula trees accepted by a QSDA by tracking the precise set of symbolic trees to be accepted in the post-QSDA and their corresponding data formulas.

We assume that the formula lattice $\mathcal{F}$ supports quantifier-elimination. We encourage the reader to keep in mind numerical domains over the theory of integers with constants (0, 1, etc.), addition, and the usual relations (like $<, \leq, =$) as an example of the formula lattice. Table 6.1[1] gives the abstract transformer $F_f^\sharp$ which takes a regular language over formula trees L_f and gives, as output, a set of formula trees. We know from Theorem 6.2.4 that for any regular set of formula trees there exists a unique minimal QSDA that accepts it. We show below (see Lemma 6.4.2) that for a QSDA $\mathcal{A}$, the language over formula trees given by $(F_f^\sharp)\ L_f(\mathcal{A})$ is regular. Hence, we can define the abstract transformer $F^\sharp$ as $F^\sharp = \lambda\mathcal{A}.\mathcal{A}'$ where $\mathcal{A}'$ is the unique minimal QSDA such that $L_f(\mathcal{A}') = (F_f^\sharp)\ L_f(\mathcal{A})$.

In Table 6.1, $label(t, p_i)$ is the set of pointer and universal variables which label the same node in t as variable p_i. The predicate $update(t, stmt, t')$ is true if symbolic trees t and t' are related such that the execution of statement *stmt* updates precisely the symbolic tree t to t'. As an example, the abstract

[1]The abstract transformer defined in Table 6.1 assumes that there are no memory errors in the program. It can be extended to handle memory errors.

Table 6.1: Abstract Transformer $F_f^\sharp$ over the language of formula trees. The abstract transformer over QSDAs $F^\sharp(\mathcal{A}) = \mathcal{A}'$ where $\mathcal{A}'$ is the unique minimal QSDA such that $L_f(\mathcal{A}') = (F_f^\sharp)\ L_f(\mathcal{A})$. The predicate *update* and the set *label* are defined below.

Statements	Abstract Transformer $F_f^\sharp$ on a regular language over formula trees
$p_i := nil$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t, \varphi) \in L_f, \mathit{update}(t, p_i := nil, t')\}\}$
$p_i := p_j$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = (p_i \to \texttt{data} = p_j \to \texttt{data}) \sqcap \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t, \varphi) \in L_f, \mathit{update}(t, p_i := p_j, t')\}\}$
$p_i := p_j \to \texttt{next}$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t, \varphi) \in L_f, \mathit{update}(t, p_i := p_j \to \texttt{next}, t')\} \bigsqcap\{p_i \to \texttt{data} = v \to \texttt{data} \mid v \in \mathit{label}(t', p_i)\}\}$
$p_i \to \texttt{next} := nil$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = \bigsqcup\{\varphi \mid (t, \varphi) \in L_f, \mathit{update}(t, p_i \to \texttt{next} := nil, t')\}\}$
$p_i \to \texttt{next} := p_j$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = \bigsqcup\{\varphi \mid (t, \varphi) \in L_f, \mathit{update}(t, p_i \to \texttt{next} := p_j, t')\}\}$
$p_i \to \texttt{data} := data\_expr$	$\lambda L_f.\ \{(t, \varphi') \mid \varphi' = \exists d.(\varphi[v_1 \to \texttt{data}/d, \ldots, v_\ell \to \texttt{data}/d] \sqcap \bigsqcap\{v \to \texttt{data} = data\_expr[v_1 \to \texttt{data}/d, \ldots, v_\ell \to \texttt{data}/d] \mid v \in V\}), V = \{v_1, \ldots, v_\ell\} = \mathit{label}(t, p_i), (t, \varphi) \in L_f\}$
$\texttt{assume}\ \psi_{struct}$	$\lambda L_f.\ \{(t', \varphi') \mid (t', \varphi') \in L_f,\ t' \models \psi_{struct}\}$
$\texttt{assume}\ \psi_{data}$	$\lambda L_f.\ \{(t', \varphi') \mid \varphi' = \varphi \sqcap \psi_{data}, (t', \varphi) \in L_f\}$
$\texttt{new}\ p_i$	$\lambda L_f.\{(t', \varphi') \mid \varphi' = (y \to \texttt{data} = p_i \to \texttt{data}) \sqcap \bigsqcup\{\exists d_1 d_2.\varphi[p_i \to \texttt{data}/d_1, y \to \texttt{data}/d_2] \mid (t, \varphi) \in L_f, \mathit{update}(t, \texttt{new}^{\{y\}}\ p_i, t')\},\ y \in Y\} \bigcup \{(t', \varphi') \mid \varphi' = \bigsqcup\{\exists d.\varphi[p_i \to \texttt{data}/d] \mid (t, \varphi) \in L_f, \mathit{update}(t, \texttt{new}^{\{-\}}\ p_i, t')\}\}$

transformer for the statement $p_i := nil$ in the first row of Table 6.1 states that the post-QSDA maps the symbolic tree t' to the data-formula φ' where φ' is the join of all formulas of the form $\exists d.\varphi[p_i \to \texttt{data}/d]$ where φ is the data-formula associated with symbolic tree t in the pre-QSDA such that $\mathit{update}(t, p_i := nil, t')$ is true.

We now briefly describe the predicate $\mathit{update}(t, \texttt{new}^{\{\hat{y}\}}\ p_i, t')$, where $\hat{y} \in Y \cup \{-\}$, which is used in the definition of the transformer for the `new` statement and is slightly more involved. The statement $\texttt{new}\ p_i$ allocates a new memory location. After the execution of this statement, pointer p_i points to this allocated node. Besides, the universal variables also need to valuate over this new node apart from the valuations over the previously existing locations in the heap. The superscript $\{y\}$ in the predicate $\mathit{update}(t, \texttt{new}^{\{y\}}\ p_i, t')$

tracks the case when variable $y \in Y$ valuates over the newly allocated node (analogously, the superscript {-} tracks the case when no universal variable valuates over the newly allocated node). Hence, if $update(t, \texttt{new}^{\{y\}}\ p_i, t')$ holds true then the symbolic trees t and t' agree on the locations pointed to by all variables except p_i and the universal variable y; both these variables point, in t', to a new location v which is not in t and a new edge exists in t' from the root to v.

An important point to note is that the abstract transformer for the statement $p_i \rightarrow \texttt{next}$ (i.e., the predicate $update(t, p_i \rightarrow \texttt{next} := p_j, t')$) assumes that the program does not introduce cycles in the heap configurations.

From the construction in Table 6.1 it can be observed that given a language of valuation trees obtained uniquely from a language of formula trees, $F_f^\sharp$ applies the most-precise concrete transformer on each valuation tree in the language, and then constructs the smallest regular language of valuation trees (or equivalently formula trees) which approximates this set. More precisely, for all formula trees $(t, \varphi) \in L_f(\mathcal{A})$, the abstract transformer $F_f^\sharp$ applies the precise concrete transformer on the symbolic tree t (only the structure with the valuations for universal variables) to obtain t'. And separately, it applies the precise concrete transformer on the data extensions of t, which is given by φ, to obtain the data formula φ' such that $(t', \varphi') \in (F_f^\sharp)L_f(\mathcal{A})$.

As we have already discussed, the abstract transformer by reasoning over valuation/formula trees (and not heap skinny-trees) leads to a loss in precision. To regain some of the lost precision, we define a function *Strengthen* which takes a formula language L_f and finds a smaller language over formula trees, which accepts the same set of heap trees. Here $t\ \lfloor_y$ stands for a $\Pi\backslash\{y\}$ -labelled tree which agrees with t on the locations pointed to by all variables except y.

$$Strengthen = \lambda y.\lambda L_f.\Big\{(t',\varphi') \mid \varphi' : \varphi'' \sqcap \phi,\ (t', \varphi'') \in L_f, \\ \phi : \bigsqcap\{\exists d.\varphi[y \rightarrow \texttt{data}/d] \mid (t, \varphi) \in L_f, t\ \lfloor_y = t'\ \lfloor_y\}\Big\}$$

We now reason about the soundness of the operator *Strengthen*. Fix a $y \in Y$. Consider a QSDA $\mathcal{A}$ with a language over formula trees L_f and consider all symbolic trees t such that $t\ \lfloor_y = t'\ \lfloor_y$. This implies that the trees t have the pointer variables pointing to the same positions as t' and have the same valuations for variables in $Y\backslash\{y\}$. Since our automaton model has a universal

semantics, any heap tree accepted by $\mathcal{A}$ should satisfy the data formulas annotated at the final states reached for every valuation of the universal variables. If we look at a fixed valuation for variables in $Y \backslash \{y\}$ (which is same as that in t') and different valuations for y, any heap tree accepted should satisfy the formula $\exists d.\varphi[y \rightarrow \texttt{data}/d]$ for all such $(t, \varphi) \in L_f$. Hence the *Strengthen* operator can safely strengthen the formula φ'' associated with the symbolic tree t' to $\varphi'' \sqcap \phi$. It can be shown that for a given universal variable y and a regular language L_f, the language over formula trees (*Strengthen*) y L_f is regular. In fact, the QSDA accepting the language (*Strengthen*) y $L_f(\mathcal{A})$ for a QSDA $\mathcal{A}$ can be easily constructed. The abstract transformer $F_f^\sharp$ can thus be soundly strengthened by an application of *Strengthen* at each step, for each variable $y \in Y$.

It is clear that the abstract transformer $F_f^\sharp$ in Table 6.1 as well as the function *Strengthen* are monotonic. We now show that the language over formula trees given by $(F_f^\sharp)L_f(\mathcal{A})$ is a regular language for any QSDA $\mathcal{A}$. This helps us to construct the abstract transformer $F^\sharp : \mathcal{Q}_\mathcal{F} \rightarrow \mathcal{Q}_\mathcal{F}$. And finally, we show that this abstract transformer is a sound approximation of the concrete transformer $F^\natural$.

Lemma 6.4.1. *The abstract transformer $F_f^\sharp$ is sound with respect to the concrete semantics.*

Lemma 6.4.2. *For a* QSDA $\mathcal{A}$*, the language $(F_f^\sharp)$ $L_f(\mathcal{A})$ over formula trees is regular.*

From Lemma 6.4.2 and Theorem 6.2.4, it follows that there exists a QSDA $\mathcal{A}'$ such that $\mathcal{A}' = (F^\sharp)\mathcal{A}$. The monotonicity of $F^\sharp$, with respect to the partial order defined in Section 6.3 over QSDAs, follows from the monotonicity of $F_f^\sharp$. The soundness of $F^\sharp$ can be stated as the following theorem.

Theorem 6.4.3. *The abstract transformer $F^\sharp$ is sound with respect to the concrete transformer $F^\natural$.*

Hence $F^\sharp$ is both monotonic, and sound with respect to the concrete transformer $F^\natural$. In the next section we introduce elastic QSDAs, a subclass of QSDAs, which form an abstract domain and we use the above defined transformer $F^\sharp$ over QSDAs to define an abstract transformer over elastic QSDAs. Note that the abstract transformer $F^\sharp$, in general, might require a

powerset construction over the input QSDA, very similar to the procedure for determinizing a tree automaton. Hence the worst-case complexity of the abstract transformer is exponential in the size of the QSDA. However our experiments show that this worst-case is not achieved for most programs in practice.

6.5 Elastic Quantified Skinny-Tree Data Automata

As we saw in Section 6.3, a least upper bound might not exist for an infinite set of QSDAs. Therefore, we identify a sub-class of QSDAs called elastic quantified skinny-tree data automata (EQSDAs) such that elastic QSDAs form a complete lattice and provide a mechanism to compute the abstract semantics of programs.

Let us denote the symbol $(b, -) \in \Pi$ indicating that a position does not contain any variable by $\underline{b}$. A QSDA $A = (Q, \Pi, \Delta, \mathcal{T}, f)$ where $\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ is called elastic if each transition on $\underline{b}$ in Δ_1 is a self loop i.e. $\Delta_1(q_1, \underline{b}) = q_2$ implies $q_1 = q_2$.

We first show that the number of states in a minimal EQSDA is bounded for a fixed set PV and Y. Consider all skinny-trees where a blank symbol $\underline{b}$ occurs only at branching points. Since the number of branching points is bounded and since every variable can occur only once, there are only a bounded number of such trees. Consider any minimal EQSDA. Consider all states that are part of the run of the EQSDA on the trees of the kind above. Clearly, there are only a bounded number of states in this set. Now, we argue that on *any* tree, the run on that tree can only use these states. For any tree t, consider the tree t' obtained by removing the nodes of degree one marked by blank. The run on tree t will label common states of t and t' identically, and the nodes that are removed will be labeled by the state of its child, since blank transitions cannot cause state-change. Since in any minimal automaton, for any state, there must be some tree that uses this state, we know that the number of state is also bounded.

We next show the following result that every QSDA $\mathcal{A}$ can be *most precisely over-approximated* by a language of valuation trees (or equivalently formula trees) that can be accepted by an EQSDA $\mathcal{A}_{\text{el}}$. We will refer to this construction, which we outline below, as *elastification*. This result is

an extension of the unique over-approximation result for quantified data automata over words (see Chapter 5). Using this result, we can show that elastic QSDAs form a complete lattice and there exists a Galois-connection $\langle \alpha, \gamma \rangle$ between a set of heap skinny trees and EQSDAs. This lets us define an abstract transformer over the abstract domain EQSDAs such that the semantics of a program can be computed over EQSDAs in a sound manner.

Let $\mathcal{A} = (Q, \Pi, \Delta, \mathcal{T}, f)$ be a QSDA such that $\Delta = (\Delta_0, \Delta_1, \ldots, \Delta_k)$ and for a state q let $R_{\underline{b}}(q) := \{q' \mid q' = q \text{ or } \exists q''.q'' \in R_{\underline{b}}(q) \text{ and } \Delta_1(q'', \underline{b}) = q'\}$ be the set of states reachable from q by a (possibly empty) sequence of $\underline{b}$-unary-transitions. For a set $S \subseteq Q$ we let $R_{\underline{b}}(S) := \bigcup_{q \in S} R_{\underline{b}}(q)$.

The set of states of $\mathcal{A}_{\text{el}}$ consists of sets of states of $\mathcal{A}$ that are reachable by the following transition function Δ^{el} (where $\Delta_i(S_1, \ldots, S_i, a)$ denotes the standard extension of the transition function of $\mathcal{A}$ to sets of states):

$$\Delta_0^{\text{el}}(a) = R_{\underline{b}}(\Delta_0(a))$$

$$\Delta_1^{\text{el}}(S, a) = \begin{cases} R_{\underline{b}}(\Delta_1(S, a)) & \text{if } a \neq \underline{b} \\ S & \text{if } a = \underline{b} \text{ and } \Delta_1(q, \underline{b}) \text{ is defined for some } q \in S \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\Delta_i^{\text{el}}(S_1, \ldots, S_i, a) = R_{\underline{b}}(\Delta_i(S_1, \ldots, S_i, a)) \text{ for } i \in [2, k]$$

Note that this construction is similar to the usual powerset construction except that in each step we apply the transition function of $\mathcal{A}$ to the current set of states and take the $\underline{b}$-closure. However, if the input letter is $\underline{b}$ on a unary transition, $\mathcal{A}_{\text{el}}$ loops on the current set if a $\underline{b}$-transition is defined for some state in the set.

It can be argued inductively, starting from the leaf states, that the type for all states in a set is the same. Hence we define the type of a set S as the type of any state in S. The final evaluation formula for a set is the least upper bound of the formulas for the states in the set: $f^{\text{el}}(S) = \bigsqcup_{q \in S} f(q)$. We can now show that $\mathcal{A}_{\text{el}}$ is the *most precise over-approximation* of the language of valuation trees accepted by QSDA $\mathcal{A}$.

Theorem 6.5.1. *For every* QSDA $\mathcal{A}$*, the* EQSDA $\mathcal{A}_{el}$ *satisfies* $L_v(\mathcal{A}) \subseteq L_v(\mathcal{A}_{el})$*, and for every* EQSDA $\mathcal{B}$ *such that* $L_v(\mathcal{A}) \subseteq L_v(\mathcal{B})$*,* $L_v(\mathcal{A}_{el}) \subseteq L_v(\mathcal{B})$ *holds.*

The proof of Theorem 6.5.1 is similar to the proof of a similar theorem in Chapter 5 for the case of words. The above theorem can also be stated

over a language of formula trees in the same way, that $\mathcal{A}_{\text{el}}$ is the most precise over-approximation of the language of formula trees accepted by QSDA $\mathcal{A}$.

We can now show that EQSDAs form a complete lattice $({\mathcal{Q}_{\mathcal{F}}}^{el}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The partial order on EQSDAs is the same as the partial order on QSDAs but now restricted to only elastic QSDAs. For EQSDAs $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 \sqsubseteq \mathcal{A}_2$ if $L_f(\mathcal{A}_1) \subseteq L_f(\mathcal{A}_2)$, meaning that for every symbolic tree t if $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$ then $\varphi_1 \sqsubseteq_{\mathcal{F}} \varphi_2$. Given EQSDAs $\mathcal{A}_1$ and $\mathcal{A}_2$ and a symbolic tree t such that $(t, \varphi_1) \in L_f(\mathcal{A}_1)$ and $(t, \varphi_2) \in L_f(\mathcal{A}_2)$, the meet $\mathcal{A}_1 \sqcap \mathcal{A}_2$ is the EQSDA which maps t to the unique formula $\varphi_1 \sqcap_{\mathcal{F}} \varphi_2$. The algorithm to construct this EQSDA involves the product-construction and is very similar to the algorithm to intersect finite-state automata. Given elastic QSDAs (which have only self-loops on states on a $\underline{b}$-transition), the data automaton obtained by the above mentioned product construction is also elastic and is the *greatest lower bound*. The meet for EQSDAs, $\mathcal{A}_1 \sqcup \mathcal{A}_2$, is obtained by constructing a QSDA which maps the symbolic tree t to the formula $\varphi_1 \sqcup_{\mathcal{F}} \varphi_2$ followed by its unique elastification to obtain an EQSDA. The construction of the intermediate QSDA is very similar to the greatest lower bound construction we described above, using a product-construction. Note that the number of states in this QSDA is bounded as its structure is almost elastic except for single occurrences of $\underline{b}$-transitions which do not self-loop. Using the most precise over-approximation result, the EQSDA obtained by the elastification of this QSDA is the *least upper bound* of the set of EQSDAs. The bottom and the top elements in the lattice ${\mathcal{Q}_{\mathcal{F}}}^{el}$ over EQSDAs are the EQSDAs taking every symbolic tree to the formulas *false* and *true* respectively.

We can now view the space of EQSDAs as an abstraction over sets of heap skinny trees. Let us define an abstraction function $\alpha : \mathcal{H} \to {\mathcal{Q}_{\mathcal{F}}}^{el}$ and a concretization function $\gamma : {\mathcal{Q}_{\mathcal{F}}}^{el} \to \mathcal{H}$ such that $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ form a Galois-connection. Note that, abstract interpretation [CC77] requires that the abstraction function α maps a concrete element (a language of heap skinny-trees) to a unique element in the abstract domain and that α be surjective; similarly γ should be an injective function. Also note that given a regular language of heap skinny-trees there might be several QSDAs (and thus EQSDAs) accepting that language. In such a case defining a surjective function α is not possible. Therefore, we first restrict ourselves to a set of EQSDAs in ${\mathcal{Q}_{\mathcal{F}}}^{el}$ where each EQSDA accepts a different language.

Under this assumption, we define an α and a γ as follows: for a set of heap configurations H, $\alpha(H) = \bigsqcap\{\mathcal{A} \mid H \subseteq L(\mathcal{A})\}$ and $\gamma(\mathcal{A}) = L(\mathcal{A})$. Note that both α and γ are order-preserving; α is surjective and γ is an injective function. Also for a set of heap configurations H, $H \subseteq \gamma(\alpha(H))$ and for an EQSDA $\mathcal{A}$, $\mathcal{A} = \alpha(\gamma(\mathcal{A}))$. Hence $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ forms a Galois-connection.

Theorem 6.5.2. *Let $(\mathcal{H}, \subseteq)$ be the class of sets of heap skinny-trees and $({\mathcal{Q}_{\mathcal{F}}}^{el}, \sqsubseteq)$ be the class of* EQSDA*s (accepting pairwise inequivalent languages) over data formulas $\mathcal{F}$, then $(\mathcal{H}, \alpha, \gamma, {\mathcal{Q}_{\mathcal{F}}}^{el})$ forms a Galois-connection.*

Let us define the abstract transformer over EQSDAs as $F_{el}^{\sharp} : {\mathcal{Q}_{\mathcal{F}}}^{el} \rightarrow {\mathcal{Q}_{\mathcal{F}}}^{el} = F_{el} \circ F^{\sharp}$ where F_{el} is the *elastification* operator which returns the most precise EQSDA over-approximating a language of valuation trees accepted by a QSDA. The soundness of $F_{el}^{\sharp}$ follows from the soundness of $F^{\sharp}$ (and the fact that F_{el} is extensive, i.e., $F_{el}(\mathcal{A}) \sqsupseteq \mathcal{A}$). Similarly its monotonicity follows from the monotonicity of $F^{\sharp}$ and the monotonicity of F_{el}. The semantics of a program can be thus computed over the abstract domain ${\mathcal{Q}_{\mathcal{F}}}^{el}$ as the least fix-point of a set of equations of the form $\psi = F_{el}^{\sharp}(\psi)$. Since the number of states in an EQSDA is bounded (for a given set of program variables PV and universal variables Y), this least fix-point computation terminates (modulo the convergence of the data formulas in the formula lattice $\mathcal{F}$ in which case termination can be achieved by defining a suitable widening operator on the data formula lattice).

6.5.1 From EQSDAs to a Decidable Fragment of Strand

In this section we show that EQSDAs can be converted to formulas that fall in a decidable fragment of first order logic, in particular the decidable fragment of Strand over lists. Hence, once the abstract semantics has been computed over EQSDAs, the invariants expressed by the EQSDAs can be used to validate assertions in the program that are also written using the decidable sublogic of Strand over lists. We assume that the assertions in our programs express quantified properties over *disjoint* lists, like sortedness of lists, etc. and properties relying on mutual sharing or aliasing of list-structures are not allowed.

Given an EQSDA $\mathcal{A}$ and for every pointer variable p, we construct a QSDA over words that are projections of trees accepted by $\mathcal{A}$ and where

the first node is p. A key property in the decidable fragment of STRAND is that universal quantification is not permitted to be over elements that are only a bounded distance away from each other, or in other words universally quantified variables are only allowed to be related by elastic relations. As a result, we can safely elastify the constructed QSDA over words and obtain an EQSDA over words expressing quantified properties in the decidable sublogic of STRAND. In Chapter 5 we have detailed the translation from an EQSDA over words to a quantified formula in the decidable fragment of STRAND over lists. Using the same translation, we obtain a formula in STRAND that precisely corresponds to the QSDA that is the abstract semantics of the program at the point of the assertion we want to verify. Since the translation ensures this formula falls in the decidable fragment of STRAND, we can use the underlying decision procedure to prove the program correct.

6.6 Experimental Evaluation

We implemented the abstract domain over EQSDAs presented in this chapter, and evaluated it on several list-manipulating programs. We now first present the implementation details followed by our experimental results. Our prototype implementation along with the experimental results and programs can be found at `http://web.engr.illinois.edu/~garg11/qsdas.html`.

Implementation Details. Given a program P we compute the abstract semantics of the program over the abstract domain ${\mathcal{Q}_{\mathcal{F}}}^{el}$ over EQSDAs. A program is a sequence of statements as defined by the grammar in Figure 6.1. In addition to those statements, a program is also annotated with a pre-condition and a bunch of assertions. The pre-condition formulas belong to a fragment of STRAND over lists and can express quantified properties over disjoint lists (aliasing of two list-structures is not allowed), like sortedness of lists, etc. Given a pre-condition formula φ, we construct the EQSDA which accepts all the heap skinny-trees which satisfy φ. This EQSDA precisely captures the set of initial configurations of the program. Starting from these configurations we compute the abstract semantics of the program over ${\mathcal{Q}_{\mathcal{F}}}^{el}$. The assert statements in the program are ignored during the fix-point computation. Once the convergence of the fix-point has been achieved, the

Table 6.2: Experimental results. Property checked — LIST: the return pointer points to a list; INIT: the list is properly initialized with some key; MAX: returned value is the maximum of all data values in the list; GEK: the list (or some parts of the list) have data values greater than or equal to a key k; SORT: the list is sorted; LAST: returned pointer is the last element of the list; EMPTY: the returned list is empty.

Programs	#PV	#Y	#DV	Property checked	#Iter	Max. size of QSDA	Time (s)
INIT	2	1	1	INIT, LIST	4	19	0.0
ADD-HEAD	2	1	1	INIT, LIST	-	11	0.1
ADD-TAIL	3	1	1	INIT, LIST	4	29	0.1
DELETE-HEAD	2	1	1	INIT, LIST	-	10	0.0
DELETE-TAIL	4	1	1	INIT, LIST	5	51	0.5
MAX	2	1	1	MAX, LIST	4	19	0.1
CLONE	4	1	1	INIT, LIST	4	44	0.7
FOLD-CLONE	5	1	1	INIT, LIST	5	57	3.2
COPY-GE5	4	1	0	GEK, LIST	9	53	2.6
FOLD-SPLIT	3	1	1	GEK, LIST	4	33	0.3
CONCAT	4	1	1	INIT, LIST	5	44	0.7
SORTED-FIND	2	2	2	SORT, LIST	5	38	0.3
SORTED-INSERT	4	2	1	SORT, LIST	6	163	5.8
BUBBLE-SORT	4	2	1	SORT, LIST	5/18	191	42.8
SORTED-REVERSE	3	2	0	SORT, LIST	5	43	1.5
EXPRESSOS-LOOKUP-PREV	3	2	1	SORT, LIST	6	73	2.2
GSLIST-APPEND	4	0	1	LIST	8	3	0.0
GSLIST-PREPEND	2	0	1	LIST	-	3	0.0
GSLIST-LAST	3	0	0	LAST, LIST	3	7	0.0
GSLIST-FREE	3	0	0	EMPTY, LIST	1	3	0.0
GSLIST-POSITION	4	0	0	LIST	3	13	0.0
GSLIST-REVERSE	3	0	0	LIST	3	5	0.0
GSLIST-CUSTOM-FIND	3	1	1	GEK, LIST	4	29	0.1
GSLIST-NTH	3	0	1	LIST	3	7	0.0
GSLIST-REMOVE	4	0	1	LIST	4	10	0.0
GSLIST-REMOVE-LINK	5	0	0	LIST	4	16	0.0
GSLIST-REMOVE-ALL	5	1	1	GEK, LIST	5	51	0.6
GSLIST-INSERT-SORTED	5	2	1	SORT, LIST	6	279	27.4

EQSDAs can be converted back into decidable STRAND formula over lists (as described in Section 6.5.1) and the STRAND decision procedure can be used for validating the assertions.

We recall that the abstract transformer $F^\sharp_{el}$ is a function composition of the abstract transformer $F^\sharp$ over QSDAs and the unique elastification operator F_{el}. So that we are as precise as possible, for every statement in the program we apply the more precise transformer $F^\sharp$ (and not $F^\sharp_{el}$). However, we apply the elastification operator F_{el} at the header of loops before the join to ensure

convergence of the computation of the abstract semantics. The intermediate semantic facts (QSDAs) in our analysis are thus not necessarily elastic.

Our abstract domains are parameterized by a quantifier-free domain $\mathcal{F}$ over the data formulas. In our experiments, we instantiate $\mathcal{F}$ with the octagon abstract domain [Min01] from the Apron library [JM09]. It is sufficient to capture the pre/post-conditions and the invariants of all our programs.

Experimental Results. We evaluate our abstract domain on a suite of list-manipulating programs (see Table 6.2). For every program we report the number of pointer variables (PV), the number of universal variables (Y), the number of data variables (DV) and the property being checked for the program. We also report the number of iterations required for the fixed-point to converge, the maximum size of the intermediate QSDAs and finally the time taken, in seconds, to analyze the programs.

The names of the programs in Table 6.2 are self-descriptive, and we only describe some of them. The program COPY-GE5, from [BDES11], copies all those entries from a list whose data value is greater than or equal to 5. Similarly, the program FOLD-SPLIT [BDES11] splits a list into two lists – one which has entries whose data values are greater than or equal to a key k and the other list with entries whose data value is less than k. The program EXPRESSOS-LOOKUP-PREV is a method from the module cachePage in a verified-for-security platform for mobile applications [MPX+13]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. This method returns the correct position in the cache at which a disc page could be inserted. The programs in the second part of the table are various methods adapted from the Glib list library which comes with the GTK+ toolkit and the Gnome desktop environment. The program GSLIST-CUSTOM-FIND finds the first node in the list with a data value greater or equal to k and GSLIST-REMOVE-ALL removes all elements from the list whose data value is greater or equal to k. The programs GSLIST-INSERT-SORTED and SORTED-INSERT insert a key into a sorted list.

All experiments were completed on an Intel Core i5 CPU at 2.4GHz with 6Gb of RAM. The number of iterations is left blank for programs which do not have loops. BUBBLE-SORT program converges on a fix-point after 18 iterations of the inner loop and 5 iterations of the outer loop. The size of the intermediate QSDAs depends on the number of universal variables and

the number of pointer variables and largely governs the time taken for the analysis of the programs. For all programs, our prototype implementation computes their abstract semantics in reasonable time. Moreover we manually verified that the final EQSDAs in all the programs were sufficient for proving them correct (this validity check is currently not mechanized due to the unavailability of the decision procedure for STRAND but can be done in the future). The results show that the abstract domain we propose in this chapter is reasonably efficient and powerful enough to prove a large class of programs manipulating singly-linked list structures.

6.7 Related Work

In this chapter we introduce a class of data automata called quantified data automata for skinny-trees (QSDA) to capture universally quantified properties over skinny-trees. The QSDA model is an extension of the QDA model we introduced in Chapter 5 [GLMN13] to capture quantified data properties over words, as opposed to trees. Also, the QDA model in [GLMN13] is parameterized by a *finite* set of data formulas and is used for *learning* invariants from examples and counterexamples. In contrast, QSDAs extend the QDA model with an (possibly-infinite) abstract domain over data formulas and, in this chapter, we develop a theory of abstract interpretation over QSDAs.

CHAPTER 7

VERIFYING SHARED MEMORY CONCURRENT PROGRAMS

In this thesis, we have developed novel techniques for verifying sequential programs by learning inductive invariants. In this and the subsequent chapter, we will develop techniques to verify concurrent programs. In this chapter, in particular, we will address the problem of verifying shared memory concurrent programs. Given a concurrent program with a safety specification, we would like to *sequentialize* it, i.e., reduce the problem of verifying the concurrent program to the verification of a sequential program. Consequently, we can use techniques for verifying sequential programs, including learning algorithms for synthesizing inductive invariants for sequential programs covered in this thesis, to verify concurrent programs. The most important aspect of this reduction is that we seek a sequential program that does not simply simulate the global evolution of the concurrent program as that would be quite complex and involve taking the product of the local state-spaces of the processes. Instead, we seek a sequential program that tracks a *bounded* number of copies of the local and shared variables, where the bound is independent of the number of parallel components.

The appeal of sequentialization is that it allows using the existing class of sequential verification tools to verify concurrent programs. A large number of sequential verification techniques and tools, like deductive verification, abstraction-based model-checking, static data-flow analysis, and invariant synthesis immediately come into play when a sequentialization is possible.

Of course, such sequentializations are not possible for all concurrent programs and specifications. In fact, in the presence of recursion and when variables have bounded domains, concurrent verification is undecidable while sequential verification is decidable, which proves that an effective sequentialization is in general impossible.

The currently known sequentializations have hence focussed on capturing *under-approximations* of concurrent programs. Lal and Reps [LR08] showed

that given a concurrent program with finitely many threads and a bound k, the problem of checking whether the concurrent program is safe on all executions that involve only k context-rounds can be reduced to the verification of a sequential program. A *lazy* sequentialization for bounded context rounds that ensures that the sequential program explores only states reachable by the concurrent program was defined by La Torre et al [LMP09]. A sequentialization for unboundedly many threads and bounded round-robin rounds of context-switching is also known [SLP12]. Lahiri, Qadeer and Rakamarić have used the sequentialization of Lal and Reps to check concurrent C-programs by unrolling loops in the sequential program a bounded number of times, and subjecting them to deductive SMT-solver based verification [LQR09, GHR10].

In this chapter, we show a general sequentializability result that is not restricted to under-approximations. We show that any concurrent program with finitely many threads can *always* be sequentialized provided there exists a *compositional proof* of correctness of the concurrent program. More precisely, we show that given a concurrent program C with assertions and a set of *auxiliary* variables A, there is a sequentialization of it, $S_{C,A}$ with assertions, and that C can be shown to compositionally satisfy its assertions by exposing the auxiliary variables A if and only if $S_{C,A}$ satisfies its assertions. The notion of C compositionally satisfying its assertions using auxiliary variables A is defined semantically, and intuitively captures the *rely-guarantee proofs* pioneered by Jones [Jon83]. Rely-guarantee proofs of concurrent programs are very standard, and perhaps the best known compositional verification technique for concurrent programs. In these proofs, *auxiliary variables* can be seen as local states that get exposed in order to build a compositional rely-guarantee proof.

Compositional proofs of programs may not always exist, and since our sequentialization only produces sequential programs that are precise when a compositional proof exists over the fixed auxiliary variables A, proving its sequentialization correct can be seen as a sound but incomplete mechanism for verifying the concurrent program. Note that our sequentialization *does not* require the compositional proof to be given; it is only parameterized by the auxiliary variables A. In fact, if the sequential program is correct, then we show that the concurrent program is always correct. Conversely, if the concurrent program is correct and has a compositional proof using variables A, then we show that the sequential program is guaranteed to be correct as

well.

The salient aspect of our sequentialization is that it can be used to prove concurrent programs *entirely* correct, as opposed to checking under-approximations of it. Moreover, though our sequentializations are sound but incomplete, we believe they are useful on most practical applications since concurrent programs often have compositional proofs. Our result also captures the *cost* of sequentialization (i.e. the number of variables in the sequentialization) as directly proportional to the number of auxiliary variables that are required to build a compositional proof. Concurrent programs that are "loosely coupled" often require only a small number of auxiliary variables to be exposed, and hence admit efficient sequentializations.

We also describe our experience in applying our sequentialization to prove a suite of concurrent programs entirely correct by using deductive verification of their sequentializations. More precisely, we wrote rely-guarantee proof annotations for some concurrent programs by formulating the rely and guarantee conditions, the loop invariants, and pre- and post-conditions for every function. We then sequentialized the concurrent program and also *transformed* the rely-guarantee proof annotations to corresponding proof annotations on the sequential program. As we show, in this translation, rely and guarantee conditions naturally get transformed to pre- and post-conditions of methods, while loop invariants and pre- and post-conditions get translated to loop invariants and pre- and post-conditions in the sequential program. Then, using an automatic sequential program verifier Boogie, we verified the sequentializations correct. Boogie takes our programs with the proof annotations, generates verification conditions, and discharges them using an automatic theorem prover (SMT solver).

The above use of sequentialization for deductive verification is not the best use of our sequentializations, as given rely-guarantee proofs, simpler techniques for statically verifying them are known [FFQ02]. However, our sequentializations can be applied even when the rely-guarantee proofs are not known, provided the sequential verification tool is powerful to prove it correct. An important corollary is that if we have a learning algorithm that synthesizes inductive loop invariants and pre-/post-condition annotations for sequential programs, we can use the same learner for synthesizing compositional rely-guarantee proofs for the given concurrent program. Indeed, we have used the sequentialization followed by sequential verification, using an automatic

predicate-abstraction tool SLAM [BR02], to prove some concurrent programs correct.

In summary, the result presented in this chapter shows a surprising connection between compositional proofs and sequentializability. We believe that this constitutes a fundamental theoretical understanding of when concurrent programs are efficiently sequentializable, and offers the first efficient sequentializations that work without underapproximation restrictions, enabling us to verify concurrent program entirely using sequentializations.

7.1 A Compositional Abstract Semantics for Concurrent Programs

We define a *non-standard compositional semantics* for concurrent programs, different from the traditional semantics, in order to capture when a parallel composition of programs can be *argued compositionally to satisfy a specification*. This semantics is parameterized by a set of auxiliary variables, and is the semantic analog of *compositional rely-guarantee proofs* pioneered by Jones [Jon83].

Let us fix two processes P_1 and P_2, working concurrently, with local variables L_1 and L_2 respectively, and a set of shared variables S (assume L_1, L_2 and S are pairwise disjoint, without loss of generality). For any set of (typed) variables V, let Val_V denote the set of valuations of V to their respective data-domains (data-domains are finite or countably infinite). For any $u \in Val_V$, let $u{\downarrow}V'$ denote the valuation u restricted to the variables in $V \cap V'$. We extend this notation to sets of valuations, $U{\downarrow}V'$. Also, for any $u \in Val_V$ and $u' \in Val_{V'}$, where $V \cap V' = \emptyset$, let $u \cup u'$ denote the unique valuation in $Val_{V \cup V'}$ that extends u and u' to $V \cup V'$.

Let $Init \subseteq (Val_{L_1} \times Val_{L_2} \times Val_S)$ be the set of initial global configurations of $P_1||P_2$. Let $\delta_1 \subseteq (Val_{L_1} \times Val_S \times Val_{L_1} \times Val_S)$ and $\delta_2 \subseteq (Val_{L_2} \times Val_S \times Val_{L_2} \times Val_S)$ be the local transition relations of P_1 and P_2, respectively.

The natural (interleaving) semantics of $P_1||P_2$ is, of course, defined by the function $\delta \subseteq (Val_{L_1} \times Val_{L_2} \times Val_S \times Val_{L_1} \times Val_{L_2} \times Val_S)$, where $\delta(l_1, l_2, s, l_1', l_2', s')$ holds iff $\delta_1(l_1, s, l_1', s')$ holds and $l_2' = l_2$, or $\delta_2(l_2, s, l_2', s')$ holds and $l_1' = l_1$. The set of *reachable states* according to this relation, *Reach*, is defined as the set of global states that can be reached from the initial state.

Let us now define the non-standard compositional semantics of $P_1||P_2$. This definition is parameterized by a set of *auxiliary variables* $A \subseteq L_1 \cup L_2$.

Definition 7.1.1. *The semantics of the compositional semantics of parallel composition with respect to the set of auxiliary variables A, denoted $P_1||_A P_2$, is defined using the* four *sets:*

$$R_1 \subseteq (Val_{L_1} \times Val_S \times Val_{A \cap L_2}),$$
$$R_2 \subseteq (Val_{L_2} \times Val_S \times Val_{A \cap L_1}),$$
$$Guar_1, Guar_2 \subseteq (Val_S \times Val_A \times Val_S \times Val_A),$$

which are defined as the least *sets that satisfy the following conditions:*

a) Initialization:

- *R_1 contains the set $\{(l_1, s, t) \mid l_1 \cup s \cup t \in Init\downarrow (L_1 \cup A \cup S)\}$.*
- *R_2 contains the set $\{(l_2, s, t) \mid l_2 \cup s \cup t \in Init\downarrow (L_2 \cup A \cup S)\}$.*

***b) Transitions of** P_1**:** If $(l_1, s, t) \in R_1$ and $\delta_1(l_1, s, l_1', s')$ holds, then*

- ***Local update:*** $(l_1', s', t) \in R_1$.
- ***Update to guarantee:*** $(s, l_1\downarrow A \cup t, s', l_1'\downarrow A \cup t) \in Guar_1$.

***c) Transitions of** P_2**:** If $(l_2, s, t) \in R_2$ and $\delta_2(l_2, s, l_2', s')$ holds, then*

- ***Local update:*** $(l_2', s', t) \in R_2$
- ***Update to guarantee:*** $(s, l_2\downarrow A \cup t, s', l_2'\downarrow A \cup t) \in Guar_2$.

d) Interference:

- *If $(l_1, s, t) \in R_1$ and $(s, l_1\downarrow A \ \cup \ t, s', t') \in Guar_2$, then $(l_1, s', t'\downarrow L_2) \in R_1$.*
- *If $(l_2, s, t) \in R_2$ and $(s, l_2\downarrow A \ \cup \ t, s', t') \in Guar_1$, then $(l_2, s', t'\downarrow L_1) \in R_2$.*

The set of reachable states *according to the* non-standard compositional semantics *with respect to the set of auxiliary variables A is defined as*

$$Reach_A = \{(l_1, s, l_2) \mid (l_1, s, l_2\downarrow A) \in R_1 \text{ and } (l_2, s, l_1\downarrow A) \in R_2\}. \qquad \square$$

Intuitively, under the compositional semantics, we track independently the view of P_1 (and P_2) using valuations of its local variables, shared variables, and the subset of the other process's local variables declared to be auxiliary (using the sets R_1 and R_2). Furthermore, we keep the set of *guarantee* transition-relations $Guar_1$ and $Guar_2$ that summarize what transitions P_1 and P_2 can take, but restricted to the auxiliary and shared variables only. The guarantee-relation of P_1 is used to update the view of P_2 (i.e. R_2), and vice versa. The crucial aspect of the definition above is that it *ignores* the correlation between local variables of P_1 and P_2 that are not defined to be auxiliary variables. The computation of $P_1||_A P_2$ hence proceeds mostly locally, with updates using the guarantee relation of the other process (which affects shared and auxiliary variables only), and is combined in the end to get the set of globally reachable configurations.

It is not hard to see that $Reach \subseteq Reach_A$, for any A. Hence, the compositional semantics is an *over-approximation* of the set of reachable states of the program, and proving that a program is safe under the compositional semantics is sufficient to prove that the program is safe. Moreover, when the auxiliary variables include all local variables (including the program counter and local call stack), the compositional semantics coincides with the natural semantics.

The above definitions and rules can be generalized to k processes running in parallel, and we can define the compositional semantics $P_1||_A P_2||_A \ldots ||_A P_n$ where A is subset of local variables of each process.

The rely-guarantee proof method of Jones:

The rely-guarantee method of Jones [Jon83] essentially builds compositional rely-guarantee *proofs* using a similar abstraction. Given *sequential programs* P_1 and P_2, and a pre-condition *pre* and a post-condition *post* for $P_1||P_2$, the rely-guarantee proof technique over a set of auxiliary variables A involves providing a pair of tuples, $(pre_1, post_1, rely_1, guar_1)$ and $(pre_2, post_2, rely_2, guar_2)$, where pre_1, $post_1$, pre_2, and $post_2$ are unary predicates defining subsets of states, and $rely_1$, $guar_1$, $rely_2$, $guar_2$ are binary relations defining transformations of the shared variables and the auxiliary variables A. The meaning of the tuple for P_1 is that, when P_1 is started with a state satisfying pre_1 and in an environment that could change the auxiliary variables and shared variables allowed by $rely_1$, P_1 would make transitions that accord to $guar_1$, and if it

terminates, will satisfy $post_1$ at the exit. An analogous meaning holds for P_2. Note that $rely_1, rely_2, guar_1$ and $guar_2$ are defined over shared variables and the auxiliary variables. The programs P_1 and P_2 are proved to satisfy these conditions using a *local* proof by considering each P_i interacting with a general environment satisfying $rely_i$; in particular invariants of P_i needed to establish the Hoare-style proof of P_i should be *invariant* or stable with respect to $rely_i$.

The following proof rule can then be used to prove partial correctness of $P_1||P_2$:

$$\frac{\begin{array}{cc} guar_1 \Rightarrow rely_2, & guar_2 \Rightarrow rely_1, \\ P \models (pre, post_1, rely_1, guar_1), & Q \models (pre, post_2, rely_2, guar_2) \end{array}}{P||Q \models (pre, post_1 \wedge post_2)}$$

The rely-guarantee method works also for *nested* parallellism compositionally; see [Jon83, XdRH97] for details.

It is easy to see that a compositional rely-guarantee proof of $P_1||P_2$, over a set of auxiliary variables A, is really a proof that the compositional semantics of $P_1||_A P_2$ is correct. Note that if $P_1||_A P_2$ is correct, it does not imply a rely-guarantee proof exists, however, as proofs have limitations of the logical syntax used to write the rely and guarantee conditions, and hence do not always exist.

The main result:

We can now state the main result of this chapter. We show that, given a parallel composition of sequential programs $P_1||P_2||\ldots||P_n$ with assertions, and a set of auxiliary variables A, we can build a sequential program S with assertions such that S has the following properties:

- At any point, the scope of S contains at most one copy of the local variables of a *single* process P_i, three copies of the auxiliary variables, and at most three copies of the shared variables.
- The compositional semantics of $P_1||_A P_2||_A \ldots||_A P_n$ with respect to the auxiliary variables A satisfies its assertions iff S satisfies its assertions.
- If S satisfies its assertions, then $P_1||P_2||\ldots||P_n$ also satisfies its assertions.

The first remark above says that the sequentialized program has less variables in scope than the naive *product* of the individual processes; the sequentialization intuitively simulates the processes *separately*, keeping track of only an extra copy of auxiliary variables and shared variables. Second, the sequentialization is a *precise* reduction of the verification problem, provided the concurrent program can be proved compositionally (i.e. if the auxiliary variables are sufficient to make the compositional semantics of the program be assertion-failure free). Finally, the sequential program is an over-approximation of the behaviors of the parallel program for *any* set of auxiliary variables, and hence proving it correct proves the parallel program correct.

The above result will be formalized in the sequel (see Theorem 1) for a class of parallel programs that has sequential recursive functions, but with no thread creation or dynamic memory allocation (the result can be extended to dynamic data-structures but will require mechanisms to cache heap-structures and compare them for equality). Our main theorem hence states that any parallel program that is amenable to compositional reasoning can be sequentialized, where the number of new variables added in the sequentialization grows with the number of auxiliary variables required to prove the program correct. We utilize the sequentialization result in one verification context, namely deductive verification, to build a compositional deductive verification tool for concurrent programs using the sequential verifier Boogie.

7.2 A high level overview of the sequentialization

In this section, we give a brief overview of our sequentialization. For ease of explanation, let us consider a concurrent program consisting of two processes P_1 and P_2. Let A be the set of auxiliary variables and assume that the compositional semantics of the concurrent program is correct with respect to A.

Assume we had functions $G_1(s^*, a^*)$ (and $G_2(s^*, a^*)$) that *somehow* takes a shared and auxiliary state (s^*, a^*) and non-deterministically returns all states (s, a) such that $(s^*, a^*, s, a) \in Guar_1$ (respectively $Guar_2$), where $Guar_1$ and $Guar_2$ are guarantees for the processes as defined in Definition 1. Then we could write a function that computes the states that P_1 can reach in

accordance with the compositional semantics (i.e. R_1 in Definition 1) using the following code:

```
while(*) {
  if (*) then
    <<simulate a transition of P1>>
  else
    (s,a) := G2(s,a);
  fi
}
return (s,a);
```

In other words, we could write a sequential program that returns precisely the states in R_1, by interleaving simulations of P_1 with calls to G_2 to compute interference according to $Guar_2$ (see Definition 1). We can similarly implement the sequential code that explores R_2 using calls to G_1.

Note that on two *successive* calls to $G_2()$, there is no preservation of the local states of P_2, *except its variables declared to be auxiliary*. However, we do *not* have to preserve the exact local state of P_2 as we are not simulating the natural semantics of the program, but only its compositional semantics with respect to auxiliary variables A. This is the crux of the argument as to why we can sequentially compute R_1 without simultaneously tracking all the local variables of P_2.

Now, turning to the function G_2 (and G_1), consider Definition 1 again, and notice that, given (s^*, a^*), in order to compute (s, a) such that $(s^*, a^*, s, a) \in Guar_2$, we must essentially be able to find a local state l_2 such that $(l_2, s^*, a^* \downarrow L_1) \in R_2$ where $l_2 \downarrow A = a^* \downarrow L_2$, and then we can take its transitive closure with respect to δ_2. We can hence write G_2 using the following sequential code:

```
G2(s*,a*) {
   <<initialize variables of P2>>
   while(*) {
       if (*) then
          <<simulate a transition of P2>>
       else
          (s,a) := G1(s,a);
       fi
   }
   assume (local and shared state is consistent with s*,a*);
   while(*) {
        <<simulate a transition of P2>>
```

```
    }
    return (s,a);
}
```

Intuitively, G_2 starts with the *initial* state of P_2 and sets about recomputing a state (l_2, s_2) that is compatible with its given input (s^*, a^*) (i.e. with $s_2 = s^*$ and $l_2 \downarrow A = a^* \downarrow L_2$). It does this by essentially running the code for R_2 (i.e. by simulating P_2 and calling G_1). Once it has found such a state, it simulates P_2 for a while longer, and returns the resulting state.

We hence get four procedures that compute R_1, R_2, $Guar_1$ and $Guar_2$, respectively, with mutually recursive calls between the functions computing $Guar_1$ and $Guar_2$. The correctness of the sequential programs follow readily from Definition 1, as it is a direct encoding of that computation. Our sequentialization transformation essentially creates these functions G_1 and G_2. However, since our program cannot have statements like "simulate a transition of P2", we perform a syntactic transformation of the concurrent code into a sequential code, where control code is inserted between statements of the concurrent program in order to define the functions G_1 and G_2. This complication combined with the handling of recursive functions makes the translation quite involved; however, the above explanation captures the crux of the construction.

7.3 Sequential and concurrent programs

Our language for concurrent programs consists of a parallel composition of recursive sequential programs. Variables in our programs are defined over integer and Boolean domains. The syntax of programs is defined by the following grammar:

$$
\begin{array}{lcl}
\langle \textit{conc-pgm} \rangle & ::= & \langle \textit{decl} \rangle^* \langle \textit{pgm-list} \rangle \\
\langle \textit{pgm-list} \rangle & ::= & \langle \textit{pgm-list} \rangle \;||\; \langle \textit{pgm-list} \rangle \mid \langle \textit{pgm} \rangle \\
\langle \textit{pgm} \rangle & ::= & \langle \textit{decl} \rangle^* \langle \textit{proc} \rangle^* \\
\langle \textit{proc} \rangle & ::= & f(\overline{x})\ \texttt{begin}\ \langle \textit{decl} \rangle^* \langle \textit{stmt} \rangle\ \texttt{end} \\
\langle \textit{stmt} \rangle & ::= & \langle \textit{stmt} \rangle; \langle \textit{stmt} \rangle \mid \texttt{skip} \mid \overline{x} := \textit{expr}(\overline{x}) \mid \overline{x} := f(\overline{y}) \mid \\
 & & f(\overline{x}) \mid \texttt{return}\ \overline{x} \mid \texttt{assume}\ \textit{b-expr} \mid \texttt{assert}\ \textit{b-expr} \mid \\
 & & \texttt{if}\ \textit{b-expr}\ \texttt{then}\ \langle \textit{stmt} \rangle\ \texttt{else}\ \langle \textit{stmt} \rangle\ \texttt{fi} \mid \\
 & & \texttt{while}\ \textit{b-expr}\ \texttt{do}\ \langle \textit{stmt} \rangle\ \texttt{od} \mid \texttt{atomic}\ \{\langle \textit{stmt} \rangle\} \\
\langle \textit{decl} \rangle & ::= & \texttt{int}\ \langle \textit{var-list} \rangle; \mid \texttt{bool}\ \langle \textit{var-list} \rangle; \\
\langle \textit{var-list} \rangle & ::= & \langle \textit{var-list} \rangle, \langle \textit{var-list} \rangle \mid \langle \textit{literal} \rangle
\end{array}
$$

A concurrent program consists of k sequential program components $P_1 ... P_k$ (for some k) communicating with each other through shared variables $\mathcal{S}$. These shared variables are declared in the beginning of the concurrent program (we assume integer variables are initialized to 0 and Boolean variables to *false*). Each sequential component consists of a procedure called *main* and a list of other procedures. The control flow for all sequential components P_i starts in the corresponding *main* procedure, which we call $main_i$. The *main* for all sequential components has *zero* arguments and no return value.

Each procedure is a declaration of local variables followed by a sequence of statements, where statements can be simultaneous assignments, function calls (call-by-value) that take in multiple parameters and return multiple values, conditionals, while loops, assumes, asserts, atomic, and return statements. In the above syntax, $\overline{x}$ represents a vector of variables. We allow non-determinism in our programs; boolean constants are *true*, *false* and $*$, where $*$ evaluates non-deterministically to *true* or *false*.

The safety specifications for both concurrent and sequential programs are expressed in our language as assert statements. The semantics of an assume statement is slightly different. If the value of the boolean expression (*b-expr*) evaluates to *true*, then the assume behaves like a skip. Otherwise, if the boolean expression evaluates to *false*, the program silently terminates. Synchronization and atomicity are achieved by the *atomic* construct. All the statements enclosed in the atomic block are executed without any interference by the other processes. Locks can be simulated in our syntax by modeling a lock l as an integer variable `l` and by modeling P_i acquiring l using the code:

```
atomic { assume(l=0); l := i;}
```

and modeling the release with the code:

```
atomic { if (l=i) then l := 0;}
```

We assume programs do not have nested atomic blocks.

The syntax of sequential programs is the same as the syntax of concurrent programs except that we disallow the parallel composition operator || and the *atomic* construct.

7.4 The Sequentialization

In this section, we describe our sequentialization for concurrent programs and argue its correctness.

Let us fix a concurrent program with shared variables S and auxiliary variables A; we assume auxiliary variables are *global* in each thread P_i. Let the concurrent program be composed of k sequential components.

The sequential program corresponding to the concurrent program will have a new function *main*, and additionally, as explained in Section 7.2, will have a procedure G_i for each sequential component P_i of the concurrent program that semantically captures the guarantee $Guar_i$ of P_i. The procedure G_i takes a shared state ($\overline{s^*}$) and auxiliary state ($\overline{a^*}$) as input and returns ($\overline{s}, \overline{a}$) such that ($\overline{s^*}, \overline{a^*}, \overline{s}, \overline{a}$) $\in Guar_i$. Finally, each $G_i()$ is formed using procedures that are obtained by transforming the process P_i (using the function $\tau_i[]$ shown below that essentially *inserts* the interference code $\mathcal{I}_i$ shown in Figure 7.1 between the statements of P_i).

The shared variables and auxiliary variables are modeled as global variables in the sequential program. Furthermore, we have an extra copy of the shared and auxiliary variables ($\overline{s^*}$ and $\overline{a^*}$) that are used to pass shared and auxiliary states between the processes $G_i()$. We also have a copy of shared and auxiliary variables ($\overline{s'}$ and $\overline{a'}$) that are declared to be *local* in each procedure to store a shared and auxiliary state and restore it after a call to a function $G_j()$ to compute interference. Besides these, the sequential program also uses global Boolean variables z and *term*; intuitively, z is used to keep track of when the shared and auxiliary state $\overline{s^*}$ and $\overline{a^*}$ has been reached and *term* (for *terminate*) is used to signal that $G_i()$ has finished computing and wants to return the value.

- Global variable declarations are:

```
// insert declaration for s̄, ā, s̄*, ā* as global variables
decl bool term, z;
```

- The function main() is defined as:

```
main() begin G_1() end
```

- Each function $G_i()$ is defined as below:

```
G_i() begin
   z := false;  term := false;
   s̄* := s̄;  ā* := ā;
   // insert code to initialize s̄, ā
   main_i();
   assume (term = true);
   return
end
```

- The function τ_i that transforms the program for P_i is defined as:
  - $\tau_i[\texttt{f}(\overline{\texttt{x}})\ \texttt{begin}\ \mathit{decl}\ \mathit{stmt}\ \texttt{end}] =$

```
f(x̄) begin decl
     // insert declaration of s̄', ā' as local variables.
     τ_i[stmt]
end
```

  - $\tau_i[\mathcal{S}_1;\ \mathcal{S}_2] = \tau_i[\mathcal{S}_1];\ \tau_i[\mathcal{S}_2]$
  - $\tau_i[\mathcal{S}] = \mathcal{I}_i;\ \mathcal{S}$ where $\mathcal{S}$ is an assignment, skip statement, assume statement, assert statement, a function call or a return statement.
  - $\tau_i[\texttt{while}\ b\text{-}\mathit{expr}\ \texttt{do}\ \mathcal{S}\ \texttt{od}] = \mathcal{I}_i;\ \texttt{while}\ b\text{-}\mathit{expr}\ \texttt{do}\ \tau_i[\mathcal{S}];\ \mathcal{I}_i\ \texttt{od}$
  - $\tau_i[\texttt{if}\ b\text{-}\mathit{expr}\ \texttt{then}\ \mathcal{S}_1\ \texttt{else}\ \mathcal{S}_2\ \texttt{fi}] =$
    $\mathcal{I}_i;\ \texttt{if}\ b\text{-}\mathit{expr}\ \texttt{then}\ \tau_i[\mathcal{S}_1]\ \texttt{else}\ \tau_i[\mathcal{S}_2]\ \texttt{fi}$
  - $\tau_i[\texttt{atomic}\ \{\mathcal{S}\}] = \mathcal{I}_i;\ \mathcal{S}$

The procedure *main* in the sequential program simply calls the method G_1. The procedure G_i is obtained from the corresponding program component P_i by a simple transformation. At a high level, this procedure first copies the incoming shared and auxiliary state into the variables $\overline{s^*}$ and $\overline{a^*}$. It then computes a local state of P_i which is consistent with the state $(\overline{s^*}, \overline{a^*})$ (at which point z turns to *true*), and then non-deterministically simulates the

transitions of P_i from this local state, to return a reachable shared state $\overline{s}$ and auxiliary state $\overline{a}$. Every time G_i is called, it starts from its initial state, and simulates P_i, interleaving it with the control code $\mathcal{I}_i$ given in Figure 7.1.

The interference code $\mathcal{I}_i$ (Figure 7.1) keeps track of whether the incoming state $(\overline{s^*}, \overline{a^*})$ has been reached through a boolean variable z which is initialized to *false*. If z is *false* (i.e. the state $(\overline{s^*}, \overline{a^*})$ has not been reached), then before any transition of P_i, the control code can non-determinsitically choose to invoke its environment (in doing so, in order to preserve its input $\overline{s^*}, \overline{a^*}$, it stores them in a local state and restores them after the call returns and restores its variables z and *term* to *false*).

When the state $(\overline{s^*}, \overline{a^*})$ is reached, z can be non-deterministically set to *true*, from which point no interference code G_j can be called, and only local computation proceeds, till non-deterministically the program decides to terminate by setting *term* to *true*. Once *term* is *true*, the code pops the control-stack all the way back to reach the function G_i which then returns to its caller, returning the new state in $(\overline{s}, \overline{a})$. Note that the state $z = false$, $term = false$ corresponds to the first while loop in the code for the guarantee G_2 in Section 7.2. Similarly, setting *term* to *true* corresponds to the termination of the second while loop. We conclude this section by stating our main theorem:

```
if(term = true) then return fi
if(!z & *) then
  while(*) do
    // call G_1
    if(*) then
          s' := s*;   a' := a*;
          G_1();
          z := false;
          term := false;
          s* := s';   a* := a'
    fi
    Similarly call G_2 ... G_k except G_i
  od
fi
if(!z & s = s* & a = a* & *) then
  z := true fi
if(z & *) then
  term := true; return
fi
```

Figure 7.1: The interference control code $\mathcal{I}_i$

Theorem 7.4.1. *Let C be a concurrent program with auxiliary variables A (assumed global), and let $S_{C,A}$ be its sequentialization with respect to A. Then the compositional semantics of C with respect to the auxiliary variables A has no reachable state violating any of its assertions iff $S_{C,A}$ violates none of its assertions.* □

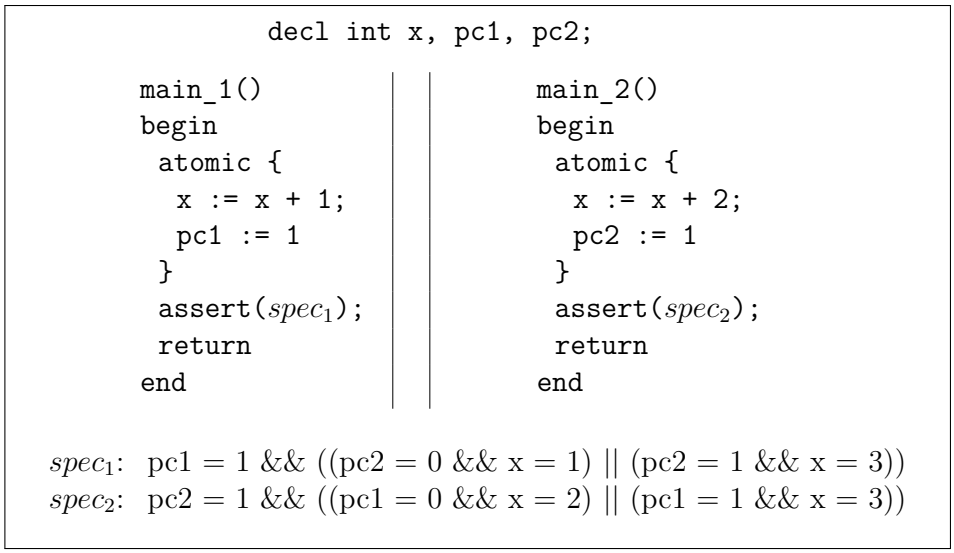


Figure 7.2: An example program

Illustration of the sequentialization:

Figure 7.2 shows a concurrent program consisting of two threads, say P_1 and P_2. The program consists of a shared variable x whose initial value is *zero*. Both the threads *atomically* increment the value of x. Let $A = \{pc1, pc2\}$ be the auxiliary variables capturing the control position in the respective processes and let the initial value of these variables also be *zero*. In general, new auxiliary variables may be needed for performing compositional proofs; these new variables are *written to* but never read from in the program, and hence do not affect the semantics of the original program; see [Jon83, OG76].

It can be easily seen that the compositional semantics of this program with respect to the auxiliary variables A is correct. Figure 7.3 shows the sequential program obtained from the sequentialization of this concurrent program with respect to these auxiliary variables. Our result allows us to verify the concurrent program in Figure 7.2 by verifying the correctness of its sequentialization with respect to the auxiliary variables A, shown in Figure 7.3.

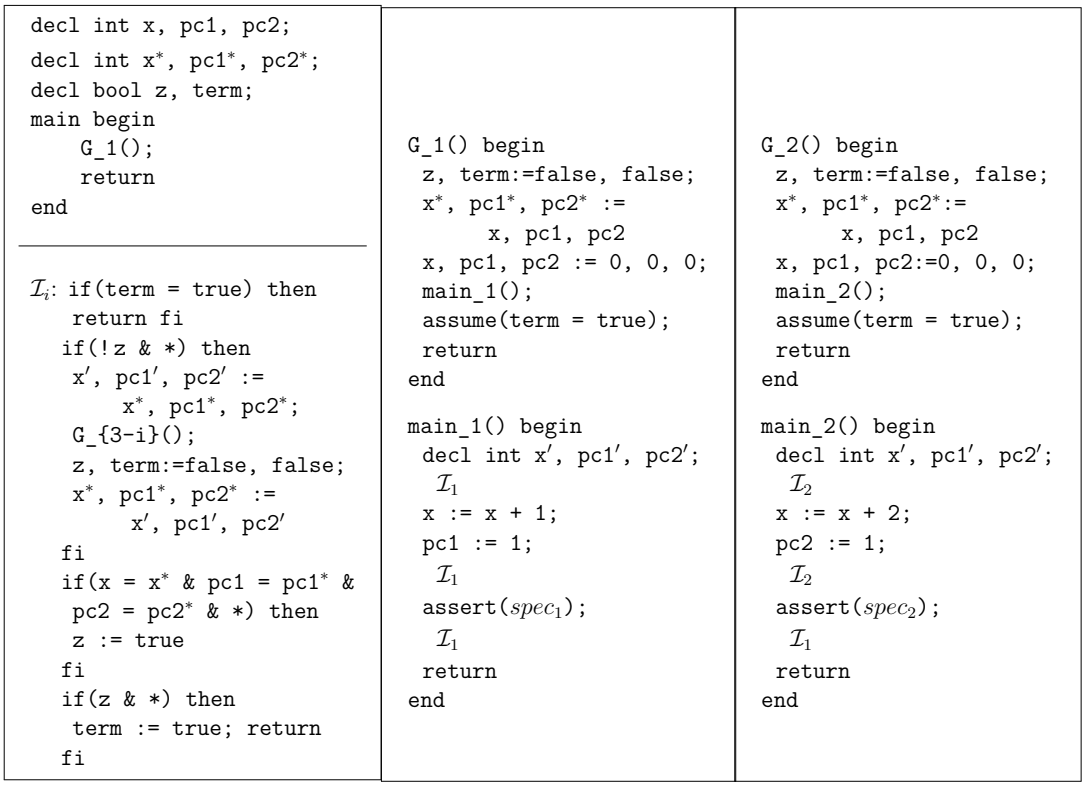


Figure 7.3: The sequential program obtained from sequentializing the concurrent program in Figure 7.2.

7.5 Experience

The sequentialization used in this chapter can be used to verify a concurrent program using *any* sequential verification tool. This includes tools based on abstract interpretation and predicate abstraction, those based on bounded model-checking, those based on deductive-verification based extended static checking as well as those based on invariant synthesis using learning (developed in this thesis).

Deductive verification: We used the sequentialization for proving concurrent programs using deductive verification. Given a concurrent program and its Jones-style rely-guarantee proof annotations (pre, post, rely, guar, and loop-invariants), we sequentialized it with respect to the auxiliary variables and syntactically transformed the user-provided proof annotations to obtain the proof annotations (pre-conditions and post-conditions of methods and loop invariants) of the sequential program. In general, the pre-condition of method G_i asserts that "$term = false$" and its post-condition asserts that the

guarantee $guar_i$ is true across the function (if *term* is *true*). Furthermore, the pre-conditions and post-conditions of every function in P_i gets translated to pre-conditions and post-conditions in its sequentialization with the extra condition that $guar_i$ is *true* when *term* is equal to *true*.

These annotated sequential programs were fed to the sequential verifier BOOGIE that generates verification conditions that are in turn solved by an SMT solver (Z3 in this case). If the sequential program is proved correct, it proves the correctness of the original concurrent program. Note that though similar static extended checking techniques are known for the rely-guarantee method [FFQ02], our technique allows one to use just a sequential verifier like BOOGIE to prove the program correct, and requires no other decision problems to be solved (the checking in [FFQ02], for example, requires a separate call to a theorem prover to check guarantees are reflexive and transitive, etc.)

We used this technique to prove correct the following set of concurrent programs: X++ (Figure 7.2), Lock [FQ03], Peterson's mutual exclusion algorithm, the Bakery protocol, ArrayIndexSearch [Jon83], GCD [Fen09], and a simplified version of a Windows NT Bluetooth driver. Lock is a simple example program consisting of two threads that modify a shared variable after acquiring a lock; the safety condition in the example asserts that these modifications cannot occur concurrently. The program ArrayIndexSearch finds the *least index* of an array such that the value at that index satisfies a given predicate, and consists of two threads, one that searches odd indices and the other that searches even indices, and communicate on a shared variable *index* that is always kept updated to the current least index value. GCD is a concurrent version of Euclid's algorithm for computing the greatest common divisor of any two numbers; here the two concurrent threads update the pair of integers.

The Windows NT bluetooth driver is a *parameterized program* (i.e. has an unbounded number of threads). It consists of two types of threads: there is one stopper-thread and an unbounded number of adder-threads. A stopper calls a procedure to halt the driver, while an adder calls a procedure to perform I/O in the driver. The I/O is successfully handled if the driver is not stopped while it executes. The program, though small, has an intricate global invariant that requires a shared variable to reflect the number of active adder threads.

Programs	Concurrent pgm			Sequential pgm		
	#Threads	#LOC	#Lines of annotations	#LOC	#Lines of annotations	Time
X++	2	38	5	113	6	8s
Lock	2	50	9	184	10	122s
Peterson	2	52	35	232	36	145s
Bakery	2	55	8	147	13	18s
ArrayIndexSearch	2	74	17	222	21	126s
GCD	2	78	23	279	29	869s
Bluetooth	unbdd	69	20	276	55	107s

Table 7.1: Experimental Results. Evaluated on Intel dual-core 1.6GHz, 1Gb RAM.

Table 1 gives the experimental results[1]. For each program, we report the number of threads in the concurrent program, the number of lines of code in the concurrent program and its sequentialization, the number of lines of annotations in both the concurrent program (which includes rely/guarantee annotations and loop invariants) and its sequentialization, and the time taken by BOOGIE to verify the sequentialized program.

BOOGIE was able to verify the correctness of all our programs. All these programs except the Windows NT bluetooth driver consist of two threads and are sequentialized as detailed in Section 5. The Bluetooth driver is an example of a parameterized program running any number of instances of the adder threads. In our sequentialization, we model the environment consisting of all the adder threads together with a single procedure. If we keep track of the number of adders at a particular program location (counter abstraction [Lub84]) and expose these auxiliary variables, it turns out that the device driver can be proved correct under compositional semantics. We used this rely-guarantee proof, sequentialized the program, and used BOOGIE to prove the Bluetooth driver correct in its full generality.

Predicate abstraction: We have also used our sequentialization followed by the predicate-abstraction tool SLAM [BR02] to prove programs automatically correct. In this case, we need no annotations and just the set of auxiliary variables. We were able to automatically prove the correctness of the programs X++, Lock, Peterson and the Bakery protocol, in negligible time. Since the sequentialization covered in this chapter is not tied to any

[1]Experiments available at http://www.cs.uiuc.edu/~garg11/tacas11

verification technique, instead of predicate abstraction, we can also learn rely-guarantee proofs by learning pre/post-conditions and loop invariants for the obtained sequential program using algorithms we have developed earlier in this thesis.

7.6 Related Work

Thread-modular verification [FFQ02, FQ03] is in fact precisely the same as compositional verification á la Jones, but has been adapted to both model-checking [FQ03] and extended static checking [FFQ02]. Our result can be hence seen as showing how thread-modular verification of concurrent programs can be reduced to pure sequential verification. There has also been work on using counter-example guided predicate-abstraction and refinement for rely-guarantee reasoning [CN07], and building rely-guarantee interfaces using *learning* [CGP03, AMN05].

CHAPTER 8

VERIFYING ASYNCHRONOUS PROGRAMS BY SYNTHESIZING ALMOST-SYNCHRONOUS INVARIANTS

Writing correct asynchronous event-driven programs, which involve concurrently evolving components communicating using messages and reacting to input events, is difficult. One approach to testing and verification of such programs is using model-checking, where the state-space of the program (or the program coupled with a test harness) is explored systematically. State-space explosion occurs due to several reasons— explosion of the underlying data-space domain, explosion due to the myriad interleavings caused due to concurrency, and explosion due to the unbounded message buffers used for communication.

In this chapter, our aim is to build model-checking techniques that synthesize invariants to *provably* verify asynchronous event-driven programs against local assertions. In particular, we are interested in proving programs written in a recently proposed programming language P [DGJ+13], which is an actor-based programming language which provides abstractions that hide the underlying data and device manipulations, thus exposing the high-level protocol. Our primary concern is to tackle the *asynchrony* of message passing which causes unbounded message buffers. Our goal is to effectively and efficiently *prove* (as opposed to systematically test) event-driven programs correct, when the number of processes and the local data are bounded, but when message buffers are unbounded.

The classical approach to tackle state-space explosion when systematically testing concurrent programs using model-checking is *partial-order reduction* [God95, FG05]. A concurrent program's execution can be viewed as a partial order that captures causality between events. Local state reachability can then be checked by exploring only one linearization of every partial order, and partial-order techniques which give methods that explore one (or a few) of these linearizations per partial order can result in considerable savings.

In the setting where we want to prove protocols correct using model-

checking, the key criterion to achieve termination is to detect cycles in the state-space. However, in message passing systems, global state-spaces are infinite, even when the local data domains and number of processes are bounded, as the message buffers get unbounded. Consider the simple scenario where a machine p sends a machine q unboundedly many messages, like in a producer-consumer setting. Even in this simple scenario, systematic model-checkers (based on partial-order reduction or otherwise) would fail to terminate checking local assertions, even when the local data stored at p and q is finite, since message buffers get unbounded [God95]. Consequently, techniques such as partial-order reduction do not typically help, as they are not aimed at exploring a finite subset of the infinite reachable state-space that can guarantee correctness. In this section, we aim to synthesize and explore such an adequate finite subset, which we call *almost-asynchronous invariants (ASI)*.

Almost-synchronous Invariants: Our primary thesis is that *almost-synchronous invariants* often suffice to prove asynchronous event-driven programs correct, and furthermore, a search for these invariants is also more effective in finding bugs. Intuitively, almost-synchronous states are those where the message buffers are close to empty, and almost-synchronous invariants are collections of such states that ensure that all local states have been discovered. For instance, in the producer-consumer example above, exploring the sends of p immediately followed by the receive in q discovers an almost-synchronous invariant where message buffers are bounded by one, though blindly exploring the state-space would never lead to termination.

The primary contribution of our work is a sound and complete reduction scheme that discovers almost-synchronous invariants using model-checking. The key idea is to explore interleavings that keep the message buffers small, while at the same time finding a closure argument that argues that all local states have been discovered, at which point we can terminate. Intuitively, for any partial order described by the system, we aim to "cover" this using a linearization that has small buffer sizes.

Verifying Asynchronous Event-driven Programs in P: One of our primary motivations in this work is to verify real-world asynchronous event-driven device-driver programs written in P against a property called *responsiveness*.

Asynchronous event-driven programs typically have layers of design, where the higher layers reason with how the various components (or machines) interact and the protocol they follow, and where lower layers manage more data-intensive computations, controlling local devices, etc. However, the programs often get written in traditional languages that offer no mechanisms to capture these abstractions, and hence over time leads to code where the individual layers are no longer discernible. High level protocols, though often first designed on paper using clean graphical state-machine abstractions, eventually get lost in code, and hence verification tools for such programs face the daunting task of extracting these models from the programs.

The natural solution to the above problem is to build a programming language for asynchronous event-driven programs that preserves the protocol abstractions in code. Apart from the difficulty in designing such a language, this problem is plagued by the reluctance of programmers to adopt a new language of programming and the discipline that it brings. However, this precise solution was pioneered in a new project at Microsoft Research recently, where, during the development of Windows 8, the team building the USB driver stack decided to use a domain-specific language for asynchronous event-driven programs called P [DGJ+13]. Programs written in P capture the high-level protocol using a collection of interacting state machines that communicate with each other by exchanging messages. The machines, internally, also have to do complex tasks such as process data and perform low level control of devices, reading sensors or controlling devices, etc., and these are modeled using external foreign functions written in C.

The salient aspect of P is that it is a programming paradigm where the protocol model and the lower level data and control are *simultaneously* expressed in the same language. P programs can be compiled to native code for execution, while the protocol model itself can be extracted cleanly from the code in order to help perform analysis, especially those relevant to finding errors in the protocol. Writing code in P gives immediate access to designers to correct errors found by analysis tools during the design phase itself, and significantly contributed to building a more reliable USB stack [DGJ+13]. Maintenance of the code in P automatically keeps these models up to date, enabling verification mechanisms to keep up with evolving code.

The primary specification that P programs are required to satisfy in [DGJ+13] is *responsiveness*. Each state in a P program declares the precise set of mes-

sages a machine can handle and the precise set of messages it will *defer*, implicitly asserting that all other messages are not expected by the designer to arrive when in this state. Receiving a message outside these sets hence signals an error, and in device drivers, often leads drivers to crash. The work reported in [DGJ+13] includes a *systematic testing* tool for the models using model-checking, where the system is explored for hundreds of thousands of states to check for errors. However, such model-checking seldom succeeds in proving the program correct, since there are many sources of infinity, including message buffer sizes.

In this work, we present a reduction based on almost-synchronous invariants for a model called event-driven automata, which closely resembles P programs (our algorithm can be easily adapted to other actor-based concurrency models as well). We show that our reduction can prove P programs correct, for arbitrary message buffer sizes. We also show that our reduction works *faster*, despite handling unbounded buffers, than the naive systematic exploration over reasonably bounded buffers. Our reduction also helps in finding bugs in incorrect P programs, exploring less states and performing faster than iterated depth-first search techniques. The high point of our experiments is the complete verification of the USB Windows Phone Driver, which our tool can prove responsive with no bound on message buffers, a proof that has hitherto been impossible to achieve using current model-checkers.

8.1 Motivation

The key idea of this work is that almost-synchronous invariants often suffice to find proofs of local assertions in event-driven asynchronous programs. Given an asynchronous program with local assertions, we would like to explore a set of reachable global states that covers all reachable local states. However, this set of global states need not be the set of all reachable global states (partial-order reduction [God95, FG05] also works this way; all global states are not explored, but all local states are covered).

Synchronous states, intuitively, is the set of global states where message buffers are empty. From the perspective of rely-guarantee reasoning [Jon83], when a machine p sends a message to machine q, p is not quite concerned with what the state of q is when the send-event happens, but rather is

concerned with the state of q when it receives/dequeues the message it sends, which is essentially what synchronous states capture. However, synchronous invariants (invariants containing synchronous states) may themselves not suffice to prove a system correct for two reasons: (a) in order to ensure that all synchronous states have been explored, we may need to explore asynchronous states (where message buffers are not empty), and (b) certain local states may manifest themselves only in asynchronous states. *Almost-synchronous invariants* are invariants of the system expressed using global states where message buffers are close to empty, but for which inductiveness of the invariant is provable and which covers all local states. The primary thesis of this work is that almost-synchronous invariants (ASI) often exist for event-driven asynchronous programs, and proofs that target finding such invariants can prove their correctness efficiently.

We will present, in Section 8.3, a reduction scheme (called *almost-synchronous reduction*) that will explore a selective set of interleavings that leads to the discovery of ASIs and simultaneously proves their inductiveness. The primary aim of the reduction is to explore interleavings that keep the message buffers to the minimal size needed, while still ensuring that all local states are eventually explored. The reduction will be *sound* and *complete*— all errors will be detected (if the search finishes) and all reported errors will be real errors.

At a very basic level, almost-synchronous reductions (presented in detail in Section 8.3) schedule *receive*-events whenever they are enabled, suppressing *send*-events. This rule ensures that messages are removed from message queues (which are FIFO and one per process) as soon as possible, thus ensuring message buffers are contained, and as we show in practice, often bounded. Moreover, this prioritization is sound as receive events that are enabled do not *conflict*, a la partial-order reduction [God95], with other receive or send events.

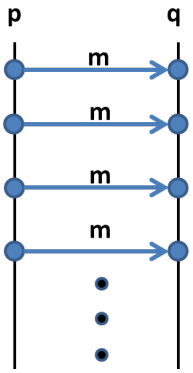


Figure 8.1: Producer-consumer Scenario

To appreciate this prioritization, consider the producer-consumer scenario on the right, where process p sends an unbounded number of messages to q, which q receives (p could do this by having a recurring state send out messages received by a recurring state of q). The reduction that we propose

will explore this scenario (partial-order) using the linearization consisting of an unbounded number of rounds, where in each round p sends to q followed by q immediately receiving the message from p, thus exploring an essentially synchronous interleaving where the message buffer is bounded by *one*. Furthermore, and very importantly, when exploring this interleaving, the search will discover that the global state repeats, which includes the local states of all machines and the contents of all message buffers. This is entirely because the message buffer gets constantly depleted causing the global state to recur.

Similarities and Differences with Partial-order Reduction: Note that techniques such as *partial-order reduction* [God95, FG05] do not necessarily help in this scenario. Even an optimal static or dynamic partial-order reduction that promises to explore every partial-order using just one linearization, cannot assuredly help. In the above example, if the linearization chosen is the one where the sends from p are all explored first (or a large number of them are explored) before the corresponding receives are explored, then each global state along this execution would be *different* because the message buffer content is different in each step. This turns out to be true for both depth-first and breadth-first searches with partial-order reduction. Note that this problem does not, in general, arise when systems communicate through bounded shared memory only; it is message-passing that causes the problem.

Partial-order reduction techniques are targeted to reducing the number of interleavings of every partially ordered execution explored, but are not aimed at choosing the interleavings explored carefully so as to reduce the global configuration, in particular the size of message buffers. Our almost-asynchronous reduction, on the other hand, chooses interleavings that reduce message-buffer sizes.

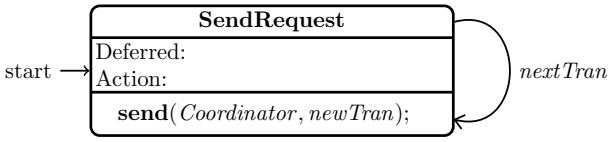
Despite the above differences, our reduction has many similarities to partial-order reductions. In particular, the *proof* that our reduction is sound and complete in discovering all local states is similar to the corresponding proofs for partial-order reduction— we show that for every execution that reaches a local state, there is another execution within our reduced system that is equivalent (respects the same partial order) and hence reaches the same local state.

As in the scenario of Figure 8.1, if a system readily always presents synchronous events (all sends enabled always have the matching receive events

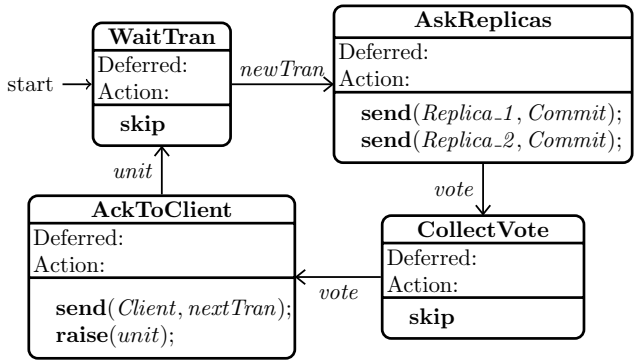
immediately enabled in the receiving process), one can solely explore the executions with synchronous events only and keep the sum of all message buffer sizes to one. While this often happens, it does not typically happen all the time in a system's evolution, which is why we need almost-synchronous global states to be explored.

In Section 8.3, we introduce the concept of *destination sets* to determine which events need to be scheduled in our reduction, in a given system configuration. A destination set is a subset of processes that is defined for every system configuration. For configurations in which none of the receive events are enabled, we construct its destination set and explore only those events that send messages to a process in the destination set. Destination sets are defined in a manner such that the above exploration covers all local states reached along linearizations that involve events which send messages to processes in the destination set. However, any form of selective exploration, such as the one we propose above using destination sets, is, in general, unfair and can completely miss local states that are reached along linearizations that involve only the events that have not been explored [God95, Val91]. To ensure that our search is not unfair with respect to some linearizations, our reduction also enables a transition that blocks processes whose events were chosen to be selectively explored. The use of blocked processes is a unique aspect of our reduction, and crucially relies on the semantics of message passing. A generic reduction technique, such as partial-order reduction, which works by handling shared memory and message passing uniformly cannot achieve such reductions, as what we do strays away from the normal semantics of transitions on the global state. In other words, we are *under-approximating* the global state description itself, while preserving soundness and completeness.

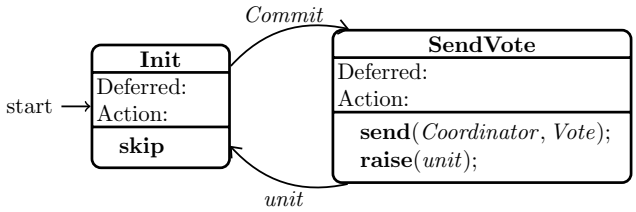
A simple P program: Figure 8.5 presents a toy example in P, that implements the distributed commit protocol. The system consists of a *Client* machine, a *Coordinator* and two *Replica* machines. The client sends new transaction (*newTran*) requests to the coordinator machine. The coordinator machine dequeues these requests and processes them by coordinating with the replicas in the system. It does so by sending *Commit* requests to the replica machines and waiting for a *Vote* from them. Once the coordinator has received votes from both the replicas, it sends a *nextTran* message back to the client. Only after receiving this message can the client send a new transaction



(a) Client Machine



(b) Coordinator Machine



(c) Replica Machine

Figure 8.2: Distributed Commit Protocol in P

to the coordinator. In this way, the protocol ensures that the client machine sends a *newTran* request only after the coordinator has finished processing the previous transaction. Note that the set of messages deferred in any state of the above P program is empty; our reduction can also handle P programs with non-empty deferred sets.

Figure 8.3 shows the queuing architecture for the distributed commit protocol indicating the communication pattern amongst the machines in the system. An edge from p to q represents that p is a potential sender of q. In this particular example, the client can only send a message to the coordinator; the coordinator sends a message to the client and to both the replicas, and the replicas in turn send messages back to the coordinator. In section 8.3, we use this example to describe our reduction algorithm.

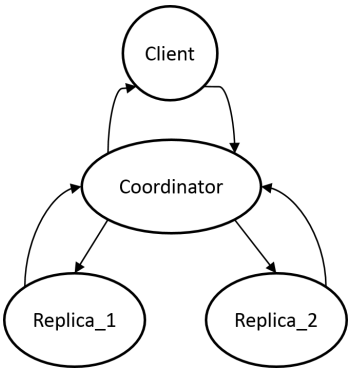


Figure 8.3: The queuing architecture for the Distributed Commit Protocol in Figure 8.5.

8.2 Event-driven Automata Model

In this section we introduce an automaton model, called event-driven automata (EDA), for modeling event-driven programs, inspired by and very similar to P programs. Event-driven automata are however a lot simpler, allowing us to define the reductions and prove precise theorems about them. We will then lift the reductions to general programs, including programs written in P (see Section 8.4).

In our automaton model, a program is a finite collection of state machines communicating via messages. Each state machine is a collection of states, has local variables and has a set of actions. Each machine also has a single FIFO queue into which other machines can enqueue messages. We will not restrict any of the sets (states, domain of local variables, payload on messages, etc.) to be finite; all of them can be infinite, and hence our automata can model event-driven software. For instance, P programs allow function calls in local machines; these can be modeled in our automata using an appropriate encoding of the call-stack in the state. Also, for simplicity, we will assume there is no process/machine creation; our reduction does extend to this setting, but for simplicity we explain our algorithms without these complications. Section 8.4 describes how we extend our algorithms to work on general P programs.

A message is modeled as a pair $\pi = (m, l)$, consisting of a message type m (from a finite set) and an associated payload l belonging to some (finite or infinite) domain. Let M be a finite set of message types and let Dom_M be the payload domain. Then, a message π belongs to $\Pi = M \times Dom_M$. We fix M, Dom_M, and Π for the rest of this chapter.

Let Dom be the domain for the local variables in the state machines. Without loss in generality, we assume that each machine in the program has a single local variable, and fix Dom for the rest of this chapter. Also let us denote $f^{\mathcal{T}}$ to be the class of all (computable) functions of type $\mathcal{T}$.

Event-driven automata (EDA): An *event-driven automaton* over $\Pi = (M \times Dom_M)$ and Dom is a tuple $\mathcal{P} = (\{P_i\}_{i \in N})$, where $N = \{1, \cdots, n\}$, $n \in \mathbb{N}$, and each $P_i = (Q_i^s, Q_i^r, Q_i^{int}, q_i^0, val_i^0, T_i, Def_i, q_i^{err})$, where

- $Q_i = Q_i^s \uplus Q_i^r \uplus Q_i^{int} \uplus \{q_i^{err}\}$ is the set of states, partitioned into states that send a message Q_i^s, states that receive messages Q_i^r, internal states Q_i^{int}, and an error state q_i^{err}.

- $q_i^0 \in Q_i^s \cup Q_i^r$ is the initial state of P_i;

- $val_i^0 \in Dom$ is the initial valuation for the local variable in P_i;

- T_i is the set of transitions for P_i and is partitioned into send transitions T_i^s, receive transitions T_i^r and internal transitions T_i^{int}.
  Send transitions are of the form:
  $T_i^s : Q_i^s \longrightarrow (Q_i^{int} \cup \{q_i^{err}\}) \times (N \backslash \{i\}) \times M \times f^{Dom \longrightarrow Dom_M}$
  Receive transitions are of the form:
  $T_i^r : Q_i^r \times M \longrightarrow (Q_i^{int} \cup \{q_i^{err}\}) \times f^{Dom \times Dom_M \longrightarrow Dom}$
  Internal transitions are of the form:
  $T_i^{int} : Q_i^{int} \longrightarrow 2^{(Q_i^s \cup Q_i^r \cup \{q_i^{err}\}) \times f^{Dom \longrightarrow Dom}}$;

- $Def_i : Q_i^r \longrightarrow 2^M$ associates a deferred set of messages to each receive state. □

When $T_i^s(q) = (q', j, m, f)$, this means that machine P_i, when in state q and local variable valuation v can transition to q', sending the message of type m with a payload $f(v)$ to machine P_j. Note that a machine cannot send messages to itself (we assume this mainly for technical convenience). Similarly, when $T_i^r(q, m) = (q', f)$, this means that P_i can receive message (m, l) when in state q and valuation v, and update its state to q' and local variable to $f(v, l)$. When $T_i^{int}(q)$ contains (q', f), it means that P_i can (non-deterministically) transition from state q and local variable valuation v to state q' and local valuation $f(v)$.

Note that, by definition, send transitions are deterministic, and receive transitions are deterministic for any received message; true local non-determinism is only present in internal transitions. Also note that every send or receive transition takes the control of the machine to an internal state and is immediately followed by an internal transition which non-deterministically transitions the machine to a send or a receive state. From the way we have defined transitions, the automaton can transition from any state to the error state q_i^{err} and from the error state, no further transitions are enabled.

As we noted above, in our automaton model, messages sent to a machine are stored in a FIFO queue. However, as in P programs, we allow the possibility to influence the order in which the messages are received by deferring them. In a given receive state q in machine P_i, some messages can be *deferred*, which is captured by the set $Def_i(q)$. When a machine is in this state, it skips all the messages that are in its deferred set and dequeues the first message that is not in its deferred set.

The communication model we follow is that whenever a machine sends a message to another machine, the message is immediately added to the receiver's queue. This is the same communication model as in P, which was mainly designed to model event-driven programs running on a single machine, for example an operating system driver. Note that this communication model is, however, general enough and can be used to also model distributed systems where messages sent by a machine reach the receiving machine after arbitrary time delay (but in FIFO order). One can model such a system by introducing a separate channel process between every pair of machines. This process dequeues messages from its sender and immediately forwards it to the receiver. Since there are multiple channel processes forwarding messages to a given machine, interleavings between them has the same effect as having messages delivered with delay.

8.2.1 Formal semantics of Event-drive Automata

A (global) configuration of an event-driven automata (EDA) consisting of n machines is a tuple $C = (\{C_i\}_{i \in N})$, where $N = \{1, \cdots, n\}$, and where C_i (denoted as $C[i]$) is the local configuration of the i^{th} machine. The configuration $C[i]$ belongs to $(Q_i \times Dom \times \Pi^*)$. The first and the second

$$\frac{C[i] = \Big(q_i, v_i, \mu_i\Big) \qquad (q_i', f) \in T_i^{int}(q_i)}{C \xrightarrow{i} C[i \mapsto (q_i', f(v_i), \mu_i)]} \text{ INTERNAL}$$

$$\frac{\begin{array}{c} C[i] = (q_i, v_i, \mu_i) \qquad C[j] = (q_j, v_j, \mu_j) \\ T_i^s(q_i) = (q_i', j, m, f) \end{array}}{C \xrightarrow{i!j} C\Big[i \mapsto (q_i', v_i, \mu_i)\Big]\Big[j \mapsto \Big(q_j, v_j, \mu_j\ (m, f(v_i))\Big)\Big]} \text{ SEND}$$

$$\frac{\begin{array}{c} C[i] = \Big(q_i, v_i, \mu_i\ (m, l)\ \mu_i'\Big) \\ \mu_i \in [Def_i(q_i) \times Dom_M]^* \qquad m \notin Def_i(q_i) \\ T_i^r(q_i, m) = (q_i', f) \end{array}}{C \xrightarrow{i?} C[i \mapsto (q_i', f(v_i, l), \mu_i\ \mu_i')]} \text{ RECEIVE}$$

$$\rightarrow \quad = \quad \xrightarrow{i} \uplus \xrightarrow{i!j} \uplus \xrightarrow{i?}$$

Figure 8.4: Semantics of EDA

component of $C[i]$ refer to the current state of the i'th machine and the value of its local variable; the third component is the incoming message queue to machine P_i, modeled as a sequence of pairs of a message type and a payload. For a given configuration C and a local configuration of the i'th machine C_i', let $C[i \mapsto C_i']$ be the configuration which is the same as C except that its i^{th} configuration is C_i'.

The initial configuration of the EDA is C_{init} where $C_{init}[i] = (q_i^0, val_i^0, \epsilon)$ for all $i \in N$. The rules for the operational semantics of EDAs are presented in Figure 8.4. The rules for the send and the internal transitions are straightforward; the rule for a receive transition is slightly more complex. From a receive state q_i, machine P_i skips all the messages in its queue that are in its deferred set and dequeues the first message m from its queue that is not in its deferred set. The state of the machine and the value of its local variable is updated according to the semantics of the receive transition.

Let $Reach_G$ be the set of global configurations of the EDA that can be reached from its initial configuration, and it can be computed as $lfp(\lambda S.\ C_{init} \cup \{C' \mid C \rightarrow C', C \in S\})$. Let $Bad_G = \{C \mid C[i] = (q_i^{err}, v_i, \mu_i)\ \ for\ some\ \ v_i, \mu_i,\ and\ i \in N\}$ be the set of error configurations of the EDA. Then we say that the EDA is safe or correct if $Reach_G \cap Bad_G = \emptyset$. Note that even when the states, Dom and Dom_M are finite, the problem of checking whether a

given EDA is safe is an undecidable problem [BZ83].

8.3 Almost-Synchronous Invariants for Event-driven Automata

Given an event-driven automaton, we describe in this section a reduction scheme that selectively explores a subset of the global reachable configurations of the EDA, such that the exploration is sufficient to cover all the local states that can be reached by the EDA. Our reduction mechanism does so by constructing *almost-synchronous invariants*, which are invariants for proving local assertions in asynchronous programs and are expressed as a set of global configurations of the system where the message buffers are close to empty [1]. Finally, we argue in this section that our reduction is both sound and complete and can be effectively used for verifying local assertions in asynchronous/distributed programs.

Given an automaton $\mathcal{P}$, we present a construction of a transition system $\mathcal{P}_{\mathcal{R}}$ such that the set of reachable states of $\mathcal{P}_{\mathcal{R}}$ correspond to a reduced set of global configurations of $\mathcal{P}$ that form an almost-synchronous invariant of the system. Unlike standard reductions, the states as well as transitions will be *different* than that of the automaton. States in $\mathcal{P}_{\mathcal{R}}$ are of the form (C, B) where C is a configuration of the automaton $\mathcal{P}$ and is of the form $(\{C_i\}_{i \in N})$, $C_i \in (Q_i \times Dom \times \Pi^*)$, and $B \subseteq N$ is a subset of *blocked machines*.

As briefly motivated in Section 8.1, transitions in $\mathcal{P}_{\mathcal{R}}$ prioritize receive actions over send transitions, thereby ensuring that the message queues remain small. From a configuration that cannot receive any further messages from its queues, $\mathcal{P}_{\mathcal{R}}$ enables a subset of send transitions whose choice depends on the communication pattern amongst the machines in the current configuration as well as the system-wide global communication pattern amongst machines that is determined statically. Naively enabling only a subset of send transitions will miss out on exploring states that can be reached on taking transitions that are never enabled. To circumvent this problem, $\mathcal{P}_{\mathcal{R}}$ allows at every step a move that blocks those machines whose send transitions were prioritized.

[1] Almost-synchronous invariants are not actually global invariants! They are in fact a subset of reachable configurations that cover all local states and that can be used to prove that no other local states are reachable.

Blocked machines remain forever blocked and can take no transitions. Furthermore, messages sent to blocked machines do not end up in its queue, but are lost to ether, since the blocked machine will anyway not receive them. Consequently, these transitions deviate from the semantics of EDA, but we will show that nevertheless the reduced transition system is sound and complete in discovering all local states.

Before we give the construction of $\mathcal{P}_{\mathcal{R}}$, let us first introduce certain concepts that are important for understanding the construction.

Definition 8.3.1 (Senders)**.** *For a given machine $j \in N$ and a configuration C of the EDA, senders(j, C) is the set of machines $i \in N$ such that $C[i] = (q_i, v_i, \mu_i)$ and $T_i^s(q_i) = (q_i', j, m, f)$, for some $q_i, q_i', v_i, \mu_i, m, f$.* □

Intuitively, *senders*(j, C) is the set of all machines i that are sending a message to machine j in configuration C. This is used to capture the communication pattern amongst the machines in the current configuration.

Definition 8.3.2 (Potential-Senders)**.** *For a given machine $j \in N$, potential-senders(j) is the set of machines $i \in N$ such that there exists a send state $q_i \in Q_i^s$ such that $T_i^s(q_i) = (q_i', j, m, f)$ for some q_i', m, f.* □

Unlike senders, the notion of potential senders is independent of the current configuration. The potential senders of a machine j is the set of all machines that can possibly send a message to it. This depends on the system-wide global communication pattern amongst the machines which can be statically determined.

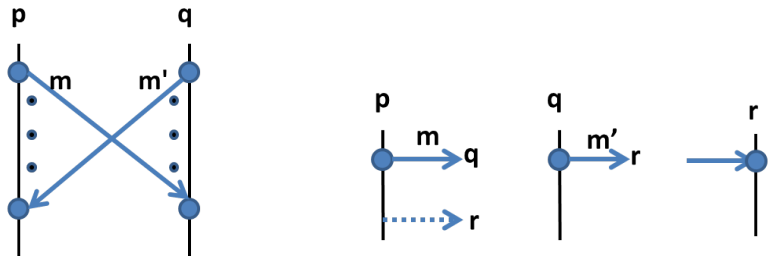
Definition 8.3.3 (Unblocked-Senders)**.** *For a given machine $j \in N$ and an extended configuration (C, B), unblocked-senders(j, C, B) is the set of machines $i \in N$ such that $i \notin B$ and $i \in$ senders(j, C).* □

For a state (C, B) of the transition system $\mathcal{P}_{\mathcal{R}}$, *unblocked - senders*$(j, C, B) =$ *senders*$(j, C) \setminus B$. Given that the machines in B are blocked and not allowed to transition, *unblocked-senders*(j, C, B) captures the set of machines that are allowed to send a message to j from the current state (C, B).

Definition 8.3.4 (isReceiving)**.** *Given a machine $j \in N$ and a configuration C such that $C[j] = (q_j, v_j, \mu_j)$, the predicate isReceiving(j, C) is true iff $q_j \in Q_j^r$.*

Example 8.3.5. *Consider the producer-consumer scenario in Figure 8.1 and let C be its starting configuration. Then, senders$(q, C) = \{p\}$, senders$(p, C) = \emptyset$, and $p \in$ potential-senders(q). Also, isReceiving$(p, C) =$ false while isReceiving$(q, C) =$ true.*

Secondly, consider the scenario in Figure 8.5b and let C be its starting configuration. Then, senders$(r, C) = \{q\}$, senders$(q, C) = \{p\}$, and potential-senders$(r) = \{p, q\}$. Further, senders$(p, C) = \emptyset$, isReceiving$(r, C) =$ true and isReceiving$(q, C) =$ isReceiving$(p, C) =$ false.



(a) (b)

Figure 8.5: Various communication scenarios– (a) process p sends message to q and process q sends message to p (b) process p sends message to q, q sends message to r, which is in a receiving state, process p sends message to r in a state different from the state it is currently in.

We now introduce an important concept, called *destination sets*. From an extended configuration (C, B) of the transition system $\mathcal{P}_{\mathcal{R}}$, our reduction mechanism only explores a subset of the possible send transitions. From a state (C, B) of $\mathcal{P}_{\mathcal{R}}$, our algorithm enables only those transitions that send a message to machines in a destination set, which is defined below.

Definition 8.3.6 (Destination sets)**.** *Given an extended configuration (C, B), $X \subseteq N$, a subset of machines, is a destination set if X contains at least one machine $x \in N$ such that unblocked-senders$(x, C, B) \neq \emptyset$ and for all machines y such that there is an $x' \in X$ with $y \in$ potential-senders(x') and $y \notin B$, the following conditions hold:*

1. *if isReceiving(y, C) is true, then $y \in X$,*
2. *if for some machine $z \in N$, $y \in$ unblocked-senders(z, C, B), then $z \in X$.* □

In other words, for an extended configuration (C, B), a destination set X is a set that includes a machine who has at least one unblocked sender, and for every machine $x \in X$, if y is a potential sender of x, then (1) if y is in receive mode, then $y \in X$, and (2) if y is unblocked and in send mode, then the machine it is sending to is in X.

Note that there could be many destination sets for an extended configuration. Also, note that the set of *all* machines is always a destination set, provided there is at least one machine with an unblocked sender.

Further, note that the two conditions on X are *monotonic*, and hence we can start with a single machine x that has at least one unblocked sender, and close it with respect to the two conditions to get the *least* set containing x that is a destination set.

We now fix a particular choice of destination set for every extended configuration (C, B) that has a machine with at least one unblocked sender. This could be the one obtained by choosing a canonical machine with an unblocked sender and closing it with respect to the two conditions, as described above.

In any case, let us fix a function *destination-set* that maps every extended configuration (C, B) to a destination set if there is at least one machine with an unblocked sender, and to the empty set otherwise.

Example 8.3.7. *In the scenario in Figure 8.5a, let C be the starting configuration and let the blocked set B be empty. Then we can argue that one of the destination sets is $\{q\}$. This can be computed by taking q, which has an unblocked sender, and closing it with respect to the conditions, which doesn't add any more machines. Note that $\{p\}$ is also a destination set.*

Thus, in the reduction, if we choose the destination set $\{q\}$, then we will enable the send from p to q. Now if p after sending the message gets to a receive state, the destination set constructed for this new state will be $\{p\}$, which will force us to enable the other send, from q to p.

Example 8.3.8. *In the scenario in Figure 8.5b, let C be the starting configuration and let the blocked set B be empty. Then notice that $\{r, q\}$ is a destination set, with r having an unblocked sender. However, $\{r\}$ is not a destination set, and in fact $\{r, q\}$ is the smallest destination set including $\{r\}$. If we choose this destination set, then our reduction will enable all the send transitions to them, i.e., the sends from p to q and from q to r. Notice the fact that p being a potential sender to r forces our reduction to also enable the send*

transition from p to q. If our reduction had not enabled the send transition from p to q, we would have not covered the behavior in which process p sends message m to q, followed by process p sending a message to r from a future state, followed by process r receiving that message before process q can send message m' to process r.

The Reduction

We are now ready to define the reduction. The informal algorithm for the reduction is as follows.

Given that the system is in an extended configuration (C, B), we will explore the following transitions from it:

- If any machine is in receive mode and there is an undeferred message on its incoming queue, then we will schedule *all* such receive events and disable all send events.
- If no receives can happen, then we construct the set $X = \textit{destination-set}(C, B)$. Then we schedule *all* sends that send to some machine in X, including sends emanating from X. Furthermore, we also enable a transition that blocks the unblocked senders to X.

The first rule prioritizes receives over sends. The second one selects a subset of sends to enable, depending on the destination set computed. Furthermore, it also enables blocking the unblocked senders to X, which results in a new configuration where senders to X will not be explored, while other send events can be explored. Also, note that sends to blocked machines will have their messages sent to ether.

Figure 8.6 describes the construction of the transition system $\mathcal{P}_{\mathcal{R}}$ with a transition relation $\longrightarrow \subseteq (C \times 2^N) \times (C \times 2^N)$. The initial state of $\mathcal{P}_{\mathcal{R}}$ is $(C_{init}, \emptyset)$ where C_{init} is the initial configuration of the EDA $\mathcal{P}$ and the set of blocked machines is empty. Let us define $Reach_R$, in the natural way, as the set of states reachable by $\mathcal{P}_{\mathcal{R}}$ from its initial state. By definition, $Reach_R$

$$\textsc{receive}\ \frac{C \xrightarrow{i?} C'}{(C,B) \longrightarrow (C',B)}$$

$$\textsc{send-to-unblocked}\ \frac{\begin{array}{cc} \mathit{NoReceivesEnabled}(C) & C \xrightarrow{i!j} C' \\ j \in \mathit{destination\text{-}set}(C,B) & i,j \notin B \end{array}}{(C,B) \longrightarrow (C',B)}$$

$$\textsc{send-to-blocked}\ \frac{\begin{array}{ccc} \mathit{NoReceivesEnabled}(C) & C \xrightarrow{i!j} C' & \\ j \in \mathit{destination\text{-}set}(C,B) & i \notin B & j \in B \end{array}}{(C,B) \longrightarrow \Big(C\big[i \mapsto C'[i]\big], B\Big)}$$

$$\textsc{block}\ \frac{\begin{array}{c} \mathit{NoReceivesEnabled}(C) \\ B' = \Big\{\ i \mid i \in \mathit{unblocked\text{-}senders}(j,C,B), j \in \mathit{destination\text{-}set}(C,B)\Big\} \qquad B' \neq \emptyset \end{array}}{(C,B) \longrightarrow (C, B \cup B')}$$

where

$$\mathit{NoReceivesEnabled}(C): \ \textit{for all } k, \ \textit{if } C[k] = (q_k, v_k, \mu_k) \ \textit{ and } \ \mathit{isReceiving}(k,C) = \ \textit{true, then } \ \mu_k \in [\mathit{Def}_k(q_k) \times \mathit{Dom}_M]^*$$

Figure 8.6: The reduced transition system $\mathcal{P}_{\mathcal{R}}$ whose reachable states $Reach_R$ is an almost-synchronous reduction that includes all local states reachable in $\mathcal{P}$.

is an almost-synchronous reduction of the set of configurations that can be reached by $\mathcal{P}$.

If $\mathcal{P}_{\mathcal{R}}$ is in state (C, B) such that a receive transition is enabled from the configuration C of EDA $\mathcal{P}$, $\mathcal{P}_{\mathcal{R}}$ prioritizes the receive transition (rule RECEIVE in Figure 8.6). The other three rules in Figure 8.6– SEND-TO-UNBLOCKED , SEND-TO-BLOCKED and BLOCK apply only when no receive transitions are enabled from configuration C (captured by the condition *NoReceivesEnabled*(C)). In this case, our reduction mechanism first constructs the destination set for the current state (C, B). Then, P_R enables all send transitions that send a message to a machine j belonging to this set. This case is split into two rules: the rule SEND-TO-UNBLOCKED handles the case where j is not blocked and the second rule SEND-TO-BLOCKED handles the case where j is blocked and the message sent is not enqueued but is lost to ether. In the latter case, note that the configuration of the sender machine i is only updated, and the receiver j's configuration is unaffected. At the same

time, to ensure that our selective exploration does not miss any behaviors, from the state (C, B), $\mathcal{P}_{\mathcal{R}}$ also blocks the machines i whose send transitions to machines j were selectively enabled (rule BLOCK). Note that Figure 8.6 does not depict the internal transitions. However, $\mathcal{P}_{\mathcal{R}}$ does include internal transitions ((C, B) can transition to (C', B) if any internal transition takes C to C'), and in fact these internal transitions are prioritized so that they immediately happen. Since there is no shared state, we do not need to interleave internal transitions in different machines, and hence they happen atomically with the earlier send/receive transition.

Observe that whenever a machine is added to the blocked set, it is in a send state (the rule BLOCK in Figure 8.6). It follows that a machine, if blocked, remains forever blocked and can take no further transitions.

We now turn to the soundness and completeness argument for our ASI reductions. Let $Bad_R = \{(C, B) \mid C \in Bad_G\}$. Also let $\longrightarrow^* \subseteq (C \times 2^N) \times (C \times 2^N)$ be the transitive closure of the single step transition relation $\longrightarrow$ of $\mathcal{P}_{\mathcal{R}}$. We next argue that only exploring states that are reachable in $\mathcal{P}_{\mathcal{R}}$ is both sound and complete with respect to proving the correctness of the automaton $\mathcal{P}$. In other words, a local state is reachable in the program iff it is reachable in the reduced transition system.

Theorem 8.3.9 (Soundness)**.** *If some state $(C_e, B_e) \in Reach_R \cap Bad_R$, then there exists a configuration $C' \in Reach_G \cap Bad_G$.*

Proof sketch: Consider the $\mathcal{P}_{\mathcal{R}}$-reachable, error trace $(C_{init}, \emptyset) \longrightarrow \cdots (C, B) \longrightarrow \cdots (C_e, B_e)$ where $(C_e, B_e) \in Bad_R$. Then we can show that essentially the same set of actions can be mimicked in $\mathcal{P}$ as well, except that the configurations may contain a bit more information on certain message buffers. As we traverse the trace in $\mathcal{P}_{\mathcal{R}}$, at any point, we construct a $\mathcal{P}$-reachable configuration C' which is same as C except for the queue contents of machines that have been already blocked along the error trace. For RECEIVE, SEND-TO-UNBLOCKED and BLOCK transitions along the error trace, the update to C' is straight forward. On a SEND-TO-BLOCKED transition along the error trace, the update to C' departs from the update to (C, B). The update to C', in this case, follows the semantics of EDA $\mathcal{P}$ and enqueues the message into the queue of the blocked machine. As we know that machines that have been blocked cannot take any further transitions, this means that the message enqueued in the blocked machine's queue will be never received by it

as we move forward along the error trace. Hence, though C' differs from C it never diverges away from it (i.e., a $\mathcal{P}_{\mathcal{R}}$-transition enabled from (C, B) will be always enabled from configuration C'; also the states of machines in C' are the same as the states of machines in C). Since $C_e \in Bad_R$, it follows that configuration C'_e we end up with is such that $C'_e \in Reach_G \cap Bad_G$. □

We next argue the completeness of our reduction mechanism. For that, let us introduce $\rightarrow_B \subseteq C \times C$ for $B \subseteq N$ such that $C \rightarrow_B C'$ if configuration C' of automaton $\mathcal{P}$ is reachable from C along a $\mathcal{P}$-trace that involves no transition by any of the machines in the set B. Formally,

$$\rightarrow_B = (\bigcup_{i \notin B} \xrightarrow{i}) \cup (\bigcup_{i \notin B} \xrightarrow{i!j}) \cup (\bigcup_{i \notin B} \xrightarrow{i?})$$

and let $\rightarrow_B^*$ be the transitive closure of $\rightarrow_B$. The completeness result, Theorem 8.3.11, follows essentially from the following lemma. This lemma asserts that whenever we can reach an error configuration from a configuration C in the original program without involving any transition of machines in the set B, we can reach an error configuration in the reduced transition system from the extended configuration (C, B).

Lemma 8.3.10. *If for configurations C, C_e and set $B \subseteq N$ such that $C \rightarrow_B^* C_e$ where $C_e \in Bad_G$, then there exists C', B' such that $(C, B) \longrightarrow^* (C', B')$ and $(C', B') \in Bad_R$.*

Proof sketch: First, we will assume that $C \notin Bad_G$, for otherwise the lemma is obvious. The proof is by contradiction. Assume that there are configurations C, C_e and set $B \subseteq N$ such that $C \rightarrow_B^* C_e$ where $C_e \in Bad_G$, and that there is no $\mathcal{P}_{\mathcal{R}}$-state $(C', B') \in Bad_R$ such that $(C, B) \longrightarrow^* (C', B')$. Let us consider an ordering over the space of C, C_e and B. Let this ordering be the standard lexicographic ordering over $(\mathbb{N} \times \mathbb{N} \times \mathbb{N})$, where the first component is the length of the trace $C \rightarrow_B^* C_e$; the second component is the sum (over all machines) of the messages pending in the queues in configuration C; and the third component is size of the complement of the blocked set B. This is a well-founded ordering. Let us pick configurations C, C_e and set B that satisfy all the assumptions and is smallest with respect to this lexicographic ordering.

We show that we can always make "progress" along the $C \rightarrow_B^* C_e$ trace via a $\mathcal{P}_{\mathcal{R}}$-transition, thereby getting a smaller counter-example with respect to the lexicographic ordering, leading to a contradiction.

We split using two cases, the first when a receive is enabled in configuration C, and the second when no receive is enabled.

Case 1: Let us first consider the case when machine i ($i \notin B$) in configuration C is ready to receive a message from its queue. Let the $\rightarrow_B^*$-trace be $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$. Now, consider the case where there is a transition of machine i in the sequence τ. Then the first transition of machine i in τ must be a receive event. Consider the trace $\tau' = C \xrightarrow{i?}_B C_1 \rightarrow_B \cdots \rightarrow_B C_e$ obtained from τ by moving this receive transition to the front; this is a valid $\rightarrow_B^*$-trace. Using rule RECEIVE in the reduced transition system, it follows that $(C, B) \longrightarrow (C_1, B)$ and also that there exists no state $(C', B') \in Bad_R$ such that $(C_1, B) \longrightarrow^* (C', B')$. Note that τ'-suffix from C_1 to C_e has a shorter length than τ. This means that the choice C_1, C_e and B is a strictly smaller counter-example, which is a contradiction.

When no transition of i is present along τ, the trace $\tau_1 : C \rightarrow_B \cdots C_e \xrightarrow{i?}_B C'_e$ obtained from τ by augmenting it with transition $i?$ is a valid $\rightarrow_B^*$-trace such that $C'_e \in Bad_G$. As before, trace $\tau'_1 = C \xrightarrow{i?}_B C_1 \rightarrow_B \cdots C'_e$ is also a valid $\rightarrow_B^*$-trace but one whose suffix from C_1 to C'_e has the same length as τ. However, note that C_1 has one less message pending in its queues compared to C. Using the same argument as the one above, the choice C_1, C'_e and B is a counter-example and is strictly smaller than C, C_e and B, which is a contradiction.

Case 2: Now consider the second case when no receive transitions are enabled in configuration C. Let $X = \textit{destination-set}(C, B)$.

Consider the subcase where the $\rightarrow_B^*$-trace $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$ contains a transition that sends a message to $x \in X$. Let $p!x$ be the first such transition occurring along τ. We will argue that the transition $p!x$ in this case can be commuted to the beginning and it is possible to construct a valid $\rightarrow_B^*$-trace $\tau' : C \xrightarrow{p!x}_B C_1 \cdots \rightarrow_B C_e$. From the rules SEND-TO-UNBLOCKED or SEND-TO-BLOCKED, it follows that $(C, B) \longrightarrow (C_1, B)$. We can argue that C_1, C_e and B is a smaller counter-example (since the suffix of τ' from C_1 is shorter), leading to a contradiction. Now let us argue that the first transition of p in τ is $p!x$ (if this is so, it is easy see that $p!x$ can be commuted to the front). By definition, $p \in \textit{potential-senders}(x)$ and $p \notin B$. We will show that in C, p is in a state sending to x. If p is in a receive state in C, then by the definition of destination sets, $p \in X$ (since $x \in X$, $p \in \textit{potential-senders}(x)$, and p is in

a receive state). This implies that in τ, before the send event by p happens, there must be a send-event by some machine to p (since we are in the case where the buffers to enabled receivers are empty). Since $p \in X$, this send is a send event to X, which contradicts the assumption that $p!x$ was the first transition along τ sending a message to a machine in X. If p is in a send state in C but it is sending a message to a machine $y \neq x$, then from the definition of destination sets, $y \in X$. Again, this implies that τ has a transition $p!y$ for $y \in X$ before the $p!x$ event, which is again a contradiction. The only option left is that $p!x$ is enabled in configuration C.

We still need to arrive at a contradiction when the $\rightarrow_B^*$-trace $\tau : C \rightarrow_B \cdots \rightarrow_B C_e$ contains no transition that sends a message to a machine $x \in X$. Let $B' = \bigcup_{x \in X} \textit{unblocked-senders}(x, C, B)$ for $x \in X$. Since τ has no transitions sending messages to X, τ involves no transitions by machines in B'. Now let us show that B' is non-empty. Note that the machine involved in the first transition along τ is an unblocked sender. This implies that $X = \textit{destination-set}(C, B)$ is non-empty. Hence, there must be an unblocked sender to X (by definition of destination sets). Hence B' is non-empty. From the rule BLOCK, it follows that $(C, B) \longrightarrow (C, B \cup B')$. Also, $C \rightarrow_{B \cup B'}^* C_e$ is *true*. Since $B \cup B'$ is strictly larger than B, the counter-example C, C_e and $B \cup B'$ is smaller, which is a contradiction. □

Theorem 8.3.11 (Completeness)**.** *If some configuration $C \in Reach_G \cap Bad_G$, then there exists C', B' such that $(C', B') \in Reach_R \cap Bad_R$.*

Proof: The theorem follows directly from the above lemma by substituting B in the lemma to be the empty set and C to be the initial state C_{init} of the automaton $\mathcal{P}$. □

8.4 Lifting ASI Reductions to P programs

A P program [DGJ+13] is a collection of state machines communicating via asynchronous events or messages. Each state machine in P is a collection of states; it has a set of local variables whose values are retained across the states of the machine, has an entry method which is the state in which the control transfers to on the creation of a new machine and finally, has a FIFO incoming queue through which other machines can send messages

to it. Each state in a P state machine has an entry function which is the sequence of statements that are first executed whenever the control reaches that state. Additionally, each state has a set of transitions associated with incoming message types, has a set of action handlers associated with the incoming message types, as well as a classification of certain message types as being deferred or ignored in the given state. After the entry function has been executed, the P machine continues to remain in the same state till it receives a message in its queue. The machine dequeues the first message from its queue that is not deferred and checks if the message is ignored in the current state. If it is, the message is simply dropped from the queue, and the machine continues to remain in the same state. If the message is not ignored, the machine dequeues the message; the next state to which the machine transitions to along with the update to its local state on dequeuing the message is determined by the state's transitions and action handlers. P statements include function calls and calls to foreign functions that are used to model interaction with the environment. A P state machine can have call statements and call transitions in it which are used to implement hierarchical state machines.

We will describe next the mapping between P features and the EDA we introduced in Section 3. EDAs do not support dynamic creation and deletion of machines. We did not find this to be a serious limitation as most driver programs written in P and distributed protocols we modeled in P had a statically determined, bounded, number of machines. Hence the global communication pattern, amongst the machines in the EDA, required for our reductions can also be determined statically.

Also note that we do not restrict the domain *Dom* for the local variables in the state machines to be finite. The entry statement in each state can have multiple sends which can be encoded as a separate send state in EDA connected by local internal transitions. The nondeterministic choice statement can be encoded in the form of nondeterminism on internal transitions. The set of outgoing transition in each P state can be easily mapped on to transitions in EDA. Actions in P can be expanded as a state transition logic implementing the action handler. Similarly, the call statements and call transitions can be handled by repeating the sub-state machines at all call points. By encoding a stack in the local state of the machines, we can model function calls in P programs, in our automaton.

Every machine P_i in an EDA has an error state q_i^{err} that can be used to model local assertions in the P program. An important safety property in P programs is to check the responsiveness of the system, i.e., for every receive state, if m is the first message in the queue that is not deferred, then there should be a receive action enabled from this state that handles m. Checking if a P program is responsive can be easily reduced to checking that the error state q_i^{err} is not reachable, for all $i \in N$.

The upshot of the above relationship is that the reduction algorithms for EDAs described in the earlier sections can be easily lifted to P programs. States in P can perform multiple actions within the state (such as internal actions and sending multiple messages), but these can be broken down into smaller states to simulate our reduction.

8.5 Implementation and Evaluation

We have implemented our ASI reductions by adapting the Zing model-checker [AQR+04]. The P compiler translates P programs into Zing models, preserving the input programs execution model. The explorer in Zing supports guided-search based on a scheduler that is external to the model checker. The ASI reduction in Zing is implemented in the form an external ASI scheduler that guides the explorer on which set of actions are enabled in the current state and the explorer then iterates over these actions. The Zing program is instrumented appropriately to communicate the current state configuration information to the ASI scheduler. This instrumentation is performed automatically by our modified P compiler. The model is instrumented to pass information such as (1) the current state of each state machine, whether its in a send or a receive state (2) size of the message queues, etc. Based on the current state of each state machine, and the communication pattern amongst the machines, the ASI scheduler calculates the destination set mentioned in Section 8.3. Using this destination set, the set of next actions to be performed are prioritized by the ASI scheduler and executed by the Zing explorer.

The implementation of the reduction can be seen as a composition of an almost synchronous ASI scheduler and the asynchronous Zing model, exploring only the state space of the composite system. Most part of the

ASI reduction can be implemented as being external to the model checker except for the case when a state machine is pushed into a *blocked* state. The blocking of a state machine is part of the state of the system and is handled as a special case. A special state machine called *blocking-state-machine* is created with respect to each state machine in the model. The job of the *blocking-state-machine* is to enqueue a special event *block* in the associated state machine. Each state machine in P is extended to handle block event in all states, such that on dequeuing the block event it enters a new state where it keeps dropping all enqueued messages. Now the ASI scheduler can block a state machine by simply scheduling the corresponding *blocking-state-machine* and atomically executing the transitions enqueuing and dequeueing the block event.

8.5.1 Evaluation

We now present an empirical evaluation of the ASI reduction approach for verifying P programs and also evaluate it for finding bugs in them. All the experiments reported are performed on Intel Xeon E5-2440, 2.40GHz, 12 cores (24 threads), 160GB machine running 64 bit Windows Server OS. The Zing model checker can exploit multiple cores during exploration as its iterative depth-first search algorithm is highly parallel [UDR11]. We report the timing results in this section for the configuration when Zing is run with 24 threads and uses iterative depth bounding by default for exploring the state space.

In order to thoroughly evaluate our ASI technique, we evaluate it on models from various domains. We used P for writing all our benchmarks, and used the P compiler to generate Zing models for verification. Our benchmark suite includes:

- the Elevator controller model described in [DGJ+13],
- the OSR driver used for testing USB devices,
- the Truck Lifts distributed controller protocol,
- Time Synchronization standards protocol used for synchronization of nodes in distributed systems,
- the German cache coherence protocol, and

- the Windows Phone (WP) USB driver, which is the actual driver shipped with the Windows Phone operating system.

Note that the lines of code reported in Table 8.1 are for the models when written in P, which is a domain specific language in which protocols can be written very compactly. We could not evaluate our ASI reduction approach on the Windows 8 USB driver used in [DGJ+13] as it was not available to us. We, however, evaluated our technique on the Windows Phone USB driver under a license agreement.

Verifying P programs: Message buffers in P programs can become unbounded and their systematic exploration by ZING will fail to prove such programs correct in the presence of such behaviors [DGJ+13]. In general, the queues can become unbounded when a state machine pushes events into them at arbitrarily fast rates. For ZING to be able to explore such models, P users are allowed to provide a bound on the maximum number of occurrences of an event in a message queue. This indirectly bounds the queue size of each state machine during the state space exploration.

Table 8.1 shows the results for the ZING Bounded Model Checker [DGJ+13] as well as our ASI based reduction technique. The ZING results are only for an under-approximation of the state space, restricted by bounding the maximum number of occurrences of an event to a constant value that was picked by the P developers on the basis of domain knowledge [DGJ+13]. On the other hand, our results for ASI reduction are for the complete verification of the models, where message buffers are unbounded. For ASI, we report the total number of states explored, the time taken by the tool, and whether it was able to prove the programs correct or not.

Our ASI reduction was able to verify completely the Windows Phone (WP) driver and the German protocol, while the ZING bounded model checker failed to explore the state space completely (even when message buffers we bounded) for these models. The P language is being mainly used in Microsoft currently for the development of the Windows Phone USB drivers. The most surprising result here is that our reduction-based technique was able to verify that this driver is responsive (i.e., there is no reachable configuration where a machine receives a message that it cannot handle).

For comparatively smaller models, ASI was able to prove the models correct

Models	Lines of code in P	Zing Model Checker (with buffer bounds)				Almost-synchronous Invariants (with *no* buffer bounds)		
		Bound on max occurrence of an event in queue	Total number of states	Time (h:mm)	State-space exhaustively Explored?	Total number of states	Time (h:mm)	Program Proved Correct?
Elevator	270	2	1.4×10^6	0:22	Yes	2.8×10^4	**0:08**	**Yes**
OSR	377	2	3.1×10^5	0:16	Yes	3.9×10^3	**0:02**	**Yes**
Truck Lifts	290	2	3.3×10^7	2:07	Yes	1.1×10^5	**0:24**	**Yes**
Time Sync (Linear Topology)	2200	4	7.4×10^{10}	5:34	Yes	1.0×10^7	**3:07**	**Yes**
German	280	3	$> 1 \times 10^{12}$	*	No	4.7×10^8	**2:32**	**Yes**
Windows Phone USB Driver	1440	3	$> 1 \times 10^{12}$	*	No	2.4×10^9	**3:48**	**Yes**

** denotes timeout after 12 hours*

Table 8.1: Results for proof based on almost-synchronous invariants for P.

much faster than ZING because of the large state space reduction obtained. ZING is a state of the art explicit-state model checker tuned for efficiently exploring P programs. It uses state caching to avoid re-explorations. However it does not implement partial-order reduction techniques as prior experience suggested they were not very useful in this domain. As described in Section 2, partial-order reduction can easily fail to keep message buffers small (as in the producer-consumer scenario) and hence often lead to infinite state-spaces, which precludes exhaustive search.

We found in the ASI exploration that, for our benchmark programs, the size of message queues never exceeded four, thus indicating that the queues remain bound to a small size under our reduction. On the other hand, even after bounding the queue sizes, ZING bounded model checker could not prove large P programs correct or took a very long time. It is important to note that exploring the system with message-buffers bounded by four does not prove the system correct for arbitrary buffer sizes. Our technique proves the system correct, and *in the end* we get to know what buffer size would have been enough (it is not possible to compute message buffer bounds that ensure completeness without doing the exploration). The experimental results illustrate that testing exhaustively even a highly under-approximated reachable space takes more time. This highly under-approximated search is the current testing strategy for P programs, where developers had chosen bounds on duplicate messages in queues based on system knowledge, to explore using Zing.

Finding bugs in P programs: To demonstrate the soundness of our approach, we created buggy versions of the models in our benchmark suite by introducing known safety errors in them. Table 8.2 shows results in terms of the number of states explored and the time taken before finding the bug, with and without ASI. The search terminates as soon as a bug is found. The table shows that our reduction technique explores orders of magnitude less states and also finds bugs faster for all the models. For the Time Synchronization model with nodes in a ring topology, ZING bounded model checker failed to find the bug while ASI was able to find it. The comparison of ASI with naive iterative DFS and the results we obtain suggest that almost-synchronous reductions may also be a good reduction strategy for finding bugs faster. Note that several bounding techniques have been studied earlier in the context of

Buggy Models	Zing Bounded Model Checker (with buffer bounds)				Almost-synchronous Invariants (with *no* buffer bounds)		
	Bound on max occurrence of an event in queue	Total number of states	Time (h:mm)	Bug Found?	Total number of states	Time (h:mm)	Bug Found?
Truck Lifts	2	950005	1:17	Yes	13453	**0:14**	Yes
Time Sync (Ring Topology)	4	*	*	No	129973	**1:37**	**Yes**
German	3	595723	0:44	Yes	2345	**0:10**	Yes
Windows Phone USB Driver	3	1616157	2:04	Yes	23452	**0:38**	Yes

** denotes timeout after 12 hours*

Table 8.2: Results for bug finding using almost-synchronous invariants for P

finding concurrent bugs [TDB14, EQR11, MQ07a]. Finding better exploration strategies that combine these bounding techniques with ASI is an interesting direction for future work.

8.6 Remarks

We would like to emphasize here that our technique is incomplete and there are simple scenarios where the ASI reduction might not terminate and thus fail to prove a program correct. Figure 8.7 shows such a scenario where p sends an unbounded number of messages to r followed by a message sent to q. The process q receives the message from p, then sends a message to r. The process r receives this message from q before it receives the (unbounded number of) messages that p has sent to it. In this system configuration, the destination set is $\{q, r\}$, and hence the event sending message from p to r would be explored successively, leading to an unbounded number of messages in r's queue, and the exploration will not terminate.

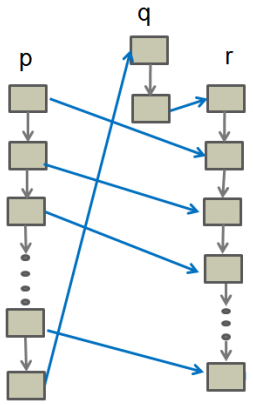


Figure 8.7:

However, as we have shown, for many real-world asynchronous event-driven programs, exploring almost-synchronous interleavings that grow the buffers only when they really need to grow captures more interesting linearizations, discovers smaller adequate invariants, and leads to faster techniques to both prove and find bugs in programs.

8.7 Related Work

The reachability problem for finite state machines communicating via unbounded fifo queues is undecidable [BZ83]. The undecidability stems from the fact that the unbounded queues can be used to simulate the tape of a Turing machine. To circumvent this undecidability barrier, there has been work in several directions. It has been shown that the problem becomes decidable under certain restrictions like when the finite state machines com-

municate via unbounded *lossy* fifo queues that may drop messages in an arbitrary manner [AJ93], when the communication is via a bag of messages and not via fifo queues [SV06, JM07a], when only one kind of message is present in the message queues [PP92], when the language of each fifo queue is bounded [GGLR87], or when the communication between the machines adheres to a forest architecture [LMP08]. For verification of these machines communicating via messages, techniques that over-approximate the set of reachable states have also been studied [BZ83, PP91].

Several under-approximate bounding techniques have been explored to find bugs, even when the machines have shared memory, including depth-bounding [God97], bounded context-switching reachability [QW04, QR05, LMP08], bounded-phase reachability [BE12], preemption-bounding [MQ07b], delay-bounding [EQR11], bounded-asynchrony [FHMP08], etc. These techniques systematically explore a bounded space of reachable states of the concurrent system and are used in practice for finding bugs. It has also been shown that several of these bounding techniques admit a decidable model-checking problem even when the underlying machines have recursion [QR05, LMP07, MP11]

Unlike the above bounding techniques which are not complete, partial order reduction (POR) methods retain completeness while trying to avoid exploring interleavings that have the same partial order [God95, Maz86]. POR techniques use persistent/stubborn sets [GP93, Val91] or sleep sets [GW93] to selectively search the space of reachable states of the concurrent system in a provably complete manner. Dynamic POR [FG05] and its variants [TKL$^+$12, PGK07, LKMA10, LDMA09] including the recently proposed optimal one [AAJS14] significantly improve upon the earlier works by constructing these sets dynamically. Dynamic POR for restrictions of MPI programs where synchronous moves are sufficient have also been explored in [Sie05, SA05, PGK07].

Message sequence charts (MSC), which provide a specification language for specifying scenarios of different communication behaviors of the system, have a partial order semantics, and high-level message sequence charts can combine them with choice and recursion. While checking linear time properties of scenarios of these graphs is undecidable [AY99], surprisingly, checking MSO properties over MSCs directly was shown to be decidable in [Mad01]. Furthermore, this kind of model-checking can be done using linearizations that

keep the message buffers bounded [MM01], similar to the almost-synchronous interleavings explored in this chapter.

The work reported in [JM07a] solves the problem of data flow analysis for asynchronous programs using an under-approximation and over-approximation bounding the counters representing the pending messages, where messages are delivered without in non-fifo order. The authors in [BBO12, BB11] present a technique where choreography of asynchronous machines can be checked when the asynchronous communication can be replaced by synchronous communication. The authors use their analysis technique for verifying channel contracts in the Singularity operating system [HL07]. Though our approach of almost-synchronous reduction has a similar flavor, we do not restrict our analysis to systems where asynchronous message passing can be entirely replaced by synchronous communication.

Our present work builds on top of P [DGJ+13], which is a language for writing asynchronous event-driven programs. While [DGJ+13] uses a model checker to systematically test P programs for responsiveness, our reduction technique provides a methodology for *verifying* P programs. In addition, our experiments strongly suggest that our reductions can be also used to find bugs much faster.

CHAPTER 9

CONCLUSIONS

In the preceding chapters of this thesis, we investigate learning approaches to synthesize inductive invariants of programs towards automatically verifying them. As we have argued in the introduction, learning presents an approach that is complementary to white-box invariant generation techniques such as interpolation, abstract interpretation, etc. The main advantage of the learning approach is that even complicated language constructs (like nonlinear arithmetic operations, operations over the heap, etc.) can be handled elegantly as the actual learner is agnostic to the considered language and the semantics of the program.

Learning approaches have been investigated before for invariant synthesis, and also in the other verification contexts such as learning likely invariants, learning rely-guarantee contracts and learning stateful interfaces for programs. However, prior learning approaches were unduly influenced by traditional machine learning models that learned concepts from positive and negative counterexamples. In this thesis, we theoretically argue that these models are not robust for synthesizing invariants. This has been acknowledged before by several authors as progress in prior learning approaches was almost always achieved at the cost of introducing arbitrary bias in the learning process and possible divergence, and was also experimentally validated by us. Consequently, we propose in this thesis a new learning model for synthesizing inductive program invariants called ICE, which learns invariant concepts from positive, negative and also implication counterexamples. We argue ICE is robust for learning inductive invariants for procedures in a program, and develop algorithms in this model for learning invariants belonging to various different concept classes and apply it towards automatic verification of programs.

We, first, develop algorithms for learning numerical program invariants based on constraint solving and machine learning algorithms for learning

decision trees. We show that our learning algorithms are promising and compare favorably to various other invariant synthesizers, including those based on interpolation, abstract interpretation, white-box template-based invariant synthesis, and other black-box learners including those based on geometric techniques and randomized search, on a suite of programs taken from the software verification competition that require complicated numerical invariants for their verification.

We next develop learning algorithms for synthesizing quantified invariants of linear data-structures, such as arrays and lists, using automata learning algorithms. Our work makes interesting theoretical contributions, which include introducing a new automata model called quantified data automata (QDA) to capture quantified data properties of linear data-structures, identifying a subclass of QDAs called elastic QDAs that are translatable to decidable heap logics and hence amenable to deductive verification and also form an abstract domain for analyzing heap properties using abstract interpretation, and developing automata based learning algorithms to learn QDAs and EQDAs. Besides these theoretical contributions, we deploy the developed learning algorithms to verify various programs over arrays and lists correct, which are taken from the literature and real-world projects including a secure mobile operating system, Linux device drivers and the Gnu library. The learning algorithms are extremely fast and we show they can synthesize invariants to prove certain data-structure programs that are out of reach of current invariant synthesis techniques, such as those based on interpolation.

Finally, we investigate techniques to verify concurrent programs and show that deductive verification of shared memory concurrent programs (under the over-approximate compositional semantics) can be reduced to sequential verification. Similarly, synthesizing rely-guarantee annotations for concurrent programs can be reduced to synthesizing invariants for sequential programs, which can possibly be done using the learning approach we discuss in this thesis. Further, for asynchronously-communicating message-passing systems, we develop an invariant synthesis technique which like the learning approach described before is based on constructing an invariant (called almost synchronous invariants) over concrete system configurations. Our approach is based on explicit model checking; we show that almost synchronous invariants for most systems can be represented finitely, and can be thereby constructed, including for the Windows USB driver that ships with Windows phone, leading

to provably correct system implementations.

Despite the work we present in this thesis and build up on, using learning as an approach for synthesizing program invariants is still in its infancy and there are several interesting directions to pursue this line of research further. A current shortcoming of the ICE learning model is that, though it can be used to prove correct programs correct, it cannot find bugs in incorrect programs. Over the years, model checking has evolved into a good bug-finding technique which searches for violations of the specification over increasingly longer executions traces of the program. Integrating model-checking with learning based invariant synthesis is a very interesting research direction that combines the best of both worlds– bug finding and verification. In fact, this integration can be very tightly coupled where model checkers are used in the ICE teacher to generate new data points for the learner.

Though we describe several advantages that black-box learning provides over white-box invariant generation, white-box program analysis techniques have their own advantages. Invariant synthesizers based on abstract interpretation over simple properties, such as constant propagation or interval analysis, are very fast polynomial-time algorithms. Integrating such white-box synthesizers for simple properties, and also other program analysis techniques such as data-flow analysis, program slicing, etc., with black-box learning-based invariant generation for more complicated properties, we believe, is the most promising direction forward for automatic program verification using invariant synthesis.

Finally, it would be interesting to see if one can use the learning algorithms we have developed to synthesize other kinds of invariants, such as data-structure invariants expressed in separation logic. We believe that building custom invariant-generation tools for particular domains, using domain knowledge to identify interesting Boolean and numerical predicates using which invariants can be synthesized, would bring our techniques to bear on larger programs. GPUVerify [BCD+12], a race-checker for GPU programs does precisely this and uses Houdini to generate conjunctive invariants. Finding more domains where such invariant synthesis will scale is, practically, the most interesting direction for future work.

Apart from annotating inductive program invariants, deductive verification of programs has other manual burdens of annotations. Specifying auxiliary variables in the program and writing auxiliary code that manipulates and maintains them is another huge burden on the programmer wishing to de-

ductively verify large real-world programs. Synthesizing auxiliary state and the program code that maintains it, though outside the scope of this thesis, is an interesting research direction that may not only lead to more practical deductive verification, but also lead to a rich source of mostly unaddressed challenging problems in program verification.

REFERENCES

[AAJS14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. POPL '14, pages 373–384, New York, NY, USA, 2014. ACM.

[ABG+12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Safari: Smt-based abstraction for arrays with interpolants. In *CAV*, volume 7358 of *LNCS*. Springer, 2012.

[ABJ+13] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *FMCAD*, pages 1–17, 2013.

[ACH+10] Christopher Ackermann, Rance Cleaveland, Samuel Huang, Arnab Ray, Charles P. Shelton, and Elizabeth Latronico. Automatic requirement extraction from test cases. In *RV*, pages 1–15, 2010.

[ACMN05] Rajeev Alur, Pavol Cerný, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for java classes. In *POPL*, pages 98–109. ACM, 2005.

[AJ93] Parosh Aziz Abdulla and Bengt Jonsson. Verifying programs with unreliable channels. In *LICS*, pages 160–170, 1993.

[AMN05] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *CAV*, volume 3576 of *LNCS*, pages 548–562. Springer, 2005.

[Ang87a] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[Ang87b] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[Ang90] Dana Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990.

[Apt81] Krzysztof R. Apt. Ten years of hoare's logic: A survey - part 1. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[AQR+04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *CAV*, pages 484–487, 2004.

[Av11] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *POPL*, pages 599–610, 2011.

[AY99] Rajeev Alur and Mihalis Yannakakis. Model checking of message sequence charts. CONCUR '99, pages 114–129, London, UK, UK, 1999. Springer-Verlag.

[BB11] Samik Basu and Tevfik Bultan. Choreography conformance via synchronizability. WWW '11, pages 795–804, New York, NY, USA, 2011. ACM.

[BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Synchronizability for verification of asynchronously communicating systems. In *VMCAI*, pages 56–71, 2012.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, pages 178–192, 2007.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.

[BCD+12] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: A verifier for gpu kernels. *SIGPLAN Not.*, 47(10):113–132, October 2012.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In *CAV*, pages 178–183, 2011.

[BDES11] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. On inter-procedural analysis of programs with lists and data. In *PLDI*, pages 578–589, 2011.

[BDES12] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *VMCAI*, volume 7148 of *LNCS*, pages 1–22. Springer, 2012.

[BE12] Ahmed Bouajjani and Michael Emmi. Bounded phase analysis of message-passing programs. TACAS'12, pages 451–465, Berlin, Heidelberg, 2012. Springer-Verlag.

[BHZ04] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. Widening operators for powerset domains. In Bernhard Steffen and Giorgio Levi, editors, *VMCAI*, volume 2937 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2004.

[BK11] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.

[BKK+10] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The Automata Learning Framework. In *CAV*, volume 6174 of *LNCS*, pages 360–364. Springer, 2010.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3. ACM, 2002.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.

[BZM08] Dirk Beyer, Damien Zufferey, and Rupak Majumdar. Csisat: Interpolation for LA+EUF. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 304–308, 2008.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118. ACM, 2011.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *TPHOLs*, pages 23–42, 2009.

[CFC+09] 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*, pages 31–45, 2009.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, volume 2619 of *LNCS*, pages 331–346. Springer, 2003.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.

[CN07] Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 55–67, 2007.

[CNS13] Wontae Choi, George C. Necula, and Koushik Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *OOPSLA*, pages 623–640, 2013.

[CR08] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.

[CSS03] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, volume 2725 of *LNCS*, pages 420–432. Springer, 2003.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[CW12] Yu-Fang Chen and Bow-Yaw Wang. Learning boolean functions incrementally. In *CAV*, volume 7358 of *LNCS*, pages 55–70. Springer, 2012.

[DGJ+13] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[dMB07] Leonardo Mendonca de Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *CADE*, pages 183–198, 2007.

[dMB08] Leonardo Mendonca de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.

[DOY06] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, pages 287–302, 2006.

[ECGN00] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE*, pages 449–458. ACM, 2000.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. POPL '11, pages 411–422, New York, NY, USA, 2011. ACM.

[Fen09] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 315–327, 2009.

[FFQ02] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 262–277, 2002.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. POPL '05, pages 110–121, New York, NY, USA, 2005. ACM.

[FHMP08] Jasmin Fisher, Thomas A. Henzinger, Maria Mateescu, and Nir Piterman. Bounded asynchrony: Concurrency for modeling cell-cell interactions. In *FMSB*, pages 17–32, 2008.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME*, volume 2021 of *LNCS*, pages 500–517. Springer, 2001.

[Flo67] Robert Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, number 19 in Proceedings of Symposia in Applied Mathematics, pages 19–32. AMS, 1967.

[FQ02] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

[FQ03] Cormac Flanagan and Shaz Qadeer. Thread-modular model checking. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 213–224, 2003.

[FR99a] Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.

[FR99b] Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theor. Comput. Sci.*, 222(1-2):77–111, 1999.

[GGLR87] M. G. Gouda, E. M. Gurari, T. H. Lai, and L. E. Rosier. On deadlock detection in systems of communicating finite state machines. *Comput. Artif. Intell.*, 6(3):209–228, July 1987.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127. ACM, 2006.

[GHR10] Naghmeh Ghafari, Alan J. Hu, and Zvonimir Rakamaric. Context-bounded translations for concurrent software: An empirical evaluation. In *Model Checking Software - 17th International SPIN Workshop, Enschede, The Netherlands, September 27-29, 2010. Proceedings*, pages 227–244, 2010.

[GKT13] Pierre-Loïc Garoche, Temesghen Kahsai, and Cesare Tinelli. Incremental invariant generation using logic-based automatic abstract transformers. In *NASA Formal Methods*, pages 139–154, 2013.

[GLMN13] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. Learning universally quantified invariants of linear data structures. In *CAV*, pages 813–829, 2013.

[GLMN14] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A robust framework for learning invariants. In *CAV 2014*, volume 8559 of *LNCS*, pages 69–87. Springer, 2014.

[GMR09] Ashutosh Gupta, Rupak Majumdar, and Andrey Rybalchenko. From tests to proofs. In *TACAS*, pages 262–276, 2009.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246. ACM, 2008.

[GNMR15] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. Technical report, University of Illinois at Urbana-Champaign, May 2015. `http://hdl.handle.net/2142/77025`.

[GNS+13] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Abstract interpretation over non-lattice abstract domains. In *SAS 2013*, volume 7935 of *LNCS*, pages 6–24. Springer, 2013.

[God95] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, 1995.

[God97] Patrice Godefroid. Model checking for programming languages using verisoft. POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods (extended abstract). CAV '93, pages 438–449, London, UK, UK, 1993. Springer-Verlag.

[GR09] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

[GSV08] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292. ACM, 2008.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata, Logics, and Infinite Games*, volume 2500. Springer, 2002.

[GVA07] Bolei Guo, Neil Vachharajani, and David I. August. Shape analysis with inductive recursion synthesis. In *PLDI*, pages 256–265, 2007.

[GW93] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, 1993.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[HHR+11] Peter Habermehl, Lukás Holík, Adam Rogalewicz, Jirí Simácek, and Tomás Vojnar. Forest automata for verification of heap manipulation. In *CAV*, pages 424–440, 2011.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL*, pages 58–70. ACM, 2002.

[HL07] Galen C. Hunt and James R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348. ACM, 2008.

[IS] Franjo Ivancic and Sriram Sankaranarayanan. NECLA Benchmarks. `http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz`.

[JM06] Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[JM07a] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. POPL '07, pages 339–350, New York, NY, USA, 2007. ACM.

[JM07b] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *CAV*, volume 4590 of *LNCS*, pages 193–206. Springer, 2007.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[Jon83] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, 5(4):596–619, October 1983.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[KJD+10] Soonho Kong, Yungbum Jung, Cristina David, Bow-Yaw Wang, and Kwangkeun Yi. Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In *APLAS*. Springer, 2010.

[KLR10] Daniel Kroening, Jérôme Leroux, and Philipp Rümmer. Interpolating quantifier-free presburger arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, pages 489–503, 2010.

[KN01] Bakhadyr Khoussainov and Anil Nerode. *Automata Theory and Its Applications*. Birkhauser Boston, 2001.

[Koh70] Zvi Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 1970.

[KPW15] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *CoRR*, abs/1501.04725, 2015.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994.

[LDMA09] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[Lit87] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987.

[LKMA10] Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. FASE'10, pages 308–322, Berlin, Heidelberg, 2010. Springer-Verlag.

[LMP07] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. A robust class of context-sensitive languages. In *LICS*, pages 161–170, 2007.

[LMP08] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Context-bounded analysis of concurrent queue systems. In *TACAS*, pages 299–314, 2008.

[LMP09] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 477–492, 2009.

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using smt solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 509–524, 2009.

[LR08] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 37–51, 2008.

[Lub84] Boris D. Lubachevsky. An approach to automating the verification of compact parallel coordination programs I. *Acta Inf.*, 21:125–169, 1984.

[Mad01] P. Madhusudan. Reasoning about sequential and branching behaviours of message sequence graphs. ICALP '01, pages 809–820, London, UK, UK, 2001. Springer-Verlag.

[Maz86] Antoni W. Mazurkiewicz. Trace theory. In *Advances in Petri Nets*, pages 279–324, 1986.

[McM92] Kenneth Lauchlin McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[McM03] Kenneth L. McMillan. Interpolation and SAT-Based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[McM06a] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV 2006*, volume 4144 of *LNCS*, pages 123–136. Springer, 2006.

[McM06b] Kenneth L. McMillan. Lazy abstraction with interpolants. In *CAV*, pages 123–136, 2006.

[McM08] Kenneth L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427. Springer, 2008.

[Min01] Antoine Miné. The octagon abstract domain. In *WCRE*, pages 310–, 2001.

[Mit97a] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[Mit97b] Tom M. Mitchell. *Machine learning*. McGraw Hill series in computer science. McGraw-Hill, 1997.

[MM01] P. Madhusudan and B. Meenakshi. Beyond message sequence graphs. In *FSTTCS*, pages 256–267, 2001.

[MP11] P. Madhusudan and Gennaro Parlato. The tree width of auxiliary storage. In *POPL*, pages 283–294, 2011.

[MPQ11] P. Madhusudan, Gennaro Parlato, and Xiaokang Qiu. Decidable logics combining heap structures and data. In *POPL*, pages 611–622. ACM, 2011.

[MPX+13] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *ASPLOS*, pages 293–304, 2013.

[MQ07a] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[MQ07b] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.

[MQ11] P. Madhusudan and Xiaokang Qiu. Efficient decision procedures for heaps using STRAND. In *SAS*, volume 6887 of *LNCS*, pages 43–59. Springer, 2011.

[MRS10] Bill McCloskey, Thomas W. Reps, and Mooly Sagiv. Statically inferring complex heap, array, and numeric invariants. In *SAS*, pages 71–99, 2010.

[MYRS05] Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, pages 181–198, 2005.

[Nei14] Daniel Neider. *Applications of Automata Learning in Versification and Synthesis*. PhD thesis, RWTH Aachen University, April 2014.

[NKWF12] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Using dynamic analysis to discover polynomial and array invariants. In *ICSE*, pages 683–693. IEEE, 2012.

[OG76] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Inf.*, 6:319–340, 1976.

[OG92] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. *Pattern Recognition and Image Analysis*, pages 49–61, 1992.

[PGK07] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of mpi-based parallel programs. PADTAD '07, pages 43–53, New York, NY, USA, 2007. ACM.

[PP91] Wuxu Peng and S. Puroshothaman. Data flow analysis of communicating finite state machines. *ACM Trans. Program. Lang. Syst.*, 13(3):399–442, July 1991.

[PP92] Wuxu Peng and S. Purushothaman. Analysis of a class of communicating finite state machines. *Acta Inf.*, 29(6/7):499–522, 1992.

[PW10] Andreas Podelski and Thomas Wies. Counterexample-guided focus. In *POPL*, pages 249–260, 2010.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. TACAS'05, pages 93–107, Berlin, Heidelberg, 2005. Springer-Verlag.

[Qui86] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[QW04] Shaz Qadeer and Dinghao Wu. Kiss: keep it simple and sequential. In *PLDI*, pages 14–24, 2004.

[RE] Zvonimir Rakamaric and Michael Emmi. SMACK: Static Modular Assertion Checker. `https://github.com/smackers/smack`.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.

[Ros58] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Inf. Comput.*, 103(2):299–347, 1993.

[RS10] Andrey Rybalchenko and Viorica Sofronie-Stokkermans. Constraint solving for interpolation. *J. Symb. Comput.*, 45(11):1212–1233, 2010.

[RSY04] Thomas W. Reps, Shmuel Sagiv, and Greta Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.

[SA05] Stephen F. Siegel and George S. Avrunin. Modeling wildcard-free mpi programs for verification. In *PPOPP*, pages 95–106, 2005.

[SA14] Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *CAV 2014*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.

[SGH+13a] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. A data driven approach for algebraic loop invariants. In *ESOP*, volume 7792 of *LNCS*, pages 574–592. Springer, 2013.

[SGH+13b] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In *SAS*, volume 7935 of *LNCS*, pages 388–411. Springer, 2013.

[Sha48] C. E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27:379–423, 623–656, 1948.

[Sie05] Stephen F. Siegel. Efficient verification of halting properties for mpi programs with wildcard receives. VMCAI'05, pages 413–429, Berlin, Heidelberg, 2005. Springer-Verlag.

[SISG06a] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS 2006*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.

[SISG06b] Sriram Sankaranarayanan, Franjo Ivancic, Ilya Shlyakhter, and Aarti Gupta. Static analysis in disjunctive numerical domains. In *SAS*, volume 4134 of *LNCS*, pages 3–17. Springer, 2006.

[SL08] Armando Solar Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[SLP12] P. Madhusudan S. La Torre and G. Parlato. Sequentializing parameterized programs. *CoRR*, abs/1207.4271v1, 2012.

[SLTB+06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[SNA12] Rahul Sharma, Aditya V. Nori, and Alex Aiken. Interpolants as classifiers. In *CAV*, volume 7358 of *LNCS*, pages 71–87. Springer, 2012.

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *SAS*, volume 5673 of *LNCS*, pages 3–18. Springer, 2009.

[SRW02] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[SV06] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. CAV'06, pages 300–314, Berlin, Heidelberg, 2006. Springer-Verlag.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 15–28, New York, NY, USA, 2014. ACM.

[Tho97] Wolfgang Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Language Theory*, volume III, pages 389–455. Springer, 1997.

[TKL+12] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. Transdpor: A novel dynamic partial-order reduction technique for testing actor programs. FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.

[TLLR13] A. Thakur, A. Lal, J. Lim, and T. Reps. PostHat and all that: Attaining most-precise inductive invariants. Technical Report TR1790, University of Wisconsin, Madison, WI, Apr 2013.

[UDR11] Abhishek Udupa, Ankush Desai, and Sriram K. Rajamani. Depth bounded explicit-state model checking. In *SPIN*, pages 57–74, 2011.

[UM14] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, pages 302–318, 2014.

[Val91] Antti Valmari. Stubborn sets for reduced state space generation. In *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pages 491–515, London, UK, UK, 1991. Springer-Verlag.

[vE98] C. A. J. van Eijk. Sequential equivalence checking without state space traversal. In *DATE*, pages 618–623, 1998.

[VV07] Abhay Vardhan and Mahesh Viswanathan. Learning to verify branching time properties. *Formal Methods in System Design*, 31(1):35–61, 2007.

[VW86] Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

[XdRH97] Qiwen Xu, Willem P. de Roever, and Jifeng He. The rely-guarantee method for verifying shared variable concurrent programs. *Formal Asp. Comput.*, 9(2):149–174, 1997.

[YBS06] Greta Yorsh, Thomas Ball, and Mooly Sagiv. Testing, abstraction, theorem proving: better together! In *ISSTA*, pages 145–156, 2006.