

© 2006 by Abhay Vardhan. All rights reserved.

LEARNING TO VERIFY SYSTEMS

BY

ABHAY VARDHAN

B.Tech., Indian Institute of Technology at Delhi, 1995
M.S., University of Illinois at Urbana-Champaign, 1997
M.S., University of Illinois at Urbana-Champaign, 1998

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, 2006

Urbana, Illinois

Abstract

Making high quality and reliable software systems remains a difficult problem. One approach to address this problem is *automated verification* which attempts to demonstrate algorithmically that a software system meets its specification. However, verification of software systems is not easy: such systems are often modeled using abstractions of infinite structures such as unbounded integers, infinite memory for allocation, unbounded space for call stack, unrestricted queue sizes and so on. It can be shown that for most classes of such systems, the verification problem is actually undecidable (there exists no algorithm which will always give the correct answer for arbitrary inputs). In spite of this negative theoretical result, techniques have been developed which are successful on some practical examples although they are not guaranteed to always work. This dissertation is in a similar spirit and develops a new paradigm for automated verification of large or infinite state systems. We observe that even if the state space of a system is infinite, for practical examples, the set of *reachable* states (or other fixpoints needed for verification) is often expressible in a simple representation. Based on this observation, we propose an entirely new approach to verification: the idea is to use techniques from *computational learning theory* to identify the reachable states (or other fixpoints) and then verify the property of interest. To use learning techniques, we solve key problems of either getting positive and negative examples for the fixpoint of interest or of answering membership and equivalence queries for this fixpoint. We show that learning-based verification is a powerful approach: as long as one has suitable algorithms which can learn the fixpoints needed and decision procedures for some common set-theoretic operations, one can guarantee that the verification procedure will either find a bug or prove that the system is correct. In particular, we have seen that for a large number of practical systems, the class of regular languages is rich enough to express these fixpoints, allowing us to automatically verify such systems using learning algorithms for regular sets.

We show how the learning-based verification paradigm can be applied to a number of systems and for different kinds of specifications. First, we use learning to verify safety properties of finite state machines communicating over unbounded *first-in-first-out* channels. We assume that the reachable set of states is regular and use two different learning algorithms: one called RPNI which is based on learning from sample

executions of the system, and the other derived from Angluin's L^* algorithm which asks membership and equivalence queries. Next, we show how the learning approach can be used to verify safety properties of integer systems, parameterized systems and other systems in which states can be encoded as strings. We then extend the learning based approach to liveness properties and show how to use learning to verify omega-regular properties as well as CTL properties with fairness constraints.

We have implemented the above techniques in a tool called LEVER. We analyze various examples using the tool and show how LEVER successfully verifies their properties. The running time is also comparable to other tools available. Moreover, since we can prove that our method will terminate whenever the target set that we are computing is regular, this is a substantial improvement over other tools which can guarantee completeness only under very specific conditions. We also present a detailed case study of a module in the Linux kernel called *read-copy-update* and successfully verify some interesting properties using our learning based method.

To my mother, my wife and my daughter.

Acknowledgments

I am very grateful to my advisors Professors Gul Agha and Mahesh Viswanathan for their constant encouragement and direction. I have known Gul for a long time and have always cherished his gentle guidance and support. I thank Mahesh for the stimulating discussions that have made this research possible.

I would like to thank Professors José Meseguer and Grigore Roşu for their feedback and guidance during the course of this research. I am also grateful to all members of the Open Systems Laboratory, present and past, with whom I had a chance to interact with. I would especially like to mention, Koushik Sen with whom I collaborated for many research efforts. Prasanna Thati and Reza Zieai have been my friends for a long time and I will always remember the many discussions we have had. I would also like to thank Mark Astley, Nalini Venkatasubramanian, James Waldby, WooYoung Kim, Carlos Varela, Nadeem Jamali, Nirman Kumar, Myeong-Wuk Jang, YoungMin Kwon, Predrag Tomic, Tom Brown, Liping Chen, Sameer Sundresh, Kirill Mechitov and Sandeep Uttamchandani. I thank the helpful staff at the Department of Computer Science, in particular, Andrea Whitesel, Barb Cicone and Dana Kennedy.

I would like to thank my colleagues at Motorola for understanding for the times I had to be away from Motorola work to focus on my research.

I thank my wife, Varsha, whom I love dearly. She has been a true friend and companion and her endless love has been a real source of strength for me. I thank my daughter Mridula for making life so beautiful. I cherish the love and support from my sister Abha, my cousin Arun, and my parents Smt. Shanta Gupta and Dr. Satya Prakash. My mother was my teacher and guide since my childhood days and I wish she were here today, but I know that her love will be with me forever.

Table of Contents

List of Tables	x
List of Figures	xi
List of Symbols	xiii
Chapter 1 Introduction	1
1.1 Learning Theory	3
1.2 Learning to Verify Safety Properties	4
1.3 Learning to Verify Liveness properties	6
1.4 Discussion	7
1.5 Implementation	7
1.6 Outline	8
Chapter 2 Background	9
2.1 Mathematical Models for Programs and Systems	9
2.1.1 Kripke Structures	10
2.2 Specification Languages	11
2.2.1 The Computational Tree Logic CTL*	11
2.2.2 CTL and LTL	13
2.2.3 ω -regular Languages	14
2.2.4 Safety and Liveness Properties	14
2.3 Model Checking	15
2.3.1 Fixpoints	15
2.3.2 Model Checking Safety Properties	15
2.3.3 CTL Model Checking	16
2.3.4 Model Checking LTL and ω -regular Specifications	16
2.4 Verification of Infinite State Systems	17
2.4.1 Regular Sets and Transducers	17
2.4.2 FIFO Automata	18
2.4.3 Regular Model Checking Framework	19
2.5 Learning Framework	22
2.5.1 RPNI	22
2.5.2 Angluin's L^*	26
2.5.3 Variations of Angluin's algorithm	29
Chapter 3 Related Work	34
3.1 Verification of Finite State Systems	34
3.2 Verification of Infinite State Systems	35
3.2.1 Deductive Methods	35
3.2.2 Abstraction	36
3.2.3 Symbolic Representations for Infinite State Systems	36

3.2.4	Acceleration and Meta-transitions	37
3.2.5	Widening	38
3.2.6	Techniques used in Regular Model Checking	38
3.2.7	Bisimulation Minimization	39
3.2.8	Techniques using Rewrite Systems	39
3.2.9	Verification of Hybrid Systems	40
3.3	Use of Learning for Verification	40
Chapter 4	FIFO Automata Safety Using Passive Learning	42
4.1	Verification Procedure	44
4.1.1	Trace Annotation for FIFO	44
4.1.2	Finding Reachable States from Annotated Traces	45
4.1.3	Recovering a Witness from an Unsafe State	46
4.1.4	From Annotated Trace to System Execution	47
4.1.5	Verification Algorithm	47
4.2	Correctness of the Verification Procedure	50
4.3	Strategies for Trace Generation	52
4.4	Example Application of the Verification Procedure	52
4.5	Comparison with Related Work	53
Chapter 5	FIFO Automata Safety Using Active Learning	57
5.1	Using Active Learning Framework for Verification	57
5.2	Representation of the Reachable States and their Witnesses	59
5.3	Answering Membership Queries	61
5.4	Answering Equivalence Queries	61
5.5	Finding Annotated Traces leading to Unsafe States	64
5.5.1	Complexity Analysis	67
5.6	Comparison with Verification based on Passive Learning	68
Chapter 6	Verifying Safety for Systems in Regular Model Checking Framework	70
6.1	Preliminaries	71
6.1.1	Computing the Image of a Regular Set under a Transducer	72
6.2	Representation of States with Witness	72
6.2.1	Bounded Space	72
6.2.2	Bounded Steps	73
6.3	Answering Membership Queries	73
6.4	Answering Equivalence Queries for Bounded Space Method	74
6.5	Answering Equivalence Queries for Bounded Steps Method	75
6.5.1	Incrementing Count of Number of Steps	75
6.5.2	Fixpoint Characterization for the Set of State Witness Pairs	75
6.6	Verification Procedure	77
6.7	Example Application of the Verification Procedure	79
6.8	Comparison with Related Work	80
Chapter 7	Verification of ω-regular Properties	81
7.1	Preliminaries	81
7.2	Learning to Verify ω -regular Properties	83
7.2.1	Fixpoint Characterization of $EGFf$	83
7.2.2	Learning Fixpoints	86
7.3	Infinite State Systems using Regular Languages	88
7.3.1	Construction of the Product Kripke Structure	89
7.3.2	Symbolic Representation for the Fixpoint X	89
7.3.3	Membership and Equivalence queries	89
7.3.4	Checking for $s_0 \in \sigma(X)$	90

7.3.5	Complexity Analysis	91
7.4	Comparison with Related Work	91
Chapter 8	Verification of CTL properties with Fairness Constraints	94
8.1	Learning to Verify CTL Properties	95
8.1.1	CTL Formulas without Fairness	96
8.1.2	CTL with Fairness Constraints	101
8.2	Representing States with Regular Sets	102
8.2.1	Representation of States with Counter	103
8.2.2	Symbolic Computation of Operators	103
8.2.3	Soundness and Completeness	104
8.3	Complexity Analysis	105
8.4	Comparison with Related Work	107
Chapter 9	Implementation and Results	108
9.1	Overview of LEVER	108
9.1.1	Representation of States	109
9.2	Regular Inference Algorithm	110
9.2.1	Scalability Issues	110
9.3	Input Syntax	111
9.3.1	Experiments and Results for FIFO automata	114
9.4	Experiments and Results for Integer and Parameterized systems	116
9.4.1	Discussion	117
9.5	Case Study	120
9.5.1	RCU in Linux kernel	122
9.5.2	Results	125
Chapter 10	Conclusions and Future Work	127
References	129
Vita	137

List of Tables

9.1	Running time in seconds for safety property verification of FIFO automata	116
9.2	Running times in seconds for safety properties for integer programs	118
9.3	Running times in seconds for LEVER for CTL formula $AG(EFp)$	120
9.4	Running times in seconds for CTL formula $AG(\text{req} \rightarrow AF\text{resp})$ with fairness constraint . . .	121

List of Figures

1.1	Learning rectangles in two dimensional space	3
2.1	A event-action program for a simple buffer problem	10
2.2	Some basic CTL* operators	14
2.3	A simple FIFO automaton	19
2.4	Transition relation for the token passing example	20
2.5	NDD representing the set $x = y$ for two integer variables x and y	21
2.6	Prefix Tree Automaton for $S^+ = \{b, aa, aaaa\}$	23
2.7	RPNI algorithm	24
2.8	Intermediate automata for a run of the RPNI algorithm	25
2.9	Angluin's algorithm	27
2.10	Intermediate automata for a run of Angluin's L* algorithm	28
2.11	Procedure <i>Sift</i> used in learning automata	31
2.12	Procedure <i>Update-Hypothesis</i> and <i>Update-Tree-Hypothesis</i> used in learning automata	32
2.13	Learning automata based on Kearns and Vazirani algorithm	33
4.1	Learning to verify procedure for safety properties using passive learning.	43
4.2	Example FIFO automaton.	45
4.3	Learning to verify algorithm based on RPNI.	48
4.4	Procedure <i>determinize</i> used in the learning algorithm.	49
4.5	Simple producer consumer FIFO automaton	53
4.6	Prefix tree automaton for producer consumer	54
4.7	Final automaton for annotated traces for producer consumer	55
4.8	A FIFO automaton for which RMC tool fails	56
5.1	Verification procedure for safety properties using active learning	58
5.2	Example FIFO automata	60
5.3	Answering equivalence query for the case $\mathcal{F}(L) - L \neq \emptyset$	63
5.4	Answering equivalence query for the case $\mathcal{F}(L) \subsetneq L$	64
5.5	Learning to verify algorithm using active learning	66
6.1	Transducer for the transition relation $x = x + 2$	71
6.2	Valid state-witness pairs for bounded steps case	74
6.3	Transducer for incrementing value of incoming string by 1	75
6.4	Transducer for $x = x + 2$ which also increments the count of steps	76
6.5	Verifying safety properties in regular model checking framework for the bounded space case	78
6.6	Example system to be analyzed using learning-based verification.	79
6.7	Various steps in the verification of an example system	80
7.1	Verification procedure for ω -regular properties	88
7.2	Verifying ω -regular properties for regular set based systems	92

8.1	Verification procedure for CTL	103
8.2	Learning to verify $E[p_1 U p_2]$ with regular sets	106
9.1	High level architecture of tools in LEVER	109
9.2	Example input file for LEVER for FIFO automata	112
9.3	Example input file for LEVER for integer systems	113
9.4	FIFO automata for data transmission with parity	115
9.5	FIFO automata for resource arbitrator	115
9.6	Comparison between LEVER and other tools for safety properties	119
9.7	List initial state	122
9.8	List deferred deletion	122
9.9	List after quiescent period	123
9.10	List after deletion	123
9.11	RCU API in Linux Kernel	124

List of Symbols

Basic Mathematical Symbols

- ω The first transfinite ordinal number.
- \mathbb{N} The set of natural numbers.
- \oplus Symmetric difference.

Symbols Related to General Automata

- ϵ The empty string.
- δ Transition function for an automaton.
- Σ Alphabet for an automaton.
- F Set of accepting states (also used to denote a FIFO automaton).
- I Initial states.
- M Deterministic Finite Automaton (also used for Büchi automaton).
- S Set of states (also used for set of access strings in regular learning).

Symbols Related to FIFO Automata

- Σ_M Set of symbols for contents of a channel.
- Θ Finite set of names of transitions in a FIFO automaton.
- $\bar{\Theta}$ Set of co-names of transitions in a FIFO automaton.
- τ Label for a FIFO automaton transition with internal action.
- F Used to denote a FIFO automaton (also used for a set of accepting states).
- q_0 Initial control state of a FIFO automaton.
- Q Control states of a FIFO automaton.
- C Channel names for a FIFO automaton.
- $L(F)$ Trace language of FIFO automaton F .
- s A state of the system or a string.

Symbols Used in Learning

E	Distinguishing strings for regular inference algorithm.
$N(L)$	Kernel of a language L .
O_T	Observation table in regular inference algorithm.
Pr	Prefixes of a language.
PTA	Prefix Tree Automaton.
S^+	Set of positive examples.
S^-	Set of negative examples.
Sp	Short prefixes of a language.

Symbols Used in Verification

\perp	Symbol used as a filler.
Φ	Fairness constraints for a Kripke structure.
π	Path in the computation tree.
Γ	Fixpoint function used for learning state-witness pairs.
ρ	Alphabet used for strings representing states in the regular model checking framework.
$\eta_f^{i,j}$	Property that there exists a path of length j with $i + 1$ states labeled f .
σ	See Definition 8.
\mathcal{A}	Annotation function used in passive learning.
\mathcal{A}_a	Annotation function used in verification with active learning.
$AL(F)$	Annotated trace language of FIFO automaton F used in verification with passive learning.
$AL_a(F)$	Annotated trace language of FIFO automaton F used in verification with active learning.
AP	Atomic propositions.
$\mathcal{C}(l)$	Control state which the annotated trace l ends in.
\mathcal{F}	Fixpoint function for various temporal operators.
h_c	Function used to get contents of a channel.
\mathcal{L}	Function labeling states to atomic propositions.
K	Kripke structure.
$KTrace(\pi)$	Sequence of labels encountered in a path π .
M	Büchi automaton (also used for Deterministic Finite Automaton).
$\mathcal{P}(K)$	Set of all paths in a Kripke structure K .
\mathcal{R}	States reached by a set of traces.

\mathcal{R}_m	States reached by a set of traces for multiple channel FIFO automata.
$Reach^t$	Set of all valid state-witness pairs for bounded steps.
$Reach^s$	Set of all valid state-witness pairs for bounded space.
R	Transition relation.
$\mathcal{S}(M)$	Language accepted by a Büchi automaton M .
T	Transducer for the transition relation of a system.
T_Q	Set of symbols introduced for keeping track of control states.
$\mathcal{T}(l)$	Prefix of annotated trace l without last symbol.
U	Set of unsafe states.
\mathcal{W}	Function which gives traces leading to unsafe states.

Chapter 1

Introduction

Software programs are often buggy. This is painfully obvious in almost all computer systems that we use daily; with almost predictable frequency, security holes are discovered, programs do not behave as expected and operating systems crash. Some well-known disasters such as Ariane 5 explosion, Patriot-Scud failure, loss of Mars Climate Orbiter and Pentium division bug, have all been attributed to very simple programming errors (see [76] for a web site discussing some of these bugs). A NIST study [122] in 2002 estimated that software errors cost \$59.5 billion annually in the United States alone.

Traditionally, the most common method of finding bugs has been *testing* [80]. However, it is recognized that as systems get more and more complex, exhaustively testing all possible scenarios is impossible. This is even more true of concurrent systems where bugs may be revealed only under some uncommon scenarios. To catch some classes of errors, it has been found that it is useful to do *formal inspections* [36] where a group of software developers review the source code line by line. Tools are also available that can do a conservative *static analysis* [62] of the code to look for common errors such as dangling pointers and unfreed memory. Some other methods that have also been used, include *simulation* of the software system to watch for any catastrophic problems and *runtime verification* [68, 69] where the system is instrumented to emit events which are then monitored against some specification.

There is a set of techniques loosely referred to as *formal methods* which attempts to establish the correctness of programs with mathematical rigor. The first step in the exercise of formal methods is to create a model of the software system which can be subjected to mathematical analysis. The next step is to formulate a precise mathematical statement of the specification that the software model is supposed to satisfy. This is by no means trivial; too often the requirements or specifications for a given piece of software are vague and even conflicting. For a complex system, documenting the specification precisely is a major undertaking. The final step in formal methods is to create a methodology for establishing whether the model meets its specification. Note that the intent of formal methods is not to completely replace other methods but to add powerful techniques to the developer's arsenal to make software systems more reliable.

One approach used in formal methods is that of *theorem proving*. In this, the meaning of various

constructs is expressed through axioms and inference rules in a proof system. To show that the program meets the specification, a *proof* is developed, in which starting from the axioms, each step logically derives from the previous one and ultimately leads to a statement showing the program's validity. This usually needs considerable human guidance and ingenuity but can be potentially used for models with large or even infinite state-spaces. In contrast to theorem proving, the approach of *algorithmic verification* attempts to minimize the need for human intervention and develops algorithms for automatically verifying correctness. An example of algorithmic verification is *model checking*: given a model of the software system and a specification of the desired property, a program automatically checks if the specification is satisfied. However, in many cases, model checking suffers from the *state-space explosion* problem, meaning that the effort required to check a model grows exponentially with the size of the model. Because software systems often include abstractions of unbounded data variables; unbounded process creation and memory allocation; and unbounded recursion, models of such systems have an infinite number of states. For infinite space models, it can actually be shown that the model checking problem is generally undecidable,¹ meaning that there is no algorithm that can solve the problem for all models and specifications. Even if the state space is not infinite, non-trivial programs typically have sufficiently large state-spaces to make their analysis infeasible through exhaustive exploration of all states.

In this dissertation, we propose a new paradigm for verification of infinite or large state-space systems. Our idea relies on the following insight. We know that verification of systems usually entails computing either the set of states reachable from the initial states or certain *fixpoints* associated with logical formulas. For many practical systems, even though the set of reachable states (or the fixpoints) is infinite, it is often highly structured and is expressible in a simple representation, for example as regular sets. There is a rich body of work referred to as *computational learning theory* which, among other things, deals with learning concepts such as regular sets and other languages. If we can provide the information needed to learn the reachable states or the fixpoints, then we can hope to compute these directly and thus solve the verification problem. This is the main question we address in this dissertation. Our central thesis is that techniques from *computational learning theory* can indeed be used effectively for verifying several classes of infinite state systems and various kinds of properties. In subsequent chapters, we develop the details of this learning-to-verify approach and demonstrate that this is an attractive technique for analyzing practical systems. We show how to solve the key problems needed for using learning techniques: either getting positive and negative examples of the fixpoint of interest or answering membership and equivalence queries for this fixpoint. To our knowledge, this is the first systematic attempt to view verification as a learning problem, although some

¹Infinite state systems for which some problems are decidable include *well-structured transition systems*; see [56] for some general results.

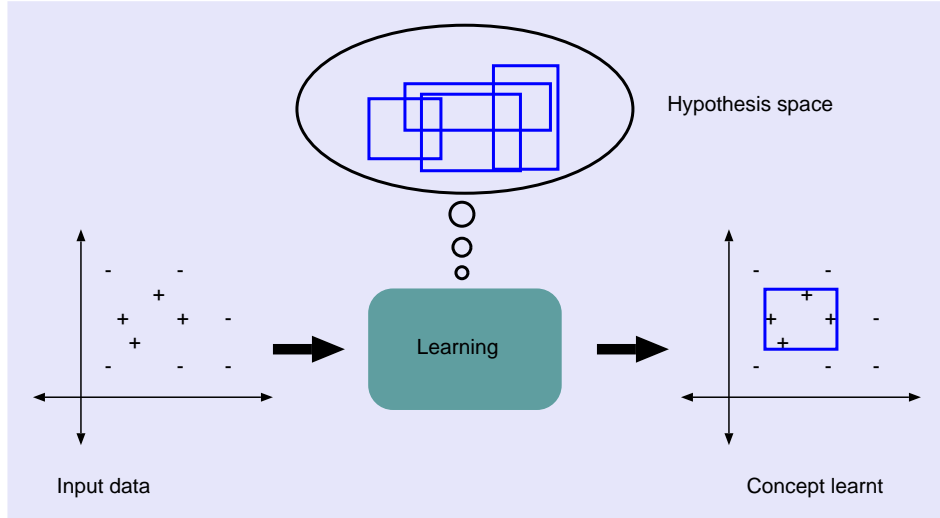


Figure 1.1: Learning rectangles in two dimensional space

ideas from learning have been used in the past for verification.

To set the context in which we use learning theory, we now briefly describe our learning framework.

1.1 Learning Theory

Learning theory deals with the process of hypothesizing a general rule from observations. For example, consider a problem where we are told to find a rectangular region in two dimensional space with its edges aligned to the axes. We are given some points labeled as positive examples (meaning these points are inside the unknown target rectangle) and others as negative examples (meaning these points are outside the target rectangle). The learning task, then, is to identify a rectangle which is compatible with these points. This is graphically illustrated in Figure 1.1.

The fundamental problem addressed by learning algorithms is to efficiently find the hypothesis from among a class of concepts which best explains the given observations. In order to evaluate efficiency, researchers have proposed various frameworks under which learning algorithms operate. In our context, we are interested in two such frameworks: *passive* learning and *active* learning.

In the passive learning framework, the learner is simply given examples included in the target concept (positive examples) and examples not included in the target concept (negative examples). The requirement imposed on the learner is that as it is given successively larger sequences of positive and negative examples, at some point, it stops changing the hypothesis it produces and this hypothesis must match the target

concept. The framework of active learning gives the learner more flexibility. The learner has access to a knowledgeable teacher (student-teacher framework [12]) which provides answers to membership queries (whether a given example belongs to a given concept) and equivalence queries (whether a given hypothesis matches the concept). In case the answer to the equivalence query is in the negative, the teacher must also return one counterexample where the hypothesis differed from the target. Based on these queries, the learner makes hypotheses and eventually converges on the target concept.

As mentioned earlier, we are interested in learning the set of *reachable states* (or other fixpoints) of the system. For the class of concepts used for learning, we choose *regular languages*. This is based on the practical success enjoyed by *regular languages* for representing sets of states in verification and our experience that a number of systems *regular sets* are expressive enough for our verification procedure.

We now briefly give intuition on how learning techniques can be used to verify different kinds of properties.

1.2 Learning to Verify Safety Properties

One of the most crucial properties that systems must satisfy are the class of *safety* properties. Informally, safety properties specify that nothing “bad” happens; some examples of safety properties are: “The temperature regulated by a thermostat will never exceed 85 degrees,” “There can never be more than one process using a critical resource,” and “No user without valid credentials will be able to access the bank account.” It is well known that verification of safety properties can be reduced to checking if any *unsafe* states are reachable from the initial states of the system. The traditional method used in model checking for computing the reachable states is to start with the initial states and iteratively apply the transition relation. At each step, the newly discovered states are added and the process is repeated until no more new reachable states are found. The resulting set is usually referred to as the fixpoint (a precise mathematical definition can be found in Chapter 2). However, for infinite state systems (or even systems with finite but large number of states), this process may never terminate or may take too long to terminate. In our verification technique, we view the identification of the reachable states as a learning problem. Let us now look at the challenges involved in learning the set of reachable states. If we consider using the *passive* learning framework, we need positive and negative examples of reachable states. We can easily find positive examples by executing some sample sequences of transitions; however, in general, we cannot get examples of unreachable states. If we use the *active* learning framework, then we need a way to answer membership and equivalence queries from the learner. Again, in general, we do not have a way of answering whether a given state is reachable or not. For the equivalence query, we do have a partial answer: if the hypothesis presented by the learner is not a

fixpoint under the transition relation, then clearly it is not the correct reachable region; but the converse does not hold. Further, when the hypothesis is not a fixpoint (and hence not equivalent to the target), we do not have an immediate way of returning a counterexample to the learner as required by the active learning framework.

The key idea that resolves the challenges discussed in the previous paragraph is that instead of learning just the set of reachable states, we also learn system executions witnessing the reachability of these states. More precisely, we learn a set of state-witness pairs such that a pair (s, w) is in the target concept if and only if the state s is shown to be reachable by the witness w . This provides us an easy way to get negative examples (in addition to the positive examples): any pair (s, w) in which w fails to demonstrate the reachability of s is a negative example. Moreover, now we can answer a membership query for (s, w) by simply checking whether w is a valid witness for the reachability of s . Equivalence query still has only a partial answer using the fixpoint check, but carrying witnesses with the states allows us to return a counterexample to the learner if the hypothesis is not a fixpoint (details are in Chapter 5). We can also show that even the partial answer to the equivalence query is enough for verification. If the learning algorithm outputs a set that is closed under the transition relation of the system and does not reach any of the unsafe states, then clearly the system can be deemed to be correct. On the other hand, if the hypothesis output by the learner intersects with the unsafe states, we also obtain a witness claiming to lead to an unsafe state. If the witness is valid, then clearly we have found a violation of the safety property and we are done. On the other hand, an invalid witness (along with the state it claimed to reach) can be used by the learner to refine the hypothesis. This process is repeated until either a valid counterexample is found or the system is shown to be correct. Our main observation is that this learning based approach is a *complete verification* method for systems whose reachable set falls in the concept class that we are learning. In other words, for such systems we will eventually either find a buggy execution that violates the safety property, or will successfully prove that no unsafe state is reachable.

We demonstrate the application of learning based verification to systems in which states can be represented using strings. We assume there exists a way to add a witness such that the state-witness pair can also be represented as a string. Based on the practical success enjoyed by *regular model checking* [25], we assume that the set of state-witness pairs such that the states are reachable from the initial states is regular. This allows us to use relatively efficient algorithms that have been developed for learning regular sets. First, we analyze a class of systems called FIFO automata which have unbounded message queues with delivery guaranteed to be in FIFO (first in first out) order. We develop a novel annotation scheme to encode the set of valid state-witness pairs. We use two different learning algorithms: a passive learning algorithm called RPNI

(Regular Positive and Negative Inference) and an active learning algorithm derived from the L^* algorithm given by Angluin in [12]. We also consider systems expressed in a framework called *regular model checking* and show how learning can be used to verify safety properties of such systems.

1.3 Learning to Verify Liveness properties

In contrast to safety properties which specify that “nothing bad happens,” there is another set of properties called *liveness* properties which assert (informally speaking) that “something good eventually happens.” Such properties are obviously very important; a system which does nothing is always safe, but for it do something useful, it typically must guarantee some liveness properties. Let us now see, how the learning approach can be applied to liveness properties as well.

For expressing general temporal properties which include both liveness and safety properties, there are two popular kinds of specification logics, namely: linear temporal logic, and branching time logic. For linear time properties, a convenient representation for the specification is the class of ω -regular properties which use automata over infinite words. It is well known that ω -regular properties can also express *fairness* constraints that are often needed for verifying liveness properties. Similar to traditional automata theory based verification of ω -regular specifications, we construct a representation of the intersection of the possible behaviors of the system and the set of undesirable behaviors. If this intersection is not empty then we have demonstrated that the system can exhibit some undesirable behavior. On the other hand, if the intersection is empty then we have successfully verified the system. The traditional way to perform this emptiness check is via a nested depth first search of an automata over infinite word or the computation of the fixpoint of a μ -calculus formula (details are in Chapter 7). However, both of the above approaches are not amenable to the learning technique. Therefore, we develop a new fixpoint based characterization for such properties. We show that this fixpoint is unique and develop membership and equivalence oracles that can be used for learning it. We again use regular sets for representing the elements of the fixpoint and use a variant of Angluin’s L^* as the learning algorithm. We show that our verification procedure is always sound and more interestingly, it is also complete under the assumption of regularity of the fixpoint.

To verify branching time properties, we use a logic called Computational Tree Logic (CTL). Since CTL itself cannot express fairness which is crucial for liveness properties, we allow fairness constraints to be specified in the model that represents the system. The first challenge to develop a learning based model checking algorithm for CTL is that CTL properties express nested fixpoints. We overcome this challenge by developing a new characterization of CTL properties in terms of functions that have unique fixpoints.

The next challenge is to adequately take into account the fairness constraints that might be associated with the system being verified. In the case of finite state systems, this is handled using the observation that it is sufficient to only consider fair computations that are *ultimately periodic* and *looping*, i.e., computations that repeatedly execute a sequence of steps that loop to a state. However, this observation does not extend to infinite state systems. In order to soundly verify an infinite state system with respect to fairness constraints, we need to also consider fair computations that are truly infinite, and are not looping. We generalize ideas that we developed for the verification of ω -regular properties (discussed in the previous paragraph) to account for fairness. We again instantiate our technique to systems in which states are encoded as strings and use a regular inference algorithm to learn the CTL fixpoints. We prove that if the fixpoints have a regular representation, our procedure will always terminate with the correct answer.

1.4 Discussion

The learning-based verification paradigm enjoys several benefits. First, our verification procedure is accompanied by a precise statement of when the method is guaranteed to work: if the set being computed for verification can be represented using the symbolic representation used by the learner, the method is guaranteed to either prove the system to be correct, or produce a counter-example demonstrating its violation. Of course, for some inputs, the fixpoint may not be representable by the class of concepts being learned; in this case our verification procedure may not halt. However, we have seen that for a large number of practical systems, even the simple class of regular languages is rich enough, allowing us to automatically verify such systems using learning algorithms for regular sets. The second major benefit of the learning based approach is that the running time of the algorithm does not depend on the time it takes to converge to the fixpoint, but rather on the size of the symbolic representation of the fixpoint set. If the fixpoint set being computed has a succinct representation, then the learning algorithm will converge to it very quickly. Finally, because intermediate approximations to the fixpoint are never computed, it avoids the space overhead of storing fixpoint approximations that may have a large symbolic representation.

1.5 Implementation

The techniques presented in this dissertation have been implemented in a tool called LEVER. The tool is written in Java and C++ and is freely available for download from [91]. We have used this tool to analyze a number of examples which include some common FIFO automata such as *alternating bit protocol*, *sliding window protocol*, cache coherence protocols such as *Dragon*, *Firefly*, *Illinois*, *MESI*, *MOESI*, *Berkeley*,

Futurebus and *Synapse*; mutual exclusion protocols such as *peterson*, *lamport*, *ticket* and *bakery*; broadcast protocols such as *consistency*, and *producer-consumer*; petri nets such as *lastinfirstserved protocol*, *Esparza-Finkel-Mayr Counter Machine*, *RTP* and *manufacturing*; and counter machines such as *lift* and *barber*. We analyze safety properties as well as some branching time and liveness properties. We report on the running times for the examples analyzed by our tool and provide comparison with some other tools that are available. The overall comparison of the performance of the various tools is mixed, with each tool taking less time in some examples but more in other examples (see Chapter 9 for details). However, as noted before, since we can prove that our method will terminate whenever the target set that we are computing is regular, we can provide more general completeness guarantees than other tools. As a case study, we also analyze the Linux implementation of *read-copy-update* mechanism which is a scalable solution for reader-writer synchronization for multiple CPUs. We model this module as an integer system and use LEVER to prove some interesting properties of this system.

1.6 Outline

In Chapter 2, we present some background and preliminaries which are used later. In Chapter 3, we survey the related work in verification. Chapters 4 and 5 describe the application of the learning based approach to verification of safety properties for FIFO automata. In Chapter 6, we show how learning can be used to verify safety of systems expressed in the *regular model checking* framework. Next, we move on the analysis of liveness properties, first using ω -regular specifications in Chapter 7 and then using CTL along with fairness constraints (Chapter 8). Chapter 9 discusses the implementation and results of analyzing the examples and we conclude with plans for future work in Chapter 10.

Parts of Chapter 4 were published in [128], parts of Chapter 5 in [127], parts of Chapter 7 in [129] and parts of Chapter 8 in [130].

Chapter 2

Background

In this chapter, we review some background material that will be used in the rest of the thesis. We start with a discussion of mathematical models used for programs and describe specification methods for expressing the properties of such models. We review some techniques for algorithmic verification of systems and discuss some popular classes of infinite state systems. Finally, we describe the learning framework and algorithms that we later use for verification.

2.1 Mathematical Models for Programs and Systems

Before a program or system can be subjected to formal verification, we have to model it as a mathematical structure. It is important that any such model captures the essence of the verification problem but at the same time does not dwell too much on syntactical and other minor details of the program as written in some specific programming language. A popular way to represent programs is using an *event-action* framework, for example, as described in [31]. In this representation, a program is a finite set of control and data variables with given initial values and has a finite set of events. Each event e has an enabling condition $enabled(e)$ which constraints the states in which the transition can be taken and an $action(e)$ specifying how the variables are transformed by the transition. Among the events whose enabling conditions are satisfied, any particular one can be chosen non-deterministically and the control and data variables modified as specified by the associated action. The data variables can be used to encode different memory structures found commonly in programs such as integer variables, call stack, message queues and so on. Event-action based languages have been used in the literature to express concurrent programs including abstractions of multi threaded Java programs, parameterized systems, petri nets, communication protocols, counter systems, broadcast protocols and cache coherence protocols [10, 54].

Example 1. A simplified example program written in this language is shown in Figure 2.1. In this example, there is a producer who adds items to a buffer b while two consumers consume these items from b . The variable o records the total number of items produced while i_1 and i_2 record the number of items consumed

<p>Data Variables: b, o, i_1, i_2: integer Control Variables: $pc : \{q_0\}$ Initial Conditions: $b = o = i_1 = i_2 = 0, pc = q_0$ Events:</p> <ol style="list-style-type: none"> 1. enabled: $true$, action: $b' = b + 1, o' = o + 1$ 2. enabled: $b > 0$, action: $b' = b - 1, i_1' = i_1 + 1$ 3. enabled: $b > 0$, action: $b' = b - 1, i_2' = i_2 + 1$
--

Figure 2.1: A event-action program for a simple buffer problem

by each of the two consumers respectively. The domain of the variables o , i_1 and i_2 is the set of natural numbers. In the action clause, the new value of a variable is given by a primed version of that variable (for brevity, only the variables that are changed are shown). Note that this program has an infinite number of reachable states.

2.1.1 Kripke Structures

While event-action based languages are rich enough to model and define the semantics of programs, they need to be enhanced in order to perform verification. In particular, states of the system need to be annotated with logical propositions that describe properties that are relevant for verification. For example, a state in which two processes are incorrectly accessing a critical resource simultaneously may be labeled *bad* while other states may be labeled *good*. Such a model of the software system where states are annotated with atomic propositions is formally called a *Kripke structure*, and is defined as follows.

Definition 1. A *Kripke structure* is a quintuple $(S, AP, R, I, \mathcal{L})$ where S is a set of (possibly infinite) states, AP is a finite set of atomic propositions, $R \subseteq S \times S$ is a (total) transition relation, $I \subseteq S$ is a set of initial states and $\mathcal{L} : S \rightarrow 2^{AP}$ is function that assigns to each state the set of propositions that are true in that state.

In this dissertation, we restrict ourselves to Kripke structures that are finitely branching, *i.e.*, if a Kripke structure has a state $s \in S$ and a relation R , the set $\{s' \mid R(s, s')\}$ is finite. We will sometimes denote $(s_1, s_2) \in R$ by $s_1 \rightarrow s_2$. A *computation* starting from state s is a sequence of states s_0, s_1, \dots such that $s_0 = s$, and $s_i \rightarrow s_{i+1}$ for each i . Constructing the Kripke structure corresponding to an event-action

program is straightforward, provided we are also given the labeling function \mathcal{L} . The states S are functions that assign to each variable of the program a value in the appropriate domain, and each event e defines a binary relation $R_e \subseteq S \times S$ to be $\{(s_1, s_2) \mid s_1 \in \text{enabled}(e) \text{ and } (s_1, s_2) \in \text{action}(e)\}$. Then, the transition relation is simply given by $R = \bigcup_{e \in E} R_e$.

Kripke structures are sometimes augmented with fairness constraints that hold in the system. Formally, a *Fair Kripke structure* is defined as follows.

Definition 2. A Fair Kripke structure is a tuple $(S, AP, R, I, \mathcal{L}, \Phi)$, where $(S, AP, R, I, \mathcal{L})$ is a Kripke structure, and $\Phi \subseteq S$ is the set of fair states. A fair computation starting from s is then a computation s_0, s_1, \dots starting from s that visits the fair states infinitely often, i.e., $s_j \in \Phi$ for infinitely many j .¹

2.2 Specification Languages

In order to proceed with any verification method, we have to first specify in some formal language, the properties that the system should exhibit. A popular framework for describing specifications is *temporal logic* which allows one to reason about properties like *eventually* or *never*.

2.2.1 The Computational Tree Logic CTL*

CTL* formulas describe properties of *computation trees*. Our treatment of CTL* is derived from [39]. If we start from a state in the Kripke structure and start unwinding the structure into an infinite tree, the resulting structure is called a computation tree. This tree shows all possible computation paths that can happen starting from some state in the Kripke structure.

Formulas in the logic are built up from the atomic propositions and are composed of *path quantifiers* and *temporal operators*. There are two path quantifiers: A (“for all computation paths”) and E (“for some computation path”). The temporal operators are:

- X (“next time”) requires that a property holds in the second state of the path.
- F (“eventually”) asserts that a property will hold at some state on the path.
- G (“always”) specifies that a property holds at every state on the path.
- U (“until”) operator is a binary operator combining two properties. It holds if there is a state on the path where the second property holds, and at every preceding state on the path, the first property holds.

¹Usually fairness is described by a *set* of constraints using the so-called *generalized* Büchi condition. However, it is known that a *generalized* Büchi condition can be converted into the kind of fairness constraint described above.

- R (“release”) is the dual of the U operator and requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

There are two types of CTL* formulas: *state formulas* which are true for a specific state and *path formulas* which are true along a specific path. Let AP be the set of atomic propositions. Then the syntax of CTL* formulas is given by the following rules:

- If $p \in AP$, then p is a state formula
- If f and g are state formulas, then $\neg f$, $f \wedge g$, $f \vee g$ are state formulas
- If f is a path formula, then Ef and Af are state formulas
- If f is a state formula, then f is also a path formula
- If f and g are path formulas, then $\neg f$, $f \wedge g$, $f \vee g$, Xf , Ff , Gf , fUg and fRg are path formulas.

The semantics of the CTL* are given with respect to a Kripke structure $M = (S, AP, R, I, \mathcal{L})$. A path in the Kripke structure is an infinite sequence of states, $\pi = s_0, s_1, \dots$ such that for every $i \geq 0$, $(s_i, s_{i+1}) \in R$. Let π^i denote the suffix of π starting at s_i . We use $M, s \models f$ to denote that a state formula f holds at a state s in M , and $M, \pi \models f$ to denote that a path formula f holds along a path π in M . The relation \models is defined inductively as follows (assuming f_1 and f_2 are state formulas and g_1 and g_2 are path formulas):

$M, s \models p$	\Leftrightarrow	$p \in \mathcal{L}(s)$
$M, s \models \neg f_1$	\Leftrightarrow	$M, s \not\models f_1$
$M, s \models f_1 \vee f_2$	\Leftrightarrow	$M, s \models f_1$ or $M, s \models f_2$
$M, s \models f_1 \wedge f_2$	\Leftrightarrow	$M, s \models f_1$ and $M, s \models f_2$
$M, s \models Eg_1$	\Leftrightarrow	there is a path π from s such that $M, \pi \models g_1$
$M, s \models Ag_1$	\Leftrightarrow	for every path π from s , $M, \pi \models g_1$
$M, \pi \models f_1$	\Leftrightarrow	s is the first state of π and $M, s \models f_1$
$M, \pi \models \neg g_1$	\Leftrightarrow	$M, \pi \not\models g_1$
$M, \pi \models g_1 \vee g_2$	\Leftrightarrow	$M, \pi \models g_1$ or $M, \pi \models g_2$
$M, \pi \models g_1 \wedge g_2$	\Leftrightarrow	$M, \pi \models g_1$ and $M, \pi \models g_2$
$M, \pi \models Xg_1$	\Leftrightarrow	$M, \pi^1 \models g_1$
$M, \pi \models Fg_1$	\Leftrightarrow	there exists a $k \geq 0$ such that $M, \pi^k \models g_1$
$M, \pi \models Gg_1$	\Leftrightarrow	for all $k \geq 0$ such that $M, \pi^k \models g_1$
$M, \pi \models g_1 Ug_2$	\Leftrightarrow	there exists a $k \geq 0$ such that $M, \pi^k \models g_2$ and for all $0 \leq j < k$, $M, \pi^j \models g_1$
$M, \pi \models g_1 Rg_2$	\Leftrightarrow	for all $j \geq 0$, if for every $i < j$ $M, \pi^i \not\models g_1$ then $M, \pi^j \models g_2$

Figure 2.2 illustrates some common CTL* operators. It can be shown that operators \vee, \neg, X, U and E are sufficient to express any other CTL* formulas.

In the presence of fairness constraints, the path quantifiers are interpreted only over fair paths. So for example, a state s satisfies EGf in the presence of fairness, if there is a *fair* computation starting from state s such that f holds in all states.

2.2.2 CTL and LTL

There are two useful sublogics of CTL*. The first one is Computation Tree Logic (CTL) in which each of the temporal operators X, F, G, U and R must be immediately preceded by a path quantifier. The other sublogic is called *linear-time logic* (LTL) consists of formulas of the form Af where f is a path formula in which the only state subformulas permitted are atomic propositions. It can be shown that both CTL and LTL have different expressive powers and are properly included in CTL*.

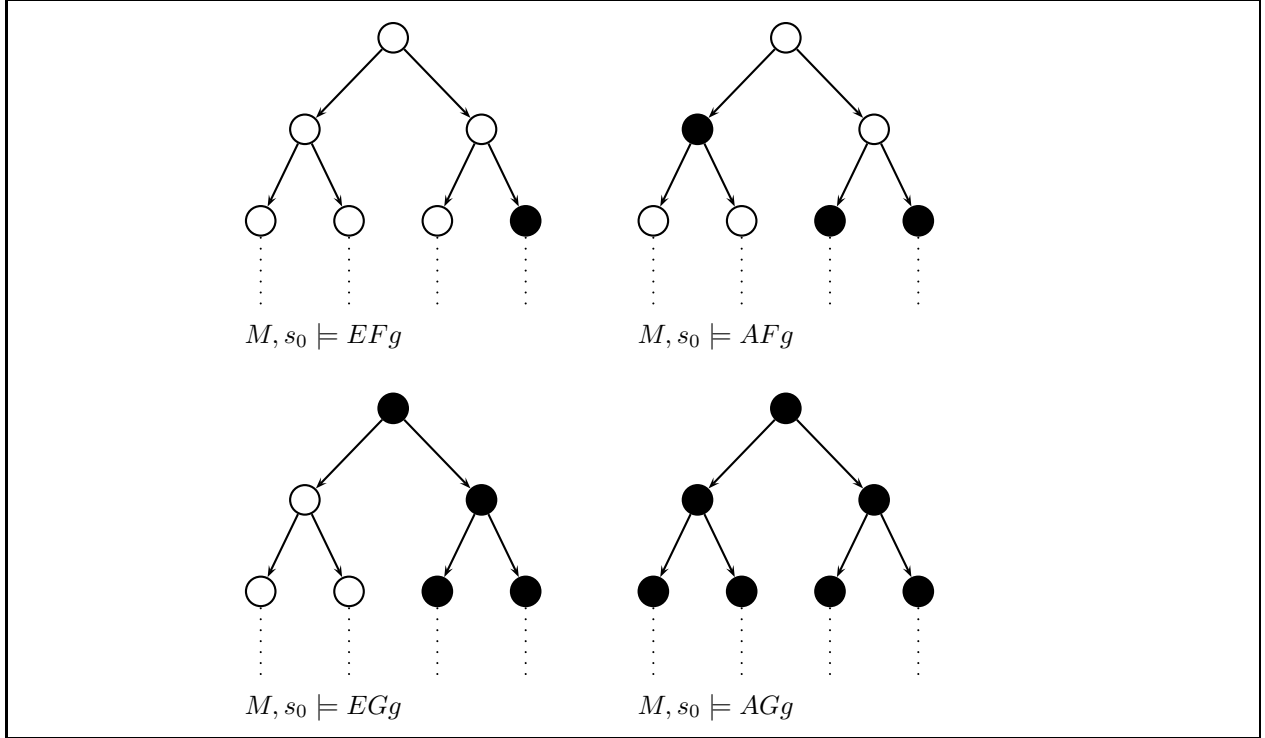


Figure 2.2: Some basic CTL* operators. Filled circle indicates that the state has the label g

2.2.3 ω -regular Languages

Another method for describing the set of acceptable or unacceptable behaviors of the system is by using automata over infinite words. A popular formulation of infinite word automata is Büchi automata.

Definition 3. A Büchi automaton [124] M is a quintuple $(S, \Sigma, I, \delta, F)$ where S is a finite set of states, $I \subseteq S$ is the set of initial states, $\delta : S \times \Sigma \rightarrow 2^S$ is the transition function, $F \subseteq S$ is a set of accepting states.

For an infinite word $v = v_0, v_1, v_2, \dots \in \Sigma^\omega$, the run of M on v is a sequence of states $\rho = s_0, s_1, s_2, \dots$, such that $s_{i+1} \in \delta(s_i, v_i)$ for every i . An infinite word v is accepted by M if there is some run ρ of M on v such that some state $s \in F$ appears infinitely often in ρ . The language accepted by M is the set of all words v accepted by M . A set of infinite words L is said to be ω -regular if there is some Büchi automaton which accepts L .

It is well-known ([60]) that all LTL formulas can be translated into a Büchi automaton.

2.2.4 Safety and Liveness Properties

Properties expressed in CTL* or ω -regular languages are often classified into *safety* and *liveness* properties. Informally, safety properties assert that something “bad” never happens. Safety properties are finitely

refutable. This means that in order to show a violation of a safety property it suffices to show a finite prefix of some computation. A simple safety property written in CTL* is $AG(p)$ where p is some invariant that the system must satisfy.

In contrast to safety, liveness properties assert that something “good” eventually happens. These are properties that are never finitely refutable, *i.e.*, one needs an infinite computation to show that a liveness property has been violated. An example of a liveness property written in CTL* is $AG(req \rightarrow AF(resp))$ which asserts that in all states, if we are ever in a *req* state, this implies that for all paths, we will see some *resp* state in future.

2.3 Model Checking

The *verification* or *model checking* problem is as follows. Given a system description in terms of a Kripke structure $(S, AP, R, I, \mathcal{L})$ (or alternatively, a fair Kripke structure $(S, AP, R, I, \mathcal{L}, \Phi)$) and a temporal logic specification f , we have to check if all states $s \in I$ satisfy the formula f .

We now introduce terminology about fixpoints and operators that we will find useful.

2.3.1 Fixpoints

Consider a function $\mathcal{F} : 2^S \mapsto 2^S$, from sets of states to sets of states. A *fixpoint* for \mathcal{F} is a set $Z \subseteq S$ such that $\mathcal{F}(Z) = Z$; it is the *least fixpoint* if it is the least, with respect to \subseteq -ordering, among all the fixpoints of \mathcal{F} , and is denoted as $\mu(\mathcal{F})$. *Greatest fixpoint* of \mathcal{F} is defined analogously and is denoted by $\nu(\mathcal{F})$. The function \mathcal{F} is said to be *monotonic* if $Z_1 \subseteq Z_2 \Rightarrow \mathcal{F}(Z_1) \subseteq \mathcal{F}(Z_2)$. It is well-known ([121]) that if \mathcal{F} is monotonic, then both $\mu(\mathcal{F})$ and $\nu(\mathcal{F})$ exist. \mathcal{F} is \cup -*continuous* if $Z_1 \subseteq Z_2 \subseteq \dots \Rightarrow \mathcal{F}(\cup_i Z_i) = \cup_i \mathcal{F}(Z_i)$. It is \cap -*continuous* if $Z_1 \supseteq Z_2 \supseteq \dots \Rightarrow \mathcal{F}(\cap_i Z_i) = \cap_i \mathcal{F}(Z_i)$. Another well-known result about fixpoints is that if \mathcal{F} is \cup -*continuous* then $\mu(\mathcal{F})$ can be calculated by starting from the empty set and repeatedly applying the function until the result does not change.² Analogously, if \mathcal{F} is \cap -*continuous* then the greatest fixpoint can be computed by starting from the entire set S and repeatedly applying the function until convergence.

2.3.2 Model Checking Safety Properties

To prove safety properties, it suffices to find the set of reachable states (or its over approximation) and show that it does not intersect with the set of “bad” states. Alternatively, one can find the set of states (or over

²Note that this procedure is guaranteed to terminate in finite number of steps only if S is finite

approximation of such a set) that can reach “bad” states but does not include any of the initial states. If U is the set of states labeled “bad”, then in terms of fixpoints, this reduces to checking one of the following:

1. The least fixpoint of the function $\mathcal{F}_f : 2^S \mapsto 2^S$ given by $\mathcal{F}_f(Z) = I \cup \{s \mid (s', s) \in R \text{ and } s' \in Z\}$ does not intersect with U .
2. The least fixpoint of the function $\mathcal{F}_b : 2^S \mapsto 2^S$ given by $\mathcal{F}_b(Z) = U \cup \{s \mid (s, s') \in R \text{ and } s' \in Z\}$ does not intersect with I .

2.3.3 CTL Model Checking

The standard algorithm [39] for CTL model checking proceeds by progressively computing the set of states that satisfy the various subformulas of f (including f itself). Initially, we compute the set of states that satisfy each of the atomic propositions in f . We know that s satisfies p iff $p \in \mathcal{L}(s)$. The algorithm then proceeds in stages. In the i th stage, the set of states corresponding to subformulas with $i - 1$ nested CTL operators are computed using the results of the computation in the previous stages. Once the states satisfying f are found, the system is deemed to be correct if and only if this set wholly contains I , the initial set of states.

The algorithm to compute the set of states satisfying a subformula g in some stage, say i , depends on the outermost logical operator of g . For the operators EU and EG , computing the set of states satisfying them, involves computing the least and greatest fixpoint, respectively, of certain functions.

2.3.4 Model Checking LTL and ω -regular Specifications

LTL model checking can be done using a *tableau* method (See Section 4.2 in [39]). Alternatively, there is a method to verify ω -regular specifications using automata theory. Since an LTL formula can be converted into an ω -regular specification, this can be used to model check LTL as well. We briefly expound on the automata-theoretic method.

We assume that we are given a ω -regular specification of the system as a Büchi automaton. Büchi automata are closed under intersection and complementation and there exists a decision procedure for checking if a Büchi automaton is empty. This suggests the following method for model checking a Kripke structure K with a finite set of states against an LTL formula L :

- Construct a Büchi automata B_1 from the Kripke structure K which models the system. A Kripke structure directly corresponds to a Büchi automata, where all the states are accepting.
- If the specification is given as an LTL formula L , negate L and translate the negated formula into a Büchi automata B_2 . If the specification is a Büchi automata giving the acceptable behaviors comple-

ment it to get a Büchi automata B_2 . (Often the specification is itself given as the set of undesirable behaviors of the system, in which case the complementation step is not needed.)

- Construct the automata for the intersection of B_1 and B_2 and check for emptiness. The system satisfies the specification if and only if the intersection is empty.

The emptiness check for a Büchi automaton is usually done by conducting a nested depth first search on the automaton. For details, the reader is referred to [39].

2.4 Verification of Infinite State Systems

We now consider some interesting classes of infinite state systems and discuss the verification problem of such systems. We first need to fix some basic notation about regular sets.

2.4.1 Regular Sets and Transducers

Let Σ be a finite set of symbols called the *alphabet* and let Σ^* be the set of strings over Σ . A *non-deterministic finite automaton (NFA)* is $M = (Q, \delta, \Sigma, q_0, F)$ where Q is a finite set of states, δ is the transition relation $\delta \subset Q \times \Sigma \times Q$, q_0 is the start state and F is a set of *final* states. A *deterministic finite automaton (DFA)* is a special case of an NFA in which δ is actually a function $Q \times \Sigma \rightarrow Q$.

We define a function $\hat{\delta}$ from $Q \times \Sigma^*$ to Q as follows:

- $\hat{\delta}(q, \epsilon) = q$. Here ϵ denotes the empty string
- for all strings w and symbol a , $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$

A string x is said to be accepted by the DFA M if $\hat{\delta}(q_0, x) = p$ for some p in F . The *language accepted* by M is the set $\{x \mid \hat{\delta}(q_0, x) \text{ is in } F\}$. A language is a regular set if it is accepted by some DFA.

Regular languages are closed under union and complementation. It is also known that for every any NFA, a DFA can be constructed which accepts the same language. For a detailed treatment of regular languages the reader is referred to [75].

A transducer T is tuple $(Q, \delta, \Sigma, \Omega, q_0, F)$ where Q is a finite set of states, $\delta \in Q \times (\Sigma \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}) \times Q$ is a transition relation, q_0 is the start state and F is a set of *final* states. Σ and Ω are finite sets and are called the input and the output alphabet respectively. A *path* from a state p_0 to another state p_{n+1} is a word in $d_1 d_2 \dots d_n \in \delta^*$ such that there is a sequence $(p_0, u_0, v_0, p_1)(p_1, u_1, v_1, p_2) \dots (p_n, u_n, v_n, p_{n+1})$ and for all i with $0 \leq i < n + 1$ it is true that $(p_i, u_i, v_i, p_{i+1}) \in \delta$. The word $u_0 u_1 \dots u_n$ is called the input string

and the word $v_0v_1 \dots v_n$ is called the output string of the path. A path from q_0 to a state in F is called a *successful* path. A transducer T defines a relation $R_T \in \Sigma^* \times \Omega^*$ given by

$$R_T = \{(f, g) \mid f \text{ and } g \text{ are input and output strings of some successful path}\}$$

For an extensive treatment of transducers, the reader is referred to [20].

2.4.2 FIFO Automata

A popular model for a variety of software systems comprises of finite state machines communicating over unbounded FIFO (first in first out) channels (*FIFO automata*). Examples of such abstraction include: networking protocols where unbounded buffers are assumed, languages like Estelle and SDL (Specification and Description Language) in which processes have infinite queue size, distributed systems and various *actor* [3] systems.

A FIFO automaton [55] is a 6-tuple $(Q, q_0, C, \Sigma_M, \Theta, \delta)$ where Q is a finite set of *control states*, $q_0 \in Q$ is the initial control state, C is a finite set of *channel names*, Σ_M is a finite alphabet for contents of a channel, Θ is a finite set of transitions names, and $\delta : \Theta \rightarrow Q \times ((C \times \{?, !\} \times \Sigma_M) \cup \{\tau\}) \times Q$ is a function that assigns a *control transition* to each transition name. For a transition name θ , if the associated control transition $\delta(\theta)$ is of the form $(q, c?m, q')$ then it denotes a *receive* action, if it is of the form $(q, c!m, q')$ it denotes a *send* action, and if it is of the form (q, τ, q') then it denotes an *internal* action. The channels are considered to be perfect and messages sent by a sender are received in the order in which they were sent. The formal operational semantics, given by a labelled transition systems, is defined below.

A FIFO automaton $F = (Q, q_0, C, \Sigma_M, \Theta, \delta)$ defines a *labelled transition system* $\mathcal{L} = (S, \Theta, \rightarrow)$ where

- The set of states $S = Q \times (\Sigma_M^*)^C$; in other words, each state of the labelled transition system consists of a control state q and a C -indexed vector of words w denoting the channel contents.
- If $\delta(\theta) = (q, c?m, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q, p' = q'$ and $w = w'[c \mapsto m \cdot w'[c]]$
- If $\delta(\theta) = (q, c!m, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q, p' = q'$ and $w' = w[c \mapsto m \cdot w[c]]$
- If $\delta(\theta) = (q, \tau, q')$ then $(p, w) \xrightarrow{\theta} (p', w')$ iff $p = q, p' = q'$ and $w' = w$.

Here $w[i \mapsto s]$ stand for the C -indexed vector which is identical to w for all channels except i , where it is s ; $w[i]$ denotes the contents of the channel i . We say $(p, w) \rightarrow (p', w')$ provided there is some θ such that $(p, w) \xrightarrow{\theta} (p', w')$. As usual, \rightarrow^* will denote the reflexive transitive closure of \rightarrow . For $\sigma = \theta_1\theta_2 \dots \theta_n \in \Theta^*$,

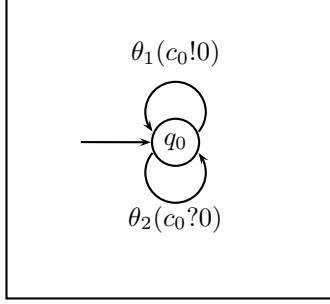


Figure 2.3: A simple FIFO automaton which sends a symbol 0 on one transition and consumes it on the other transition

we say $(p, w) \xrightarrow{\sigma} (p', w')$ when there exist states $(p_1, w_1) \dots (p_{n-1}, w_{n-1})$ such that $(p, w) \xrightarrow{\theta_1} (p_1, w_1) \xrightarrow{\theta_2} \dots (p_{n-1}, w_{n-1}) \xrightarrow{\theta_n} (p', w')$. The trace language of the FIFO automaton is

$$L(F) = \{\sigma \in \Theta^* \mid \exists s = (p, w). s_0 \xrightarrow{\sigma} s\}$$

where $s_0 = (q_0, (\epsilon, \dots, \epsilon))$, i.e., the initial control state with no messages in the channels.

Figure 2.3 shows a simple FIFO automaton. Some of the strings in the trace language of this automaton are: $\theta_1, \theta_1\theta_1, \theta_1\theta_1\theta_1, \theta_1\theta_2, \theta_1\theta_2\theta_1\theta_2, \dots$

It is shown in [27] that FIFO automata are powerful enough to simulate a Turing machine even if they are restricted to using one channel and an alphabet with two letters. It is an easy consequence of this result that any non trivial property about FIFO automata is undecidable: there cannot exist an algorithm which will halt in all cases and correctly answer whether an arbitrary FIFO automaton satisfies the given property. This means that verifying CTL or LTL properties (and in particular safety properties) of FIFO automata is undecidable. However, as we will show in subsequent chapters, using learning techniques we can give a semi-algorithm for such verification (recall that a semi-algorithm is a procedure which produces a correct answer if it terminates but is not guaranteed to do so).

2.4.3 Regular Model Checking Framework

In a series of recent papers [1, 26, 2, 25, 77, 105], a common framework has been proposed which can represent a number of classes of infinite state systems including parameterized systems, integer systems, systems with push down stacks and even FIFO automata. The main idea is to represent states as strings over an alphabet and the transition relation is given by a transducer. Sometimes, the transition relation is restricted to be length-preserving and padding symbols are added to account for any differences in length. Using this framework, a set of states can be encoded as a regular language.

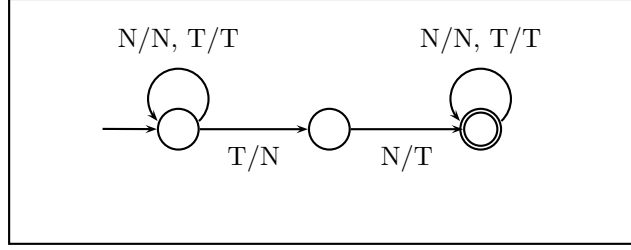


Figure 2.4: Transition relation for the token passing example

It is easy to show that the regular model checking framework is powerful enough to encode Turing machines; therefore, verification of such systems is undecidable in general.

The Kripke structure representing a system is given by $(\Sigma^*, AP, R_T, I, \mathcal{L})$ with I as a regular set in Σ^* , R_T as the transition relation of a transducer with input and output alphabet Σ . The labeling function is usually given implicitly by defining regular sets A_i for each atomic proposition $a_i \in AP$. More precisely, \mathcal{L} is given by $\mathcal{L}(s) = \{a_1, a_2, \dots, a_k\}$ iff for all $i \in 1 \dots k$ the string s is in A_i .

We now briefly mention the standard encodings used for some classes of systems in this method. The details can be found in [105].

Parameterized Systems in Array Topology. For system parameterized on the number of processes arranged in an array topology, a string $c_1c_2c_3 \dots c_n$ of length n is used to denote the global state of a system with n processes. The state of the i -th process is given by the letter c_i . Thus, the alphabet consists of letters which describe the local states that a particular process can be in.

An example of a system described by the regular model checking framework is the token passing protocol. The system consists of an arbitrary but unchanging number of processes. There is a single token in the system which can be passed by the process holding the token to the process to its left. The states of the system are modeled by a string over the alphabet $\{N, T\}$, where the length of the string is the number of processes in the system, the i^{th} element is T if the i^{th} process holds the token and the rest of the elements are N . Since this is a parameterized system, the length of the string could be arbitrary. The initial configuration is the set of strings TN^j which denotes $j + 1$ processes with a single token being held by the first process. There is a single transition in the system which is given by the transducer shown in Figure 2.4.

Integer Systems. A particularly useful class of systems expressible in the regular model checking framework is that of integer systems. The system consists of a finite number of control states and variables which can take values from the set of non-negative integers. This can also be used for representing parameterized systems in which all processes are identical allowing the configuration to be represented by a count of the

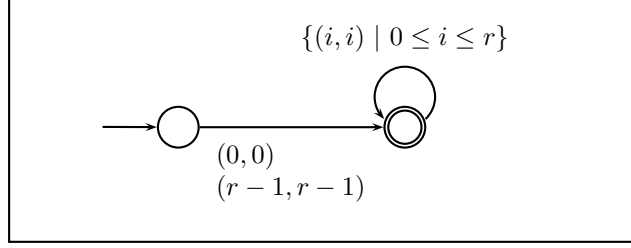


Figure 2.5: NDD representing the set $x = y$ for two integer variables x and y .

number of processes in finitely many control states.

Integer systems can be encoded in several different ways. One approach used in [105] is as follows. Consider a system with n integer variables x_1, x_2, \dots, x_n . Let each x_i be associated with a boolean variable b_i . Take the alphabet $\Sigma = \{true, false\}^n$ and let the state of the system be represented as a word over Σ such that b_i is *true* at position j iff $x_i = j$. This can encode sets constrained by equations like $x_1 < x_2$, $x_1 \geq x_2$ and $x_1 = x_2 + c$ for any constant c .

In another approach, integer systems can be encoded using Number Decision Diagrams (NDDs) [21]. Let \mathbb{Z} be the set of integers. First, we choose a *numeration basis* r which is greater than one. Any positive integer z can be encoded as a finite word $w = a_{p-1}.a_{p-2} \dots a_1a_0$ of digits belonging to $\{0, 1, \dots, r-1\}$, such that $z = \sum_{0 \leq i < p} a_i r^i$. Negative values can be encoded similarly using the method of *r*'s *complement*. To encode a vector value, each component is encoded using an identical number of digits (appending 0 digits if needed) and the digits that share the same position are grouped together. The encoding can then be viewed a word over the alphabet $\{0, 1, \dots, r-1\}^n$. It is shown in [21] that this encoding is powerful enough to express any Presburger set and in particular linear relations in integers. For example, the set $x = y$ with two integer variables can be expressed using the NDD shown in Figure 2.5. Decision procedures are available for the basic set operations of union, complement, projection and emptiness check.

FIFO Automata. FIFO automata can also be encoded in the *regular model checking* framework as follows. Using the notation in Section 2.4.2, the alphabet taken is $\Sigma = Q \cup \Sigma_M \cup \{\perp\}$. Here \perp is a filler symbol use to represent empty slots in the channels. For a single channel FIFO automata, each state can be represented as a word in $Q\perp^*\Sigma_M^*\perp^*$. The filler symbols allow the messages in the channel to grow or shrink as needed. The encoding for multi-channel FIFO automata is similar with additional “marker” symbols separating the letters for different channels.

Stack This is very similar to FIFO automata. The filler symbols are only needed at one end of the stack.

2.5 Learning Framework

As mentioned earlier in Section 1.1, a learning algorithm is usually set in a framework which describes the types of input data and queries available to the learner. There are two kinds of frameworks that are of interest to us: *passive learning* and *active learning*. We now briefly recapitulate these two frameworks.

In passive learning, the learner is simply given examples included in the target concept (positive examples) and examples not included in the target concept (negative examples). The learner is given successively larger sequences of positive and negative examples. If the learner is able to converge on the target language after being given a sufficiently large sample of positive and negative examples, it is said to identify the *language in the limit* [63]. The sample that is needed to guarantee this identification is said to be *characteristic*.

In the framework of *active learning* [12], the learning algorithm is given access to a knowledgeable teacher who can be queried. The teacher can be thought of as a pair of oracles: a *membership oracle* and an *equivalence oracle*. The membership oracle provides answers to queries about whether an example belongs to the concept being learnt or not. The equivalence oracle is an oracle which answers question about whether a hypothesis proposed by the learning algorithm is indeed equivalent to the concept being learnt. If at some point the learning algorithm's hypothesis is deemed correct by the equivalence oracle then the learning process stops. If on the other hand, the learner submits a hypothesis which is not equivalent to the target concept, the equivalence oracle not only says *no*, but also provides a counter-example to demonstrate why the hypothesis is wrong. The counter-example is either an example belonging to the hypothesis but not to the target concept, or it is an example belonging to the target concept but not to the submitted hypothesis.

We now describe some common algorithms used for the inference of regular languages.

2.5.1 RPNI

RPNI (regular positive and negative inference) [108, 48] is a well-known algorithm for the inference of regular languages in the passive learning framework. In this algorithm, the target concept to be learned is a *deterministic finite automata* (DFA) which accepts a regular language. Our treatment of RPNI follows [110].

For a regular language $L \in \Sigma^*$, the prefixes are defined as $Pr(L) = \{u \mid \exists v, uv \in L\}$. Let $L/u = \{v \mid uv \in L\}$ denote the right-quotient of L by u . The standard order of strings in Σ^* is denoted by $<$, *e.g.* for $\Sigma = \{a, b\}$, the ordering is $\epsilon, a, b, aa, ab, ba, bb, aaa, \dots$

Definition 4. The set of short prefixes of a language L is defined as follows:

$$Sp(L) = \{x \in Pr(L) \mid \nexists u \in \Sigma^* \text{ with } L/u = L/x \text{ and } u < x\}$$

Given a canonical (minimal) DFA M for a regular language L , every state of M corresponds to a unique element in the set $Sp(L)$ and M has as many states as elements in $|Sp(L)|$.

Definition 5. The kernel $N(L)$ is defined as follows:

$$N(L) = \{\epsilon\} \cup \{xa \mid x \in Sp(L), a \in \Sigma, xa \in Pr(L)\}$$

The kernel is an extension of a short prefix by a transition in the canonical DFA. Note that $Sp(L) \subseteq N(L)$. Thus, the set of short prefixes represent the states of the canonical DFA and kernel represents the set of transitions of the canonical DFA.

The input to RPNI consists of a set of positive samples S^+ accepted by the target DFA and a set of negative samples S^- rejected by the target DFA. Let $PTA(S^+)$ denote the *prefix tree acceptor* for S^+ which is a DFA containing a path from the start state to an accepting state for each string in S^+ modulo the common prefixes. The states of $PTA(S^+)$ are associated with the prefix of the strings that reach them, starting from the initial state. Further, they are numbered according to the standard ordering imposed on these prefixes. For example, the PTA for $S^+ = \{b, aa, aaaa\}$ is shown in Figure 2.6.

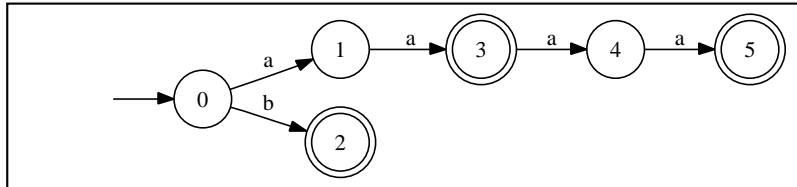


Figure 2.6: Prefix Tree Automaton for $S^+ = \{b, aa, aaaa\}$

Given a DFA M and a partition π on the set of states Q of M , we define the *quotient automaton* $M_\pi = (Q_\pi, \delta_\pi, \Sigma, B(q_0, \pi), F_\pi)$ obtained by merging states of M that belong to the same block of the partition as follows: $Q_\pi = \{B(q, \pi) \mid q \in Q\}$ is the new set of states with one for each block $B(q, \pi)$ of the partition, $F_\pi = \{B(q, \pi) \mid q \in F\}$ and $\delta_\pi : Q_\pi \times \Sigma \mapsto 2^{Q_\pi}$ is the transition function such that $\forall B(q_i, \pi), B(q_j, \pi) \in Q_\pi, \forall a \in \Sigma, B(q_j, \pi) \in \delta_\pi(B(q_i, \pi), a)$ iff $q_j = \delta(q_i, a)$.

RPNI performs an ordered search in $PTA(S^+)$ for possible merges of states while ensuring that after a merge, the resulting automaton does not accept any string in S^- . The algorithm is outlined in Figure 2.7.

```

algorithm RPNI
Input:  $S^+ \in \Sigma^*$ ,  $S^- \in \Sigma^*$ 
Output: a regular language L
begin
   $M \leftarrow PTA(S^+)$ 
  for  $i = 2$  to  $|M|$  do
    for  $j = 1$  to  $i - 1$  do
      if  $q_i, q_j$  not merged with smaller state then
         $M' \leftarrow merge(M, q_i, q_j)$ 
         $M'' \leftarrow determinize(M', q_j)$ 
        if compatible( $M'', S^-$ )
           $M = M''$ ; exit  $j$ -loop
    return language defined by  $M$ 
end

algorithm determinize
Input:  $M, x$ ; Output:  $M$ 
begin
  for any  $x \xrightarrow{a} x_1, x \xrightarrow{a} x_2$  and  $x_1 \neq x_2$ 
     $M \leftarrow merge(M, x_1, x_2)$ 
     $M \leftarrow determinize(M, \text{smaller of } x_1, x_2)$ 
  return  $M$ 
end

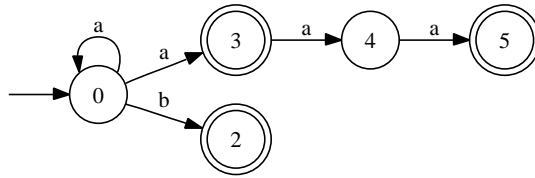
```

Figure 2.7: RPNI algorithm

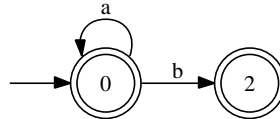
Some of the intermediate automata encountered by RPNI for an input set consisting of $S^+ = \{b, aa, aaaa\}$ and $S^- = \{\epsilon, a, aaa, baa\}$ are shown in Figure 2.8.

It has been shown in [108] that if (S_c^+, S_c^-) form what is called a *characteristic* sample then RPNI is guaranteed to find the correct automaton in time polynomial in the sample size. A set of examples (S_c^+, S_c^-) is *characteristic* for the RPNI algorithm for a regular language $L \in \Sigma^*$ if the following conditions hold:

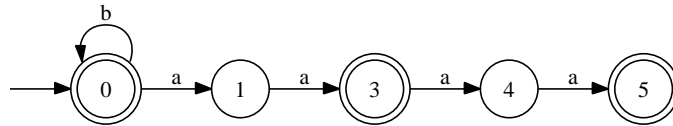
1. $\forall x \in N(L)$ if $x \in L$ then $x \in S_c^+$ else $\exists u \in \Sigma^*$ with $xu \in S_c^+$
2. $\forall x \in Sp(L), \forall y \in N(L)$ if $L/x \neq L/y$ then $\exists u \in \Sigma^*$ such that $(xu \in S_c^+, yu \in S_c^-)$ or $(yu \in S_c^+, xu \in S_c^-)$.



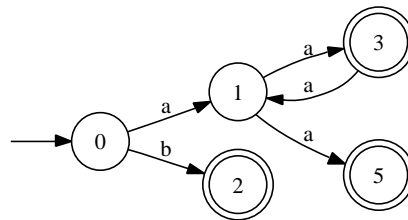
a) M' after merging states 0 and 1



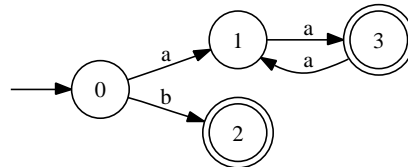
b) M'' after determinizing for the merge between states 0 and 1. M'' incompatible with S^- ; hence the merge is rejected



c) M' after merging states 0 and 2. The automaton M'' obtained after determinizing for the merge is the same as M' . Since M'' is incompatible with S^- , the merge is rejected



d) M' after merging states 1 and 4



e) M'' after determinizing for the merge between states 1 and 4. M'' compatible with S^- so the merge is accepted

Figure 2.8: Intermediate automata for a run of the RPNI algorithm

2.5.2 Angluin's L^*

Angluin's L^* algorithm [12] is an active learning algorithm for regular sets. This requires what is called a *Minimally Adequate Teacher* which provides an oracle for membership (whether a given string belongs to a target regular set) and equivalence queries (whether a given hypothesis matches the target regular set). In case the teacher answers *no* to an equivalence query, it also provides a string in the symmetric difference of the hypothesis and the target sets. The main idea behind Angluin's L^* algorithm is to systematically explore strings for membership and create a DFA with minimum number of states to make a conjecture for the target set. If the conjecture is incorrect, the string returned by the teacher is used to make corrections, possibly after more membership queries. The algorithm maintains a prefix closed set S representing different possible states of the target DFA, a set $S.\Sigma$ for the transition function consisting of strings from S extended with one letter of the alphabet and a suffix closed set E denoting *experiments* to distinguish between states.

An observation table O_T can be visualized as a two-dimensional array with rows labelled by elements of $(S \cup S.\Sigma)$ and columns labeled by elements of E , with the entry for row s and column e equal to $O_T(s.e)$. If s is an element of $(S \cup S.\Sigma)$ then let $row(s)$ denote the function f from E to $\{0,1\}$ defined by $f(e) = O_T(s.e)$. An observation table is called *closed* provided that for each t in $S.\Sigma$ there exists an s in S such that $row(t) = row(s)$. An observation table is called consistent provided whenever s_1 and s_2 are elements of S such that $row(s_1) = row(s_2)$, for all $a \in \Sigma$, $row(s_1.a) = row(s_2.a)$. If O_T is a closed, consistent observation table, we can define a DFA $M(S, E, O_T) = (Q, \delta, \Sigma, q_0, F)$ as follows: $Q = \{row(s) \mid s \in S\}$, $q_0 = row(\epsilon)$, $F = \{row(s) \mid s \in S \text{ and } O_T(s) = 1\}$ and $\delta(row(s), a) = row(s.a)$. It can be shown that the above definition is well defined for closed, consistent observation tables.

Angluin's algorithm is guaranteed to terminate in polynomial time with the minimal DFA representing the target set. Figure 2.9 shows the outline of the algorithm.

As an example, consider a regular set over the alphabet $\Sigma = \{0,1\}$ consisting of all strings in which the number of 1's is 3 modulo 4. The various intermediate automata and the final target automata for the Angluin's algorithm are shown in Figure 2.10.

```

algorithm Angluin
begin
  Initialise  $S$  and  $E$  to  $\{\epsilon\}$ 
  Ask membership queries for  $\epsilon$  and each  $a \in \Sigma$ 
  Construct the initial observation table  $(S, E, O_T)$ 
  Repeat
    While  $(S, E, O_T)$  is not closed or not consistent
      If  $(S, E, O_T)$  is not consistent
        find  $s_1$  and  $s_2$  in  $S$ ,  $a \in \Sigma$  and  $e \in E$  such that
           $row(s_1) = row(s_2)$  and  $O_T(s_1.a, e) \neq O_T(s_2.a, e)$ 
        add  $a.e$  to  $E$ 
        extend  $O_T$  to  $(S \cup S.\Sigma).E$  using membership queries
      If  $(S, E, O_T)$  is not closed
        find  $s_1 \in S$ ,  $a \in \Sigma$  such that
           $row(s_1.a)$  is different from  $row(s)$  for all  $s \in S$ 
        add  $s_1.a$  to  $E$ 
        extend  $O_T$  to  $(S \cup S.\Sigma).E$  using membership queries
    Once  $(S, E, O_T)$  is closed and consistent, let  $M = M(S, E, O_T)$ 
    Make the conjecture  $M$ 
    If Teacher replies with counter-example  $t$ 
      add  $t$  and all its prefixes to  $S$ 
      extend  $O_T$  to  $(S \cup S.\Sigma).E$  using membership queries
  Until Teacher replies yes to the conjecture
  return language defined by  $M$ 
end

```

Figure 2.9: Angluin's algorithm

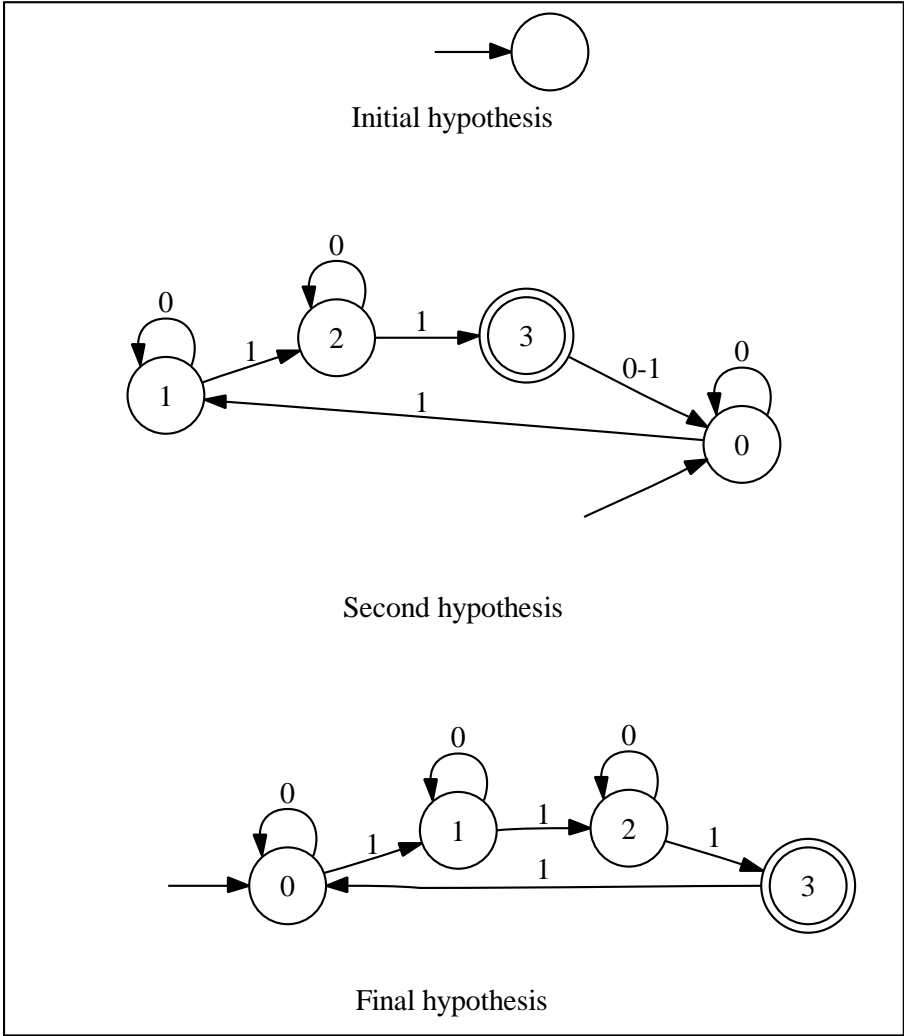


Figure 2.10: Intermediate automata for a run of Angluin's L* algorithm

2.5.3 Variations of Angluin’s algorithm

A number of researchers have proposed variations of the classical Angluin’s L^* algorithm. The specific algorithm that we use to learn regular languages is due to Kearns and Vazirani [82]. The algorithm maintains a candidate DFA for the language being learnt. This DFA is minimal- every pair of states can be *distinguished* by a string that leads one to an accept state and the other to a non-accept state. Progressively, the algorithm queries the equivalence oracle and the membership oracle, and stops when the equivalence oracle deems the candidate DFA to be correct. In this process, the learner may discover inconsistencies with the current DFA, which it resolves by splitting certain states. The details of the algorithm determine how the DFA is represented, which queries are asked, and how inconsistencies when discovered are resolved by splitting states. In what follows, we present only a brief overview of the algorithm, giving only the details that are relevant to highlight our improvements. For a comprehensive presentation, the reader is referred to [82].

Each state of the candidate DFA is represented as a string, namely, the input string that takes the DFA from the initial state to the given state. These strings are called *access strings* and the learner maintains a set S of such access strings. In addition, the learner also maintains a set E of *distinguishing strings*, that demonstrate why each string in S will correspond to a different state in the desired minimal DFA. So for any two access strings s_1 and s_2 there is a string $d \in E$ such that exactly one out of $s_1.d$ and $s_2.d$ belongs to the regular language being learnt. The algorithm maintains the sets S and E in a binary *classification tree* where each internal node is labeled by a string in E and each leaf by a string in S . Any string d that distinguishes between two access strings is placed at the root of the tree. All access strings s such that sd belongs to the concept being learnt are placed in the right subtree and the rest are placed in the left subtree. This process is repeated for each subtree until each access string is at its own leaf. The classification tree implicitly represents the current candidate automaton in the following manner. The states correspond to the access strings S . The transition on a symbol a from a state s can be obtained by “sifting” through the classification tree as follows. Start from the root (say labeled d), and check whether the string $s.a.d$ belongs to the learnt concept or not, by asking the membership oracle. If $s.a.d$ belongs to the concept then move to the right child else move to the left child. Repeat this process recursively, until a leaf is reached. This is the target state of the transition.

Observe, that the process of “sifting” can be used to determine whether two strings s_1 and s_2 must go to different states in the smallest DFA for the concept: if sifting s_1 and s_2 result in reaching different leaves then s_1 and s_2 must go to different states. This observation plays a crucial role in determining how the candidate DFA must be refined when an inconsistency is encountered. Suppose that the equivalence oracle declares that the proposed candidate DFA M is different from the concept DFA \hat{M} as evidenced by a counterexample

w . In our variant of the Kearns and Vazirani algorithm, we use Rivest and Schapire’s [114] idea of a binary search to discover a distinguishing string needed to resolve the inconsistency shown by w . We now briefly explain this idea. Since w is a counterexample, it is accepted by only one of M and \hat{M} (say it is accepted by M). Let s_i be the access string of the state reached in M on processing the first i symbols of w . Further, let $\alpha(i)$ be a function defined as follows. We concatenate s_i with the remaining symbols in w after position i and process the resulting string on \hat{M} ; if the state reached in \hat{M} is accepting, $\alpha(i) = \text{true}$ else $\alpha(i) = \text{false}$. From our previous comments, it follows that $\alpha(0) = \text{false}$ and $\alpha(|w|) = \text{true}$. Using a kind of binary search, we can find some j such that $\alpha(j) \neq \alpha(j+1)$. Let u be the prefix of w such that $|u| = j$, let b be the next symbol in w and let v make up the rest of the symbols in w . Sifting ub in the classification tree leads us to the access string s_{j+1} . However, if we sift u to get access string s_j , by construction of the candidate DFA, s_jb also has to sift to s_{j+1} . Thus, two strings s_{j+1} and s_jb go to the same state in M but the counterexample shows that they can be distinguished in the concept DFA by the string v (since $\alpha(j) \neq \alpha(j+1)$). Therefore, M must be refined by splitting the state corresponding to the access string s_{j+1} into two states: one with the original access string s_{j+1} and another with access string s_jb . The classification tree also needs to be updated correspondingly by changing the leaf with s_{j+1} into an internal node with v as the distinguishing string. The new node gets two leaves as children, one labeled with s_jb and the other with s_{j+1} .

To update the transitions of the candidate DFA M , a naive approach requires that for each state s and for each $a \in \Sigma$, we find the target of the a -transition by sifting $s.a$ down the classification tree. However, on closer examination, we see that sifting is needed only for the transitions coming out of the new state labeled s_jb . We do need to update any transition incoming to the old state labeled s_{j+1} that was split. Consider one such a -transition coming from some state labeled s . The destination of this transition will either remain as the state labeled s_{j+1} or will be the new state labeled s_jb depending on whether membership of $s.a.v$ in the target concept matches the membership of $s_{j+1}.v$ or that of $s_j.b.v$.

Figures 2.11, 2.12 and 2.13 list the algorithm.

Proposition 1. *The regular inference algorithm described above always terminates after making $O(|\Sigma|n^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. Here, Σ is the alphabet, n is the number of states in the minimal automaton representing the target regular language, and m is the size of the longest counterexample returned by the teacher.*

Proof. The algorithm executes the main loop exactly $n - 2$ times. Therefore, the number of equivalence queries is $O(n)$. In the i th iteration of the main loop, the classification tree has $i+1$ leaves and the hypothesis has $i + 1$ states. While processing *Update-Tree-Hypothesis*, the search for a distinguishing string needs $O(\log m)$ membership queries. Further, *Update-Hypothesis* needs $O(i)$ membership queries to update the

```

Procedure Sift
Input: String  $s$  in  $\Sigma^*$ , classification tree  $CT$ 
Output: Access string for some leaf in  $CT$ 
begin
  Set current node to be root of  $CT$ 
  While current node is not a leaf do
    Let  $d$  be the distinguishing string at the current node
    If  $isMember(sd)$  {  $isMember$  checks if string is in the target concept }
      update current node to be the right child of current node
    else
      update current node to be the left child of current node
  return access string of current node
end

```

Figure 2.11: Procedure *Sift* used in learning automata

transitions coming into the old state and $O(|\Sigma|)$ sifting operations to find destinations of the transitions for the new state. Each sifting operation can result in $O(i)$ membership queries. Thus, we need $O(i|\Sigma| + \log m)$ membership queries for the call to *Update-Tree-Hypothesis* in the i th iteration. Since the number of iterations is $O(n)$, the total number of membership queries is $O(|\Sigma|n^2 + n \log m)$.

□

Procedure *Update-Hypothesis*

Input: Classification tree CT , hypothesis M , old access string s_{old} ,
new access string s_{new} , new distinguishing string v

Output: DFA M updated for new hypothesis

begin

 Create a new state q_{new} in M labeled s_{new}

 If $isMember(s_{new})$ then make q_{new} accepting

 Let q_{old} be the state labeled s_{old}

 For each a -transition coming from some state q into q_{old}

 Let s be the access string for q

 if $isMember(s.a.v) = isMember(s_{new}.v)$

 Change destination of transition to q_{new}

 For each $a \in \Sigma$

$s' \leftarrow Sift(s_{new}.a, CT)$

 Direct the the a -transition from q_{new} to the state labeled s'

end

Procedure *Update-Tree-Hypothesis*

Input: Classification tree CT , Current hypothesis M ,

Counterexample $w = a_0a_1 \dots a_{|w|-1}$

Output: Classification tree CT updated

begin

$l = 0, h = |w|$

$\alpha_l = isMember(w), \alpha_h = \neg\alpha_l$

 do

$m = \lfloor \frac{l+h}{2} \rfloor$

$prefix \leftarrow a_0a_1 \dots a_{m-1}$

$suffix \leftarrow a_ma_{m+1} \dots a_{|w|-1}$

$s_m \leftarrow$ access string for state reached by processing $prefix$ on M

$\alpha_m \leftarrow isMember(s_m.suffix)$

 if $\alpha_m = \alpha_h$ then $h \leftarrow m$ else $l \leftarrow m$

 while $l < h - 1$

$s_j \leftarrow$ access string for state reached by processing $a_0a_1 \dots a_{l-1}$ on M

$s_{j+1} \leftarrow$ access string for state reached by processing $a_0a_1 \dots a_{h-1}$ on M

$v \leftarrow a_ha_{h+1} \dots a_{|w|-1}$

 Replace the node labeled with s_{j+1} with an internal node with two leaf nodes

 One leaf node is labeled with s_{j+1} , other is labeled with $s_j.a_l$

 New internal node gets the distinguishing string v

Update-Hypothesis($CT, M, s_{j+1}, s_j.a_l, v$)

end

Figure 2.12: Procedure *Update-Hypothesis* and *Update-Tree-Hypothesis* used in learning automata

```

Algorithm Learn-Automaton
Input: Access to membership and equivalence oracle
Output: DFA which accepts target regular language
begin
  if isMember( $\epsilon$ )
    Create a one state hypothesis DFA  $M$  which accepts all strings
  else
    Create a one state hypothesis DFA  $M$  which rejects all strings

  Perform an equivalence query with  $M$ , let  $w$  be the counterexample
  Initialize a classification tree  $CT$  with root labeled  $\epsilon$ ,
  and two leaves labeled  $\epsilon$  and  $w$ 
  Update-Hypothesis( $CT, M, \epsilon, w, \epsilon$ )

  do
    Perform an equivalence query with  $M$ , let  $w$  be the counterexample
    Update-Tree-Hypothesis( $CT, M, w$ )
  while equivalence query does not answer yes
end

```

Figure 2.13: Learning automata based on Kearns and Vazirani algorithm

Chapter 3

Related Work

In this chapter, we review the related work in the area of verification of software systems. The discussion is kept fairly broad and the detailed comparison of methods which can be directly contrasted to our learning-based verification is deferred to subsequent chapters.

3.1 Verification of Finite State Systems

Algorithmic verification of finite state systems goes back to the 70's and there is a large amount of literature devoted to this problem. The method was pioneered independently by Clarke and Emerson [37] and Queille and Sifakis [78] and came to be known as *model checking*. The initial algorithms for model checking were given for computational tree logic (CTL) but a tableau-based algorithm for checking LTL specifications was given a few years later by Lichtenstein and Pnueli [107]. For a thorough treatment of the verification of finite state systems, the reader is referred to Clarke's textbook [39].

One of the main problems faced by algorithmic verification is the *state explosion* that can occur as the number of components in the system being verified increase. To mitigate this problem, sets of states are sometimes represented symbolically in the hope of having a compact representation. For finite systems, a popular symbolic representation is that of ordered binary decision diagrams (BDD) [28]. The symbolic approach has enjoyed considerable success in hardware verification [79] where the state space exhibit considerable regularity and allows for compact BDD based representations. There are a large number of tools implementing this technique, two of the well known ones are: SMV [117] and NuSMV [35].

Another popular approach for verifying finite state systems is using the theory of automata on infinite objects [124] for specifying both the specification as well as its implementation. Vardi and Wolper [131] were the first to reformulate LTL model checking in terms of automata on infinite words. The verification of the system is reduced to checking whether there is an intersection in the language accepted by the automata representing the system and the language of the automata representing the "bad" behaviors. The intersection is checked using nested depth first search and is usually constructed on the fly to avoid constructing the

entire state space of the modeled system. A description of such an algorithm is given by Gerth *et al.* in [61]. Some of the tools that include this method are: SPIN [74] and the Maude LTL model checker [51].

The automata-theoretic approach has also been applied to branching time logics by Kupferman *et al.* [84] using tree automata.

3.2 Verification of Infinite State Systems

The verification problem can be shown to be undecidable for most classes of infinite state systems. For a survey on known decidability results, the reader is referred to [103]. An interesting class of systems for which verification of safety properties turns out to be decidable is the class of *well-structured* transition systems [56] which include Petri Nets, Basic Process Algebra and Lossy FIFO automata.

We now discuss some of the techniques employed for verification of general infinite state systems.

3.2.1 Deductive Methods

The technique of *deductive methods* involves a *proof system* consisting of a set of *axioms* which are assumed to be self evident and a number of inference steps which allow new facts to be shown true based on existing ones. Using this system, a person can write down a sequence of logical formulas referred to as a *proof* which demonstrate that a system satisfies a given specification. Each formula in a proof is either an axiom or can be derived from formulas earlier in the proof using some inference steps. For verification, usually the proof is checked mechanically. Often a proof assistant is available which allows the user to concentrate on large steps in the proof while automatically proving some of the easier steps. The main advantage of this method is that it is possible to prove systems of any size. However, it requires a high degree of human guidance and effort and deep understanding of both the system and the specification.

Various proof systems have been developed specifically tailored for verification needs. If the system is expressed in rewriting logics such as Maude [41], proof rules of the underlying logic can be used to show that system is correct [101, 40]. Manna and Pnueli [94] present various proof rules tailored for proving temporal properties. These proof rules can also be recast into a graphical form of *verification diagrams* [66, 95]. Tools available in theorem proving vary in the amount of expressiveness and human guidance they require. One example is the theorem prover ACL2 [81] which allows a high degree of automation and has been successfully used a number of industrial sized projects. Some other well known theorem proving tools include PVS [109], HOL [64], Coq [18] and Isabel [111].

3.2.2 Abstraction

A technique often employed to reduce the state space of a large or even infinite state system is to use *abstraction*. The idea behind abstraction is to construct a model of the system by collecting many concrete states into a single state. Usually, the abstract model exhibits more behaviors than the concrete one which introduces the possibility of a spurious answer. In rewriting logic, *equational abstractions* [99] are used to create abstract models of even infinite state systems by collapsing concrete states using equations. In the approach of *predicate abstraction* [116, 46], the abstract states are states satisfying certain predicates which are usually derived from the guards of the concrete program. There has been considerable interest in predicate abstraction in recent years and it has been used to analyze critical safety properties of programs written in high level languages such as C [72, 16, 33, 45]. The main idea is as follows. Given a C program and a set of seed predicates (possibly empty), an abstraction is created by essentially considering only boolean variables as the available data variables. These boolean variables correspond to the current set of predicates maintained by the verifier. Each true or false value of a boolean variable corresponds to the set of concrete states that make the corresponding predicate true or false respectively. Since the only variables available in the abstract domain are booleans, the abstract state space is necessarily finite and model checking tools can be applied to verify its properties. It has to be ensured that the abstract transition system is a conservative approximation of the concrete program (meaning that it exhibits at least all the executions of the concrete system). For this, a conservative pointer analysis is used to find any potential aliases and a theorem prover is used to check if a transition can take place from one abstract state to another. Model checking the abstract transition relation can either prove the property (in which case we are done) or otherwise return a counterexample. The counterexample is symbolically executed on the concrete system to check if it is spurious (which is possible due to the additional executions introduced due to the abstraction). If the counterexample turns out to be valid then the program has been shown to have violated the desired property. Otherwise, a mechanism is used which generates additional predicates from the proof that the counterexample is invalid. These predicates are then added to the existing set of predicates and the process repeated. Note that the process is not guaranteed to terminate.

3.2.3 Symbolic Representations for Infinite State Systems

Since the domain of the variables in infinite state systems is unbounded, the state space has to necessarily be represented via symbolic means. Some common representations are: regular sets [25, 2], Presburger formulas [30], Queue Decision Diagrams and Number Decision Diagrams [21], semi-linear regular expressions [55], constrained QDDs [24], Hilbert's basis for integer sets [115] and constrained facts [47].

3.2.4 Acceleration and Meta-transitions

In the approach using *meta-transitions* and *acceleration* [21, 24, 55], a sequence of transitions, referred to as a *meta-transition*, is selected and the effect of its infinite iteration calculated. The idea is that by selecting enough sequences of such transitions, the entire set of reachable states is hoped to be captured. One feature of this approach is that the set of states computed is always equal to or an under-approximation of the actual reachable set. Therefore, there is no possibility of a spurious result claiming that a bad state is reachable. However, it is possible that not all sequence of transitions can be accelerated. Further, even if all sequences of transitions can be accelerated there may not be any finite set of sequences that is able to account for all the reachable states.

An acceleration scheme has been given by Boigelot [21] for FIFO automata using Queue Decision Diagrams (QDD). It is shown that for systems with one channel, any sequence of transitions can be accelerated. More precisely, for a set of states Q , let $\sigma(Q)$ denote the set of states that is obtained by taking a sequence of transitions σ from some state in Q . Then, for any set of states Q representable by a QDD, the set of states corresponding to $\sigma^*(Q)$ can also be represented as a QDD and is effectively computable. However, this result does not extend for systems with more than one channel. Let $\sigma_{|i}$ denote the sequence obtained by dropping all transitions in σ that do not affect channel i . For multi-channel FIFO automata, given a QDD representable set Q , $\sigma^*(Q)$ is representable by a QDD if and only if there is at most channel i such that $\sigma_{|i}$ is a “counting” sequence. Here, “counting” is a technical condition which compares the number of output operations with the number of input operations on that channel. Boujjani and Habermehl [24] extend the acceleration method using another symbolic representation called *Constrained Queue Decision Diagrams* and show that any sequence of transitions can be accelerated with this representation. Finkel *et al.* [55] propose *Semi Linear Regular Expressions* which are less expressive than both QDDs and CQDDs but are shown to have more efficient decision procedures that are needed for verification.

For integer systems, Boigelot [21] proposed Number Decision Diagrams (NDDs) as a symbolic representation and gave conditions under which a sequence of transitions can be accelerated. Finkel and Leroux [57] generalized this result and showed that any sequence of transitions for *finite linear system* can be accelerated. Here, a *finite linear system* is defined as follows. First, a *Presburger-linear* function f is a triple (M_I, v, ϕ) such that M is a square matrix of integers, v is a vector of integers and ϕ is a Presburger formula such that $f(s) = M_I \cdot s + v$ for all s which satisfy ϕ . A finite linear system is collection of such Presburger linear functions such that the multiplicative monoid generated by the matrices of the functions is finite. Another important result is that for a finite linear system, the effect of accelerating all cycles of length less than or equal to k can be computed exhaustively by considering only a polynomial (in k) number of accelerations.

Bardin *et al.* [17] describe a tool called FAST which implements these accelerations along with some heuristics for improved performance on some practical examples. Other tools based on acceleration are LASH [87] and TReX [13].

Leroux and Sutre [89] define a notion of *flatness* to identify sufficient conditions when acceleration-based methods will terminate (recall that the acceleration method is only a *semi*-algorithm). A system is called *flat* if it can be decomposed into a finite number of subsystems such that each subsystem contains no nested loops and the reachable states of the entire system is the union of the reachable states of each subsystem. It is shown that flatness is a necessary and sufficient condition for the acceleration based methods to terminate. Further, they show that some restricted classes of systems such as two-dimensional vector addition systems with states are flat. However, this does not extend to higher dimensions as shown by a 3-dimension counterexample in [89]. It can also be shown that even if the set of reachable states of a system is Presburger-definable, there may not be any finite set of acceleration that can capture it.

3.2.5 Widening

Widening is an approach used in the framework of *abstract interpretation* [43] to ensure termination in fixpoint iterations. The idea is to define a widening operator ∇ which overapproximates the iterates of the fixpoint calculation at every step and has the property that for every increasing sequence x_0, x_1, \dots the “widened” sequence given by y_0, y_1, \dots such that $y_0 = x_0$ and $y_n = y_{n-1} \nabla x_n$ is not strictly increasing. This ensures that the widened sequence always terminates and the final set obtained is larger than the actual least fixpoint. Clearly, if this set does not intersect the “bad” states, the system verifies the safety property. However, the converse is not true due to potential overapproximation.

A simple widening principle in the context of regular model checking is given in [25] which is extended in [125] for parametric systems. Bultan [30] uses a widening technique for Presburger formulas and Bartzis *et al.* [19] present a widening technique for arithmetic automata.

3.2.6 Techniques used in Regular Model Checking

A popular approach for FIFO, parametric, integer and stack systems is *regular model checking* [25, 2]. A regular set is used to represent the states and a transducer is used to represent the transition relation (say R). The systems they consider are length preserving, therefore the transducer can also be seen as a finite automaton over pairs of symbols. The problem is reduced to finding a finite transducer R^+ representing the infinite composition of the transition relation. A finite quotient of R^+ is found by identifying states that are equivalent in some way. There are two main methods used for finding equivalence. In the first

method [25], the underlying equivalence is found using a technique called *saturation* which is employed while determinizing on the fly and minimizing R^+ . The second method described initially by Dams *et al.* [44] is as follows. A *forward* bisimulation and a *backward* bisimulation is defined on states of finite compositions of the transducer representing the transition relation. It is shown that the forward and backward simulations can be combined into an equivalence relation which preserves the transitive closure of the transducer. This can be used to collapse states in intermediate approximations of R^+ in the hope of capturing R^+ . The technique was extended to simulations (as opposed to bisimulations) in [2]. Abdulla *et al.* [1] also extend the results for checking liveness properties using a specification logic called LTL(MSO).

Another related technique for finding the transitive closure of the transition relation labeled as “iterating transducers” is described by Boigelot *et al.* in [22]. In this method, finite compositions of the transducer are compared to detect certain “increments”. These increments are then extrapolated in the hope of capturing the transitive closure. However, it is ensured that all extrapolations are *safe* in the sense that it is guaranteed that the resulting construction will not exceed the transitive closure.

3.2.7 Bisimulation Minimization

Bisimulation minimization algorithms [88, 23] partition a state space into equivalence classes such that states in the same class are observationally equivalent. The resulting quotient graph preserves the same CTL* (actually same μ -calculus) properties and hence can be used for verifying the temporal logic formula of interest, instead of the original system. The main idea is as follows. One starts with a partition of the states such that states in each block agree on the labeling of the atomic propositions and then repeatedly splits the blocks into new ones until all states in a block agree on the next-state transition to other blocks. The main differences in the available algorithms are in how they attempt to avoid splitting blocks which are unreachable from a set of initial states. A related technique is to find a finite simulation quotient ([70]) which is coarser than bisimulation quotient but only preserves the universal fragment of CTL* properties.

3.2.8 Techniques using Rewrite Systems

Rewrite systems offer a convenient representation of concurrent systems and a number of different techniques have been developed for verification of infinite state systems in this context. Monniaux [104], Takai [119] and Genet *et al.* [59] use abstract interpretation techniques with tree automata. Meseguer and Thati [100, 123] have developed *narrowing* techniques for reachability problems in rewrite systems. Narrowing is a method that was originally developed for generating all solutions of an equational unification problem. Escobar *et al.* [53] propose an efficient *lazy* narrowing strategy called *natural narrowing*.

3.2.9 Verification of Hybrid Systems

Hybrid systems [7, 118, 5, 6, 71] are a class of infinite state systems in which there can be both discrete and continuous variables. A hybrid system is typically modeled as a finite automaton augmented with some continuous variables which follow given dynamic laws. If the invariant, initial, flow and the jump conditions of the hybrid automaton are boolean combinations of linear inequalities, the automaton is called a *linear hybrid automaton*. If all dynamic variables follow continuous trajectories with a unit value of slope, the automaton is also called a *timed automaton*. Another class of hybrid systems are *rectangular automata* in which the trajectories are confined to piecewise linear envelopes. It is known that some verification problems for classes of hybrid systems can be solved algorithmically [73]. For example, the control state reachability problem for timed and rectangular automata are decidable. For other systems, the state space is usually represented by symbolic representations such as polyhedra [34] and ellipsoids [86] and approximations developed which allow verification in some cases.

3.3 Use of Learning for Verification

Note that in our approach of using learning techniques for verification, we are not trying to learn an unknown system model but rather the behavior of a system which is already fully described. This is closest in spirit to the work of Habermehl *et al.* [67] who use learning for verification of systems whose transition can be represented by a length-preserving transducer. They find all strings of a certain length that can be reached from the initial state and use a state merging algorithm to learn the regular set representing the reachable region. Their work generalizes the inference based method of verifying parameterized systems used by Fribourg *et al.* [58]. However, the correctness of their algorithm critically depends on the length-preserving aspect of the transition relation. Further, their learning algorithm requires collecting all samples of a given length which can be intractable for large examples.

We now describe some other approaches where learning has been used for verification.

Peled *et al.* [112] give a method called “Black Box Checking” which is extended by Groce *et al.* [65] as *Adaptive Model Checking*. Briefly, in this method, one starts with a possibly inaccurate model and incrementally updates it using Angluin’s [12] query based learning of regular sets.

Learning techniques have been popular for automated assume-guarantee reasoning. Cobleigh *et al.* [42] also use a variant of Angluin’s algorithm to learn the assumptions about the environment to aid compositional verification. Alur *et al.* [9] introduce a symbolic implementation of the learning technique for compositional verification which helps to mitigate the exponential explosion in the alphabet size with the number of

variables in the system. Chaki *et al.* [38] learn deterministic tree automata with membership and equivalence queries for applying assume-guarantee reasoning for simulation conformance.

Boigelot *et al.* [15] present a technique for constructing a finite state machine that simulates all observable operations of a given reactive program. Ammons *et al.* [11] use learning to discover formal specifications of the protocols that a client of an application program interface must observe. Similarly, Alur *et al.* [4] use Angluin’s L^* algorithm to synthesize interface specifications from Java classes. Inductive learning techniques have been applied ([92]) for abstraction refinement in program analysis targeted towards programs which manipulate pointers and heap-allocated data structures. Edelkamp *et al.* [50] consider the problem of finding “bad” states in a model as a directed search problem and use AI heuristic search methods to attempt to find these states. Ernst *et al.* [52] have developed a system called *Daikon* which attempts to discover likely invariants in a program by analyzing the values taken by its variables while the program is exercised in a test suite.

Chapter 4

FIFO Automata Safety Using Passive Learning

Finite state machines communicating over unbounded FIFO (first in first out) channels (*FIFO automata*) are a popular model for a variety of software systems. Examples of this abstraction include: networking protocols where unbounded buffers are assumed, languages like Estelle and SDL (Specification and Description Language) in which processes have infinite queue size, distributed systems and various *actor* systems.

In this chapter, we describe a method of verifying safety properties for FIFO automata using a learning algorithm called RPNI. The focus is on *passive learning*, *i.e.*, we assume that the learner is simply given positive and negative examples of the concept being learned.

For verifying safety properties of FIFO automata (or in fact in general for any system), we are usually given a set of states labelled “unsafe” and are required to check if any of these unsafe states can be reached from the initial states. Thus, a generic task in the verification of safety properties is to compute a representation for the set of reachable states. In traditional model checking methods, this is accomplished by starting from the initial states and iteratively computing its image under the transition relation until the resulting set stops changing (the least fixpoint is reached). However, for FIFO automata (and most other infinite state systems), clearly, this procedure may not terminate.

As mentioned in Chapter 1, the central idea in our approach is to learn the set of reachable states instead of computing it by iteratively applying the transition relation. However, to use learning algorithm (in the passive learning framework), we need positive and negative examples of the reachable states. As we pointed out before, we can easily find positive examples by executing some sample sequences of transitions; but in general, we cannot get examples of unreachable states. The solution is to not just learn the set of reachable states, but to actually learn a set of state-witness pairs where for a pair (s, w) , the witness w shows how the state s is reachable. The witness could be anything for which checking validity is decidable. With the witness, the problem of supplying *positive examples* and the *negative examples* is solved as follows. We simulate the FIFO automata under consideration using either a test suite, or randomly selecting from the executable transitions, or any other strategy which ensures a fair chance for all reachable states to be explored. As we execute the transitions, we see examples of the reachable states and can also record information about

the transition executions which gives us witnesses for those states (the details of the witness are discussed later in this chapter). For negative examples, we can use any (s, w) pair such that the witness w does not demonstrate the reachability of the state s .

With this idea of learning state-witness pairs, we can use passive learning algorithms to verify safety properties of FIFO automata. Figure 4.1 shows the overall framework of the verification procedure. The intuitive explanation of the procedure is as follows. We start with some system executions which give us the initial set of positive and negative examples. These are given to a passive learning algorithm (in our case, an algorithm derived from RPNI [108]) which then produces a candidate set of the state-witness pairs. We now check if these states includes any of the unsafe states. If there is some unsafe state and the corresponding witness is valid, then we have found a valid counterexample to the safety property and can stop. On the other hand, an invalid state-witness pair can be added as an additional negative example for the learner. If the set of state-witness pairs does not include any unsafe states, we check if the set is a fixpoint under the transition relation. If yes, we have found some set greater than the reachable states (which is the least fixpoint) and does not include any unsafe states. Clearly, this shows that the safety property holds for the system. If the fixpoint test fails, then we generate more system executions and add the corresponding positive and negative examples to the learner input.

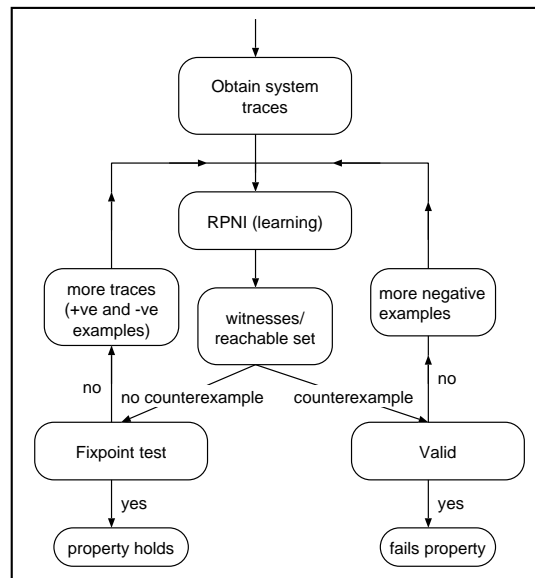


Figure 4.1: Learning to verify procedure for safety properties using passive learning.

4.1 Verification Procedure

We now describe the learning-based verification procedure in detail.

Recall from Chapter 2 that a FIFO automaton [55] is a 6-tuple $(Q, q_0, C, \Sigma_M, \Theta, \delta)$ where Q is a finite set of *control states*, $q_0 \in Q$ is the initial control state, C is a finite set of *channel names*, Σ_M is a finite alphabet for contents of a channel, Θ is a finite set of transitions names, and $\delta : \Theta \rightarrow Q \times ((C \times \{?, !\} \times \Sigma_M) \cup \{\tau\}) \times Q$ is a function that assigns a *control transition* to each transition name.

As mentioned before, the key idea to make the our method work is to learn the set of state-witness pairs. Let us now consider the language which can allow us to find both reachable states and their witnesses. The first choice that comes to mind is the language of the traces:

$$L(F) = \{\sigma \in \Theta^* \mid \exists s = (p, w). s_0 \xrightarrow{\sigma} s\}$$

Here $s_0 = (q_0, (\epsilon, \dots, \epsilon))$, i.e., the initial control state with no messages in the channels. Since each trace uniquely determines the final state in the trace, $L(F)$ has the information about the states that can be reached. While it is easy to compute the state s such that $s_0 \xrightarrow{\sigma} s$ for a *single* trace σ , it is not clear how to obtain the set of states reached, given a *set of traces*. In fact, even if $L(F)$ is regular, there is no known algorithm to compute the corresponding set of reachable states of the labelled transition system.¹ The main difficulty is that determining if a receive action can be executed depends non-trivially on the sequence of actions executed before the receive. We overcome this difficulty by annotating the traces in a way that makes it possible to compute the set of reachable states.

4.1.1 Trace Annotation for FIFO

Consider a set $\bar{\Theta}$ of *co-names* defined as follows:

$$\bar{\Theta} = \{\bar{\theta} \mid \theta \in \Theta \text{ and } \delta(\theta) \neq \tau\}$$

In other words, for every send or receive action in our FIFO automaton, we introduce a new transition name with a “bar”. We say $s \xrightarrow{\bar{\theta}} s'$ if $s \xrightarrow{\theta} s'$; executions over sequences in $(\Theta \cup \bar{\Theta})^*$ are defined naturally. The intuition of putting the annotation of a “bar” on some transitions of a trace is to indicate that the message sent or received as a result of this transition does not play a role in the channel contents of the final state. In other words, a “barred” transition $\bar{\theta}$ in an annotated trace of the system denotes either a message sent

¹This can sometimes be computed for simple loops using meta-transitions.

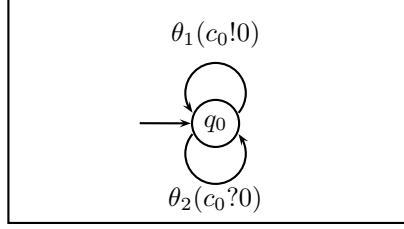


Figure 4.2: Example FIFO automaton.

that will later be received, or the receipt of a message that was sent earlier in the trace. Thus, annotated traces of the automaton will be obtained by marking send-receive pairs in a trace exhibited by the FIFO automaton. Let \mathcal{A} be the function that correctly annotates an execution to produce a string over $\Sigma = \Theta \cup \overline{\Theta}$. Observe, that each execution is annotated uniquely, or to put it formally, \mathcal{A} is an injective function. The *annotated trace language* of the automaton F is:

$$AL(F) = \{\mathcal{A}(t) \mid t \in L(F)\}$$

This language consists of all strings in $(\Theta \cup \overline{\Theta})^*$ that denote correctly annotated traces of F . For example, consider the FIFO automaton shown in Figure 4.2. Some of the words in $AL(F)$ are: θ_1 , $\theta_1\theta_1$, $\theta_1\theta_1\theta_1$, $\overline{\theta_1\theta_2}$, $\overline{\theta_1\theta_2}\theta_1$.

4.1.2 Finding Reachable States from Annotated Traces

Since our objective is to identify the reachable region, we need a way to find the reachable states corresponding to a set of annotated traces. For a channel c , consider a function $h_c : (\Theta \cup \overline{\Theta}) \rightarrow \Sigma_M^*$ defined as follows:

$$h_c(t) = \begin{cases} m & \text{if } t \in \Theta \text{ and } \delta(t) = c!m \\ \epsilon & \text{otherwise} \end{cases}$$

Let h_c also denote the unique homomorphism from $(\Theta \cup \overline{\Theta})^*$ to Σ_M^* that extends the above function. Given an annotated trace ρ , the contents of channel c in the final state are clearly given by $h_c(\rho)$.

FIFO automata with one channel: Let $F = (Q, q_0, \{c_0\}, \Sigma_M, \Theta, \delta)$ be a single channel FIFO automaton, with c_0 being the only channel. As usual $s_0 = (q_0, \epsilon)$ will denote the starting state of F . Given a set of annotated traces L , let $L_q \subseteq L$ be the set of annotated traces in L whose last transition ends in control state

q . Now the set of states reached (by traces in L) is given by

$$\mathcal{R}(L) = \{(q, m) \mid q \in Q \text{ and } m \in h_{c_0}(L_q)\}$$

For a regular set L , it can be seen that L_q is regular, and $\mathcal{R}(L)$ can be computed by a simple homomorphism, and so $\mathcal{R}(L)$ is regular.

Multi-channel FIFO automata: Consider a FIFO automaton $F = (Q, q_0, C, \Sigma_M, \Theta, \delta)$ communicating over channels $C = \{c_0, c_1, \dots, c_k\}$. Now the set of states reached (by traces in L) is given by

$$\mathcal{R}_m(L) = \{(q, (h_{c_0}(\sigma), h_{c_1}(\sigma), \dots, h_{c_k}(\sigma))) \mid q \in Q \text{ and } \sigma \in L_q\}$$

As we will see shortly, we need a test for inclusion for the reachable states corresponding to a set of annotated traces. In this respect, we cannot hope to work with $\mathcal{R}_m(L)$, since as soon as we have even two channels, given a regular L , $\mathcal{R}_m(L)$ can be seen to be a rational relation for which inclusion is undecidable [20]. However, if we compute the contents of each channel independently of others, we can compute an upper approximation of $\mathcal{R}_m(L)$ as follows:

$$\mathcal{R}(L) = \bigcup_{q \in Q} \{q\} \times h_{c_0}(L_q) \times h_{c_1}(L_q) \cdots h_{c_k}(L_q)$$

It can be easily seen that $\mathcal{R}(L)$ is a regular language if L is regular. In general, $\mathcal{R}_m(L) \subseteq \mathcal{R}(L)$, however for many FIFO systems encountered in practice (most network protocols like Alternating Bit Protocol, Sliding Window Protocol), this gives the exact reachable region when applied to $AL(F)$, *i.e.* $\mathcal{R}_m(AL(F)) = \mathcal{R}(AL(F))$. We show later that this is sufficient for the applicability of our learning approach.

4.1.3 Recovering a Witness from an Unsafe State

If the reachable states corresponding to a learned set of annotated traces have a nonempty intersection with the set of “unsafe” states, we need to extract a sequence of transitions of the system which witnesses the reachability of some unsafe state. The motivation is that such a sequence can then be used as a counterexample demonstrating the violation of the safety property or a negative example for the learning algorithm.

We assume that for each control state $q \in Q$, we are given a recognizable set [20] describing the unsafe channel configurations. Equivalently, for each q , the unsafe channel contents are given by a finite union of

products of regular languages: $\bigcup_{0 \leq i \leq n_q} P_{q,i}$ where $P_{q,i} = \prod_{0 \leq j \leq k} U_q(i, c_j)$ and $U_q(i, c_j)$ is a regular language for contents of channel c_j . For each $P_{q,i}$, an unsafe state s_u is some $(q, u_0, u_1, \dots, u_k)$ such that for $j = 0 \dots k$ we have $u_j \in U_q(i, c_j)$. Let U denote the set of all unsafe states. Given a regular set of annotated traces, L , recall that $L_q \subseteq L$ represents the set of annotated traces in L whose last transition ends in control state q . For each $P_{q,i}$, for each channel j , we can find the intersection of $h_{c_j}(L_q)$ and $U_q(i, c_j)$ and calculate the traces $L_{q_i,j} \subseteq L_q$ such that $h_{c_j}(L_{q_i,j}) = h_{c_j}(L_q) \cap U_q(i, c_j)$. Intuitively, this gives us annotated traces which lead to a potential unsafe configuration for channel c_i . Now, if the intersection $\bigcap_{0 \leq j \leq k} L_{q_i,j}$ is non empty, an annotated trace t in this intersection leads to an unsafe configuration for each channel and hence an unsafe state. By repeating this check for each state, for each $P_{q,i}$, we can find if there is any annotated trace t which leads to an unsafe state. Let \mathcal{W} be the function which given L and U outputs the set of annotated traces that can reach unsafe states.

4.1.4 From Annotated Trace to System Execution

In order to convert $\mathcal{W}(L) \in \Sigma^*$ into a sequence of transitions, we need a way to extract the presumed system execution from a given annotated trace. Essentially, we want a substitution $\mathcal{A}^{-1} : \Sigma \mapsto \Theta^*$ which “reverses” the annotation \mathcal{A} . This can be done simply by removing the “bars” on the annotated trace. Formally, we can define $\mathcal{A}^{-1}(\bar{\theta}) = \mathcal{A}^{-1}(\theta) = \theta$ for all $\theta \in \Theta$. Extending \mathcal{A}^{-1} to strings in the usual way, it can be seen that that $\mathcal{A}^{-1}(\mathcal{A}(t)) = t$.

4.1.5 Verification Algorithm

We are now ready to formally describe the *learning-to-verify* procedure as shown in Figures 4.3 and 4.4. We first collect positive and negative examples of labels in Σ^* as follows. A set T of sequences of transitions that can be exhibited by the system is obtained by invoking a function *GetTraces*. Typically, *GetTraces* might either execute a test suite or it may select transitions to execute randomly. Positive examples, S^+ are simply the correct “annotations” which put bars on the send-receive pairs in the strings in T , *i.e.* $S^+ = \{\mathcal{A}(t) \mid t \in T\}$. For the negative examples, we have three different sets. The first set $S_1^- = \{t\theta_d \mid t \in T \text{ and } \theta_d \text{ is a disabled transition}\}$ consists of sequences of transitions extended by a disabled transition (a transition that cannot be taken at a certain state). The second set $S_{2a}^- = \{\sigma \in \Sigma^* \mid \exists t \in T \text{ such that } \mathcal{A}^{-1}(\sigma) = t \text{ and } \sigma \neq \mathcal{A}(t)\}$ corresponds to “incorrect” annotations. Notice that since \mathcal{A} is injective, all annotations of a trace $t \in T$ other than $\mathcal{A}(t)$ cannot be exhibited by the system. The third set, S_{2b}^- , is a collection of spurious counterexamples; initially this is empty.

The positive and negative examples are given to a learning algorithm based on RPNI (see Section 2.5.1).

```

algorithm learnToVerify
Input:
   $F$  : model of system,
   $U$  : recognizable set of "unsafe states"
Output: Property valid OR
  path to an unsafe state
begin
   $S_{2b}^- = \emptyset$ 
   $(S^+, S_1^-, S_{2a}^-) = \text{GetTraces}()$ 
  while(true)do
     $L = \text{modifiedRPNI}(S^+, S_1^-, S_{2a}^- \cup S_{2b}^-)$ 
    if  $\mathcal{R}(L) \cap U \neq \emptyset$ 
       $l_c \in \mathcal{W}(L, U)$ 
      if  $\mathcal{A}^{-1}(l_c)$  valid execution of  $F$ 
        Output  $\mathcal{A}^{-1}(l_c)$ ; stop
      else
         $S_{2b}^- = S_{2b}^- \cup l_c$ 
      else
        if  $\mathcal{R}(L)$  is a fixpoint
          Output "Property holds"; stop
        else
           $T_{\text{new}} = \text{GetTraces}()$ 
          add  $T_{\text{new}}$  to  $(S^+, S_1^-, S_{2a}^-)$ 
    end

algorithm modifiedRPNI
Input:  $S^+ \in \Sigma^*$ ,  $S_1^- \in \Theta^*$ ,  $S_2^- \in \Sigma^*$ 
Output: a regular language  $L$ 
begin
   $D \leftarrow \text{PTA}(S^+)$ 
  for  $i = 2$  to  $|D|$  do
    for  $j = 1$  to  $i - 1$  do
      if  $q_i, q_j$  not merged with smaller state then
         $D' \leftarrow \text{merge}(D, q_i, q_j)$ 
         $D' \leftarrow \text{determinize}(D', q_j)$ 
         $D'' \leftarrow \mathcal{A}^{-1}(D')$ ; all states in  $D''$  made final
        if  $\text{compatible}(D'', S_1^-)$  and  $\text{compatible}(D', S_2^-)$ 
           $D = D'$ ; exit  $j$ -loop
    return language defined by  $D$ 
end

```

Figure 4.3: Learning to verify algorithm based on RPNI.

```

algorithm determinize
Input:  $D, x$ ; Output:  $D$ 
begin
  for any  $x \xrightarrow{\theta} x_1, x \xrightarrow{\theta} x_2$  and  $x_1 \neq x_2$ 
     $D \leftarrow \text{merge}(D, x_1, x_2)$ 
     $D \leftarrow \text{determinize}(D, \text{smaller of } x_1, x_2)$ 
  return  $D$ 
end

```

Figure 4.4: Procedure *determinize* used in the learning algorithm.

Similar to RPNI, this algorithm first constructs a *prefix tree automata* (PTA) from S^+ . Recall from Section 2.5.1 that the PTA is simply a collection of the strings in S^+ as paths with common prefixes merged together. Each state in the PTA is associated with the string generated by following the path to that state from the initial state. The states are assigned numbers according to the standard ordering² imposed by the associated strings. The learning algorithm attempts to generalize from the positive examples by merging states in the PTA in a *specific order*: for i going from 1 to the largest state in the PTA, it attempts to merge q_i with all states less than q_i in ascending order. A merge may cause non-determinism which is removed by further merges using the operation *determinize* which results in a finite automaton D' . Another finite automaton D'' is obtained from D' by applying the substitution \mathcal{A}^{-1} and making all states final. If D'' is compatible with the negative set S_1^- (all strings in S_1^- are rejected by D'') and D' is compatible with the negative set $S_2^- = S_{2a}^- \cup S_{2b}^-$, the merge is accepted. The learning algorithm is essentially the same as the traditional RPNI algorithm except for the use of the additional kind of negative examples corresponding to S_1^- .

Let the output of the modified RPNI algorithm be the regular language L . If $\mathcal{R}(L)$ intersects with the unsafe states U , then a counterexample $l_c \in \mathcal{W}(AL(F), U)$ is obtained. By attempting to simulate the counterexample on the system, we can check if $\mathcal{A}^{-1}(l_c)$ is executable. If yes, then we have found a real counterexample and are done, otherwise l_c is added to S_{2b}^- . If $\mathcal{R}(L)$ does not intersect with the unsafe states U , then it is tested for being a fixpoint under the reachability relation by checking the following condition:

$$\{s_0\} \cup \{s \mid \exists s' \in \mathcal{R}(L). s' \rightarrow s\} = \mathcal{R}(L)$$

If it is a fixpoint, we declare that the safety property holds. Otherwise, we get more traces by invoking the function *GetTraces* (successive calls to this function generate new traces) and continue the learning

²For $\Sigma = \{a, b\}$, the ordering is $\epsilon, a, b, aa, ab, ba, bb, aaa, \dots$

procedure.

4.2 Correctness of the Verification Procedure

The soundness of the procedure is straightforward. For a learned set of traces L , if $\mathcal{R}(L)$ has an empty intersection with the set of unsafe states, U , and is a fixpoint under the transition relation, the safety property holds. Any counterexample is finite and gives a supposed execution of the system leading to an unsafe state which can then be automatically checked for validity by simulation of the system.

We can also show completeness (*i.e.* the procedure terminates with the correct answer) under the condition that $AL(F)$ is regular. Then, given a “fair” method of generating the system traces, in the limit, the learning paradigm will either prove that the system satisfies the property or find a valid counterexample. By a fair method, we mean one which will eventually generate any given finite trace. There can be many different ways of generating fair traces, one of the simplest being a breadth first traversal of all traces.

Before going through the proof, the reader may want to recall the definitions of *short prefixes* and *kernel* given in Chapter 2, Section 2.5.1.

Lemma 1. *If $AL(F)$ is regular, then using any fair strategy for generating traces, in the limit, given a sufficiently large sample (S^+, S_1^-, S_2^-) , the modified RPNI algorithm outputs a DFA which generates $AL(F)$.*

Proof sketch. Let D be the canonical DFA which generates $AL(F)$. Let Sp and N respectively denote the set of *short prefixes* and *kernel* of $AL(F)$. Since the short prefixes and kernel are finite, given a fair strategy of generating the traces, we are guaranteed to eventually get a set of positive examples, S^+ , whose prefixes include all short prefixes and kernel strings. It is known from the proof of RPNI [48], that if we have such positive examples, then the ordered merge used by RPNI outputs a DFA isomorphic to D provided that $\forall x \in Sp, \forall y \in N$ if x and y are *non-equivalent*, *i.e.*, if $AL(F)/x \neq AL(F)/y$, then there are enough negative examples in S_1^- and S_2^- to prevent a merge of x and y . We show that in the limit, we are guaranteed to get enough negative examples. Consider some $x \in Sp$ and some $y \in N$ such that $AL(F)/x \neq AL(F)/y$. First consider the case that there is a string u such that $xu \in AL(F)$ but $yu \notin AL(F)$. Recalling that $L(F)$ is the language of all allowed sequences of transitions, there are two possibilities why $yu \notin AL(F)$:

- $\mathcal{A}^{-1}(yu) \in L(F)$ but $\mathcal{A}(\mathcal{A}^{-1}(yu)) \neq yu$. Intuitively, the sequence of transitions is correct but they have a wrong “annotation”. By a fair strategy, we get negative examples S_{2a}^- corresponding to all wrong “annotations” of valid transition sequences, hence in the limit we will have some negative example to detect this situation.

- $\mathcal{A}^{-1}(yu) \notin L(F)$. Let v be the minimal prefix of u such that $\mathcal{A}^{-1}(xv) \in L(F)$ but $\mathcal{A}^{-1}(yv) \notin L(F)$. Since any string shorter than $\mathcal{A}^{-1}(yv)$ is in $L(F)$, $\mathcal{A}^{-1}(yv)$ is a sequence of valid transitions extended by a disabled transition. By using a fair strategy, we will eventually get a negative example in S_1^- equal to $\mathcal{A}^{-1}(yv)$. When we check for compatibility with S_1^- , we set all states final in the DFA obtained after merge. This enables the detection of the case that $\mathcal{A}^{-1}(xv) \in L(F)$ but $\mathcal{A}^{-1}(yv) \notin L(F)$

The other possibility that $yu \in AL(F)$ but $xu \notin AL(F)$ is handled in the same manner. □

Theorem 1. *If $AL(F)$ is regular and $\mathcal{R}(AL(F))$ is the set of all reachable states, then the learning to verify procedure will eventually either prove that the system satisfies the property or find a valid counterexample.*

Proof sketch. Since we use a fair strategy for generating traces, by Lemma 1, if we keep generating more traces, eventually we will learn $AL(F)$. In this case, $\mathcal{R}(AL(F))$ would be a fix point and if the safety property holds, the procedure does not find a counterexample and proves that the system satisfies the property. If the safety property does not hold then the reachable states have a non empty intersection with the unsafe states and we are guaranteed to find a valid counterexample.

The only way that the procedure may not terminate is if it keeps getting spurious counterexamples in an infinite sequence. At any stage of the learning process, let $(S^+, S_1^-, S_{2a}^-, S_{2b}^-)$ be the set of positive and negative examples and let L be the language learned by RPNI based on these examples. As mentioned before, RPNI constructs a prefix tree automata (PTA) which simply consist of paths leading to strings in S^+ with common prefixes merged together. L is obtained by merging states in the PTA while making sure none of the negative examples are accepted. For a given S^+ , if the counterexample l_c for safety property is found to be spurious, then in the next stage RPNI is provided with $(S^+, S_1^-, S_{2a}^-, S_{2b}^- \cup \{l_c\})$. The new language L'_i is necessarily different than L_i since they do not agree on the acceptance of l_c . Since the PTA is finite, merging different states can generate only finitely many languages. In the extreme case, no state in the PTA is merged; but then there cannot be any spurious counterexample since the PTA is based on S^+ which are obtained from valid executions. Hence, the procedure cannot get “stuck” getting an infinite sequence of negative examples without any change in the traces obtained. □

The running time of the algorithm is dependent on the strategy for getting the traces. For a simple breadth-first strategy, in the worst case, the algorithm might need to explore all traces up to a depth D . Here, D is the length of the longest path starting from the initial state in the minimal DFA representing $AL(F)$ (assuming $AL(F)$ is regular). Thus, the running time can be exponential in the size of the DFA for $AL(F)$. However, as discussed in the following section (Section 4.3), we can use some heuristics to prune

down the number of traces needed. In practice, for a number of FIFO systems, the learning procedure is able to converge to the correct answer in a fairly small time period which is comparable to other tools.

Note that the conditions required by Theorem 1 are merely sufficient for termination of the learning procedure and the verification procedure can be successfully used for many systems even if $AL(F)$ is not regular. In fact, an important observation is that for a number of systems with nonregular $AL(F)$, there exists a regular subset $L' \subseteq AL(F)$ such that the traces in L' “cover” all the reachable states, *i.e.* $\mathcal{R}(L') = \mathcal{R}(AL(F))$. In other words, every reachable state in F is witnessed by some trace in L' . For example, the set of annotated traces corresponding to the automaton in Figure 4.2 is not regular but the regular language $L' = \theta_1^*$ covers all the reachable states. Note that $\mathcal{R}(L')$ is not an approximation; we are simply content with finding any regular set of annotated traces that can cover the reachable states. In Chapter 9, we analyze FIFO systems which have a regular $AL(F)$ as well as systems for which $AL(F)$ is not regular but a “covering” $L' \subseteq AL(F)$ is regular. In all cases, the algorithm terminates with the correct reachable set.

4.3 Strategies for Trace Generation

For generating the annotated traces that are used for the positive and negative examples, we use the following strategy. Starting from the initial state, we explore the system states (cross product of the control state and channel contents) in a breadth-first manner. To limit the number of traces generated, we do not distinguish between FIFO states if they have the same control state and same channel contents up to a position d from the start of the channel. We start with $d = 1$ and keep increasing d if more traces are needed. We have seen that this heuristic works quite well in practice to generate sufficient traces for the learning procedure.

4.4 Example Application of the Verification Procedure

In this section, we analyze in detail the application of the learning procedure to a simple producer-consumer FIFO automaton which is shown in Figure 4.5. We have deliberately chosen a small example for ease of illustration; in general, FIFO automata can be much more complex and non-trivial to analyze. The transitions are labeled with a unique identifier and the associated action is shown in parentheses. For illustration of a safety property, we add another state before the producer enters the idle-send loop. This state has a transition to an “unsafe” state through a receive action. Thus, control state 2 with any channel contents is deemed to be unsafe while all other states are safe.

Annotated traces of the system are generated and provided to the RPNI algorithm as the set of positive samples. RPNI constructs the prefix tree automaton (PTA) as shown in Figure 4.6. A bar on a transition

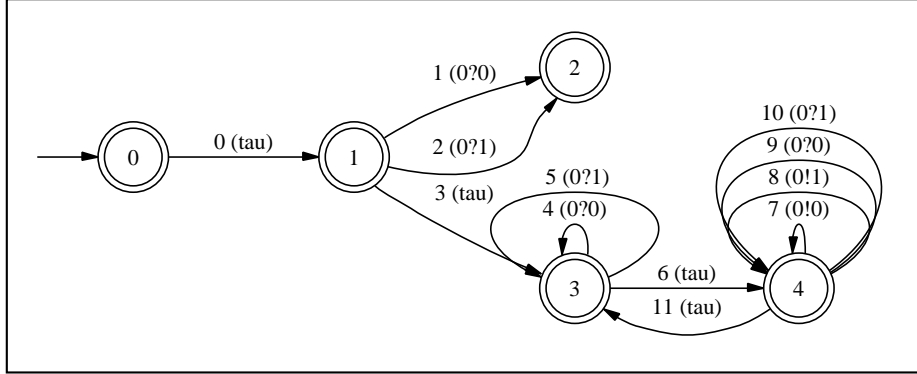


Figure 4.5: Simple producer consumer FIFO automaton

is indicated by a trailing apostrophe. A state of the PTA with the set of incoming edges I and the set of outgoing edges O is labeled with the control state corresponding to the target of the transitions in I and the source of transitions in O in the control graph.³

RPNI proceeds to merge states in the PTA in an ordered fashion, each time checking to see if the automaton formed after the merge fails any of the negative tests. After a few merges, RPNI is able to find a fix point for the reachable set. Since no unsafe state is included in the fix point, the system is declared to satisfy the safety property and the algorithm terminates with the automaton for the set of annotated traces as shown in Figure 4.7.

4.5 Comparison with Related Work

We now discuss the comparison of the learning-to-verify approach with other techniques used for verification of infinite state systems.

Recall from Chapter 3 that in the approach using *meta-transitions* and *acceleration* [21, 24, 55], a sequence of transitions, referred to as a *meta-transition*, is selected and the effect of its infinite iteration calculated. This is complementary to our learning approach, since meta-transitions can be also be incorporated into our learning algorithm. Another popular approach for FIFO, parametric, integer and stack systems is *regular model checking* [25, 2]. A regular set is used to represent the states and a transducer is used to represent the transition relation. The problem is reduced to finding a finite transducer representing the infinite composition of this relation. However, there are some examples in which even if such a finite transducer exists, the procedure may not be able to converge to it. One such example of a FIFO automaton is shown in Figure 4.8. We used the regular model checking tool from [106] to analyze this example, but the tool failed to

³This control state is unique by construction of the automaton

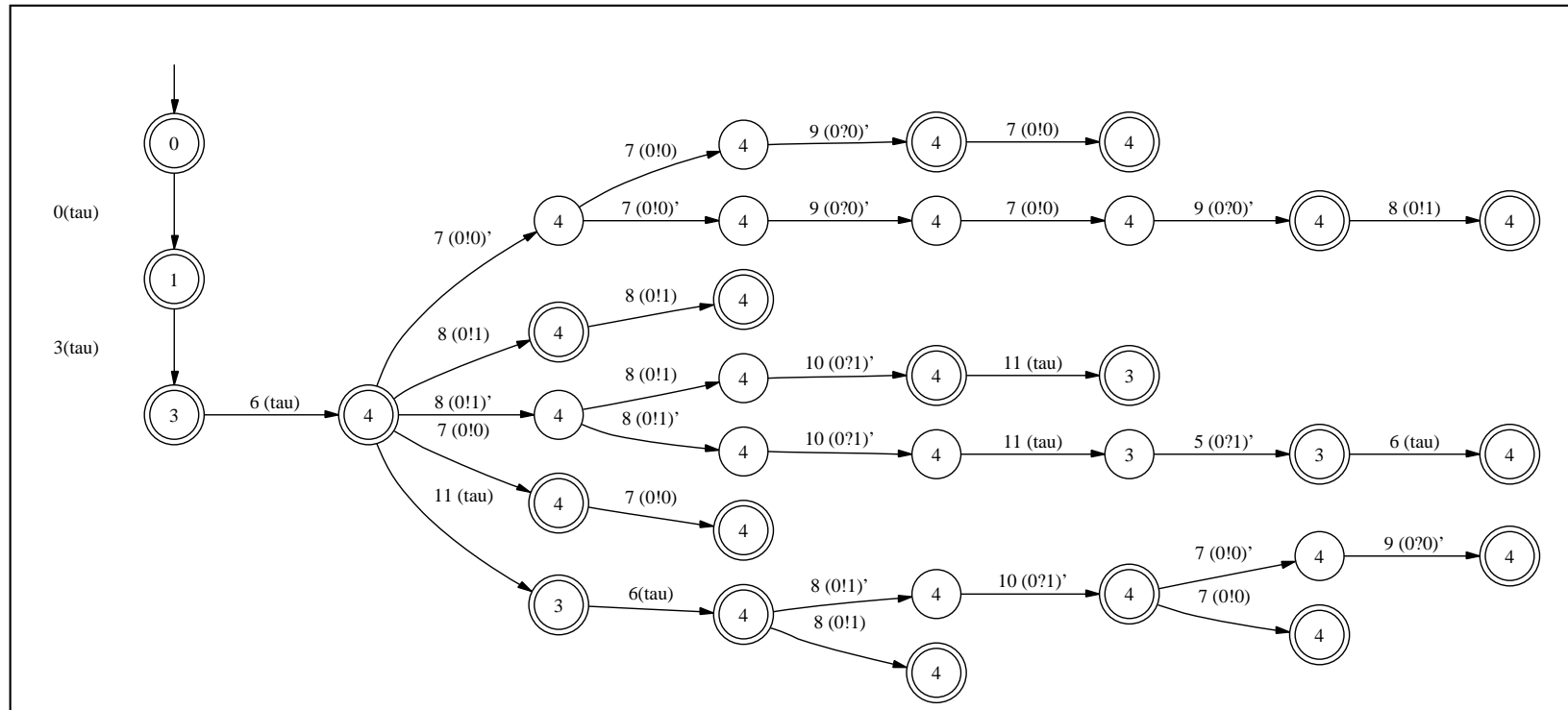


Figure 4.6: Prefix tree automaton for producer consumer. The learning algorithm will find candidate states for merging in an ordered fashion starting from the start state. Merges will be allowed only if they do not cause any negative example to be accepted.

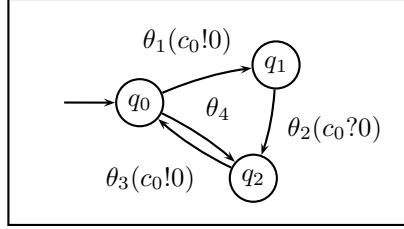


Figure 4.8: A FIFO automaton for which RMC tool fails. Here, θ_4 does not change channel contents.

terminate even after two hours. A careful analysis revealed that for this example, at each step of the regular model checking method, the intermediate transducer keeps accumulating new states in a pattern that causes its size to grow without bound. On the other hand, our learning-based tool is able to automatically find the reachable set in about fifty milliseconds.

An approach for computing the reachable region that is closely related to ours is *widening*. In this approach, the transition relation is applied to the initial configuration some number of times and then by comparing the sets thus obtained, the limit of the iteration is guessed. A simple widening principle in the context of regular model checking is given in [25] which is extended in [125] for parametric systems. Bultan [30] uses a widening technique for Presburger formulas to enable faster convergence for fixpoint. Bartzis *et al.* [19] present a widening technique for arithmetic automata. At a very high level, both *widening* and our approach use similar ideas. In both methods, based on certain sample points obtained using the transitions, a guess is made for the fixpoint being searched for. One important difference between widening and our approach is that widening (except for certain special contexts where it can be shown to be exact) is a mechanism to prove the correctness of a system and cannot be used to prove a system to be incorrect. On the other hand, the approach presented here allows one to both prove a system to be correct and to detect bugs.

Another technique that has been used for verifying infinite state systems is that of *bisimulation minimization*. However, a crucial requirement for the success of bisimulation technique is that the reachable set of equivalence classes be finite. It is easy to construct FIFO automata which fail to satisfy this property but are quite simple and can be analyzed using our learning framework automatically. The FIFO automaton shown earlier in Figure 4.2 is one such example.

Chapter 5

FIFO Automata Safety Using Active Learning

In this chapter, we extend the learning-to-verify method for safety properties of FIFO automata with two main new ideas. Firstly, we give a new scheme for the *annotations* on traces. With this annotation scheme, many more practical FIFO systems have regular annotated trace languages, thus enlarging the class of systems that can be provably verified by our method. Secondly and more significantly, we use the framework of active learning (as opposed to passive learning that we used in the previous chapter). For this, we provide a method to devise a *knowledgeable teacher* which can answer membership (whether a string belongs to the target) as well as equivalence-queries (given a hypothesis, whether it matches the concept being learnt). In the context of learning annotated traces, equivalence queries can be answered only to a limited extent. However, we overcome our limitation to answer equivalence queries exactly and present an approach that is still able to use the powerful query-based learning framework. We assume that the annotated traces of the system to form a regular language and use a variant of Angluin’s L^* algorithm [12] which is a well-known algorithm for learning regular sets. Using this algorithm gives us significant benefits. First, the number of samples we need to consider is polynomial in the size of the automaton representing the annotated traces. Second, we are guaranteed to learn the *minimal* automaton that represents the annotated traces. Finally, we can show that the running time is bounded by a polynomial in the size of the minimal automaton representing the annotated traces and the time taken to verify if an annotated trace is valid for the FIFO system.

5.1 Using Active Learning Framework for Verification

To use an active learning algorithm for the reachable states, we need to answer both membership and equivalence queries for the reachable set. However, there is no way of verifying if a certain state is really reachable or not. As before, this problem is solved by keeping a candidate witness (in terms of the transitions of the system) to a reachable state. Using this, given a purported reachable state and its witness, we can answer a query about its membership in the actual reachable region. Therefore, instead of learning the set

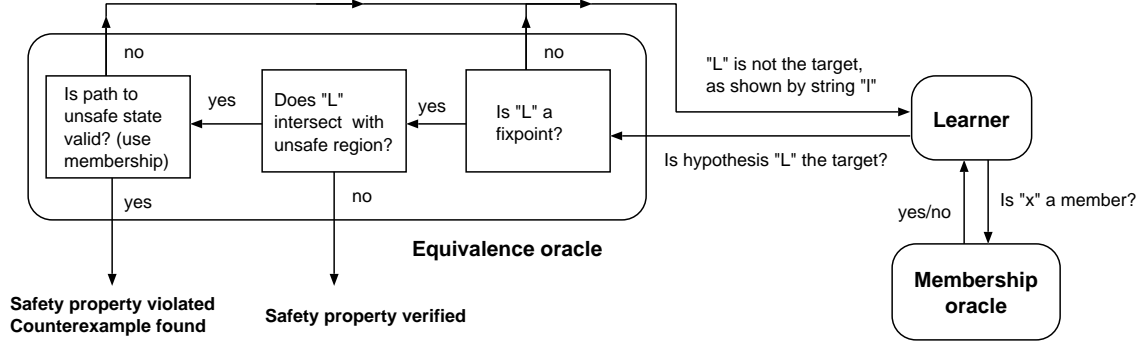


Figure 5.1: Verification procedure for safety properties using active learning

of reachable states directly, we learn a language which allows us to identify both the reachable states and their witnesses.

For equivalence queries, we can provide an answer in one direction. We will show that the reachable region with its witness executions can be seen as the least fixpoint of a relation derived from the transitions. Hence, an answer to the equivalence query can come from checking if the proposed language is a fixpoint under this relation. If it is not a fixpoint then it is certainly not equivalent to the target; but if it is a fixpoint, we are unable to tell if it is also the least fixed point. However, we are ultimately interested in only checking whether a given safety property holds. If the proposed language is a fixpoint but does not intersect with the unsafe region, the safety property clearly holds. On the other hand, if the fixpoint does intersect with unsafe states, we can check if such an unsafe state is indeed reachable using the membership query. If the unsafe state is reachable then we have found a valid counterexample to the safety property and are done. Otherwise the proposed language is not the right one since it contains an invalid trace.

Figure 5.1 shows the high level view of the verification procedure. The main problems we have to address now are:

- What is a suitable representation for the reachable states and their witnesses?
- Given a language representation, we need to answer the following questions raised in Figure 5.1:
 - (Membership Query) Given a string x , is x a valid string for a reachable state and its witness?
 - (Equivalence Query(I)) Is a hypothetical language L a fixpoint under the transition relation? If not, we need a string which demonstrates that L is not a fixpoint.
 - (Equivalence Query(II)) Does any string in L witness the reachability of some “unsafe” state?

5.2 Representation of the Reachable States and their Witnesses

In the previous chapter, we presented the states and their witnesses by annotating the traces in a way that makes it possible to compute the set of reachable states. This annotation scheme allowed us to calculate the reachable set for any regular set of annotated traces by a simple homomorphism. However, one difficulty we encountered is that for some practical FIFO systems, the annotated trace language is not regular; the nonregularity often came from the fact that a receive transition has to be matched to a send which could have happened at an arbitrary time earlier in the past. To alleviate this problem, we use a new annotation scheme in which only the send part of the send-receive pair is kept. This gives an annotated trace language which is regular for a much larger class of FIFO systems (although we cannot hope to be able to cover all classes of FIFO systems since they are Turing complete). We now describe this annotation in detail.

Recall from Chapter 2 that a FIFO automaton [55] is a 6-tuple $(Q, q_0, C, \Sigma_M, \Theta, \delta)$ where Q is a finite set of *control states*, $q_0 \in Q$ is the initial control state, C is a finite set of *channel names*, Σ_M is a finite alphabet for contents of a channel, Θ is a finite set of transitions names, and $\delta : \Theta \rightarrow Q \times ((C \times \{?, !\} \times \Sigma_M) \cup \{\tau\}) \times Q$ is a function that assigns a *control transition* to each transition name.

As before, we have a new set of barred names but this time only for the send transitions:

$$\bar{\Theta} = \{\bar{\theta} \mid \theta \in \Theta \text{ and } \delta(\theta) = c_i!a_j \text{ for some } c_i, a_j\}$$

We also define another set of names $T_Q = \{t_q \mid q \in Q\}$ consisting of a symbol for each control state in the FIFO.

Let the alphabet of *annotated traces* Σ be defined as $(\Theta - \Theta_r) \cup \bar{\Theta} \cup T_Q$ where Θ_r is the set of receive transitions, $\Theta_r = \{\theta_r \mid \delta(\theta_r) = c_i?a_j \text{ for some } c_i, a_j\}$.

Given a sequence of transitions l in $L(F)$, let \mathcal{A}_a be a function which produces an annotated string in Σ^* . \mathcal{A}_a takes each receive transition θ_{r_i} in l and finds the matching send transition θ_{s_i} which must occur earlier in l . Then, θ_{r_i} is removed and θ_{s_i} replaced by $\bar{\theta}_{s_i}$. Once all the receive transitions have been accounted for, \mathcal{A}_a appends the symbol $t_q \in T_Q$ corresponding to the control state q which is the destination of the last transition in l . Intuitively, for a send-receive pair which cancel each other's effect on the channel contents, \mathcal{A}_a deletes the received transition and replaces the send transition with a barred symbol. As before, a barred symbol indicates that the message sent does not play a role in the channel contents of the final state. Notice that in the old annotation scheme both the send and the receive were replaced with a barred version; here the receive transition is dropped altogether. The reason we still keep the send transition with a bar is, as we will show shortly, that this allows us to decide whether any given string is a valid annotated trace. The

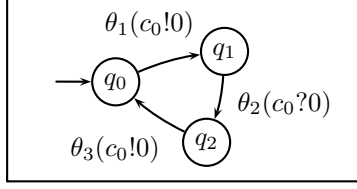


Figure 5.2: Example FIFO automata

symbol t_q is appended to the annotated trace to record the fact that the trace l leads to the control state q .

As an example, consider the FIFO automaton shown in Figure 5.2. For the following traces in $L(F)$: $\theta_1\theta_2\theta_3$, $\theta_1\theta_2\theta_3\theta_1\theta_2$, the strings output by \mathcal{A}_a are respectively: $\overline{\theta_1}\theta_3t_{q_0}$, $\overline{\theta_1}\theta_3\theta_1t_{q_2}$.

Let the language of annotated traces be $AL_a(F) = \{\mathcal{A}_a(t) \mid t \in L(F)\}$ which consists of all strings in Σ^* that denote correctly annotated traces of F . Recall that $AL(F)$ is the annotated trace language corresponding to the old annotation scheme described earlier (in which we keep both parts of a send-receive pair). The following proposition shows that the new annotation scheme has regular annotated trace language for more FIFO automata than the old scheme.

Proposition 2. *The set of FIFO automata for which $AL_a(F)$ is regular is strictly larger than the set of FIFO automata for which $AL(F)$ is regular.*

Proof sketch. If $AL(F)$ is regular, let D be the DFA for it. Now create a new DFA D' by making a copy of D and adding one more state s_{new} . Further, for any final state s_{final} in D' add a transition t_q from s_{final} to s_{new} . Here, q is the target control state for all transitions incoming on s_{final} (with *bars* ignored) and t_q is the symbol in T_Q for q . Make s_{new} the only final state in D' . It is easy to see that D and D' are essentially the *same* except that we have explicitly added symbols for the control state that any trace accepted by D ends with. We can now create a finite automaton for $AL_a(F)$ by replacing all barred receives in D' with ϵ transitions. This shows that $AL_a(F)$ is regular.

It can be shown that $AL_a(F)$ for the automaton in Figure 5.2 is regular while $AL(F)$ is not. Thus, the set of FIFO automata for which $AL_a(F)$ is regular is strictly larger. \square

$AL_a(F)$ can be seen to represent both the reachable states of the FIFO system and the annotated traces which in some sense witness the reachability of these states. Thus, $AL_a(F)$ is a suitable candidate for the language to use in the verification procedure shown in Figure 5.1.

Given a string l in Σ^* , we say that l is well-formed if l ends with a symbol from T_Q and there is no other occurrence of symbols from T_Q . We say that a language L is well-formed if all strings in L are well-formed. For a well-formed string l ending in symbol t_q , let $\mathcal{T}(l)$ denote the prefix of l without t_q and let $\mathcal{C}(l)$ denote

the control state q .

5.3 Answering Membership Queries

In order to answer a membership query for $AL_a(F)$, given a string l in Σ^* we need to verify if l is a correct annotation for some valid sequence of transitions l' in $L(F)$. Let $\mathcal{A}_a^{-1}(l)$ be a function which gives the set (possibly empty) of all sequences of transitions l' for which $\mathcal{A}_a(l') = l$. First, if l is not well-formed, $\mathcal{A}_a^{-1}(l) = \emptyset$ since all valid annotations are clearly well-formed. Assuming l is well-formed, if we ignore the bars in $\mathcal{T}(l)$, we get a string l'' which could potentially be in $\mathcal{A}_a^{-1}(l)$ except that the transitions corresponding to any receives are missing. We can identify the possible missing receive transitions by looking at the barred symbols in $\mathcal{T}(l)$; each barred send can potentially be matched by a receive transition that operates on the same channel and has the same letter. However, we do not know the exact positions where these receive transitions are to be inserted in l'' . We can try all possible (finitely many) positions and simulate each resulting transition sequence on the fly on the FIFO system. Any transition sequence which is valid on the FIFO and gives back l on application of \mathcal{A}_a is then a member of $\mathcal{A}_a^{-1}(l)$. If $\mathcal{A}_a^{-1}(l) \neq \emptyset$ then l is a valid annotated trace.

For illustration, let us consider a membership query for the string $\overline{\theta_1\theta_3}\theta_1t_{q_2}$ for the FIFO automata shown in Figure 5.2. We identify the possible missing receive transitions as two instances of θ_2 . Since a receive can only occur after a send for the same channel and letter, the possible completions of the input string with receives are $\{\theta_1\theta_2\theta_3\theta_2\theta_1, \theta_1\theta_2\theta_3\theta_1\theta_2, \theta_1\theta_3\theta_2\theta_2\theta_1, \theta_1\theta_3\theta_2\theta_1\theta_2, \theta_1\theta_3\theta_1\theta_2\theta_2\}$. Of these, $\theta_1\theta_2\theta_3\theta_1\theta_2$ can be correctly simulated on the FIFO system and gives back the input string $\overline{\theta_1\theta_3}\theta_1t_{q_2}$ on application of \mathcal{A}_a . Therefore, the answer to the membership query is *yes*. An example for a negative answer is $\overline{\theta_1}t_{q_0}$.

5.4 Answering Equivalence Queries

For learning $AL_a(F)$ in the active learning framework, we need a method to verify whether a hypothesis language of annotated traces is equivalent to $AL_a(F)$. If not, then we also need to identify a string in the symmetric difference of $AL_a(F)$ and the hypothesis language to allow the learner to make progress.

Given a string $l \in L$ and a transition θ in the FIFO, we can find if it is possible to *extend* l using θ . More precisely, we define a function $Post(l, \theta)$ as follows. If l is well-formed, let $source(\theta)$ and $target(\theta)$ be the

control states which are respectively the source and the target of θ .

$$Post(l, \theta) = \begin{cases} \emptyset & \text{if } l \text{ not well-formed or if } \mathcal{C}(l) \neq source(\theta) \\ \{\mathcal{T}(l)\theta t_{target(\theta)}\} & \text{otherwise if } \delta(\theta) = \tau \text{ or } \delta(\theta) = c_i!a_j \\ \{deriv(\mathcal{T}(l), \theta) t_{target(\theta)}\} & \text{otherwise if } \delta(\theta) = c_i?a_j \end{cases}$$

$deriv(\mathcal{T}(l), \theta)$ checks the first occurrence of a send θ_s in $\mathcal{T}(l)$ for channel c_i and if the send is for the character a_j , replaces θ_s with $\overline{\theta_s}$. $deriv(\mathcal{T}(l), \theta)$ is empty if no such θ_s could be found or if θ_s outputs a character other than a_j . Intuitively, $deriv$ is similar to the concept of the derivative in formal language theory, except that we look at only the channel that θ operates upon.

Let $Post(l)$ be $\bigcup_{\theta \in \Theta} Post(l, \theta)$ and $Post(L)$ be $\bigcup_{l \in L} Post(l)$.

Lemma 2. *Each string in $AL_a(F)$ is either t_{q_0} or in $Post(l)$ for some $l \in AL_a(F)$*

Proof sketch. A string l' is in $AL_a(F)$ because it is the annotation of at least one sequence of transitions ρ in $L(F)$, i.e. $l' = \mathcal{A}_a(\rho)$. If ρ is the empty string then $l' = \mathcal{A}_a(\epsilon) = t_{q_0}$. Otherwise, let ρ_{-1} be the prefix of ρ without the last transition. Consider $l = \mathcal{A}_a(\rho_{-1})$. From the definition of $Post$, it is easy to see that $l' = Post(l)$. \square

Theorem 2. *Let $\mathcal{F}(L) = Post(L) \cup \{t_{q_0}\}$ where q_0 is the initial control state. $\mathcal{F}(L)$ is a monotone set operator, i.e. it preserves set-inclusion. Moreover, $AL_a(F)$ is the least fixpoint of the function $\mathcal{F}(L)$.*

Proof sketch. Since $Post(L)$ is simply the union of $Post$ of all strings in L , monotonicity of \mathcal{F} is immediate.

From the definition of $Post$, we can see that if $l \in AL_a(F)$, it is also true that $Post(l) \in AL_a(F)$. This implies $\mathcal{F}(AL_a(F)) \subseteq AL_a(F)$ since a string in $\mathcal{F}(AL_a(F))$ is either t_{q_0} or $Post(l)$ for some $l \in AL_a(F)$. By Lemma 2, $\mathcal{F}(AL_a(F)) \supseteq AL_a(F)$ since any string in $AL_a(F)$ has to be $Post(l)$ for some other $l \in AL_a(F)$. Thus, $AL_a(F)$ is a fixpoint for \mathcal{F} .

To see that $AL_a(F)$ is also the least fixpoint, by way of contradiction assume a (strictly) smaller fixpoint L' . Applying $Post$ to some string either increases its length by one or increases the number of barred symbols in it. Therefore, given a finite string l , it is not possible to have an infinite chain l_0, l_1, l_2, \dots with $l = l_0$ such that $Post(l_{i+1}) = l_i$. Let l be some string in $AL_a(F)$ which is not in L' . By Lemma 2, there must be some l_1 such that $Post(l_1) = l$. Now l_1 can be t_{q_0} or be $Post(l_2)$ for some l_2 . Since this chain of l_1, l_2, \dots cannot be infinite, it has to end in t_{q_0} . Clearly, t_{q_0} is in any fixpoint, hence $t_{q_0} \in L'$. Consider the smallest i for which l_i in the chain is not in L' . But since $l_{i-1} = Post(l_i)$, l_{i-1} has to be in L' giving a contradiction. Hence, $AL_a(F)$ is the least fixpoint of \mathcal{F} . \square

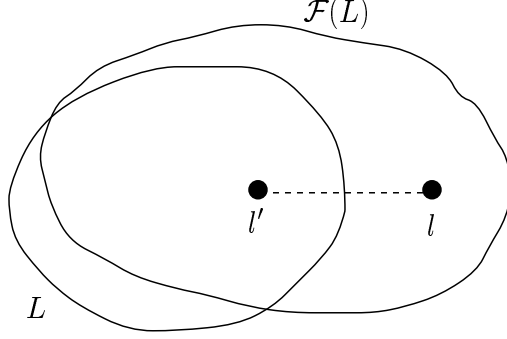


Figure 5.3: Answering equivalence query for the case $\mathcal{F}(L) - L \neq \emptyset$

Theorem 2 gives us a method for answering equivalence queries for $AL_a(F)$. If L is not a fixpoint, it cannot be equivalent to $AL_a(F)$. In this case, we can also find a string in $L \oplus AL_a(F)$ as required for the learning algorithm. Here, $A \oplus B$ denotes the symmetric difference of two sets. Consider the following cases:

1. $\mathcal{F}(L) - L \neq \emptyset$. As illustrated in Figure 5.3, let l be some string in this set. If l is t_{q_0} then it is in $AL_a(F) \oplus L$. Otherwise, we can check if l is a valid annotation using a membership query. If yes, then l is in $AL_a(F) \oplus L$. Otherwise, it must be true that $l \in Post(l')$ for some $l' \in L$. If l is not valid, l' cannot be valid since $Post$ of a valid annotation is always valid. Hence $l' \notin AL_a(F)$ or $l' \in AL_a(F) \oplus L$.
2. $\mathcal{F}(L) \subsetneq L$. This case is graphically depicted in Figure 5.4. From standard fixpoint theory, since $AL_a(F)$ is the least fixed point under \mathcal{F} , it must be the intersection of all prefixpoints of \mathcal{F} (a set Z is a prefixpoint if it *shrinks* under the functional \mathcal{F} , i.e. $\mathcal{F}(Z) \subseteq Z$). Now, L is clearly a prefixpoint. Applying \mathcal{F} to both sides of the equation $\mathcal{F}(L) \subsetneq L$ and using monotonicity of \mathcal{F} , we get $\mathcal{F}(\mathcal{F}(L)) \subsetneq \mathcal{F}(L)$. Thus, $\mathcal{F}(L)$ is also a prefixpoint. Let l be some string in the set $L - \mathcal{F}(L)$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint $AL_a(F)$. Hence, l is in $AL_a(F) \oplus L$.
3. $\mathcal{F}(L) = L$. Let $\mathcal{W}(L)$ be the set of annotated traces in L which can reach unsafe states (We will describe how $\mathcal{W}(L)$ is computed in the next section). If $\mathcal{W}(L)$ is empty, since L is a fixpoint, we can abort the learning procedure and declare that the safety property holds. For the other case, if $\mathcal{W}(L)$ is not empty then let l be some annotated trace in this set. We check if l is a valid annotation using the procedure described in Section 5.3. If it is valid, we have found a valid counterexample and can again abort the whole learning procedure since we have found an answer (in the negative) to the safety property verification. Otherwise, l is in $AL_a(F) \oplus L$.

A subtle point to note is that although we attempt to learn $AL_a(F)$, because of the limitation in the equivalence query, the final language obtained after the termination of the verification procedure may not be

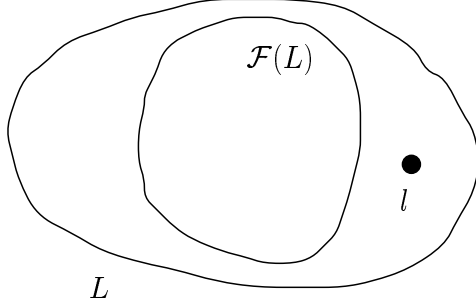


Figure 5.4: Answering equivalence query for the case $\mathcal{F}(L) \subsetneq L$

$AL_a(F)$. It might be some fixpoint which contains $AL_a(F)$ or it might be simply some set which contains a valid annotated trace demonstrating the reachability of some unsafe state. However, this is not a cause for concern to us since in all cases the answer for the safety property verification is correct.

5.5 Finding Annotated Traces leading to Unsafe States

In the previous section, we referred to a set $\mathcal{W}(L)$ in L which can reach unsafe states. We now show how this can be computed.

As before, we assume that for each control state $q \in Q$, we are given a recognizable set [20] describing the unsafe channel configurations. Equivalently, for each q , the unsafe channel contents are given by a finite union of products of regular languages: $\bigcup_{0 \leq i \leq n_q} P_{q,i}$ where $P_{q,i} = \prod_{0 \leq j \leq k} U_q(i, c_j)$ and $U_q(i, c_j)$ is a regular language for contents of channel c_j . For each $P_{q,i}$, an unsafe state s_u is some $(q, u_0, u_1, \dots, u_k)$ such that $u_j \in U_q(i, c_j)$.

For a channel c , consider a function $h_c : \Sigma \rightarrow \Sigma_M^*$ defined as follows:

$$h_c(t) = \begin{cases} m & \text{if } t \in \Theta \text{ and } \delta(t) = c!m \\ \epsilon & \text{otherwise} \end{cases}$$

Let h_c also denote the unique homomorphism from Σ^* to Σ_M^* that extends the above function.

Let L_q be the subset of an annotated trace set L consisting of all well-formed strings ending in t_q , *i.e.* $L_q = \{l \mid l \in L \text{ and } \mathcal{C}(l) = q\}$.

If an unsafe state $s_u = (q, u_0, u_1, \dots, u_k)$ is reachable, then there must exist a sequence of transitions $l_\theta \in \Theta^*$ such that $s_0 \xrightarrow{l_\theta} s_u$, where s_0 is the initial state. In l_θ , if the receives and the sends which match the receives are taken out, only the remaining transitions which are sends can contribute to the channel contents in s_u . Looking at the definition of h_c , it can be seen that for each channel content u_j in s_u , $u_j = h_{c_j}(\mathcal{A}_a(l_\theta))$

(recall that \mathcal{A}_a converts a sequence of transitions into an annotated trace). Thus, for s_u to be reachable, there must be some annotated trace $l \in AL_a(F)$ such that $s_u = (\mathcal{C}(l), h_{c_0}(l), h_{c_1}(l), \dots, h_{c_k}(l))$.

Let $h_{c_j}^{-1}(U_q(i, c_j))$ denote the inverse homomorphism of $U_q(i, c_j)$ under h_{c_j} . Note that for each $P_{q,i}$, $\bigcap_{0 \leq j \leq k} h_{c_j}^{-1}(U_q(i, c_j))$ gives a set of annotated strings which can reach the unsafe channel configurations for control state q . Intersecting this with L_q verifies if any string in L can reach these set of unsafe states. If we perform such checks for all control states for all $P_{q,i}$, we can verify if any unsafe state is reached by L . Thus, the set of annotated traces in L that can lead to an unsafe state is given by:

$$\mathcal{W}(L) = \bigcup_{q \in Q} \left(\bigcup_{0 \leq i \leq n_q} (L_q \cap \bigcap_{0 \leq j \leq k} h_{c_j}^{-1}(U_q(i, c_j))) \right)$$

We summarize the verification algorithm in Figure 5.5.

Theorem 3. *For verifying safety properties of FIFO automata, the learning to verify algorithm satisfies the following properties:*

1. *If an answer is returned by algorithm, it is always correct.*
2. *If $AL_a(F)$ is regular, the procedure is guaranteed to terminate.*

Proof sketch. Soundness of the procedure is straightforward. We declare that the safety property holds only when we have found a fixpoint L which does not intersect with the “unsafe” traces. Since L is a fixpoint, it must be larger or equal to $AL_a(F)$ which is the least fix point. If $\mathcal{W}(L)$ is empty then $\mathcal{W}(AL_a(F))$ must also be empty implying that no execution of the system can reach an unsafe state.

If we say that the safety property does not hold and provide a path leading to an unsafe state, this counterexample has to be valid since we always first check it against the FIFO system.

For showing completeness, assuming that $AL_a(F)$ is regular, we can rely on the termination guarantees of Angluin’s algorithm. The only caveat is our limited ability to answer equivalence queries. Consider a hypothetical teacher which can answer all membership and equivalence queries for $AL_a(F)$ correctly. For an equivalence query, whenever our teacher says *no* the hypothetical teacher must also say *no*; however our teacher is unable to decide when to say *yes*. Notice that a *yes* answer to an equivalence query is only given once and marks the end of the algorithm. Imagine a session of a learner with the hypothetical teacher and a parallel session of another learner with our limited teacher. Further, assume that in case the answer to the equivalence query is *no*, the hypothetical teacher returns the same string (for the symmetric difference) as our teacher. Both learners start off by making identical queries and proceed in lock step since both

```

algorithm learner
begin
Regular inference algorithm
end

algorithm isMember
Input: Annotated trace  $l$ 
Output: is  $l \in AL_a(F)$ ?
begin
  if  $l$  not well-formed return no
  else
    find receives matching barred symbols
    find possible positions for receives
    simulate resulting strings on FIFO
    system on the fly
    if any string reaches  $\mathcal{C}(l)$  with
    correct annotation, return yes
  return no
end

algorithm Equivalence Check
Input: Annotated trace set  $L$ 
Output: is  $L = AL_a(F)$ ?
If not, then some string in  $L \oplus AL_a(F)$ 
begin
   $\mathcal{F}(L) = Post(L) \cup \{t_{q_0}\}$ 
  if  $\exists l \in (\mathcal{F}(L) - L)$ 
    if isMember( $l$ )
      return (no,  $l$ )
    else
      return (no,  $l'$  where  $l = Post(l')$ )
  else if  $\mathcal{F}(L) \subsetneq L$ 
    return (no,  $l \in (L - \mathcal{F}(L))$ )
  else if  $\exists l \in \mathcal{W}(L)$ 
    if isMember( $l$ )
      Print (safety prop. does not hold,  $l$ ); stop
    else
      return (no,  $l$ )
  else
    Print (safety prop. holds); stop
end

```

Figure 5.5: Learning to verify algorithm using active learning

teachers provide the same answers. Let us say that at some point, our teacher declares that it has solved the verification problem and aborts the learning procedure. Consider the two cases possible:

- Our teacher finds some fixpoint which does not intersect with the unsafe states. In this case, the hypothetical teacher might still continue if the fixpoint is not the least fixpoint ($AL_a(F)$) or say *yes* if it is the least fixpoint. The hypothetical teacher could not have said *yes* earlier, since if the learner proposed $AL_a(F)$, our teacher would have found it as a fixpoint which does not intersect with the unsafe states.
- Our teacher finds a valid counterexample to the safety property. Again the hypothetical teacher could not have said *yes* earlier. If $AL_a(F)$ had been proposed by the learner, since the safety property does not hold, some string in $AL_a(F)$ is a valid counterexample to the safety property and our teacher would have found it.

Thus, in all cases, our teacher will end the learning procedure sooner or at the same time as the hypothetical teacher. □

5.5.1 Complexity Analysis

As shown in Proposition 1 in Chapter 2, the learning algorithm makes $O(|\Sigma|n^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. Here m is the size of the longest string returned by the teacher in a negative answer to an equivalence query and n is the size of the minimal automaton representing $AL_a(F)$ (assuming $AL_a(F)$ is regular). When we answer an equivalence query for a hypothesis L , we search for a string in $L \cap (\neg Post(L))$, $Post(L) \cap (\neg L)$ and $\mathcal{W}(L)$. The size of $Post(L)$ is bounded by $O(|\Theta|n)$, hence $L \cap (\neg Post(L))$ and $Post(L) \cap (\neg L)$ are at most of size $O(|\Theta|n^2)$. Assuming that the unsafe states are described by an automaton smaller than the minimal one for $AL_a(F)$, $|\mathcal{W}(L)|$ is bounded by $O(n^2)$. Therefore, the longest string that can be returned as an answer to the equivalence query is $O(|\Theta|n^2)$. Hence, $m = O(|\Theta|n^2)$.

Let $t(l, k)$ be the time taken for a membership query for a string of length l on a FIFO automata with k receive transitions. The running time for the verification procedure is dominated by the cost of equivalence and membership queries. Let us now consider these in turn.

1. Equivalence queries: Each equivalence query can also result in a membership query. Following the reasoning in the previous paragraph, the cost of one equivalence query is bounded by $O(m + t(m, k))$ which can be simplified to $O(t(|\Theta|n^2, k))$. For maximum of n such queries, the total cost is $O(nt(|\Theta|n^2, k))$

2. Membership queries from learner: For $O(|\Sigma|n^2 + n \log m)$ membership queries with the maximum length of a string being $O(m + n)$, the bound on the cost is $O((|\Sigma|n^2 + n \log(|\Theta|n^2))t(|\Theta|n^2, k))$

The cost of answering the membership queries from learner clearly dominates the total cost. Thus, the running time is $O((|\Sigma|n^2 + n \log(|\Theta|n^2))t(|\Theta|n^2, k))$ which is a polynomial in the size of the minimal automata for $AL_a(F)$ and the time needed for a membership query for $AL_a(F)$. The longest string for which membership may need to be checked is quadratic in the size of the minimal automata.

Let us now consider $t(l, k)$, the cost of a membership query for a string of length l . This cost depends on the annotation scheme used. For instance, in the old annotation scheme (which keeps both parts of a send-receive pair) this is simply $O(l)$. For the new annotation scheme, we drop the receive part to allow more FIFO systems to have regular annotated trace languages (making them amenable to automatic analysis). However, this forces us to do more work for the membership query. There can be at most l receive transitions that have to be inserted in the queried string to get a trace that can be simulated on the FIFO system. A trivial upper bound for $t(l, k)$ can be derived as follows. For a query string of length l , we may have to add l more receives. The receives can be put in the $l + 1$ positions in $(l + 1)^l$ or $O(l^l)$ ways. For each place, the number of choices for receives is at most equal to the minimum of k and l . Let $p = \min(k, l)$. Then, the cost of a membership query is $O(p^l l^l)$. The bound could possibly be improved but this is deferred for future work.

5.6 Comparison with Verification based on Passive Learning

The verification procedure based on the active learning approach has several advantages over the one based on passive learning. First, the number of samples we need to consider is polynomial in the size of the *minimal* automaton representing the annotated traces as opposed to exponential for the passive learning approach. Second, we are guaranteed to learn the minimal automaton that represents the annotated traces. Finally, for the active learning case, we can bound the running time by a polynomial in the size of the minimal automaton representing the annotated traces and the time taken to verify if an annotated trace is valid for the FIFO system.

We observe that for most examples we analyzed, the active learning method is indeed able to analyze the system faster. However, as discussed later in Chapter 9, for a few examples, the passive learning approach performs better. It should also be noted that the encodings used for the traces in FIFO for passive and active approaches are different. In particular, the encoding used in the active approach where only the send part of a send-receive pair cannot be used for the passive approach. This is because calculating the disabled

transitions at any step critically depends on having representatives for all transitions taken.

In some scenarios, the passive learning approach may be better. For instance, a test suite designed to cover the reachable space of the system under execution can be used to converge to the solution quickly in the case of the passive learning approach. On the other hand, the active learning approach is not able to leverage on the availability of any test suite.

Chapter 6

Verifying Safety for Systems in Regular Model Checking Framework

In previous chapters, we looked at a learning based approach for verifying safety properties of FIFO automata. In this chapter, we extend this to systems that are expressed in the Regular Model Checking framework. We will focus on using the *active learning* framework..

Recall from Chapter 2, that in the Regular Model Checking framework, states are represented by strings over an alphabet and the transition relation is given as a transducer. This framework has been successfully used ([25]) for modeling *parameterized systems*, *FIFO automata*, *systems with integer variables* and *push down stacks*. For verifying safety properties of systems expressed in this framework, we can follow the same high level idea as described for FIFO automata in Figure 5.1. Again, instead of just learning reachable states themselves, we learn a set of pairs of states and witnesses, where a witness shows how the corresponding state is reachable. As the reader can recall, the key questions that we have to answer to use the learning-to-verify approach for safety are:

- What is a suitable representation for the reachable states and their witnesses?
- Given a language representation:
 - (Membership Query) For an element x , is x a valid pair of reachable state and its witness?
 - (Equivalence Query(I)) Is a hypothetical language a fixpoint under the transition relation? If not, we need a element which demonstrates that the hypothetical language is not a fixpoint.
 - (Equivalence Query(II)) Does any element in the hypothetical language witness the reachability of some “unsafe” state?

In FIFO automata, we used trace annotations to represent reachable state and their witnesses. But in general, what should be the representation to use? The key property that the witnesses needs to satisfy is: given a state witness pair, the teacher should be able to decide algorithmically whether the witness demonstrates that the claimed state is reachable. For systems expressed in the Regular Model Checking framework, there are several possibilities for the witness. In some cases, the the encoding of the states itself

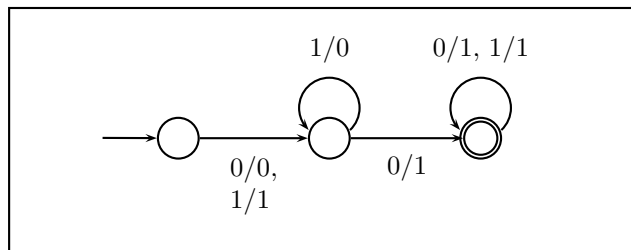


Figure 6.1: Transducer for the transition relation $x = x + 2$

provides a way for the teacher to decide if a certain state is reachable. For example, if the transition relation of the system is length-preserving, then given a state represented by a string of length n , we can decide membership as follows. Starting with the all initial states of length n , we repeatedly apply the transition relation until we get a set of states which is a fixpoint. This iteration has to converge eventually since there are only finitely many strings possible for a given length. Then, membership can be answered by checking if the state of interest is in this fixpoint. For a general transition relation, which may not be length-preserving, this solution does not work. However, a simple yet powerful idea is to keep as witness, the number of transitions that are needed to be taken, starting from the initial state to the state that is claimed to be reachable. The teacher can then decide if the state is indeed reachable by simply simulating those many steps from the initial states.

6.1 Preliminaries

We assume that the states of the system can be encoded as strings over some finite alphabet ρ . Let $S = \rho^*$ represent the set of all states. We use T_r for the transducer representing the transition relation, I for the DFA representing the set of initial states and U for the DFA representing the set of “unsafe” states.

As a running example, we will use a system with one positive integer variable x whose only transition increments x by 2. The value of x is represented in binary (the first letter in the string is the least significant bit). The initial state is $x = 0$ (represented by 0^*) and the unsafe region is $x = 3$ (represented by 110^*). The transducer for the transition relation is given in Figure 6.1. A label a/b on an edge represents input symbol a and output symbol b . An example of input and output strings related by the transducer is the pair 00 and 01 which corresponds to incrementing x from 0 to 2. Note that the transition relation is length-preserving in this case.

6.1.1 Computing the Image of a Regular Set under a Transducer

Given a transducer $T = (Q_T, \delta_T, \Sigma, \Sigma, q_{0T}, F_T)$ and an NFA $D = (Q_D, \delta_D, \Sigma, q_{0D}, F_D)$, let L be the set of strings such that $s \in L$ if and only if D accepts some string s' and T accepts (s', s) . It can be shown that L is regular. An NFA (denoted by $T(D)$) accepting L can be constructed in a manner similar to finding the intersection of the languages of two DFAs. More precisely, $T(D) = (Q_D \times Q_T, \delta, \Sigma, q_{0D} \times q_{0T}, F_D \times F_T)$ where δ is given as follows:

- $((q_1, q_2), a, (q'_1, q'_2)) \in \delta$ if $\exists b. (q_1, b, q'_1) \in \delta_D$ and $(q_2, b, a, q'_2) \in \delta_T$. Here a can also be ϵ .
- $((q_1, q_2), a, (q_1, q'_2)) \in \delta$ if $\exists a. (q_2, \epsilon, a, q'_2) \in \delta_T$

Note that the NFA $T(D)$ can have epsilon transitions but an equivalent NFA without epsilon transitions can be obtained through standard automata operations. Further, for ease of notation, wherever there is no chance of confusion, we let D represent both the NFA and the regular languages accepted by that NFA. Similarly, we let $T(D)$ represent the NFA obtained by the operation described above and the regular language accepted by that NFA. We also let $T^2(D)$ represent $T(T(D))$, $T^3(D)$ represent $T(T(T(D)))$ and so on for any i .

6.2 Representation of States with Witness

6.2.1 Bounded Space

If the transition relation of the system is length-preserving (meaning that a state represented by a string of length k can only go to another string also of length k), then a witness for the reachability of a state is simply its length. We call this a *bounded space witness*. The reason that the string length suffices as a witness is as follows. For any given length k , there can only be finitely many strings reachable from the initial states since the transition relation is length-preserving. Let all initial states of length k be denoted by $I_{|k}$ and let $T_{|k} : 2^{\rho^*} \mapsto 2^{\rho^*}$ denote a function defined by: $T_{|k}(Z) = I_{|k} \cup T_r(Z)$. Then, to answer if a string s of length k is reachable, we first compute $T_{|k}(\emptyset), T_{|k}^2(\emptyset), T_{|k}^3(\emptyset) \dots$ until for some n , we have $T_{|k}^{n-1}(\emptyset) = T_{|k}^n(\emptyset)$. This computation is guaranteed to terminate since there are only finitely many strings of length k . Now, reachability of s can be decided by checking if $T_{|k}^n(\emptyset)$ accepts s . In our running example, if we wanted to answer the membership query for the string 01 ($x = 2$), we would first take all initial states of length 2: in our case this is just 00. Applying $T_{|2}$ repeatedly gives the least fixpoint as $\{00, 01\}$. Since 01 is in this set, we can answer the membership query in the affirmative.

Note that for the bounded space method, since the witness is implicit in the state representation, the target for the learning algorithm is simply the set of reachable states. We denote this set by $Reach^s$. In our running example, this is given by the set corresponding to the regular expression $0(0|1)^*$.

6.2.2 Bounded Steps

The bounded space witness method works only if the transition relation is length preserving. For general systems, a natural candidate for the witness is simply the number of steps needed to reach a particular state. We denote the state-witness pair by (s, i) where s is the state and i is some number of steps in which s is reached. Now, in general, a set Z of pairs (s, i) is a subset of $\rho^* \times \mathbb{N}$, but to encode Z as a regular set we use the alphabet Σ given by $(\rho \cup \{\perp\}) \times \{0, 1\}$. Here \perp is a new “filler” symbol. An element (s, i) is encoded as a string over Σ such that projecting the symbols on the first component gives us s (the \perp symbols are ignored); and projecting on the second component gives i in binary notation. Alternatively, a unary notation for i can also be used in which case the alphabet used by the learning algorithm would be $(\rho \cup \{\perp\}) \times \{0, \perp\}$. Note that for integer systems, if we use binary notation, then the symbol 0 implicitly functions as a filler symbol for the first component also, hence it can be used in place of \perp . In our running examples, the alphabet for the learning algorithm would be $\Sigma = \{0, 1\} \times \{0, 1\}$

Let $Reach^t$ be the set of all valid state witness pairs. This will be the target for the learning algorithm. In our running example, the DFA for $Reach^t$ is shown in Figure 6.2. For instance, a valid state-witness pair accepted by this DFA is the string $(0, 0)(0, 1)(1, 0)$ which corresponds to $x = 4$ and number of steps of 2.

6.3 Answering Membership Queries

For the bounded space witness, answering the membership query is simply checking if a state is reachable which has already been addressed in the previous section.

For the bounded steps method, to answer a query whether an element $x = (s, i)$ is in $Reach^s$, we simply compute the set of states reached from the initial states in i steps and check if s is in this set.

Given the initial states represented by the DFA I and transducer T_r , the set of states that can be reached in one step are given by $T_r(I)$. Applying T_r repeatedly up to i times gives us the set of states $T_r^i(I)$ reached in i steps. The membership query for (s, i) is then answered by checking if this set contains s . For instance, in our running example, to answer a query for $x = 2$ and number of steps of 1, we will compute $T_r^1(I)$ which is the set $\{x = 2\}$ and since $x = 2$ is a member of this set, we will answer the query in the affirmative.

Note that $T_r^i(I)$ can be used to answer queries for all strings with witness i . Therefore, for efficiency, we

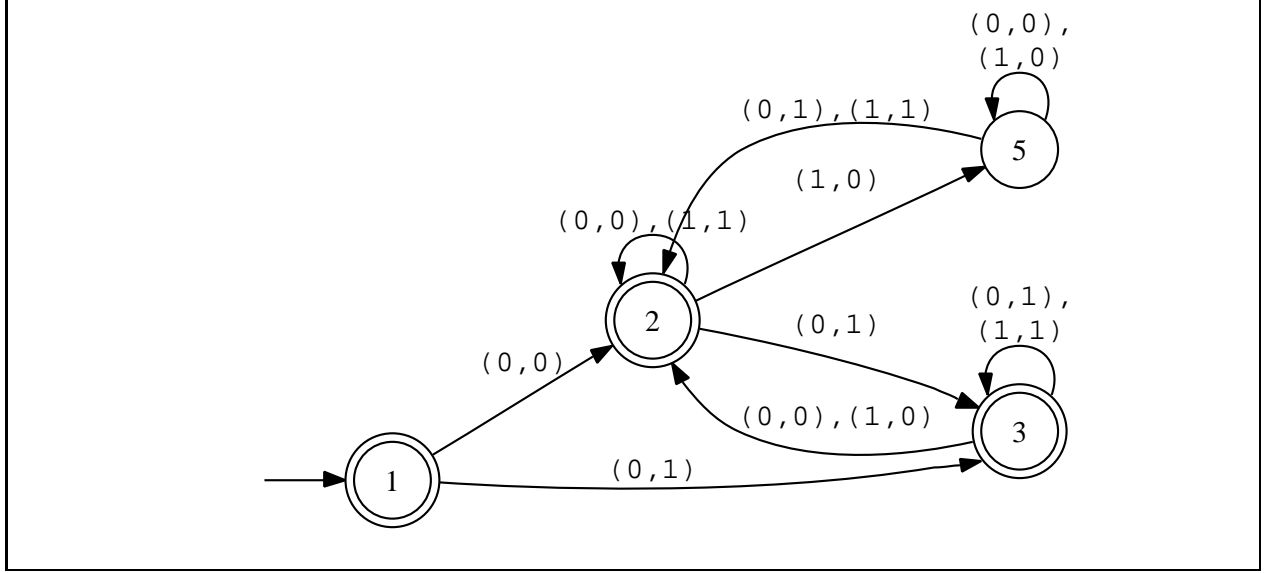


Figure 6.2: Valid state-witness pairs for bounded steps case for the transition relation where x is incremented by 2.

save the calculation of $T_r^i(I)$ and reuse it for answering subsequent queries.

6.4 Answering Equivalence Queries for Bounded Space Method

For the equivalence query, recall that we are given a hypothesis Z and have to answer if this is the target set $Reach^s$ (for the bounded space case, this is just the reachable states). Further, in case the answer is in the negative, we have to provide an example in the symmetric difference of the target and the hypothesis.

We use an idea similar to the one described in Section 5.4. For a set of states Z , let \mathcal{F}^s be a function given by $\mathcal{F}^s(Z) = T_r(Z) \cup I$. Since the reachable states are a (least) fixpoint of \mathcal{F}^s , if the hypothesis is correct, then it should not change under \mathcal{F}^s . As before, we can break the analysis down to three cases:

1. $\mathcal{F}^s(Z) - Z \neq \emptyset$. Let l be some string in this set. If $l \in I$ then it is in $Reach^s \oplus Z$. Otherwise, we can check if l is a reachable state using a membership query. If yes, then l is in $Reach^s \oplus Z$. Otherwise, it must be true that l must be reachable from some $l' \in Z$. If l is not valid, l' cannot be valid. Hence $l' \notin Reach^s$ or $l' \in Reach^s \oplus Z$.
2. $\mathcal{F}^s(Z) \subsetneq Z$. From standard fixpoint theory, since $Reach^s$ is the least fixed point under \mathcal{F}^s , it must be the intersection of all prefixpoints of \mathcal{F}^s . As before, we can show that, in this case, both Z and $\mathcal{F}^s(Z)$ are prefixpoints. Let l be some string in the set $Z - \mathcal{F}^s(Z)$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint $AL_a(F)$. Hence, l is in $Reach^s \oplus Z$.

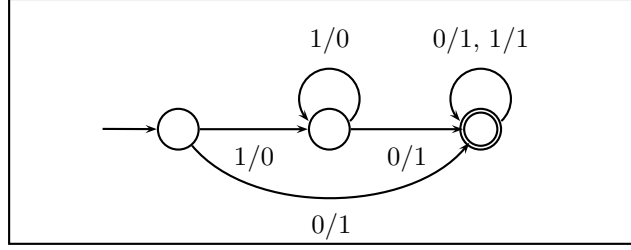


Figure 6.3: Transducer for incrementing value of incoming string by 1. This can be used for keeping the count of steps taken.

3. $\mathcal{F}^s(Z) = Z$. If $Z \cap U$ is empty, since Z is a fixpoint, we can abort the learning procedure and declare that the safety property holds. For the other case, if the intersection is not empty then let l be some state in this set. We check if l is reachable using a membership query. If it is valid, we have found a valid counterexample and can again abort the whole learning procedure since we have found an answer (in the negative) to the safety property verification. Otherwise, l is in $Reach^s \oplus Z$.

6.5 Answering Equivalence Queries for Bounded Steps Method

6.5.1 Incrementing Count of Number of Steps

A generic operation we need is that of incrementing the count for the number of steps for all elements of a set of state-witness pairs.

Definition 6. Given Z a set of strings in the alphabet of Σ , define

$$Inc(Z) = \{(s, i) \mid (s, i - 1) \in Z\}$$

If natural numbers are represented by strings in binary, a transducer which simply increments the value represented by its input string is shown in Figure 6.3.¹ An easy construction also exists for the case of unary representation. This transducer can be used to create a transducer T_{Inc} for the operation Inc . Essentially, T_{Inc} only changes the “component” for the counter i in any input string and copies the “component” for s .

6.5.2 Fixpoint Characterization for the Set of State Witness Pairs

Let T'_r be a transducer constructed from T_r such that it ignores the counter component and only changes the state component. Further, let T be the transducer whose relation is the composition of the relations of

¹For pedagogical reasons, we are visualizing a transducer as taking an input string and producing possibly different output strings.

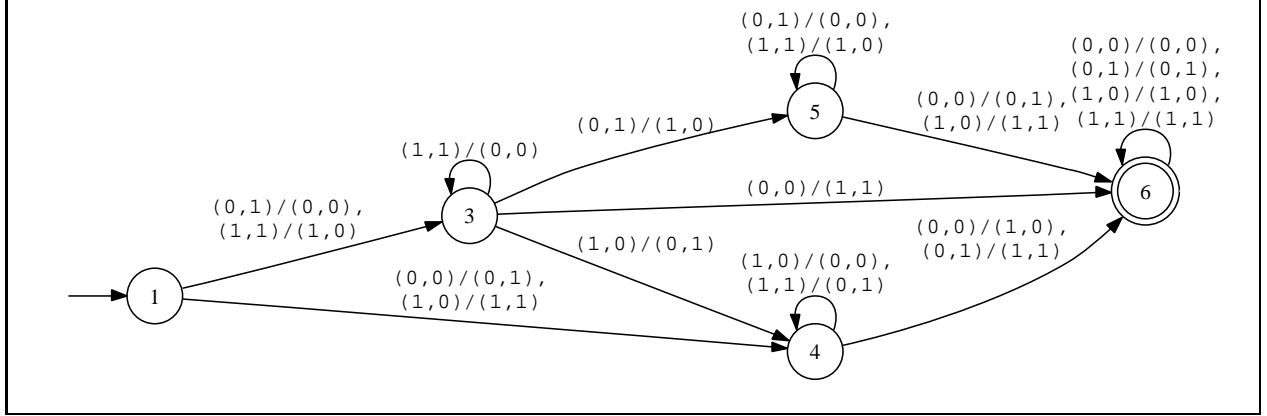


Figure 6.4: Transducer for $x = x + 2$ which also increments the count of steps

T_{Inc} and T'_r (this can be obtained by an operation similar to language intersection for NFAs). Intuitively, T takes a set of state-witness pairs S_{old} and finds a new set of state-witness pairs S_{new} such that the states in S_{new} are reachable from the states in S_{old} in one step and the count of the number of steps is incremented by one.

In our running example, the transducer T (the number of states has been minimized for clarity) is given in Figure 6.4. For instance, this transducer relates the string $(1,0)(0,1)$ ($x = 1$ and number of steps of 2) to $(1,1)(1,1)$ ($x = 3$ and number of steps of 3). Note that both the input $(1,0)(0,1)$ and output strings $(1,1)(1,1)$ are actually invalid state-witness pairs.

For the bounded steps case, let I_0 denote a DFA representing all initial states with the value of witness as 0. I_0 can be constructed from the DFA for I using a simple homomorphism defined by a function which takes a letter a in ρ to $(a, 0)$. We define a function $\mathcal{F}^t(Z)$ by

$$\mathcal{F}^t(Z) = T(Z) \cup I_0$$

Proposition 3. *The set of all valid state-witness pairs, $Reach^t$ is the least fixpoint of the function \mathcal{F}^t .*

Proof sketch. First we show that $Reach^t$ is a fixpoint of \mathcal{F}^t . Consider some $(s, i) \in Reach^t$. By definition of a valid state-witness pair, there has to be a sequence $(s_0, 0), (s_1, 1), \dots, (s_i, i)$ with $s_i = s$. Either i is 0 in which case, $(s, i) \in \mathcal{F}^t(Reach^t)$, or $(s_{i-1}, i-1)$ in the above sequence is also in $Reach^t$ (since it too is a valid state-witness pair). In the latter case, by definition $(s, i) \in \mathcal{F}^t(Reach^t)$. This shows that $(s, i) \in Reach^t \rightarrow (s, i) \in \mathcal{F}^t(Reach^t)$. Similarly, if $(s, i) \in \mathcal{F}^t(Reach^t)$ then the first possibility is that i is 0 in which case, $(s, i) \in Reach^t$. If i is not zero then there must be some $(s', i-1)$ such that $s' \rightarrow s$ and $(s', i-1) \in Reach^t$; but again if $(s', i-1)$ is a valid state-witness pair, so is (s, i) and hence $(s, i) \in Reach^t$.

This shows that $(s, i) \in \mathcal{F}^t(\text{Reach}^t) \rightarrow (s, i) \in \text{Reach}^t$. Since $(s, i) \in \mathcal{F}^t(\text{Reach}^t) \leftrightarrow (s, i) \in \text{Reach}^t$, we can see that Reach^t is a fixpoint.

To see that Reach^t is also the least fixpoint, we actually show a stronger property that \mathcal{F}^t has a unique fixpoint. Let Z and Z' be two fixpoints for \mathcal{F}^t . Let $Z_{\leq i}$ be the set $\{(s, j) \mid (s, j) \in Z \text{ and } j \leq i\}$. We will prove by induction on i that for all i , $Z_{\leq i}$ and $Z'_{\leq i}$ are equal. The base case for $i = 0$ is trivial since for all fixpoints of \mathcal{F}^t , the set of states with counter value of i as 0 has to be I_0 . Assume that the inductive hypothesis holds up to some $j > 1$. We need to show that $(s, j+1) \in Z_{\leq j+1} \Leftrightarrow (s, j+1) \in Z'_{\leq j+1}$. If $(s, j+1) \in Z$ then $(s, j+1) \in \mathcal{F}^t(Z)$. This implies there is some $(s', j) \in Z$ or $(s', j) \in Z_{\leq j}$ such that $s \rightarrow s'$. By the inductive hypothesis, $(s', j) \in Z'_{\leq j}$ or $(s', j) \in Z'$. But then, $(s, j+1) \in \mathcal{F}^t(Z')$ or $(s, j+1) \in Z'$ or $(s, j+1) \in Z'_{\leq j+1}$. Similarly, if $(s, j+1) \in Z'_{\leq j+1}$ then $(s, j+1) \in Z_{\leq j+1}$. This establishes $Z_{\leq j+1} = Z'_{\leq j+1}$. \square

The above proposition allows us to answer the equivalence query for Reach^t in the same manner as that of the bounded space case. The only issue left to be resolved is for the last case in the check when we have to determine if any state-witness pair in the hypothesis leads to an unsafe state.

State-witness pairs leading to unsafe states: Assume that we are given a DFA $U = (Q, \delta, \Sigma, q_0, F)$ representing the set of unsafe states. An NFA representing the set of state witness pairs which correspond to unsafe states with all possible witnesses is given by $(Q, \delta', (\rho \cup \{\perp\}) \times \{0, 1\}, q_0, F)$ where

$$\begin{aligned} (q, (a, 0), q') &\in \delta' \text{ if } q' = \delta(q, a) \\ (q, (a, 1), q') &\in \delta' \text{ if } q' = \delta(q, a) \\ (q, (\perp, 0), q) &\in \delta' \\ (q, (\perp, 1), q) &\in \delta' \end{aligned}$$

6.6 Verification Procedure

The overall verification procedure to check safety properties of models in the regular model checking framework is outlined in Figure 6.5 for the bounded space case. The procedure for the bounded steps is similar and is skipped. Again we assume that the target set to be learned is regular and use a variant of Angluin's L* algorithm for learning as described in Section 2.5.3. It is straightforward to show that the following theorem holds.

```

algorithm learner
begin
Regular inference algorithm
end

algorithm isMember
Input:  $s$ 
Output: is  $s$  in fixpoint?
begin
 $k = \text{length}(s)$ 
Compute least fixpoint  $Z_i = T_{|k}^n(\emptyset)$  for some  $n$ 
such that  $T_{|k}^{n-1}(\emptyset) = T_{|k}^n(\emptyset)$ 
{Here  $I$  is the set of initial states}
Is  $s \in Z_i$ ?
If yes return true
else return false
end

algorithm Equivalence Check
Input: Hypothesis  $Z'$ 
Output: For fixpoint  $Z$ , is  $Z' = Z$ ?
If not, then some string in  $Z' \oplus Z$ 
begin
If  $\mathcal{F}^s(Z') \setminus Z' \neq \emptyset$  {fixpoint check}
let  $s \in \mathcal{F}^s(Z') \setminus Z'$ 
Find  $s'$  which causes  $s$  to be in  $\mathcal{F}^s(Z')$ 
if isMember( $s$ )
return (no,  $s$ )
else
return (no,  $s'$ )
else if  $\mathcal{F}^s(Z') \subsetneq Z'$ 
return (no,  $l \in (Z' \setminus \mathcal{F}^s(Z'))$ )
else {found fixpoint}
else if  $\exists s \in U \cap Z'$ 
if isMember( $s$ )
Print (safety prop. does not hold); stop
else
return (no,  $s$ )
else
Print (safety prop. holds); stop
end

```

Figure 6.5: Verifying safety properties in regular model checking framework for the bounded space case

Theorem 4. *For verifying safety properties in the regular model checking framework, the learning to verify algorithm satisfies the following properties:*

1. *If an answer is returned by the algorithm, it is always correct.*
2. *For the bounded space case, if the set of reachable states is regular, the procedure is guaranteed to terminate.*
3. *For the bounded steps case, if the set of valid state-witness pairs is regular, the procedure is guaranteed to terminate.*

6.7 Example Application of the Verification Procedure

In this section, we show the steps taken during verification of a simple example using the learning-based verification approach. The system we analyze is depicted in Figure 6.6. In this system, there is a single integer variable x which is initially set to 0. There is a transition from the initial control state q_0 to itself which increments x by 2 and another transition which goes to the second control state if x is equal to 1001. The unsafe set of states is considered to be the second control state (labeled “bad”) with any value of x . It is easy to see that the reachable states consist of non-negative even values of x in control state q_0 and the system is safe. We now show how the learning algorithm will operate on this system.

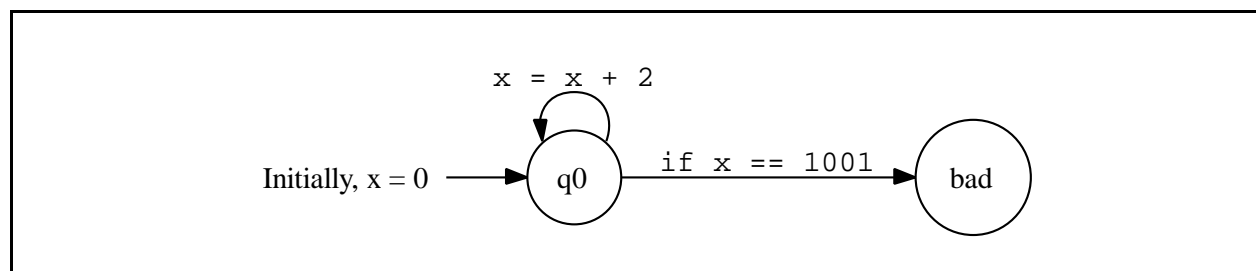


Figure 6.6: Example system to be analyzed using learning-based verification.

We use the bounded steps method for demonstration, therefore, we keep a counter (say n) which is incremented with every step. For ease of exposition, we will not explicitly show the encoding for the control state. Then, the encoding for the state-witness pairs is simply an encoding for the value of x and n . We encode over an alphabet $\{0, 1\}^2$ so that given a string s , projecting all letters in s to their first component gives the value of x in binary with the least significant bit first and projecting onto the second component gives the value of n . For example, $x = 4$ and $n = 2$ is encoded by the string $(0, 0)(0, 1)(1, 0)$. Figure 6.7 shows the different iterations of the learning algorithm and the final set of state-witness pairs that is learned.

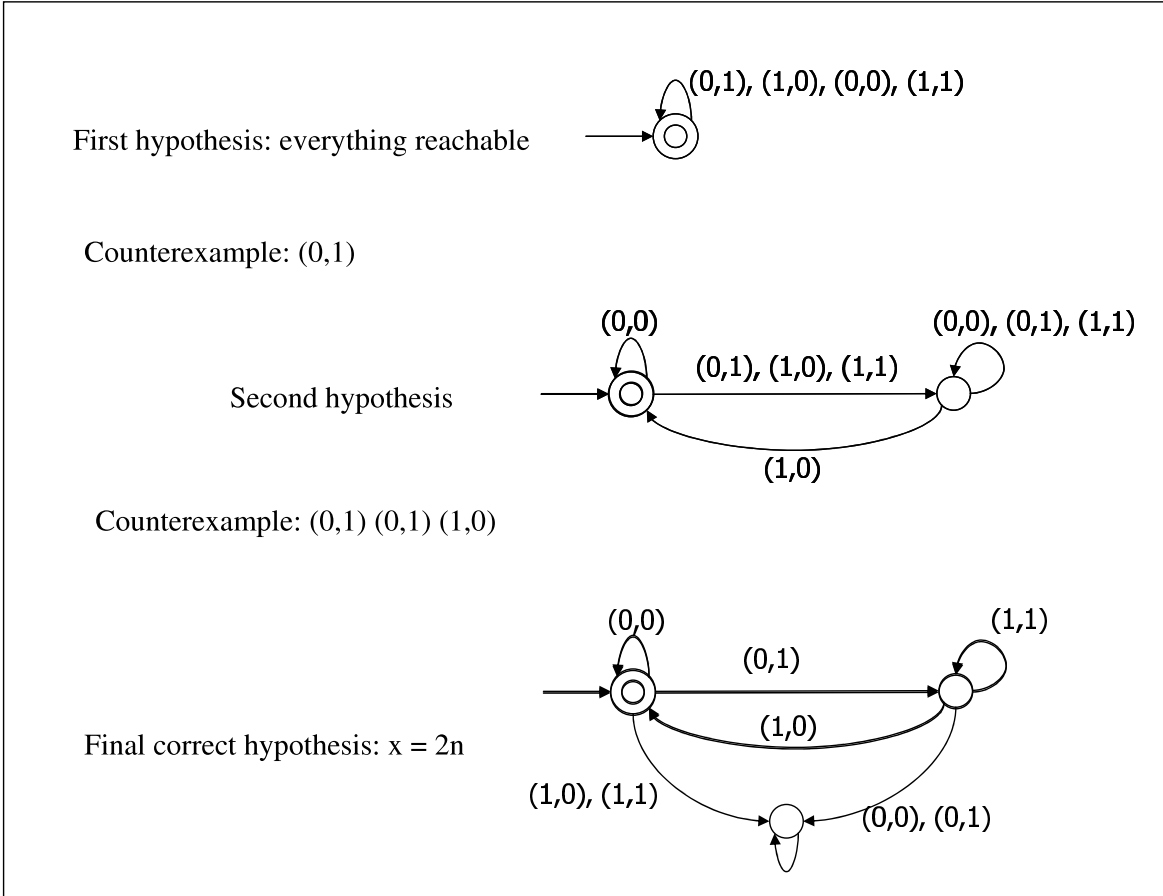


Figure 6.7: Various steps in the verification of the example system shown in Figure 6.6. Note that the final automaton shows that no unsafe state is reached, therefore, the safety property holds.

6.8 Comparison with Related Work

The discussion in Section 4.5 for FIFO automata is also applicable for the case of systems expressed in the regular model checking framework. One work that is closely related to our learning method with bounded number of steps is that of Habermehl *et al.* [67]. They propose a learning-based verification method using a variant of Trakhtenbrot-Barzdin [126] algorithm. However, this requires constructing a prefix tree automata (PTA) for all strings up to a length r where r is the so called *degree of reconstructibility* of the target automaton. The size of this PTA is exponential in r . In contrast, we use an active learning algorithm which needs only a polynomial number of membership and equivalence queries. Further, the correctness of their algorithm critically depends on the length-preserving aspect of the transition relation whereas we can handle non length-preserving transition relations using the bounded steps method.

Chapter 7

Verification of ω -regular Properties

In this chapter, we extend the learning based approach to verifying general ω -regular properties. It is well known that ω -regular properties can express all linear temporal logic specifications which include both safety and liveness properties along with fairness constraints. We first develop a new fixpoint based characterization for the verification of ω -regular properties. Next, we describe the verification method for general infinite state systems and then instantiate it to systems expressible in the *regular model checking* framework. Using a variant of Angluin's L^* algorithm for learning regular languages, we then develop an algorithm for verification of ω -regular properties of such infinite state systems and prove that it is a complete verification procedure if the fixpoint can be represented as a regular set.

7.1 Preliminaries

We use Kripke structures to model the system being verified and Büchi automata for the specification. The formal definitions of Kripke structure and Büchi automaton is given in Chapter 2.

Let K be a *Kripke structure* $(S^k, \Sigma, R^k, S_0^k, \mathcal{L})$. A *path* starting from state s is an infinite sequence $s_0, s_1, s_2 \dots$ such that $s = s_0$ and for every i , $(s_i, s_{i+1}) \in R^k$. A *path* of a Kripke structure K is just a path starting from some initial state $s \in S_0^k$. The set of all paths of K will be denoted by $\mathcal{P}(K)$. For a path $\pi = s_0, s_1, s_2, \dots$, $KTrace(\pi)$ is the sequence of labels $\ell_0, \ell_1, \ell_2, \dots$ such that for every i , $\mathcal{L}(s_i) = \ell_i$. For a set of paths Π , $KTrace(\Pi)$ is taken to be $\{KTrace(\pi) \mid \pi \in \Pi\}$.

We will use the definition of a specific CTL^* property (recall definition from Section 2.2.1), namely $EGFp$. A state s in a Kripke structure K satisfies $EGFp$ if and only if there exists a path $\pi = s_0, s_1, s_2, \dots$ starting from s such that for all i , $\mathcal{L}(s_j) = p$ for some $j \geq i$; in other words, the path encounters states labelled p infinitely often. When s satisfies $EGFp$, we will say $s, K \models EGFp$; when K is clear from the context we will simply write this as $s \models EGFp$. We will denote by $\llbracket EGFp \rrbracket_K$ the set of all states s , such that $s, K \models EGFp$.

Similar to the traditional approach used in model checking with automata theory, we assume that the

system specification is given in terms of the *bad behaviors* that the implementation must not exhibit. The bad behaviors are specified using a Büchi automaton $M = (S^m, \Sigma, S_0^m, \delta, F^m)$. Recall that the language accepted by M , which we denote by $\mathcal{S}(M)$, is the set of all words accepted by M .

For a Kripke structure K and a Büchi automaton M , K is said to be *correct* with respect to M iff $K\text{Trace}(\mathcal{P}(K)) \cap \mathcal{S}(M) = \emptyset$. Since Büchi automata are closed under complementation even if we are given the specification as an automaton M_g specifying the *good behaviors*, we can complement M_g to get M which specifies the bad behaviors.

We will reduce the problem of checking if the system satisfies the specification to the problem of checking if the CTL^* formula $EGFp$ is satisfied. In order to do this, we first define the Kripke structure obtained by taking the cross product of a Kripke structure and a Büchi automaton.

Definition 7. *The cross-product of a Büchi automaton $M = (S^m, \Sigma, S_0^m, \delta, F^m)$ and a Kripke structure $K = (S^k, \Sigma, R^k, S_0^k, \mathcal{L})$ is the Kripke structure $M \times K = (S^m \times S^k, \{f, \tilde{f}\}, R', S_0^m \times S_0^k, \mathcal{L}')$. Here, $((s_1^m, s_1^k), (s_2^m, s_2^k)) \in R'$ if and only if $(s_1^k, s_2^k) \in R^k$ and $s_2^m \in \delta(s_1^m, \mathcal{L}(s_1^k))$. A state (s^m, s^k) in $M \times K$ is labelled by f if $s^m \in F^m$ and by \tilde{f} otherwise.*

Lemma 3. *There is a path $\pi = (s_0^m, s_0^k)(s_1^m, s_1^k)(s_2^m, s_2^k) \dots$ in the product Kripke structure $M \times K$ if and only if $s_0^m s_1^m s_2^m \dots$ is a run in the Büchi automaton M on $K\text{Trace}(s_0^k s_1^k s_2^k \dots)$ where $s_0^k s_1^k s_2^k \dots$ is a path in K .*

Proposition 4. *For an automaton M , $K\text{Trace}(\mathcal{P}(K)) \cap \mathcal{S}(M) = \emptyset$ if and only if $\llbracket EGFf \rrbracket_{M \times K} \cap (S_0^m \times S_0^k) = \emptyset$ (In other words, no initial state of $M \times K$ satisfies $EGFf$).*

Proof. Suppose $K\text{Trace}(\mathcal{P}(K)) \cap \mathcal{S}(M) \neq \emptyset$. Then there is a path $\pi \in \mathcal{P}(K)$ such that $K\text{Trace}(\mathcal{P}(K))$ is accepted by M . Let $s_0^m s_1^m s_2^m \dots$ be the accepting run in M . By Lemma 3, there is a path $\pi = (s_0^m, s_0^k)(s_1^m, s_1^k)(s_2^m, s_2^k) \dots$ in $M \times K$. But since an accepting run of a Büchi automata visits an accepting state infinitely often, then by the product construction, the path $\pi = (s_0^m, s_0^k)(s_1^m, s_1^k)(s_2^m, s_2^k) \dots$ in $M \times K$ visits states labeled f infinitely often. Thus, $M \times K$ satisfies $EGFf$.

If $M \times K$ satisfies $EGFf$ then there is a path $\pi = (s_0^m, s_0^k)(s_1^m, s_1^k)(s_2^m, s_2^k) \dots$ which infinitely often visits states labeled f . By Lemma 3, there is a run $s_0^m s_1^m s_2^m \dots$ in M on $K\text{Trace}(s_0^k s_1^k s_2^k \dots)$. This is an accepting run because the product construction labels a state $(s^m, s^k) \in M \times K$ as f only if s^m is an accepting state. But then M accepts $K\text{Trace}(s_0^k s_1^k s_2^k \dots)$. Hence, $K\text{Trace}(\mathcal{P}(K)) \cap \mathcal{S}(M) \neq \emptyset$.

□

7.2 Learning to Verify ω -regular Properties

In this section, we present a general framework to verify a system described as a Kripke structure K . We assume that we are given a Büchi automaton M that describes the set of behaviors that the system K *must not* exhibit. As show by Proposition 4, we observed that the problem of checking if $K\text{Trace}(\mathcal{P}(K)) \cap \mathcal{S}(M) = \emptyset$ can be reduced to the problem of checking if an initial state of $M \times K$ satisfies $EGFf$. We first characterize $\llbracket EGFf \rrbracket$ using fixpoints of a function that we define in Section 7.2.1. Next, we show that the fixpoint is unique and has certain key properties that we need for our problem. Finally, we will show how a learning algorithm can be used to learn the fixpoint, and therefore help verify if K satisfies M .

7.2.1 Fixpoint Characterization of $EGFf$

From now on, in this chapter, we assume that we are interested in checking if some initial state of a Kripke structure $K = (S, \{f, \tilde{f}\}, R, I, \mathcal{L})$ satisfies $EGFf$. Traditionally, the fixpoint characterization of $EGFf$ is given by $\nu Z_1. EX(\mu Z_2. Z_1 \wedge (f \vee EX Z_2))$ (see [49]). Notice that this formula involves nesting of the fixpoint operators which we wish to avoid in our learning-based technique for technical reasons. Therefore, we develop a novel characterization of $EGFf$ that does not use nesting but adds some counters to be associated with states. Note that the idea of associating counters with states has been used before in [85, 132] but for different applications. We prove that the fixpoint obtained is always unique which makes it possible to answer equivalence queries exactly. As far as we know, this is a new characterization and may be of independent interest. We now proceed to describe this fixpoint.

Let X be a set of triples (s, i, j) such that $s \in S$ and $i, j \in \mathbb{N}$, where \mathbb{N} denotes the set of natural numbers. We define the function $\mathcal{F} : 2^{S \times \mathbb{N} \times \mathbb{N}} \rightarrow 2^{S \times \mathbb{N} \times \mathbb{N}}$ such that $\mathcal{F}(X) = \mathcal{F}_1(X) \cup \mathcal{F}_2(X) \cup \mathcal{F}_3(X)$, where

$$\begin{aligned} \mathcal{F}_1(X) &= \{(s, 0, j) \mid \mathcal{L}(s) = f \text{ and } j \in \mathbb{N}\} \\ \mathcal{F}_2(X) &= \{(s, i, j) \mid \mathcal{L}(s) = \tilde{f} \text{ and } \exists s'. s \rightarrow s' \exists j' < j. (s', i, j') \in X\} \\ \mathcal{F}_3(X) &= \{(s, i, j) \mid \mathcal{L}(s) = f \text{ and } \exists s'. s \rightarrow s' \exists j' < j. (s', i - 1, j') \in X\} \end{aligned}$$

The intuition behind the definition of \mathcal{F} is as follows. Consider a property $\eta_f^{i,j}$ such that a state s satisfies $\eta_f^{i,j}$ if there is a path of length j such that we encounter (at least) $i + 1$ states that are labeled f . Formally, $s \models \eta_f^{i,j}$ iff there is a finite path $s_0, s_1, s_2, \dots, s_j$ from state s such that there are indices k_1, k_2, \dots, k_{i+1} such that $\mathcal{L}(s_{k_\ell}) = f$ for every $1 \leq \ell \leq i + 1$. Now the intuition behind \mathcal{F} is that if X is a fixpoint of \mathcal{F} and $(s, i, j) \in X$ then $s \models \eta_f^{i,j}$.

Proposition 5. \mathcal{F} is monotonic and \cup -continuous.

Proof sketch. Monotonicity of \mathcal{F} is immediate from the definition.

We first prove \cup -continuity of \mathcal{F} . We have to show that $Z_0 \subseteq Z_1 \subseteq Z_2 \subseteq \dots$ implies:

$$\cup_k \mathcal{F}(Z_k) = \mathcal{F}(\cup_k Z_k)$$

We prove this equality by showing containment of each set in the other one as follows:

- $\cup_k \mathcal{F}(Z_k) \subseteq \mathcal{F}(\cup_k Z_k)$.

Clearly, for any k , $Z_k \subseteq \cup_l Z_l$. By monotonicity, $\mathcal{F}(Z_k) \subseteq \mathcal{F}(\cup_l Z_l)$. Taking union over all k , we get:

$$\begin{aligned} \cup_k \mathcal{F}(Z_k) &\subseteq \cup_k \mathcal{F}(\cup_l Z_l) \\ \text{i.e., } \cup_k \mathcal{F}(Z_k) &\subseteq \mathcal{F}(\cup_l Z_l) \\ \text{i.e., } \cup_k \mathcal{F}(Z_k) &\subseteq \mathcal{F}(\cup_k Z_k) \end{aligned}$$

- $\cup_k \mathcal{F}(Z_k) \supseteq \mathcal{F}(\cup_k Z_k)$.

Consider some $(s, i, j) \in \mathcal{F}(\cup_k Z_k)$. From the definition of \mathcal{F} , (s, i, j) is in at least one of $\mathcal{F}_1(\cup_k Z_k)$, $\mathcal{F}_2(\cup_k Z_k)$ and $\mathcal{F}_3(\cup_k Z_k)$. If $(s, i, j) \in \mathcal{F}_1(\cup_k Z_k)$, from the definition of \mathcal{F}_1 , (s, i, j) would be in $\mathcal{F}_1(Z_k)$ for all k and hence it will be in $\cup_k \mathcal{F}(Z_k)$. If $(s, i, j) \in \mathcal{F}_2(\cup_k Z_k)$ then there must be some (s', i, l) such that $s \rightarrow s'$, $\mathcal{L}(s) \in \tilde{f}$, $l < j$ and (s', i, l) in $\cup_k Z_k$ or (s', i, l) in some Z_k . But then, $(s, i, j) \in \mathcal{F}_2(Z_k)$ and hence $(s, i, j) \in \cup_k \mathcal{F}(Z_k)$. The last case of $(s, i, j) \in \mathcal{F}_3(\cup_k Z_k)$ is handled in a similar manner.

We have demonstrated that $(s, i, j) \in \mathcal{F}(\cup_k Z_k) \rightarrow (s, i, j) \in \cup_k \mathcal{F}(Z_k)$ which shows that $\cup_k \mathcal{F}(Z_k) \supseteq \mathcal{F}(\cup_k Z_k)$.

□

Since \mathcal{F} is monotonic, it has fixpoints. In addition, we can show that \mathcal{F} has a unique fixpoint. This is the objective of the next few observations.

Lemma 4. *Let X be a fixpoint of \mathcal{F} . The following two facts hold about elements of X .*

1. *If $\mathcal{L}(s) = \tilde{f}$ then $\forall i \geq 0. \forall j. (s, i, j) \in X$ if and only if $\exists s'. s \rightarrow s' \exists j' < j. (s', i, j') \in X$*
2. *If $\mathcal{L}(s) = f$ then $\forall i \geq 1. \forall j. (s, i, j) \in X$ if and only if $\exists s'. s \rightarrow s' \exists j' < j. (s', i - 1, j') \in X$*

Proof. The results follow from the definition of the fixpoint under \mathcal{F} . We illustrate this for one direction of 1; the proof for other cases is similar. Suppose $\mathcal{L}(s) = \tilde{f}$ and suppose $(s, i, j) \in X$. If $\exists s'. s \rightarrow s' \exists j' < j. (s', i, j') \in X$ does not hold then $(s, i, j) \notin \mathcal{F}(X)$ which contradicts the fact that X is a fixpoint. □

Proposition 6. *If X_1 is a fixpoint of \mathcal{F} and X_2 is also a fixpoint of \mathcal{F} then $X_1 \subseteq X_2$. Hence there is a unique fixpoint of \mathcal{F} .*

Proof. Let $(s, i, j) \in X_1$. We show that then $(s, i, j) \in X_2$. The proof will proceed by induction on i and j .

Consider the base case when $i = 0$. We will prove the claim by induction on j . Clearly $(s, 0, 0) \in X_1$ iff $\mathcal{L}(s) = f$ iff $(s, 0, 0) \in X_2$. Suppose the claim holds for $(s, 0, j')$ for all $j' < j$. Consider $(s, 0, j) \in X_1$. If $\mathcal{L}(s) = f$ then $(s, 0, j) \in X_2$ for every j by the definition of \mathcal{F}_1 . Now if $\mathcal{L}(s) = \tilde{f}$ then by Lemma 4, it must be the case that there is s' and j' such that $s \rightarrow s'$, $j' < j$ and $(s', 0, j') \in X_1$. By the induction hypothesis, we know that $(s', 0, j') \in X_2$. Again, by Lemma 4, this means that $(s, 0, j) \in X_2$.

Assume that for every $i' < i$ and for every j' , if $(s, i', j') \in X_1$ then $(s, i', j') \in X_2$. The induction step for (s, i, j) is proved by induction on j . For the base case, when $j = 0$, we observe that $(s, i, 0)$ is not a member of any fixpoint of \mathcal{F} (Lemma 4). The proof of the induction step is similar to the case of $i = 0$, and is skipped in the interests of space.

By symmetry, $X_2 \subseteq X_1$, hence $X_1 = X_2$ giving the uniqueness of the fixpoint for \mathcal{F} . \square

Henceforth, we use X to denote the *unique* fixpoint of \mathcal{F} . We are now ready to state the proposition that formally proves our intuition behind defining \mathcal{F} .

Proposition 7. *Suppose X is the fixpoint of \mathcal{F} . Then, $(s, i, j) \in X$ if and only if $s \models \eta_f^{i,j}$*

Proof. (\Rightarrow) We prove this by induction on i and j . For the base case consider $i = 0$. We now induct on j . When $j = 0$, $(s, 0, 0) \in X$ iff $\mathcal{L}(s) = f$, which means that there is a path of length 0 starting from s where we encounter one state labeled f . Now suppose $j > 0$. If $\mathcal{L}(s) = f$ then it trivially follows that there is a path of length $j > 0$ starting from s where we encounter at least one state labeled f . Suppose $\mathcal{L}(s) = \tilde{f}$. Then by Lemma 4, there is s' and $j' < j$ such that $s \rightarrow s'$ and $(s', 0, j') \in X$. Then by induction hypothesis, $s' \models \eta_f^{0,j'}$ which then implies that $s \models \eta_f^{0,j}$.

Consider $i > 0$. Once again we induct on j . Observe that since by Lemma 4, $(s, i, 0)$ is not in any fixpoint when $i > 0$, the claim holds vacuously. The induction step goes through in manner similar to the case of $i = 0$ and the proof is therefore skipped.

(\Leftarrow) We prove the converse direction also by induction. Consider $i = 0$. If $j = 0$ and $s \models \eta_f^{0,0}$ then it must be the case that $\mathcal{L}(s) = f$. This means that $(s, 0, 0) \in X$. Suppose $j > 0$ and $s \models \eta_f^{0,j}$. If $\mathcal{L}(s) = f$ then once again $(s, 0, j) \in X$. If $\mathcal{L}(s) = \tilde{f}$ then it must be the case that there is some s' such that $s \rightarrow s'$ and $s' \models \eta_f^{0,j-1}$. Thus by induction hypothesis $(s', 0, j-1) \in X$ and therefore by Lemma 4, $(s, 0, j) \in X$.

Consider $i > 0$ and $s \models \eta_f^{i,j}$. If $\mathcal{L}(s) = f$ then it is definitely the case that there is s' such that $s \rightarrow s'$ and $s' \models \eta_f^{i-1,j-1}$. By induction hypothesis, $(s', i-1, j-1) \in X$, and that implies (by Lemma 4) that

$(s, i, j) \in X$. On the other hand, if $\mathcal{L}(s) = \tilde{f}$ then we can conclude that there is s' such that $s \rightarrow s'$ and $s' \models \eta_f^{i, j-1}$. By induction hypothesis this means that $(s', i, j-1) \in X$, and by this we can conclude that $(s, i, j) \in X$ because of Lemma 4. \square

We are now ready to characterize $\llbracket EGFf \rrbracket$ in terms of the fixpoint X of \mathcal{F} . This is the formal content of Proposition 8. But before presenting that proposition, we need a technical definition.

Definition 8. $\sigma(X) = \{s \mid \forall i \exists j. (s, i, j) \in X\}$

Proposition 8. *Suppose X is the fixpoint of \mathcal{F} . Then $s \in \sigma(X)$ if and only if $s \models EGFf$*

Proof. (\Leftarrow) Suppose $s \models EGFf$. Then there is a path $\pi = s_0, s_1, s_2, \dots$ starting from s , such that for infinitely many k , $\mathcal{L}(s_k) = f$. Define j_i to be the least k such that $\mathcal{L}(s_k) = f$ and there are $i+1$ states before s_k on π that are also labeled f . It is clear that $s \models \eta_f^{i, j_i}$ and therefore by Proposition 7, $(s, i, j_i) \in X$. Hence $s \in \sigma(X)$.

(\Rightarrow) Suppose $s \in \sigma(X)$. By definition, for every i , there is some j such that $(s, i, j) \in X$. Hence, by Proposition 7, $s \models \eta_f^{i, j}$. Construct a tree with root s , containing edges appearing in all shortest paths that witness s satisfying $\eta_f^{i, j}$. A few observations about this tree are in order. First, the tree is finite branching; an immediate consequence of the Kripke structure being finite branching. Second, all leaves are labeled f since the tree is constructed using the shortest witnesses. Third, if s' is an internal node in the tree then every path from s' in the tree will reach a state labeled f . Finally, this tree has infinitely many vertices. By König's Lemma, there must be an infinite path in the tree. Let us call this infinite path π . We claim that this infinite path witnesses $EGFf$. Consider any state s' on path π . Since s' is an internal node in the tree, it must be the case that on every path from s' in the tree we encounter a state labeled f . In particular on the path π , we encounter a state labeled f beyond s' . Thus π has infinitely many states labeled f . \square

7.2.2 Learning Fixpoints

We are now ready to present our general framework for verifying ω -regular properties using learning. We make the following assumptions about the system K being verified.

1. The system K can be simulated from any state.
2. There is a convenient symbolic representation for sets consisting of triples (s, i, j) , where s is a state and i, j are natural numbers. This means that the representation is closed under complementation and decision procedures are available for membership in a set, containment of one set in another, and emptiness of a set.

3. Given the representation of a set Y of triples (s, i, j) and a state s it is possible to check if $s \in \sigma(Y)$
4. Given a representation of a set Y of triples (s, i, j) it is possible to compute the representation of $\mathcal{F}(Y)$
5. There is an active learning algorithm for concepts encoded in the symbolic representation.

Based on these assumptions, we show how learning can be used to verify ω -regular properties. The central idea is to use the learning algorithm to learn the fixpoint X of \mathcal{F} . After we learn the fixpoint, based on Propositions 4 and 8, we can reliably answer whether or not the system satisfies the specification. Thus to verify ω -regular properties using learning, we need to implement the membership and equivalence oracles that the learning algorithm needs.

Proposition 7 suggests a method to answer membership queries about whether (s, i, j) belongs to the fixpoint X of \mathcal{F} . To check if (s, i, j) belongs to X , we will simulate the system for j steps starting from state s and check if on some path, we encounter $i + 1$ states labeled f . Further, given a representation for a set Y , we can also answer whether Y is in fact equal to X . Since \mathcal{F} has a unique fixpoint, all we need to do is check if $\mathcal{F}(Y) = Y$. If $\mathcal{F}(Y) \neq Y$ then the equivalence query must provide a counterexample. In other words, we need to produce an element in the symmetric difference of Y and X . Similar to the case of safety properties (described in Section 5.4), this can be done as follows for the different possible cases.

- $\mathcal{F}(Y) \setminus Y \neq \emptyset$. Let $l = (s, i, j)$ be some element in this set. If $l = (s, 0, 0)$ then $l \in X$, because the only way we can have any $(s, 0, 0)$ in $\mathcal{F}(Y)$ is if $\mathcal{L}(s) = f$. In this case, l is in X and hence in $X \oplus Y$. If $l = (s, 0, j)$ and $\mathcal{L}(s) = f$ then once again $l \in X$ and hence in $X \oplus Y$. If $l = (s, i, j)$ for some $j \neq 0$, we can check if $l \in X$ using the membership query. If the answer is yes, then l is also in $X \oplus Y$ and we are done. Otherwise, $l \in \mathcal{F}(Y)$ because of the existence of some triple $(s', i', j') \in Y$ which satisfies the conditions \mathcal{F}_2 or \mathcal{F}_3 . (s', i', j') cannot be in X otherwise (s, i, j) would have to be in X . Hence $(s', i', j') \in X \oplus Y$.
- $\mathcal{F}(Y) \subsetneq Y$. From standard fixpoint theory, since X happens to also be the least fixpoint under \mathcal{F} , it must be the intersection of all prefixpoints of \mathcal{F} (a set Z is a prefixpoint if it *shrinks* under the function \mathcal{F} , *i.e.* $\mathcal{F}(Z) \subseteq Z$). Now, Y is clearly a prefixpoint. Applying \mathcal{F} to both sides of the equation $\mathcal{F}(Y) \subsetneq Y$ and using monotonicity of \mathcal{F} , we get $\mathcal{F}(\mathcal{F}(Y)) \subsetneq \mathcal{F}(Y)$. Thus, $\mathcal{F}(Y)$ is also a prefixpoint. Let l be some string in the set $Y \setminus \mathcal{F}(Y)$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint X . Hence, l is in $X \oplus Y$.

Once we have learned the fixpoint X , we can verify if the initial states of the Kripke structure satisfy *EGFf* using Proposition 8. By Proposition 4, this provides an answer to the verification problem. The

7.3.1 Construction of the Product Kripke Structure

Let M be the Büchi automaton specifying the bad behaviors that must not be exhibited by the system. Since ω -regular languages are powerful enough to express fairness constraints, we assume that such constraints, if any, are already embodied in the Büchi automaton. We now show how to construct the product Kripke structure $M \times K$. We extend the alphabet ρ^k to $\rho^{M \times K}$ with new symbols b_{s^m} , one for each state s^m in M . A state (s^m, s^k) in $M \times K$ is encoded as a string with the first letter as b_{s^m} and the remaining part of the string as the original string encoding s^k . Initial states in $I^{M \times K}$ are given by concatenating a letter b_{s_0} for $s_0 \in S_0^m$ and a string in I^k . The set of states $S_{\tilde{f}}$ (resp. S_f) labelled with \tilde{f} (resp. f) is given by a DFA which looks at the first letter of the input string and accepts if this is b_{s^m} for some $s^m \notin F^m$ (resp. $s^m \in F^m$). The transducer $T^{M \times K}$ representing the transition relation for $M \times K$ is a bit more tedious but can be constructed using standard automata operations.

Henceforth, we restrict our attention to the Kripke structure $M \times K$. For ease of notation, we drop the superscript $M \times K$ in T , I , ρ and so on.

7.3.2 Symbolic Representation for the Fixpoint X

As discussed in Section 7.2.2, we now need to learn the fixpoint X of the function \mathcal{F} . In general, X is a subset of $\rho^* \times \mathbb{N} \times \mathbb{N}$. To encode X as a regular set we use the alphabet ρ^X given by $(\rho \cup \{\perp\}) \times \{0, 1\} \times \{0, 1\}$. This is the alphabet that will be used by the learning algorithm. Here, \perp is a new “filler” symbol. An element (s, i, j) is encoded as string over ρ^X such that projecting the symbols on the first component gives us s (the \perp symbols are ignored); and projecting on the second and third components gives i and j respectively in binary notation. Alternatively, the numbers can also be represented in unary.

7.3.3 Membership and Equivalence queries

As discussed before, membership queries for X can be answered using Proposition 7. For answering equivalence queries, we need a symbolic way to calculate $\mathcal{F}(X)$. Apart from the standard operations on regular set we define the following.

Definition 9. *Given Y a set of strings in the alphabet of ρ^X , define*

$$\begin{aligned} Inc^i(Y) &= \{(s, i, j) \mid (s, i-1, j) \in Y\} \\ Inc^j(Y) &= \{(s, i, j) \mid (s, i, j-1) \in Y\} \end{aligned}$$

Given a DFA for Y , the DFA for $Inc^i(Y)$ and $Inc^j(Y)$ can be constructed using a method similar to the

one described in Section 6.5.1 for checking safety properties of systems in regular model checking framework. The only difference is that we now have three components in the alphabet instead of two.

Checking hypothesis for upward closure in j . A property that we will find useful in answering equivalence queries is that by definition of \mathcal{F} , its fixpoint X is upward closed in the j component, *i.e.*, if (s, i, j) in X then for all $j' > j$, (s, i, j') is also in X . A set Y is upward closed in the j component if and only if $Inc^j(Y) \subseteq Y$. If Y is not upward closed then let (s, i, j) be the string in $Inc^j(Y) \setminus Y$. Clearly, $(s, i, j) \notin Y$. Now we use membership query to check if $(s, i, j) \in X$. If (s, i, j) is indeed in X then (s, i, j) is in the symmetric difference $X \oplus Y$. Otherwise $(s, i, j - 1)$ is also not in X (since X has the upward closed property). In this case $(s, i, j - 1) \in X \oplus Y$.

Symbolic computation of \mathcal{F}_1 . A finite automaton for $\mathcal{F}_1(Y)$ is obtained by taking the DFA for f and taking its cross product with a DFA that accepts 0 for the i component and another DFA which accepts any j .

Symbolic computation of \mathcal{F}_2 . If we always first check for upward closure in j , we can assume that we would need to compute \mathcal{F}_2 only for sets which are upward closed. Let $T^{-1}(Y)$ be the inverse of T lifted to the triples (s, i, j) so that it simply copies the second and the third components. It can be seen that if Y is upward closed then $\mathcal{F}_2(Y) = S_{\bar{f}} \cap Inc^j(T^{-1}(Y))$.

Symbolic computation of \mathcal{F}_3 . For \mathcal{F}_3 , $T^{-1}(Y)$ gives the set of states which have a successor in Y . It is easy to see that $\mathcal{F}_3(Y) = S_f \cap Inc^i(Inc^j(T^{-1}(Y)))$.

Using the fixpoint check. From the previous paragraphs, we have a symbolic method to compute $\mathcal{F}(Y) = \mathcal{F}_1(Y) \cup \mathcal{F}_2(Y) \cup \mathcal{F}_3(Y)$. Now, the equivalence oracle simply needs to check if $Y = \mathcal{F}(Y)$. We also need a method of extracting strings in the symmetric difference of Y and the fixpoint in case Y is not the fixpoint. It can be seen that the approach outlined in Section 7.2.2 can be applied to regular sets.

7.3.4 Checking for $s_0 \in \sigma(X)$.

Proposition 9. $\sigma(X) = \overline{Proj_1(Proj_{1,2}(X))}$. Here, $Proj_1$ is the projection to the first component and $Proj_{1,2}$ the projection to the first and second components.

Sketch. Recall that $\sigma(X) = \{s \mid \forall i \exists j. (s, i, j) \in X\}$. Equivalently, $\sigma(X) = \{s \mid \neg(\exists i \neg(\exists j. (s, i, j) \in X))\}$. The claim follows from the fact that \exists can be eliminated using projection and the \neg operator corresponds to taking the complement. \square

Given a regular representation of X we can calculate $\sigma(X)$ using standard regular set operations. Then the system is correct if and only if $I \cap \sigma(X) = \emptyset$.

The verification algorithm is summarized in Figure 7.2.

7.3.5 Complexity Analysis

Let m be the length of the longest string returned by the teacher in a negative answer to an equivalence query, n be the number of states of the minimal automaton representing the fixpoint X , k be the size of the alphabet of the learned language and t be the number of states of the automaton representing the transducer for the transition relation. By Proposition 1, the language inference algorithm makes $O(kn^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. The worst case for the equivalence query for a hypothesis Y occurs when we look for a string in the difference of Y and $\mathcal{F}(Y)$. The size of DFA representing Y is bounded by n . Looking at \mathcal{F} , it can be seen that the DFA representing the difference of Y and $\mathcal{F}(Y)$ would be $O(nt)$. Thus the length of the longest string returned by an equivalence query is $m = O(nt)$.

The cost of answering membership queries dominates the total runtime cost of the algorithm. Using $m = O(nt)$, the number of membership queries is $O(kn^2 + n \log nt)$. For efficiency, given a query for (s, i, j) , we build a DFA D_j for $\mathcal{F}^{j+1}(\emptyset)$ where \mathcal{F}^{j+1} denotes the composition of \mathcal{F} $j + 1$ times with itself. Once D_j has been built, all queries with the same value of j can be answered by checking if the queried element is accepted by D_j . Thus the cost of the membership queries is equal to the number of membership queries and the cost of building the DFAs. The cost for D_j is $(O(t))^j$ which leads to the total cost of membership queries of $O(t^{O(nt)} + kn^2 + n \log nt)$ (using maximum value of j to be $m = O(nt)$).

7.4 Comparison with Related Work

Verification of ω -regular properties for infinite state systems has also been addressed in [25] and [113]. In a similar line of work, Abdulla *et al.* [1] present a “two-dimensional” modal logic called LTL(MSO) for verification of liveness properties. The above approaches rely on loop detection for checking liveness and assume that the transition relation is length preserving. Recently, Bouajjani *et al.* [26] have analyzed liveness properties of non-length preserving systems using a notion of simulation between states.

Our learning based technique for verifying ω -regular specifications enjoys several advantages. First, we do not need the transition relation to be restricted to be length-preserving as has been assumed in some other approaches such as [67, 1]. In fact, our general framework can potentially be used to verify systems symbolically represented using *polyhedra* or *ellipsoids*, not just regular languages, provided appropriate

```

algorithm learner
begin
Regular inference algorithm
end

algorithm isMember
Input:  $(s, i, j)$ 
Output: is  $(s, i, j) \in X$ ?
begin
  From  $s$  simulate system for  $j$  steps
  Does any path in above encounter
  at least  $i + 1$  states labelled  $f$ ?
  If yes return true
  else return false
end

algorithm Equivalence Check
Input: Hypothesis  $Y$ 
Output: For fixpoint  $X$ , is  $Y = X$ ?
If not, then some string in  $Y \oplus X$ 
begin
  If  $Inc^j(Y) \setminus Y \neq \emptyset$  {upward closure check}
  let  $(s, i, j) \in Inc^j(Y) \setminus Y$ 
  if isMember $((s, i, j))$ 
    return (no,  $(s, i, j)$ )
  else
    return (no,  $(s, i, j - 1)$ )
  else if  $\mathcal{F}(Y) \setminus Y \neq \emptyset$  {fixpoint check}
  let  $(s, i, j) \in \mathcal{F}(Y) \setminus Y$ 
  Find  $(s', i', j')$  which causes  $(s, i, j)$  to be in  $\mathcal{F}(Y)$ 
  if isMember $((s, i, j))$ 
    return (no,  $(s, i, j)$ )
  else
    return (no,  $(s', i', j')$ )
  else if  $\mathcal{F}(Y) \subsetneq Y$ 
    return (no,  $l \in (Y \setminus \mathcal{F}(Y))$ )
  else {found fixpoint}
  if  $I \cap \overline{Proj_1(Proj_{1,2}(X))} \neq \emptyset$ 
    print "System incorrect"
  else
    print "System correct"
end

```

Figure 7.2: Verifying ω -regular properties for regular set based systems

learning algorithms can be plugged in. Second, our algorithm for checking containment of the system's trace language in the specification automata's language, is not based on discovering loops where final states of the automata are visited infinitely often (as is the case in [67]). Thus, our algorithm will successfully identify faulty systems, even when there is no ultimately periodic execution that witnesses the violation. This is important because for general infinite state systems, it is often the case that there is no such ultimately periodic execution witnessing the violation of a liveness property. Finally, since we use a variant of Angluin's L^* algorithm, we are guaranteed to not only learn the smallest automaton representing the fixpoint, but are also guaranteed to only make polynomially many calls to the membership and equivalence oracles.

Chapter 8

Verification of CTL properties with Fairness Constraints

Computation Tree Logic (CTL) [39] is a temporal logic in which one can express properties about the branching, non-deterministic behavior of the system. Properties about information flow in the system, which cannot be expressed in a specification language that reasons only about individual computations, can be written in CTL. For these reasons, CTL is very often used to describe the correctness requirements of a system. Note that the expressive powers of CTL and ω -regular are incomparable; there exists some properties that can be expressed only in CTL and vice versa.

In this chapter, we present a learning based model checking algorithm for infinite state systems with respect to CTL properties. The algorithm presented here is the first CTL model checker (based on learning or otherwise) for infinite state systems with fairness constraints; the CTL model checker for infinite state systems in [32] did not account for fairness constraints. Finally, there is precise characterization of the class of systems for which this model checking algorithm is complete; for every subformula, if the set of states satisfying it form a regular language, then the algorithm presented here is guaranteed to terminate with the right answer.

In order to apply the learning based verification method to CTL, we need to overcome two fundamental challenges. To better understand these problems, let us recall the case of verifying invariants. As outlined before, the learning based method to verify invariants calls an algorithm that attempts to learn the set of reachable states, and then checks whether the invariant is violated in any of the states in the learnt set. To guarantee soundness, the model checker has to check if the set returned by the learning algorithm is indeed the set of reachable states, without actually computing the set of reachable states again. This turns out to be a difficult problem because while it is easy to check if a set is closed with respect to the transition relation (and hence contains all reachable states), there is no easy way to check if it is the *smallest* such set. Instead, the learning based method checks whether the learnt set contains all reachable states and does not violate the invariant, or whether a specific unsafe state in the learnt set is reachable. These *approximate* tests turn out to be feasible, and sufficient for the purposes of verifying the invariant. Thus, in the case of invariant verification, the learning based model checker does not ever know whether it has actually computed the set

of reachable states, but only knows whether it has discovered a proof of correctness or a proof of violation.

To extend the learning approach to verify CTL, the following approach suggests itself immediately. Similar to the classical model checking algorithm for CTL [39], progressively compute the set of states satisfying each of the subformulas, starting from simple atomic propositions; the only difference being that we learn the sets instead of computing them iteratively. However, this approach runs into the problem that unlike safety properties and ω -regular properties, CTL properties are nested fixpoints, where the set of states satisfying inner subformulas is used in the computation of the set of states satisfying outer subformulas. Hence, we will need a test that checks whether the learning algorithm has learnt the exact set of states satisfying a particular subformula (and not some over or under-approximation). Once again, while it is easy to check if a set is a fixpoint, it is unclear how to check if it is the *least* fixpoint or the *greatest* fixpoint. We overcome this central problem by presenting a new characterization of CTL operators in terms of functions with unique fixpoints. The output of a learning algorithm trying to discover these unique fixpoints can then be easily checked, and this allows us to get a learning based model checking algorithm for CTL.

The next challenge is to adequately take into account the fairness constraints that might be associated with the system being verified. In the case of finite state systems, this is handled using the observation that it is sufficient to only consider fair computations that are *ultimately periodic* and *looping*, i.e., computations that repeatedly execute a sequence of steps that loop to a state. However, this observation does not extend to infinite state systems. In order to soundly verify an infinite state system with respect to fairness constraints, we need to also consider fair computations that are truly infinite, and are not looping. We generalize ideas that we developed for the verification of ω -regular properties in Chapter 7 to account for fairness.

We instantiate our technique to systems in which states are encoded as strings and use a regular inference algorithm to learn the CTL fixpoints. We prove that if the fixpoints have a regular representation, our procedure will always terminate with the correct answer.

8.1 Learning to Verify CTL Properties

Recall (from Section 2.3) that the classical model checking algorithm for CTL proceeds by inductively determining the set of all states that satisfy each of the subformulas. For each subformula, the algorithm to determine the set of states satisfying it is determined based on the outermost logic operator. Given a suitable representation for sets of states, $\neg f$ and $f_1 \vee f_2$ correspond to performing the boolean operations of complementation and union on the sets of states satisfying f , f_1 and f_2 . In the case of EX , it involves computing predecessors: $EX(Z) = \{s \mid \exists s' \rightarrow s' \text{ and } s' \in Z\}$.

The most interesting cases are those of EU and EG , which are handled by computing fixpoints. To illustrate the challenges in developing a model checking algorithm for infinite-state systems, let us consider a formula $E[\text{true } U \ f]$ which is also sometimes written as EFf . The set of states satisfying EFf consists of the states which can reach a state labeled by f . Thus, EFf can be found by starting with a set Z_0 consisting of states satisfying f and in the i th iteration adding the states that can reach a state satisfying f in i -steps. Clearly this method of computing the set of state satisfying EFf may not terminate for a system with infinitely many states. As mentioned before, our idea is to learn this set instead of performing this iterative computation. In order to do this, we have to answer membership and equivalence queries for the set of states satisfying EFf . We do have a weak test for equivalence; if the set hypothesized for EFf changes under backward reachability then it is certainly not the right set. However, even if it does not change under backward reachability, it may be just an overapproximation of EFf (in the case of EGf this can be an underapproximation). For membership queries the situation is even more difficult; it is unclear how we can answer whether a state s satisfies EFf without solving the original verification problem. As before, the solution is to learn a set with more information from which EFf can be computed and which allows answering membership and equivalence queries. In the case of EFf , one simple way to achieve this is to learn a set X of (s, i) pairs where $(s, i) \in X$ means that the state s can reach some state labeled f in i steps. Now, a membership check $(s, i) \in X$ involves checking if in i steps a computation from s can reach X , which is an easier problem. Moreover, it can be shown that there is a unique set that is a fixpoint for the function $\Gamma(Z) = \{(s, j) \mid \exists s \rightarrow s' \text{ and } (s', j-1) \in Z\} \cup \{(s, 1) \mid s \models f\}$. This allows us to answer the equivalence query for EFf exactly.

Using the ideas informally presented above, we can develop a learning based algorithm for CTL, in the absence of fairness constraints. This is formally presented next. After this, we consider the case of model checking in the presence of fairness constraints.

8.1.1 CTL Formulas without Fairness

First, let us consider the problem of model checking a Kripke structure that does not have any fairness constraints. As we saw before, \neg , \vee , EX can be handled in a fairly straightforward manner. From [39], we know that $E[f_1 U f_2]$ is the least fixpoint of the function $\mathcal{F}_{E[f_1 U f_2]} : 2^S \mapsto 2^S$ given by $\mathcal{F}_{E[f_1 U f_2]}(Z) = \llbracket f_2 \rrbracket \cup (\llbracket f_1 \rrbracket \cap EX \ Z)$ or equivalently:

$$\mathcal{F}_{E[f_1 U f_2]}(Z) = \{s \mid s \in \llbracket f_1 \rrbracket \text{ and } \exists s'. s \rightarrow s' \text{ and } s' \in Z\} \cup \llbracket f_2 \rrbracket$$

Here, $\llbracket f \rrbracket$ denotes the set of states satisfying the subformula f . Further, EGf is the greatest fixpoint of the function $\mathcal{F}_{EGf}(Z) : 2^S \mapsto 2^S$ given by $\mathcal{F}_{EGf}(Z) = \llbracket f \rrbracket \cap EX Z$ or equivalently:

$$\mathcal{F}_{EGf}(Z) = \{s \mid \exists s'. s \rightarrow s' \text{ and } s' \in Z\} \cap \llbracket f \rrbracket$$

As discussed before, we want to derive new functions which will allow us to use learning techniques.

Let \mathbb{N} be the set of natural numbers, and $\Gamma_{E[f_1 U f_2]} : 2^{S \times \mathbb{N}} \mapsto 2^{S \times \mathbb{N}}$ and $\Gamma_{EGf} : 2^{S \times \mathbb{N}} \mapsto 2^{S \times \mathbb{N}}$ be two functions defined as follows.

Definition 10.

$$\Gamma_{E[f_1 U f_2]}(Z) = \{(s, i + 1) \mid s \in \llbracket f_1 \rrbracket \text{ and } \exists s'. s \rightarrow s' \text{ s.t. } (s', i) \in Z\} \cup \llbracket f_2 \rrbracket \times \{1\}$$

Definition 11.

$$\Gamma_{EGf}(Z) = \{(s, i + 1) \mid s \in \llbracket f \rrbracket \text{ and } \exists s'. s \rightarrow s' \text{ s.t. } (s', i) \in Z\} \cup \llbracket f \rrbracket \times \{1\}$$

Intuitively, $\Gamma_{E[f_1 U f_2]}$ (resp. Γ_{EGf}) keeps an additional counter i with each state, with the counter value of 1 associated with all states in $\llbracket f_2 \rrbracket$ (resp. $\llbracket f \rrbracket$). Both functions do a similar backward transition step as $\mathcal{F}_{E[f_1 U f_2]}$ and \mathcal{F}_{EGf} except that at each step the value of the counter is incremented by one.

Proposition 10. $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} are monotonic and \cup -continuous.

Proof sketch. The proof proceeds along similar lines as the proof of Proposition 5.

Monotonicity of $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} is immediate from the definition.

We first prove \cup -continuity of $\Gamma_{E[f_1 U f_2]}$. We have to show that $Z_0 \subseteq Z_1 \subseteq Z_2 \subseteq \dots$ implies:

$$\cup_i \Gamma_{E[f_1 U f_2]}(Z_i) = \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$$

We prove this equality by showing containment of each set in the other one as follows:

- $\cup_i \Gamma_{E[f_1 U f_2]}(Z_i) \subseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$.

Clearly, for any j , $Z_j \subseteq \cup_i Z_i$. By monotonicity, $\Gamma_{E[f_1 U f_2]}(Z_j) \subseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$. Taking union over all j , we get:

$$\begin{aligned} \cup_j \Gamma_{E[f_1 U f_2]}(Z_j) &\subseteq \cup_j \Gamma_{E[f_1 U f_2]}(\cup_i Z_i) \\ \text{i.e., } \cup_j \Gamma_{E[f_1 U f_2]}(Z_j) &\subseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i) \\ \text{i.e., } \cup_i \Gamma_{E[f_1 U f_2]}(Z_i) &\subseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i) \end{aligned}$$

- $\cup_i \Gamma_{E[f_1 U f_2]}(Z_i) \supseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$.

Consider some $(s, k) \in \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$ with $k \geq 1$. From the definition of $\Gamma_{E[f_1 U f_2]}$, (s, k) is either in $\llbracket f_2 \rrbracket \times \{1\}$ or there exists some s' such that $s \rightarrow s'$ and $(s', k-1) \in \cup_i Z_i$. In the former case, (s, k) would be in $\Gamma_{E[f_1 U f_2]}(Z_i)$ for all i and hence it will be in $\cup_i \Gamma_{E[f_1 U f_2]}(Z_i)$. In the latter case, $(s', k-1)$ must be in some Z_j . But then, (s, k) must be in $\Gamma_{E[f_1 U f_2]}(Z_j)$ and hence in $\cup_i \Gamma_{E[f_1 U f_2]}(Z_i)$. We have demonstrated that $(s, k) \in \Gamma_{E[f_1 U f_2]}(\cup_i Z_i) \rightarrow (s, k) \in \cup_i \Gamma_{E[f_1 U f_2]}(Z_i)$ which shows that $\cup_i \Gamma_{E[f_1 U f_2]}(Z_i) \supseteq \Gamma_{E[f_1 U f_2]}(\cup_i Z_i)$.

The proof of \cup -continuity of Γ_{EGf} proceeds along exactly the same lines. \square

Instead of trying to learn the least (greatest) fixpoint of $\mathcal{F}_{E[f_1 U f_2]}(\mathcal{F}_{EGf})$, we can learn $\Gamma_{E[f_1 U f_2]}(\Gamma_{EGf})$ and then retrieve the desired fixpoint of $\mathcal{F}_{E[f_1 U f_2]}(\mathcal{F}_{EGf})$. The first challenge is to answer equivalence queries. For this, we have the following proposition.

Proposition 11. $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} have unique fixpoints.

Proof sketch. Since $\Gamma_{E[f_1 U f_2]}$ is a monotonic operator on sets of states, it has fixpoints. Let Z and Z' be two fixpoints for $\Gamma_{E[f_1 U f_2]}$. Let $Z_{\leq i}$ be the set $\{(s, j) \mid (s, j) \in Z \text{ and } j \leq i\}$. We will prove by induction on i that for all i , $Z_{\leq i}$ and $Z'_{\leq i}$ are equal. The base case for $i = 1$ is trivial since for all fixpoints of $\Gamma_{E[f_1 U f_2]}$, the set of states with counter value of i as 1 has to be $\llbracket f_2 \rrbracket$. Assume that the inductive hypothesis holds up to some $j > 1$. We need to show that $(s, j+1) \in Z_{\leq j+1} \Leftrightarrow (s, j+1) \in Z'_{\leq j+1}$. If $(s, j+1) \in Z$ then $(s, j+1) \in \Gamma_{E[f_1 U f_2]}(Z)$. This implies $s \in \llbracket f_1 \rrbracket$ and there is some $(s', j) \in Z$ or $(s', j) \in Z_{\leq j}$ such that $s \rightarrow s'$. By the inductive hypothesis, $(s', j) \in Z'_{\leq j}$ or $(s', j) \in Z'$. But then, $(s, j+1) \in \Gamma_{E[f_1 U f_2]}(Z')$ or $(s, j+1) \in Z'$ or $(s, j+1) \in Z'_{\leq j+1}$. Similarly, if $(s, j+1) \in Z'_{\leq j+1}$ then $(s, j+1) \in Z_{\leq j+1}$. This establishes $Z_{\leq j+1} = Z'_{\leq j+1}$. Since $Z' = \cup_{j \geq 1} Z'_{\leq j}$ and $Z_{\leq j} = Z'_{\leq j}$ for all j , we can conclude that $Z = Z'$.

The proof for Γ_{EGf} goes through in the same manner. \square

The above proposition helps us answer equivalence queries as follows. The query asks whether a proposed hypothetical set Z' is the same as the fixpoint Z of Γ (Γ could be $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf}). Since the fixpoint of Γ is unique, this can be correctly answered by checking if Z' itself is a fixpoint, *i.e.* comparing Z' with the image $\Gamma(Z')$. It is important to note that in general the fixpoints encountered in CTL verification are not unique; it is due to our construction of Γ that the fixpoint is unique.

The next challenge is to answer a membership query asking if (s, i) is in the fixpoint of $\Gamma_{E[f_1 U f_2]}$ or Γ_{EGf} . The following proposition shows that in order to check if some pair (s, i) is in the fixpoint of $\Gamma_{E[f_1 U f_2]}$ (Γ_{EGf}) we only need to check the i -fold composition $\Gamma_{E[f_1 U f_2]}^i(\emptyset)$ ($\Gamma_{EGf}^i(\emptyset)$).

Proposition 12.

1. Let Z be the fixpoint of $\Gamma_{E[f_1 U f_2]}$. Then, for all $i > 0$, $(s, i) \in \Gamma_{E[f_1 U f_2]}^i(\emptyset)$ if and only if $(s, i) \in Z$
2. Let Z be the fixpoint of Γ_{EGf} . Then, for all $i > 0$, $(s, i) \in \Gamma_{EGf}^i(\emptyset)$ if and only if $(s, i) \in Z$

Proof sketch.

1. To show that $(s, i) \in \Gamma_{E[f_1 U f_2]}^i(\emptyset)$ implies $(s, i) \in Z$ we observe that $\Gamma_{E[f_1 U f_2]}$ is \cup -continuous. This gives us $Z = \cup_i \Gamma_{E[f_1 U f_2]}^i(\emptyset)$.

We prove by induction on i that $(s, i) \in Z$ implies $(s, i) \in \Gamma_{E[f_1 U f_2]}^i(\emptyset)$. Base case for $i = 1$ is trivial. Assume the inductive hypothesis up to j . If $(s, j + 1) \in Z$ then since Z is a fixpoint, $(s, j + 1) \in \Gamma_{E[f_1 U f_2]}(Z)$. But then $s \in \llbracket f_1 \rrbracket$ and there must be some $(s', j) \in Z$ such that $s \rightarrow s'$. By the inductive hypothesis, $(s', j) \in \Gamma_{E[f_1 U f_2]}^j(\emptyset)$ which easily leads us to $(s, j + 1) \in \Gamma_{E[f_1 U f_2]}^{j+1}(\emptyset)$.

2. The proof follows the same steps as the case for $\Gamma_{E[f_1 U f_2]}$.

□

For the equivalence query, if Z is the unique fixpoint that we are seeking and Z' is the hypothesis proposed by the learner, in case $Z' \neq Z$, the learner typically also needs an element in the symmetric difference $Z' \oplus Z$ to make progress. We can obtain such an element using the method in Chapter 5 which is summarized for $\Gamma_{E[f_1 U f_2]}(Z')$ ($\Gamma_{EGf}(Z')$ can be done in a similar manner) as follows for the different cases possible.

- $\Gamma_{E[f_1 U f_2]}(Z') \setminus Z' \neq \emptyset$. Let $l = (s, i) \in \Gamma_{E[f_1 U f_2]}(Z') \setminus Z'$. If $l = (s, 1)$ then $l \in Z$, because the only way we can have any $(s, 1)$ in $\Gamma_{E[f_1 U f_2]}(Z')$ is if $(s, 1) \in \Gamma_{E[f_1 U f_2]}(\emptyset)$. In this case, l is in Z and hence in $Z' \oplus Z$. If $l = (s, i)$ for some $i > 1$, we can check if $l \in Z$ using the membership query. If yes, then l is also in $Z' \oplus Z$ and we are done. Otherwise, $l \in \Gamma_{E[f_1 U f_2]}(Z')$ because of the existence of some pair $(s', i - 1) \in Z'$. Clearly $(s', i - 1)$ cannot be in Z otherwise (s, i) would have to be in Z . Hence $(s', i - 1) \in Z' \oplus Z$.
- $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$. From standard fixpoint theory, since Z happens to also be the least fixpoint of $\Gamma_{E[f_1 U f_2]}$, it must be the intersection of all prefixpoints of $\Gamma_{E[f_1 U f_2]}$ (a set Y is a prefixpoint if it *shrinks* under the function \mathcal{F} , *i.e.* $\mathcal{F}(Y) \subseteq Y$). Now, Z' is clearly a prefixpoint. Applying $\Gamma_{E[f_1 U f_2]}$ to both sides of the equation $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$ and using monotonicity of $\Gamma_{E[f_1 U f_2]}$, we get

$$\Gamma_{E[f_1 U f_2]}(\Gamma_{E[f_1 U f_2]}(Z')) \subsetneq \Gamma_{E[f_1 U f_2]}(Z')$$

Thus, $\Gamma_{E[f_1 U f_2]}(Z')$ is also a prefixpoint. Let l be some string in the set $Z' \setminus \Gamma_{E[f_1 U f_2]}(Z')$. Since l is outside the intersection of two prefixpoints, it is not in the least fixpoint Z . Hence, l is in $Z' \oplus Z$.

Finally, once the learning procedure is done, we need to retrieve the set of states satisfying $E[f_1 U f_2]$ (resp. EGf) from fixpoint of $\Gamma_{E[f_1 U f_2]}$ (resp. Γ_{EGf}). The following proposition addresses this.

Proposition 13.

1. Suppose Z be the fixpoint of $\Gamma_{E[f_1 U f_2]}$. Then, $\llbracket E[f_1 U f_2] \rrbracket = \{s \mid \exists i \text{ s.t. } (s, i) \in Z\}$
2. Suppose Z is the fixpoint of Γ_{EGf} . Then, $\llbracket EGf \rrbracket = \{s \mid \forall i > 0 (s, i) \in Z\}$.

Proof sketch.

1. First, we show that if $(s, i) \in Z$ then $s \in \llbracket E[f_1 U f_2] \rrbracket$. We use induction on i . Base case for $i = 1$ is trivial. Next, if $(s, j + 1) \in Z$ then $(s, j + 1) \in \Gamma_{E[f_1 U f_2]}(Z)$. But then $s \in \llbracket f_1 \rrbracket$ and there must exist some $(s', j) \in Z$ such that $s \rightarrow s'$. The inductive hypothesis for j gives $s' \in \llbracket E[f_1 U f_2] \rrbracket$. This means that there is a path from s' to a state satisfying f_2 such that all states in the path satisfy f_1 . Since, $s \rightarrow s'$ and s satisfies f_2 , this path can be extended to start from s . Hence, $s \in \llbracket E[f_1 U f_2] \rrbracket$. For the other direction, we show that if $s \in \llbracket E[f_1 U f_2] \rrbracket$ then for some $i \in \mathbb{N}$, $(s, i) \in Z$. If a state s satisfies $E[f_1 U f_2]$ then there exists a path from s to a state in f_2 such that all the states in this path satisfy f_1 . If the length of the path is i , it can be seen that (s, i) will get included in $\Gamma_{E[f_1 U f_2]}^i(\emptyset)$. Hence, $(s, i) \in Z$.
2. First, we show that if for all $i > 0$, $(s, i) \in Z$ then $s \in \llbracket EGf \rrbracket$. Construct a tree with root s , containing edges appearing in all shortest paths such that all states in this path satisfy f . A few observations about this tree are in order. First, the tree is finite branching; an immediate consequence of the Kripke structure being finite branching. Second, all nodes of the tree satisfy f . Finally, this tree has infinitely many vertices because $(s, i) \in Z$ for all $i > 0$. By König's Lemma, there must be an infinite path in the tree. Clearly, this infinite path witnesses EGf .

For the other direction, we need to show that if $s \in Z$ then for all $i > 0$, $(s, i) \in Z$. If a state s satisfies EGf then there exists an infinite path from s such that all the states in this path satisfy f . If s_i is the i th state on this path (counting s as s_1), it can be seen that (s, i) will get included in $\Gamma_{EGf}^i(\emptyset)$. Hence, for all $i > 0$, $(s, i) \in Z$.

□

Example 2. We illustrate the verification procedure using the system described in Figure 2.1. Suppose we want to verify the CTL property $AG(i_1 + i_2 \leq o)$ which says that in all states reachable from the initial states, the number of items consumed is always less than the number of items generated. This can be written as $\neg EF(\neg(i_1 + i_2 \leq o))$ or $\neg E[true \ U \ (i_1 + i_2 > o)]$. Since there is only a single control state q_0 , we can represent the global state of the system by a four-tuple giving the values of the data variables, $x = (b, o, i_1, i_2)$. Then, we have to calculate the least fixpoint of the function

$$\mathcal{F}(Z) = \{(b, o, i_1, i_2) \mid i_1 + i_2 > o\} \cup \{x \mid x \rightarrow x' \text{ and } x' \in Z\}$$

This is transformed into another function

$$\Gamma(Z) = \{(b, o, i_1, i_2, 1) \mid i_1 + i_2 > o\} \cup \{(x, j + 1) \mid x \rightarrow x' \text{ and } (x', j) \in Z\}$$

Once the fixpoint for Γ is learnt, we can project away the fifth component of the states of the fixpoint to get the states satisfying $EF(i_1 + i_2 > o)$. We complement this set and then check if all the initial states are included in the complement. If the answer is yes, then the system verifies the property otherwise it does not.

8.1.2 CTL with Fairness Constraints

We are now ready to consider the problem of model checking a Kripke structure that has fairness constraints Φ . Evaluating CTL formulas with fairness constraints is known to be harder than the case where there are no fairness constraints. As shown in [39], the problem can be reduced to the following. Let *fair* denote the set of all states s such that there is a fair computation starting from s . It can be shown that $EX(f)$ under fairness conditions is equivalent to $EX(f \wedge \textit{fair})$ without fairness conditions. Similarly, $E[f \ U \ g]$ under fairness conditions is equivalent to $E[f \ U \ g \wedge \textit{fair}]$ without fairness conditions. The set *fair* can be shown to be the states satisfying $EG\textit{true}$ under the fairness constraint. Therefore, if we can evaluate a formula EGf under fairness constraints, we can compute all other CTL formulas using the method in Section 8.1.1.

Let us now look at the learning problem for EGf under a fairness constraint Φ . As described in Section 2.3, EGf means that there exists a path beginning with the current state on which f holds globally and states in Φ are encountered infinitely often on this path. The set of such states Z is the largest set with the following properties: a) all of the states in Z satisfy f , b) for all states $s \in Z$, there is a sequence of states of length one or greater to a state in Z which is also in Φ such that all states on the path satisfy f . This set can be written as the greatest fixpoint of a function $\mathcal{F}(Z) = f \wedge EX \ E[f \ U \ (Z \wedge \Phi)]$ but we cannot directly

use the procedure outlined in the Section 8.1.1 because each application of the function requires evaluating an EU formula which itself needs a fixpoint computation. However, we can adapt a fixpoint characterization we developed in Chapter 7 to EGf .

Definition 12. Let $\Gamma_{EGf}^{fair} : 2^{S \times \mathbb{N} \times \mathbb{N}}$ be a function defined by $\Gamma_{EGf}^{fair}(Z) = (\Gamma_1(Z) \cup \Gamma_2(Z) \cup \Gamma_3(Z)) \cap (\llbracket f \rrbracket \times \mathbb{N} \times \mathbb{N})$ as follows.

$$\begin{aligned} \Gamma_1(Z) &= \{(s, 0, j) \mid s \in \Phi \text{ and } j \in \mathbb{N}\} \\ \Gamma_2(Z) &= \{(s, i, j) \mid s \notin \Phi \text{ and } \exists s'. s \rightarrow s' \text{ and } \exists j' < j. (s', i, j') \in Z\} \\ \Gamma_3(Z) &= \{(s, i, j) \mid s \in \Phi \text{ and } \exists s'. s \rightarrow s' \text{ and } \exists j' < j. (s', i - 1, j') \in Z\} \end{aligned}$$

Intuitively, we associate two counters with each state. Let Z_{EGf} be the fixpoint of Γ_{EGf}^{fair} . A triple (s, i, j) in Z_{EGf} means that there exists a path of length j starting from s which encounters at least $i + 1$ states labeled with Φ and all states in this path satisfy f . Since this can be checked in finite time, we have a method of answering membership queries.

Proposition 14. Γ_{EGf}^{fair} has a unique fixpoint (Z_{EGf}). Further, the set of states satisfying EGf is given by

$$\{s \mid \forall i. \exists j. (s, i, j) \in Z_{EGf}\}$$

Proof. The proof follows the same steps as in Proposition 6. □

The uniqueness of the fixpoint allows equivalence queries to be answered as before. The proposition also gives us a way to compute EGf from Z_{EGf} .

To recapitulate, we have developed fixpoint characterizations for all CTL operators with or without fairness such that each such fixpoint can be computed using a learning procedure. This allows us to model check any CTL formula by starting from the innermost sub-expressions and finally checking to see if all initial states are in the states for the outermost expression.

The overall verification procedure is depicted in Figure 8.1.

8.2 Representing States with Regular Sets

In the previous section, we presented a general set of conditions under which we can use a learning based approach to verify CTL properties of systems. In this section, we give details of how this can be achieved within the context of using regular languages to represent sets of states.

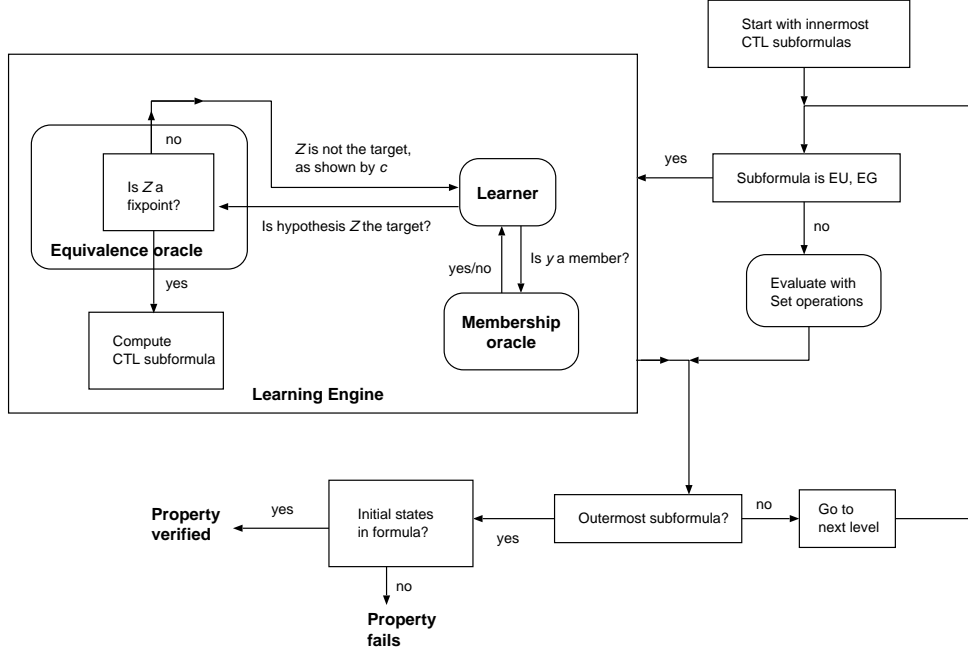


Figure 8.1: Verification procedure for CTL

We assume that the states of the system can be encoded as strings over some finite alphabet ρ . We further assume that the *enabled* and *action* pairs of the events the program can be used to create a transducer representing the transition relation of the Kripke structure. Recall that the transducer is a finite state machine which accepts a pair of strings (s_1, s_2) if the state corresponding to s_1 can transition to the state corresponding to s_2 . We assume that the set of initial states; the set of states with labeled with a atomic proposition; and the fairness constraint Φ are all given as regular sets.

8.2.1 Representation of States with Counter

In general, a set Z of pairs (s, i) is a subset of $\rho^* \times \mathbb{N}$. To encode Z as a regular set we use the alphabet Σ given by $(\rho \cup \{\perp\}) \times \{0, 1\}$. This is the alphabet that will be used by the learning algorithm. Here \perp is a new “filler” symbol. An element (s, i) is encoded as string over Σ such that projecting the symbols on the first component gives us s (the \perp symbols are ignored); and projecting on the second component gives i in binary notation. A similar encoding can be used to represent sets of triples (s, i, j) .

8.2.2 Symbolic Computation of Operators

The various operations required for verification can be done efficiently using regular sets. Standard procedures are available for complementation and union of regular sets represented by finite automata. Let T_{inv} represent a transducer whose relation is the inverse of the Kripke relation. Given a DFA M_Z representing a set of states Z , the set of states $EX(Z)$ can be found using the construction in Section 6.1.1 as $T_{inv}(M_Z)$.

Next, we discuss how to compute the image of a regular set of states under the function $\Gamma_{E[f_1 U f_2]}$ (the case for Γ_{EGf} and Γ_{EGf}^{fair} can be handled similarly).

Definition 13. Given Z a set of strings in the alphabet of Σ , define

$$\begin{aligned} Inc(Z) &= \{(s, i) \mid (s, i - 1) \in Z\} \\ Dec(Z) &= \{(s, i) \mid (s, i + 1) \in Z\} \end{aligned}$$

A transducer for computing $Inc(Z)$ can be constructed using a method similar to the one described in Section 6.5.1 for checking safety properties of systems in regular model checking framework. The transducer construction for $Dec(Z)$ is similar.

Given a DFA M_Z representing a set of states Z , $\Gamma_{E[f_1 U f_2]}(Z)$ is found by computing $T_{Inc}(T_{inv}(M_Z))$, intersecting the resulting regular set with the regular set $\llbracket f_1 \rrbracket \times \mathbb{N}$ and finally applying union with the regular set $\llbracket f_2 \rrbracket \times \{1\}$.

Let $Proj_1$ denote the projection to the first component, $Proj_2$ to the second component and $Proj_{1,2}$ to the first and second component (when there are more than two components). These projections can be done on regular sets using homomorphisms. For example, $Proj_1$ can be done by a homomorphism $h : \Sigma \mapsto \rho^*$ which takes each letter in Σ and maps it to a letter in ρ which corresponds to the state component.

Finally, given a regular set Z we need a way to calculate the following.

- $\{s \mid \exists i \text{ s.t. } (s, i) \in Z\}$ (needed for EU subformulas): This can be calculated as $Proj_1(Z)$.
- $\{s \mid \forall i > 0 (s, i) \in Z\}$ (needed for EG subformulas without fairness): This is the complement of the set $\{s \mid \exists i. (s, i) \notin Dec(Z)\}$. Thus, the desired set is $\overline{Proj_1(Dec(Z))}$.
- $\{s \mid \forall i. \exists j. (s, i, j) \in Z\}$ (needed for EG subformulas with fairness): This can be written as

$$\{s \mid \neg(\exists i. \neg(\exists j. (s, i, j) \in Z))\}$$

or equivalently, $\overline{Proj_1(Proj_{1,2}(Z))}$.

Figure 8.2 outlines the procedure that can be used to compute an EU CTL subformula. The procedure for EG subformulas is similar.

8.2.3 Soundness and Completeness

The learning based verification procedure is always sound since it computes all CTL subformulas exactly. Further, we have the following completeness result.

Theorem 6. *Given a CTL property to verify, assume that the following conditions hold:*

1. *Given a Kripke structure K , for every subformula of the form $E[f_1 U f_2]$ or EGf , the fixpoints of $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf} have a regular representation*
2. *Given a fair Kripke structure K , for every subformula of the form $E[f_1 U f_2]$ or EGf , the fixpoints of $\Gamma_{E[f_1 U f_2]}$ and Γ_{EGf}^{fair} have a regular representation*

Then the learning-based verification procedure will always terminate and correctly infer whether the system satisfies the given CTL property.

Remark 1. Note that the set of states for the subformulas for other CTL operators \neg , \vee and EX always have regular representation since they can be obtained using standard automata operations.

8.3 Complexity Analysis

The main cost of the verification procedure is learning the fixpoints. Let f be the number of CTL subformulas requiring fixpoint computations. We now analyze the cost of the learning procedure. This analysis is similar to the one done in Section 7.3.5 for ω -regular specifications.

Let m be the length of the longest string returned by the teacher in a negative answer to an equivalence query, n be the number of states of the minimal automaton representing a fixpoint, k be the size of the alphabet of the learned language and t be the number of states of the automaton representing the transducer for the function Γ whose fixpoint is being learned. By Proposition 1, the language inference algorithm makes $O(kn^2 + n \log m)$ membership queries and $O(n)$ equivalence queries. The worst case for the equivalence query for a hypothesis Y occurs when we look for a string in the difference of Y and $\Gamma(Y)$. The size of DFA representing Y is bounded by n . Looking at $\Gamma(Y)$, it can be seen that the DFA representing the difference of Y and $\Gamma(Y)$ would be $O(nt)$. Thus the length of the longest string returned by an equivalence query is $m = O(nt)$.

The cost of answering membership queries dominates the total runtime cost of the procedure. Using $m = O(nt)$, the number of membership queries is $O(kn^2 + n \log nt)$. The cost of the membership queries is equal to the number of membership queries and the cost of building the DFA D_j representing $\Gamma^j(\emptyset)$. The cost for D_j is $(O(t))^j$ which leads to the total cost of membership queries of $O(t^{O(nt)} + kn^2 + n \log nt)$ (using maximum value of j to be $m = O(nt)$).

Hence, the overall running time is $O((t^{O(nt)} + kn^2 + n \log nt)f)$.

```

algorithm learner
begin
Regular inference algorithm
end

algorithm isMember
Input:  $(s, i)$ 
Output: is  $(s, i)$  in fixpoint?
begin
  Compute  $Z_i = \Gamma_{E[f_1 U f_2]}^i(\emptyset)$  if not done already
  Is  $(s, i) \in Z_i$ ?
  If yes return true
  else return false
end

algorithm Equivalence Check
Input: Hypothesis  $Z'$ 
Output: For fixpoint  $Z$ , is  $Z' = Z$ ?
If not, then some string in  $Z' \oplus Z$ 
begin
  If  $\Gamma_{E[f_1 U f_2]}(Z') \setminus Z' \neq \emptyset$  {fixpoint check}
  let  $(s, i) \in \Gamma_{E[f_1 U f_2]}(Z') \setminus Z'$ 
  Find  $(s', i')$  which causes  $(s, i)$  to be in  $\Gamma_{E[f_1 U f_2]}(Z')$ 
  if isMember $((s, i))$ 
    return (no,  $(s, i)$ )
  else
    return (no,  $(s', i')$ )
  else if  $\Gamma_{E[f_1 U f_2]}(Z') \subsetneq Z'$ 
    return (no,  $l \in (Z' \setminus \Gamma_{E[f_1 U f_2]}(Z'))$ )
  else {found fixpoint}
    CTL subformula =  $Proj_1(Z')$ 
end

```

Figure 8.2: Learning to verify $E[p_1 U p_2]$ with regular sets

8.4 Comparison with Related Work

Most previous approaches for verification of infinite state systems have either focused on safety properties or on linear time logic properties. To our knowledge, the only other major work that has addressed CTL properties (using techniques different from learning) is that of Bultan *et al.* [32] where a combination of widening and finite iteration are used for *conservative approximations*. The notable differences between their work and our work are as follows. First, we also are able to handle fairness constraints which are needed for liveness properties. Second, we never return a “do not know” answer which a conservative analysis can return. Finally, as pointed out before, the main advantage of the learning-based verification procedure is that as long as the fixpoints corresponding to the subformulas of the CTL formula are regular, the verification procedure is guaranteed to terminate.

Chapter 9

Implementation and Results

The learning based verification techniques presented in the previous chapters have been implemented in a tool suite called LEVER which is available for download from [91]. In this chapter, we discuss the details of the implementation of the tool suite and present some examples that we have analyzed. We also compare the performance of LEVER with other tools that are available. The results indicate that no tool is a clear winner for all of the examples and every tool outperforms the rest on some of the examples. We conclude the chapter with a case study of verification of a module called *read-copy-update* which is used as a synchronization method in the Linux kernel.

9.1 Overview of Lever

The LEVER suite currently provides the following main tools:

1. Tool for verifying safety properties of FIFO automata through either passive or active learning.
2. Tool for verifying ω -regular and CTL properties of integer systems. These systems include programs with positive integer variables and parameterized systems in which we only need to keep track of the number of processes in any state.

The primary motivation for using two different tools rather than a single one is to allow domain specific optimizations for FIFO automata and integer and parameterized systems and to enable using an automata library that is better suited for encoding the states in each system. The tool for FIFO automata uses the BRICS automata package [102] in which the states and transitions are kept explicitly. On the other hand, for integer and parameterized systems, we use MONA [83] since it provides a very efficient library for automata based on representing the transition relation as Binary Decision Diagrams. The preferred programming language of choice was Java and this is the one that was used for FIFO automata. However, for integer and parameterized systems, we used C++, since MONA is implemented in C which can be easily interfaced with C++ but has limited interfacing capabilities with Java (using the Java Native Interface). Figure 9.1 shows the high level architecture for the both tools.

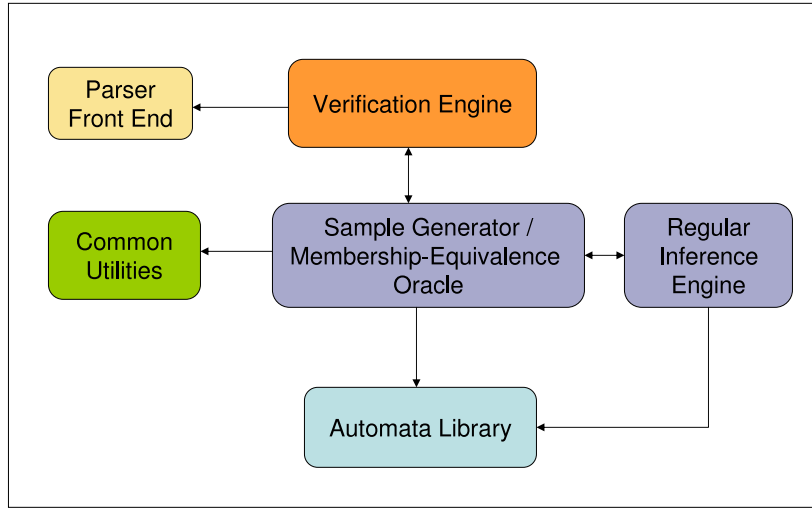


Figure 9.1: High level architecture of tools in LEVER

The *parser front end* takes the input program consisting of variable declarations, transition guards and actions, initial states, the states labeled with the atomic propositions and the property to be verified. The *verification engine* sets up either the membership and equivalence oracles for active learning or a generator of positive and negative samples for passive learning. The regular inference algorithm uses the oracles or the samples to learn the regular set which is provided back to the verification engine. The verification engine analyzes the learnt set and uses this to either learn another fixpoint in case of nested CTL formulas or verify whether the desired property holds.

9.1.1 Representation of States

For FIFO systems, instead of learning the reachable states directly, we learn a set of annotated traces as described in Chapters 4 and 5. Recall that the set of annotated traces is a language over the alphabet consisting of labels of the transitions in the FIFO automata and some additional symbols which allow the reachable states to be computed for any regular set of annotated traces.

For integer and parameterized systems, a vector (x_1, x_2, \dots, x_n) in \mathbb{N}^n is encoded as a string s over an alphabet $\Sigma = \{0, 1\}^n$. For a letter a in Σ , let a^i stand for the i -th component in a . Let the string s be $a_1 a_2 \dots a_m$. Then the i -th component x_i of the vector (x_1, x_2, \dots, x_n) is represented in binary by $a_1^i a_2^i \dots a_m^i$. In other words, $x_i = \sum_{j=1 \dots m} a_j^i 2^{j-1}$. This representation is essentially the same as Boigelot's Number Decision Diagrams (NDD) [21] (in NDD, for base 2, the alphabet is $\{0, 1\}$ and for a string $b_1 b_2 \dots b_{nm}$ denoting a vector $x \in \mathbb{N}^n$, the component x_i is given in binary by $b_i b_{i+m} \dots b_{i+n(m-1)}$). It is known that any Presburger set can be encoded as an automata in this representation. The additional counters used in the verification of ω -regular or CTL properties are encoded by adding integer variables which are initialized

to zero and incremented according to the procedures mentioned in Chapters 7 and 8. For efficiency, we first calculate the set of reachable states and restrict all intermediate states calculated during fixpoint computations to be reachable.

As mentioned before, for integer and parameterized systems, we use the automata library from MONA [83]. MONA keeps the states of the automata explicitly but the transition relation is encoded as a multi-terminal shared BDD [29]. This allows a compact representation of the automata even when the alphabet is of large size. The transducer representing the transition relation is also encoded as an automaton in MONA with a set of new variables added to represent the values taken by system variables after application of a transition.

9.2 Regular Inference Algorithm

Recall from Chapter 4 that the passive learning algorithm uses a variant of RPNI (Regular Positive and Negative Inference). One important optimization in the algorithm is to record the changes done during a merge of states in an automaton and roll back these changes if the merge is not accepted. This obviates the need to make a copy of the entire automaton to be used if the merge on the original automaton is not accepted. Since most merges are rejected, this improves the performance significantly.

For active learning, we use a modified version of the regular inference algorithm described in [82]. As mentioned in Section 2.5.3, one major change in the original algorithm is to use the idea of analyzing counterexamples in a binary-search manner as described in Rivest *et al.* [114].

9.2.1 Scalability Issues

In the active learning algorithm, the run-time cost is linear in the alphabet size of the language being learned. However, a problem is that the alphabet size itself grows exponentially in the number of variables in the system. We now discuss some of the techniques we use to manage this cost.

In Section 2.5.3 we described the details of the learning algorithm used for regular sets. We recall that when the learner introduces a new state in its hypothesis, it has to make membership queries to find the transition function from the new state. Let the access string for the new state being formed be s . For each symbol b , to find the destination for b -transition out of s , we have to sift sb down the classification tree. This requires membership queries for strings in $s.b.E$. We first compute a DFA D^k representing $\Gamma(\emptyset) \cup \Gamma^2(\emptyset) \cup \dots \cup \Gamma^k(\emptyset)$ (where Γ is the function whose fixpoint is being computed) up to a sufficient depth k such that all the membership queries for $s.\Sigma.E$ can be answered by looking up acceptance in D^k . A naive implementation would now need $O(|\Sigma|)$ membership queries; since $|\Sigma|$ is exponential in the number

of variables, this is an extremely high cost. To mitigate this, we can use a symbolic learning algorithm that symbolically sifts $s.\Sigma$ for the entire alphabet at one go, rather than individually sifting sb for each $b \in \Sigma$. We present its details next.

Let the automaton D^k have states Q_A and transition function δ_A , and let $q_s \in Q_A$ be the state of D^k reached on reading s . Consider b_1 and b_2 such that $\delta_A(q_s, b_1) = \delta_A(q_s, b_2)$. From this, we know that for every $d \in E$, $\delta_A(q_s, b_1.d) = \delta_A(q_s, b_2.d)$. Since D^k was chosen to be such that all membership queries for strings in $s.\Sigma.E$ can be faithfully answered by D^k , it follows that for every $d \in E$, $s.b_1.d$ belongs to the concept iff $s.b_2.d$ belongs to the concept. Thus, the strings $s.b_1$ and $s.b_2$ will sift in the same way in the classification. One important consequence of this observation is that one need not sift all strings in $s.\Sigma$; we only need to sift strings that go to different states in D^k , which can be significantly less than that number of symbols in Σ . Furthermore, if the transition relation of all our DFAs are represented using ordered BDDs, then all this computation can be done implicitly and efficiently.

9.3 Input Syntax

For FIFO automata, the input program declares the number of channels and the alphabet used for the messages (the alphabet is restricted to be a value from 0 to some fixed number). This is followed by an enumeration of the control states. Next, a list of transitions is given; each transition specifies the start control state and the destination control state along with the action to be taken on the transition. Figure 9.2 shows an abstraction of the Alternating Bit Protocol expressed in this syntax.

The input syntax for integer and parameterized programs is similar to that used by FAST [54]. The easiest way to understand the syntax is via an example listed in Figure 9.3. This example demonstrates a simplified version of the producer consumer problem. It has three variable `size`, `spaceLeft`, `produced` which correspond to the size of a buffer, the space left in the buffer and items produced so far respectively. The domain of each variable is the set of natural numbers. There is only one control state in the system called `normal`. The transition `prod` has a guard which checks to see if space is available in the buffer and if so, adds an item while decrementing `spaceLeft`. On the other hand, the transition `cons`, checks to see if there are items in the buffer and if so, decrements `produced` while incrementing `spaceLeft`. The region `init` specifies the initial states of the system. In this case, we are interested in checking a safety property so we also specify a set of *bad* states. Constraints on states, guards and actions are all expressed in Presburger arithmetic.

```

// Alternating Bit Protocol

numChannels = 2; // number of channels in the FIFO automaton
maxLetter = 1;  // value of the maximum letter in the FIFO alphabet

// Control States
{
// sij denotes sender's ith state and receiver's jth
// The first state is the initial state
s00; s10; s01; s11;
}

// Transitions
{
// Sender's transitions
s00 -> (0!0) s00;
s00 -> (1?1) s00;
s00 -> (1?0) s10;
s10 -> (0!1) s10;
s10 -> (1?0) s10;
s10 -> (1?1) s00;

s01 -> (0!0) s01;
s01 -> (1?1) s01;
s01 -> (1?0) s11;
s11 -> (0!1) s11;
s11 -> (1?0) s11;
s11 -> (1?1) s01;

// Receiver's transitions
s00 -> (1!1) s00;
s00 -> (0?1) s00;
s00 -> (0?0) s01;
s01 -> (1!0) s01;
s01 -> (0?0) s01;
s01 -> (0?1) s00;

s10 -> (1!1) s10;
s10 -> (0?1) s10;
s10 -> (0?0) s11;
s11 -> (1!0) s11;
s11 -> (0?0) s11;
s11 -> (0?1) s10;
}

```

Figure 9.2: Example input file for LEVER for FIFO automata

```

model producer_consumer {

    var    size, spaceLeft, produced;

    states normal;

    transition prod := {
        from := normal;
        to := normal;
        guard := spaceLeft >= 1;
        action := produced' = produced + 1, spaceLeft' = spaceLeft - 1;
    };

    transition cons := {
        from := normal;
        to := normal;
        guard := produced >= 1;
        action := produced' = produced - 1, spaceLeft' = spaceLeft + 1;
    };

}

strategy s1 {

    Region init := {state=normal && size = spaceLeft && produced = 0};
    Transitions t := {prod, cons};

    Region bad := {!(spaceLeft + produced = size)};
}

```

Figure 9.3: Example input file for LEVER for integer systems

9.3.1 Experiments and Results for FIFO automata

We have used LEVER to analyze some canonical FIFO automata verification problems described below.

[Producer Consumer] A simple producer consumer problem with one FIFO channel. The producer can either be in an “idle” or in a “send” state in which it transmits either 0 or 1 to the FIFO channel.

[Data with parity] A simple data communication protocol in which the sender sends data and a parity bit for the number of 1’s sent. The receiver uses the parity bit as a simple check for data integrity.

[Resource arbitrator] In this example, two senders wish to broadcast over a shared channel and use a resource manager to arbitrate which one is allowed to use it at any time.

[Alternating bit protocol (ABP)] This consists of a sender and receiver communicating over a data and an acknowledgment channel. We consider a non-lossy version of ABP.

[Sliding window protocol] This is similar to ABP except that the sender can keep multiple data messages in flight. We use a window size of 2 and maximum sequence number also of 2.

“Producer Consumer,” “Alternating bit protocol” and “Sliding window protocol” are fairly well-known in the FIFO research community, see for example [120]. For the other two systems, a detailed description is given below.

Data with parity

The system consists of a sender and a receiver whose automata are shown in Figure 9.4. The sender attaches a “parity” bit if the number of 1’s is not even which is verified by the receiver. The FIFO system is simply a cross product of the two automata with the bad states being any state in which the receiver is at the control state q_b .

Resource arbitrator

The system consists of two senders which share a channel c_0 for broadcasting data. In order to ensure that only one sender transmits at a time, there is a resource manager to which a request has to be sent for the use of channel c_0 . The resource manager grants the request to one sender and waits for that sender to indicate that it is done before listening for more requests. For simplicity, we assume that the first sender transmits a pattern $(01)^*$ on c_0 while the second sender transmits 2^* . The safety property we want to verify that in any control state, the channel c_0 does not have transmissions from the two senders mixed up, *i.e.*, for any

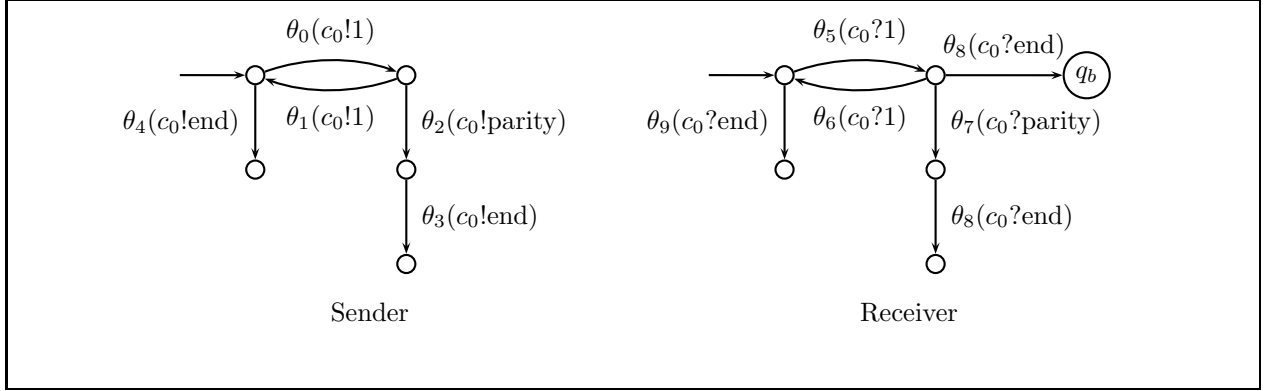


Figure 9.4: FIFO automata for data transmission with parity

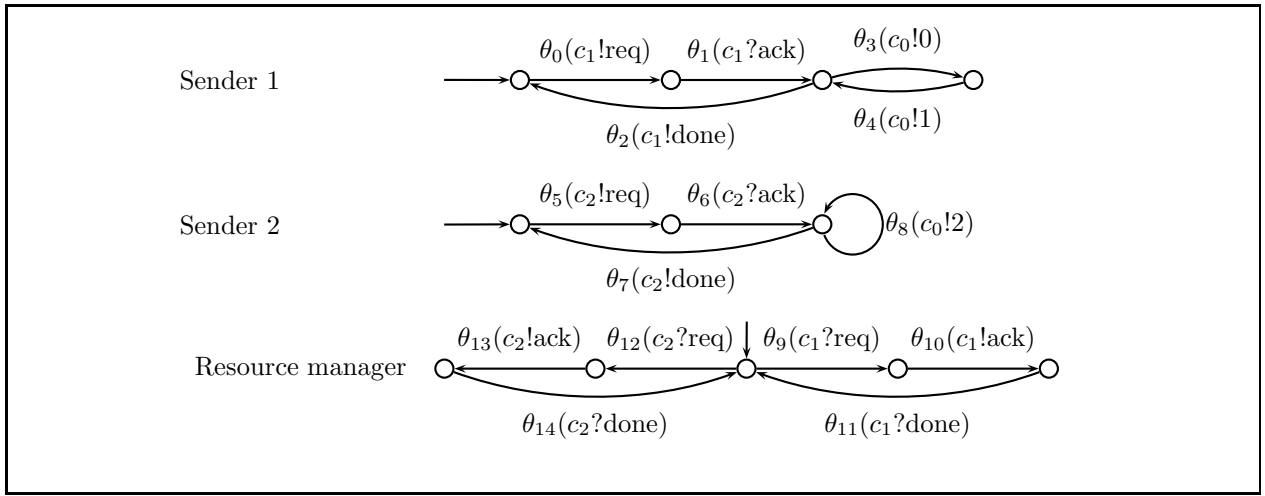


Figure 9.5: FIFO automata for resource arbitrator

control state, contents of c_0 other than $((01)^*|2^*)^*$ is considered “unsafe”. Figure 9.5 shows the description of the senders and the resource manager; the automaton to be analyzed is the cross product of all these three automata. It can be seen that the annotated trace language $AL(F)$ for this automaton is regular.

Results

We compare the running times (T) using the verification procedures using active learning and passive learning (columns $\text{Size}_{\text{passive}}$ and T_{passive}). All executions were done on a 1594 MHz notebook computer with 512 MB of RAM using Java virtual machine version 1.4.1 from Sun Microsystems. We also report the time taken (T_{rnc}) by the regular model checking tool [106] on the same examples. It can be seen the running time of LEVER is slightly better than the regular model checking tool.

In most of the examples, the active learning approach is faster. However, in certain cases it is possible that the passive approach outperforms the active one. For instance, for the FIFO automata for *Data Parity*

	T_{active}	T_{passive}	T_{rmc}
Producer Consumer	0.3	0.4	3.3
Data Parity	305.0	0.5	12.7
Resource Arbiter	0.5	0.7	33.2
Alternating Bit	2.0	4.1	24.7
Sliding Window	54.0	81.2	78.4

Table 9.1: Running time in seconds for safety property verification of FIFO automata

given in Figure 9.4, the time taken by the active learning approach is much higher. The intuitive reason why the passive learning approach is able to find the answer quickly in this case is that the characteristic sample needed for the reachable state happens to be quite small and is found using relatively a small number of traces executed by the system. On the other hand, due to the cross product of two FIFO automata involved in the system, the number of transitions that are enabled at any state is large. Most of these transitions can be executed independently of each other and reach the same state regardless of order. The active learning approach has to reconstruct each possible order causing it to slow down.

9.4 Experiments and Results for Integer and Parameterized systems

For integer and parameterized systems, we analyze cache coherence protocols such as *Dragon*, *Firefly*, *Illinois*, *MESI*, *MOESI*, *Berkeley*, *Futurebus* and *Synapse*; mutual exclusion protocols such as *peterson*, *lamport*, *ticket* and *bakery*; broadcast protocols such as *consistency*, and *producer-consumer*; petri nets such as *lastin-firstserved protocol*, *Esparza-Finkel-Mayr Counter Machine*, *RTP* and *manufacturing*; and counter machines such as *lift* and *barber*. A number of these examples are taken from the FAST [54] web site. We also analyze an example called *noaccel* for which the reachability set is regular but on which the *acceleration* methods employed in tools such as FAST cannot be applied because the transition relation does not satisfy the conditions needed for *acceleration*. Another system called *flatcounter* is derived from an example given in [90] which illustrates a counter machine for which the reachable set is regular but the machine is not *flat*. It is known that the acceleration method does not terminate for systems which are not *flat*. Note that for most of these examples, the safety property of interest holds; but this is not very surprising since these examples are well-studied in the literature. All these examples are available for download along with the LEVER tool.

We analyze three different properties:

1. A simple safety property which says that the property p holds invariantly.

2. A CTL property $AG(EFp)$ which asserts that it is always true in all states that it is possible to get to a state in future which satisfies p . This is often used to assert properties such as *proper termination*. Note that this is a branching time property that cannot be expressed in linear time logics.
3. Another CTL property $AG(\text{req} \rightarrow AF\text{resp})$. This illustrates the application to a liveness property which also requires fairness constraints. The fairness constraints were manually added to the examples taken from the literature but due to the significant work involved, this was done only for some of the more well known examples. Hence, $AG(\text{req} \rightarrow AF\text{resp})$ was not analyzed for all the examples. This property along with the fairness constraints can also be expressed as an ω -regular specification.

In order to evaluate the performance of our tool, we compare our LEVER tool with three other tools popular for verifying safety properties of infinite state systems: FAST [54], BRAIN [115] and ALV [10]. FAST uses acceleration techniques to compute the effect of infinite iteration of certain loops. BRAIN does a backward search from the “unsafe states” and uses Hilbert’s bases for symbolic representation of integer sets. Finally, ALV uses *widening* and can also employ acceleration techniques. In the case of LEVER, we explore two options: one in which learning uses the bound on the space as witness (see Section 6.2.1) and the other in which the witness is the number of steps taken to reach a state (see Section 6.2.2). The former option is labeled as “LEVER (space)” while the latter is labeled as “LEVER (steps)”. For the ALV tool, the default backward search strategy was used and the computation of loop closures was turned on using the option ‘-L’.

9.4.1 Discussion

The running time for analyzing safety properties using LEVER, FAST, BRAIN and ALV shown in Table 9.2. All analysis was done on Intel Xeon based Linux machine running at 1.70GHz with 1GB memory. For some examples, the analysis could not be completed either because the tool did not terminate in two hours, or it exhausted available memory, or (in the case of ALV) it reported that it cannot provide an answer. For these cases, the table shows an entry of \uparrow .

The comparison of running times between LEVER, FAST, BRAIN and ALV is summarized in Figure 9.6. The y-axis shows the logarithm (base 10) of the time in milliseconds to analyze the given examples. For cases where a tool is unable to give an answer, we assign a value of 10^{10} milliseconds as the time taken.

It is interesting to note that since LEVER only relies on regularity of the set being learnt, it is able to analyze the example *noaccel* trivially, while FAST is unable to make any headway since the transition relation does not lend itself to *acceleration* based methods. Similarly, for the system *flatcounter*, LEVER

	LEVER(space)	LEVER(steps)	FAST	BRAIN	ALV
noaccel	0.03	0.04	↑	0.004	0.03
synapse	0.1	0.11	0.22	0.004	0.06
flatcounter	0.15	↑	↑	0.004	0.05
efm	0.17	0.26	0.59	0.01	0.2
berkeley	0.26	0.27	0.27	0.004	0.07
mesi	0.32	0.28	0.34	0.004	0.09
manufacturing	0.82	0.94	2.42	10.97	↑
moesi	0.5	1.23	0.42	0.004	0.28
rtp	0.52	1.02	1.6	0.02	3.37
barber	0.55	1.78	1.39	0.06	0.64
ticket2i	0.59	3.81	0.68	↑	↑
lamport	0.74	1.42	1.78	0.05	6.8
consistency	0.93	6.07	142.81	0.06	571.47
dragon	0.96	1.26	1.07	0.01	0.66
firefly	1.2	0.85	0.65	0.01	0.11
peterson	1.37	2.82	3.4	0.25	441.71
illinois	1.6	1.24	0.71	0.01	0.17
lift	2.73	7.45	4.12	↑	0.4
bakery2	2.74	6.22	↑	0.01	0.09
producer-consumer	3.5	20.99	0.32	↑	0.04
kanban	3.95	9.61	7.08	↑	↑
readwrit	5.05	9.76	7.07	0.06	1104.65
dekker	12.17	54.19	14.41	21.29	↑
ticket3i	12.62	118.46	2.78	↑	↑
futurbus	14.49	11.18	1.58	0.03	2.32
centralserver	17.09	6.73	14.18	0.05	18.48
csm	85.93	167.36	31.29	0.21	1428.52
lastinfirstserved	126.35	13.91	1.44	0.01	0.45
multipoll	4400	1513	11.2	287.88	↑
train	↑	↑	5.98	0.02	0.27
fms	↑	↑	112.5	410.92	↑
swimmingpool	↑	↑	116.41	0.01	↑

Table 9.2: Running times in seconds for safety properties for integer programs

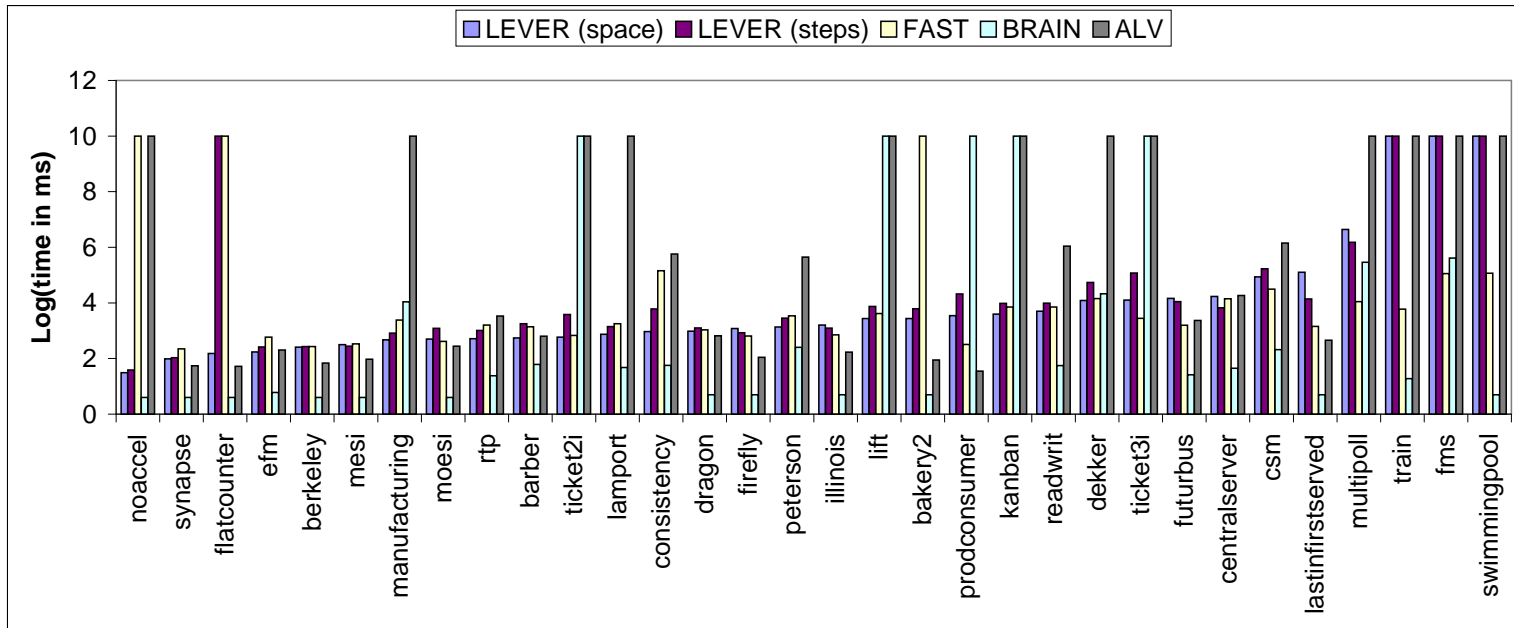


Figure 9.6: Comparison between LEVER and other tools for safety properties

Example	$AG(EFp)$
dragon	2.70
firefly	1.82
illinois	3.81
mesi	1.63
moesi	1.84
prod/consumer	6.99
synapse	0.56
bakery2	17.00
barber	1.01
berkeley	0.61
consistency	5.04
efm	0.19
futurbus	20.95
lamport	2.62
lastinfirstserved	132.72
lift	6.93
manufacturing	3.41
noaccel	0.09
peterson w	2.35
rtp	15.34
ticket2i	12.13

Table 9.3: Running times in seconds for LEVER for CTL formula $AG(EFp)$

successfully analyses with witness using bounded space; however FAST is unable to terminate as the system is not flat.

The overall comparison of the performance of the various tools is mixed. There are some examples for which one tool is better than all the others and there are other examples in which some tools are unable to give an answer while others are successful. In some cases, LEVER is unable to terminate in two hours. However, the important observation is that the performance of LEVER is comparable to the other tools and for some examples it is significantly better. Another significant advantage of using LEVER is that, given enough time and memory, the learning based technique gives is guaranteed to terminate as long as the set being learned is regular.

Tables 9.3 and 9.4 report the running times for the CTL formulas $AG(EFp)$ and $AG(\text{req} \rightarrow AF\text{resp})$ respectively. Comparison with other tools could not done for these formulas, since most of them are restricted to analysis of safety properties.

9.5 Case Study

We now present a case study for the application of learning techniques for verification. The system we analyze is an abstract model derived from a module in the Linux kernel. Linux is a popular open source operating

Example	$AG(\text{req} \rightarrow AF\text{resp})$
bakery2	29.28
barber	3.40
consistency	2.36
lamport	5.24
lift	9.78
ticket2i	45.92

Table 9.4: Running times in seconds for CTL formula $AG(\text{req} \rightarrow AF\text{resp})$ with fairness constraint

system originally developed by Linus Trovaldis and has been used extensively for desktop, embedded and enterprise environments.

The module we analyze in Linux is the *read-copy-update (RCU)* mechanism designed as a scalable reader-writer synchronization mechanism for data structures that are accessed concurrently on multiple CPU systems. The RCU work is part of the *Linux Scalability Effort* [93] which aims to scale Linux to systems with large processor counts and IO configurations. A major challenge in multi processor systems is to design efficient mechanisms for mutual exclusion for critical sections of code. The traditional methods for mutual exclusion use spin locks, semaphores, reader-writer locks and so on. However, with the advent of faster CPUs and the fact that memory interconnects are not advancing as rapidly, the relative cost of acquiring locks is increasing. Therefore, a large effort has been put into looking at alternatives to conventional locking methods. *Read-copy-update* [14, 96, 97] is one such alternative which is designed for the case where most of the accesses are by readers (who do not modify data) and only occasional accesses by writers (who modify data). Such scenarios are quite common in modern operating systems, for example, looking up *routing tables* for IP network layer and accessing *file structures*.

RCU is a two-phase update method, where readers can access the data without any conventional locks, but writers have to use a special callback scheme to update the data. Writers update all the global references to the changed data with a new copy and use the callback scheme to free the old copy after all the CPUs have lost local references to it by going through a quiescent state (like a context switch). To explain the RCU mechanism, let us take a concrete example of list updates from [98]. Suppose that there are two CPUs accessing a list which is initially of three elements as shown in Figure 9.7. CPU 0 is updating element B while CPU 1 is doing a lock-free read traversal.

Now, suppose that CPU 0 needs to make change to element B. It cannot modify B in place because CPU 1 may be reading it at that time. Therefore, CPU 0 makes a copy into a new element B', modifies B' and changes A's pointer to B'. This does not interfere with CPU 1 which still has the old element B. Next CPU 0 waits for every CPU to go through a *quiescent state* (a state where the CPU gives up any references to

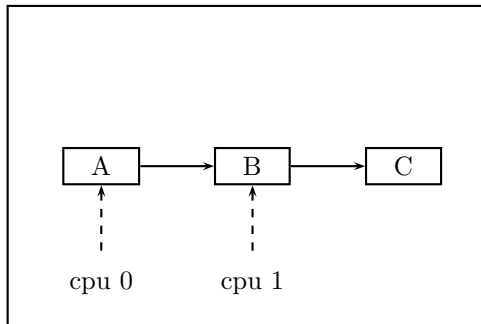


Figure 9.7: List initial state

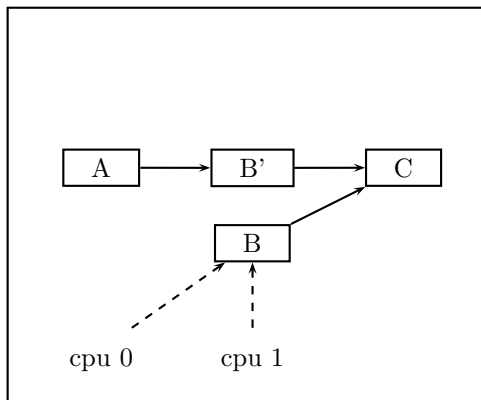


Figure 9.8: List deferred deletion

shared data structures) and then deletes B. Since, CPU 1 has gone through a quiescent state, it cannot have any stale reference to B. At this time, CPU 0 proceeds to free B and the final list is shown in Figure 9.10.

9.5.1 RCU in Linux kernel

We analyzed the RCU implementation in Linux kernel version 2.6.12.5 (obtained from <http://www.kernel.org>). The RCU application programming interface is shown in Figure 9.11. The functions `rcu_read_lock` and `rcu_read_unlock` are used by the readers and are extremely light weight. In fact, for non-preemptive kernel they do not generate any code at all; they are used only in preemptive kernels because the current implementation uses a context switch as a quiescent state which is valid only if there is no preemption. The function `call_rcu` is used by the writer to schedule a call back to update the change that it needs after every CPU has gone through a quiescent state. The `call_rcu` function uses its `struct rcu_head` argument to remember the call back function. The function `synchronize_rcu` blocks until all CPUs have gone through a quiescent state. There are similar functions `rcu_read_lock_bh`, `rcu_read_unlock_bh` and `call_rcu_bh` which are dedicated to kernel bottom half handling but since the overall algorithm is the same, we do not discuss

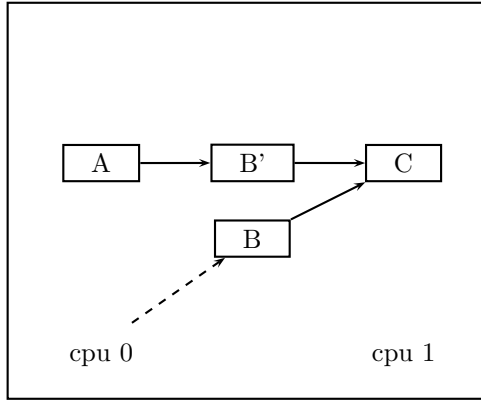


Figure 9.9: List after quiescent period

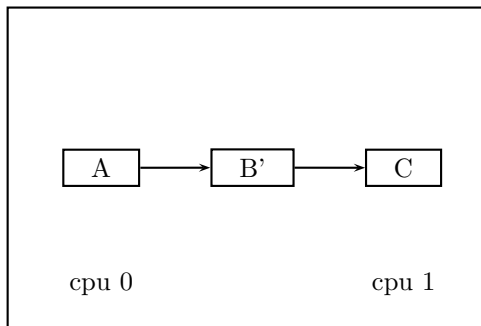


Figure 9.10: List after deletion

```

struct rcu_head {
    struct rcu_head *next;
    void (*func)(struct rcu_head *head);
};

extern void call_rcu(struct rcu_head *head,
                    void (*func)(struct rcu_head *head));
extern void synchronize_rcu(void);

void rcu_read_lock(void);
void rcu_read_unlock(void);

```

Figure 9.11: RCU API in Linux Kernel

them separately.

The overall flow of the RCU mechanism proceeds as follows:

1. Readers access data between calls to `rcu_read_lock` and `rcu_read_unlock` but as we saw this is light weight.
2. Writers do not modify data in place; they make a copy and schedule a call back function to do the update via `call_rcu`. The calls get queued in a per CPU queue.
3. If a CPU has pending updates, it informs all other CPUs of the beginning of a *quiescent period* (a time interval during which each CPU passes through a *quiescent state*).
4. Each CPU keeps track of the times it goes through a quiescent state with counters. As each CPU learns of a new *quiescent period* it takes a snapshot of its quiescent-state counters.
5. Each CPU periodically checks if its current quiescent-state counter has exceeded the snapshot. If yes, it clears a bit in a global bit mask to record the fact that it has passed through a quiescent state for the current quiescent period.
6. Once the bit mask changes to all zeroes, the CPU that happens to do this also records the fact that the current quiescent period has ended, and restarts another one if needed.
7. As each CPU learns that a quiescent period has ended it invokes any callbacks that were waiting for that quiescent period.

In the actual implementation, there are three variables which are shared across all CPUs:

- `curr` which numbers the current quiescent period,

- `completed` which records the number of the quiescent period that was last completed and
- `cpubitmask` which is a bit mask of all the CPUs.

In addition to the above variables, there are per CPU variables:

- `batch` which records the number of the quiescent period that this CPU is waiting for to schedule callbacks.
- `quiescbatch` which stores the number of the current quiescent period that this CPU is aware of; `passed_quiesc` records whether the CPU has passed a quiescent state in the current period; and `qpending` records whether the the quiescent period is already completed for this CPU or not. In our model, we abstract away from the details of the updates to these variables and assume that for each CPU the `cpubitmask` is updated when it passes through a quiescent state.
- `nxtlist`, `currlist` and `donelist`. These respectively keep track of the new callbacks to be queued, the callbacks that are waiting for the current quiescent period to end and the callbacks that are ready to be invoked. In our model, instead of a separate queue for each of these lists we keep a single integer variable called `list`. We abstract away from the details of the queue management and let `list` store 3 if there are pending elements in the `nxtlist`, 2 if there are pending elements in the `currlist`, 1 if there are pending elements in the `donelist` and 0 if there is nothing pending. This is valid because we only analyze the run of the RCU mechanism for a single update for each CPU and each callback passes through `nxtlist`, `currlist` and `donelist` sequentially.

We abstract away from the exact details of the function calls and present the RCU implementation as an integer transition system. In this system, all the above variables are represented as unbounded natural numbers with one copy per CPU for the case of CPU private variables. The full model is available along with the distribution of the LEVER tool from [91].

9.5.2 Results

We analyze a run of the RCU mechanism for a system with two CPUs with a single update occurring at each CPU. This implies that each CPU has pending updates each of which will require all CPUs to go through a quiescent period. Thus, initially, the `list` variable for each CPU is 3. First, we analyze a safety property which asserts that it is not possible for the callbacks for all CPUs to be invoked (all the `list` variables being 0) and the `cpubitmask` not being 0 (some CPU has not gone through a quiescent state). The LEVER tool successfully analyzes that the model satisfies this safety property in 5 minutes and 7 seconds. We also

analyze a CTL property stating that from all states, it is possible to go to a state in which all the updates are done (value of `list` for each CPU is 0). LEVER takes 12 minutes and 38 seconds to report that the property holds for the model. All tests were performed on an Intel Xeon based Linux machine running at 1.70GHz with 1GB memory.

Chapter 10

Conclusions and Future Work

The central thesis of this dissertation is that learning algorithms can be effectively used for verification of models of software systems. In previous chapters, we described an application of this technique for verifying properties of various classes of infinite state systems. We first showed how passive learning can be used to verify safety properties of FIFO automata. Then, we developed a new annotation scheme for representing the system executions and states reached with those executions. We also showed that the framework of active learning can also be used for verification of safety properties. Next, we applied the learning to verify approach to systems expressible in the *regular model checking* framework and in particular, parameterized systems and integer systems. We demonstrated verification of safety properties of such systems using learning with witnesses consisting of a bounded number of steps and witnesses using bounded space.

We have also shown that learning to verify approach can be used not only for safety properties but also for liveness properties. Both, linear time properties using ω -regular specifications and branching time properties using CTL with fairness constraints can be analyzed.

The techniques mentioned above have been implemented and are available in the LEVER tool suite. We have analyzed various examples taken from the literature and compared the performance of the tool with other tools available for analyzing infinite state systems. Although, there is no clear winner which performs better than the others for all the examples, we note that the LEVER tool suite performs better in a number of cases. As stated earlier, the main advantage of the learning based approach is that given enough time and space, it is guaranteed to find a solution if the set being learned is in fact regular. We have also presented a case study for the application of the learning based verification technique to analyze the *read-copy-update* mechanism used in the Linux kernel for reader-writer synchronization.

In recent years, there has been a trend towards model driven development of software systems; therefore, verification of system models is likely to become increasingly important. The learning-to-verify approach is a promising new method for automatic verification of such systems.

Since the application of learning techniques to verification is fairly new, there are a large number of interesting directions to pursue. In this dissertation, we have focused on mostly infinite state systems arising

as models of software systems. Another class of systems that the learning technique could be applied to is the class of hardware systems. In hardware systems, the set of states are usually represented symbolically to allow for a compact encoding. It has often been noted that even when the representation of the final fixpoint needed for verification is not large, intermediate approximations to the fixpoint could still be very complex. In such cases, the learning-based technique can potentially be better since it does not compute the intermediate approximations.

Application of the learning based technique to hybrid systems is also very intriguing. In hybrid systems, set of states are often represented using geometrical shapes such as *polyhedra* and *ellipses*. It would be interesting to see if techniques for learning geometric shapes can be applied fruitfully for verification of such systems. Another class of systems that could be analyzed is that of probabilistic systems.

In this dissertation, we used learning algorithms for regular sets and showed that the verification procedure is complete if the sets being learned are indeed regular. An interesting direction of research is to investigate if there any useful sufficient condition that can be used to check if the sets to be learned for a particular system are regular. If the check succeeds, then the user has the assurance that the learning based technique will eventually terminate with the right answer. Of course, even if the check fails, it might be still worthwhile to run the tool but the termination is no longer guaranteed. It is to be noted that we cannot hope to develop sufficient as well as *necessary* conditions since the underlying problem is typically undecidable.

We have so far used regular sets as the representation of choice for learning. It would be interesting to investigate techniques for learning sets more expressive than regular sets. This could expand the class of systems that can be successfully analyzed using the learning-to-verify method. Of special interest is learning *visibly pushdown languages* [8] since these languages have been shown to useful for a number of program analysis problems.

Another interesting challenge to pursue for future work is to apply the learning technique to actual software code written in traditional programming languages like C, C++, Java itself. The technique of abstraction is likely to be necessary for a viable verification procedure. Learning methods could be used to learn abstractions of the system which either verify that the system is correct or produce counterexamples which are then checked for validity. Similar to predicate abstraction techniques, spurious counterexamples could be used to refine the abstraction that was hypothesized by the learner.

References

- [1] P. A. Abdulla, B. Jonsson, M. Nilsson, J. d’Orso, and M. Saksena. Regular model checking for LTL(MSO). In *Proc. of CAV’04, USA, LNCS 3114*, 2004.
- [2] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d’Orso. Algorithmic improvements in regular model checking. In *Computer-Aided Verification (CAV’03)*, volume 2725 of *LNCS*, pages 236–248. Springer, 2003.
- [3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass., 1986.
- [4] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *2nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages*, 2005.
- [5] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
- [6] Rajeev Alur, Costas Courcoubetis, Thomas A. Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, pages 209–229, 1992.
- [7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.
- [8] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211. ACM Press, 2004.
- [9] Rajeev Alur, P. Madhusudan, and Wonhong Nam. Symbolic compositional verification by learning assumptions. In *Proceedings of the 17th International Conference on Computer Aided Verification*, 2005.
- [10] ALV. Action language verifier. <http://www.cs.ucsb.edu/~bultan/composite/>, 2004.
- [11] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *ACM SIGPLAN Notices*, 37(1):4–16, January 2002.
- [12] D. Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, November 1987.
- [13] A. Annichini, A. Bouajjani, and M. Sighireanu. TRex: a tool for reachability analysis of complex systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of the international conference on computer aided verification (CAV’01)*, Paris, France, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2001.
- [14] Andrea Arcangeli, Mingming Cao, Paul E. McKenney, and Dipankar Sarma. Using read-copy-update techniques for System V IPC in the Linux 2.5 kernel. In *USENIX Annual Technical Conference, FREENIX Track*, pages 297–309, 2003.

- [15] B. Boigelot and P. Godefroid. Automatic synthesis of specifications from the dynamic observation of reactive programs. In E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 321–334, Enschede, The Netherlands, 1997. Springer Verlag, LNCS 1217.
- [16] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
- [17] Sébastien Bardin, Alain Finkel, and Jérôme Leroux. FASTER acceleration of counter automata in practice. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590, Barcelona, Spain, March 2004. Springer.
- [18] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report 0203, INRIA, August 1997.
- [19] C. Bartzis and T. Bultan. Widening arithmetic automata. In *CAV: International Conference on Computer Aided Verification*, 2004.
- [20] J. Berstel. *Transductions and Context-Free-Languages*. B.G. Teubner, Stuttgart, 1979.
- [21] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Collection des Publications de la Faculté des Sciences Appliquées de l'Université de Liège, 1999.
- [22] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large (extended abstract). In *CAV: International Conference on Computer Aided Verification*, 2003.
- [23] A. Bouajjani, Jean-Claude Fernandez, and Nicholas Halbwachs. Minimal model generation. In Edmund M. Clarke and Robert P. Kurshan, editors, *Proceedings of Computer-Aided Verification (CAV '90)*, volume 531 of *LNCS*, pages 197–203, Berlin, Germany, June 1991. Springer.
- [24] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with non-regular sets of configurations. *Theoretical Computer Science*, 221(1–2):211–250, June 1999.
- [25] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 403–418. Springer, 2000.
- [26] Ahmed Bouajjani, Axel Legay, and Pierre Wolper. Handling liveness properties in (ω -)regular model-checking. In *Proc. of Infinity'04, London, UK*, 2004.
- [27] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *JACM*, 30(2):323–342, April 1983.
- [28] R. E. Bryant. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *International Conference on Computer Aided Design (ICCAD '95)*, pages 236–245, Los Alamitos, Ca., USA, November 1995. IEEE Computer Society Press.
- [29] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [30] T. Bultan. *Automated symbolic analysis of reactive systems*. PhD thesis, Dept. of Computer Science, University of Maryland, College Park, Md., 1998.
- [31] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state programs using presburger arithmetic. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, pages 400–411, 1997.

- [32] Tevfik Bultan and Tuba Yavuz-Kahveci. Action language verifier. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, pages 382–386, 2001.
- [33] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering (TSE)*, 30(6):388–402, June 2004.
- [34] A. Chutinan and B. K. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In *Hybrid Systems: Computation and Control, Second International Workshop*, pages 76–90, 1999.
- [35] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.
- [36] Marcus Ciolkowski, Oliver Laitenberger, and Stefan Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, 2003.
- [37] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings of Workshop on Logics of Programs, LNCS 131*, pages 52–71, 1981.
- [38] Ed Clarke, Sagar Chaki, Nishant Sinha, and Prasanna Thati. Automated assume-guarantee reasoning for simulation conformance. In *Proceedings of the 17th International Conference on Computer Aided Verification*, 2005.
- [39] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. Number ISBN:0262032708. The MIT Press, 2000.
- [40] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Building equational proving tools by reflection in rewriting logic. In *Cafe: An Industrial-Strength Algebraic Formal Method*. Elsevier, 2000.
- [41] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.
- [42] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 331–346, 2003.
- [43] P. Cousot and R. Cousot. Abstract interpretation: A unified model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.
- [44] Dennis Dams, Yassine Lakhnech, and Martin Steffen. Iterating transducers. *Journal of Logic and Algebraic Programming*, 52-53:109–127, 2002.
- [45] Satyaki Das and David L. Dill. Counter-example based predicate discovery in predicate abstraction. *Lecture Notes in Computer Science*, 2517:19–29, 2002.
- [46] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, pages 160–171, 1999.
- [47] G. Delzanno and A. Podelski. Model checking in CLP. *LNCS*, 1579:223–239, 1999.
- [48] P. Dupont. Incremental regular inference. In *Proceedings of the 3rd International Colloquium on Grammatical Inference (ICGI-96): Learning Syntax from Sentences*, volume 1147 of *LNAI*, pages 222–237, Berlin, September 1996. Springer.

- [49] E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, Washington, D.C., 1986. IEEE Computer Society Press.
- [50] S. Edelkamp, A. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI Symposium on Model-based Validation of Intelligence*, 2001.
- [51] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker and its implementation. In *Model Checking Software: Proc. 10 th Intl. SPIN Workshop*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
- [52] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE'99)*, pages 213–224, 1999.
- [53] S. Escobar, J. Meseguer, and P. Thati. Natural narrowing for general term rewriting systems. In *International Conference on Rewriting Techniques and applications (RTA)*, 2005.
- [54] FAST. Fast acceleration of symbolic transition systems. <http://www.lsv.ens-cachan.fr/fast/>, 2004.
- [55] A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems: Application to FIFO automata. *Information and Computation*, 181(1):1–31, 2003.
- [56] A. Finkel and Ph. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1–2):63–92, 2001.
- [57] Alain Finkel and Jérôme Leroux. How to compose Presburger-accelerations: Applications to broadcast protocols. In Manindra Agrawal and Anil Seth, editors, *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'02)*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156, Kanpur, India, December 2002. Springer.
- [58] L. Fribourg and H. Olsén. Reachability sets of parametrized rings as regular languages. In *Proc. 2nd Int. Workshop on Verification of Infinite State Systems (INFINITY'97)*, Bologna, Italy, July 1997, volume 9. Elsevier Science, 1997.
- [59] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Conference on Automated Deduction*, pages 271–290, 2000.
- [60] Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
- [61] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
- [62] Roberto Giacobazzi, editor. *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*. Springer, 2004.
- [63] E. M. Gold. Language indentity in the limit. *Inform. Control*, 10:447–474, 1967.
- [64] Mike Gordon. Introduction to the HOL system. In Myla Archer, Jennifer J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the International Workshop on the HOL Theorem Proving System and its Applications*, pages 2–3, Los Alamitos, CA, USA, August 1992. IEEE Computer Society Press.
- [65] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *LNCS*, pages 357–371, 2002.

- [66] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 208–219, New Brunswick, NJ, USA, / 1996. Springer Verlag.
- [67] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In *Proc. of Infinity'04, London, UK*, 2004.
- [68] Klaus Havelund and Grigore Rosu. Foreword - selected papers from the first international workshop on runtime verification held in paris, july 2001 (rv'01). *Formal Methods in System Design*, 24(2):99–100, 2004.
- [69] Klaus Havelund and Grigore Rosu. An overview of the runtime verification tool Java pathexplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
- [70] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *Annual Symposium on Foundations of Computer Science (FOCS)*, 1995.
- [71] Thomas Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.
- [72] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
- [73] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the 27th Annual Symposium on Theory of Computing (STOC)*, pages 373–382, 1995.
- [74] Gerard J. Holzmann. *The SPIN Model Checker*. Pearson Education, 2003.
- [75] John E. Hopcroft, Rajeev Motwani, Rotwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co. Inc., 2000.
- [76] Thomas Huckle. Collection of software bugs. <http://www5.in.tum.de/~huckle/bugse.html>, 2005.
- [77] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *6th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *LNCS*, pages 220–234. Springer, 2000.
- [78] J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.
- [79] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.
- [80] Gregory M. Kapfhammer. *The Computer Science and Engineering Handbook*, chapter Software Testing. CRC Press, Boca Raton, FL, second edition, To appear 2004.
- [81] Matt Kaufmann and J. S. Moore. Industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, April 1997.
- [82] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Cambridge, Massachusetts, 1994.
- [83] Nils Klarlund and Anders Møller. Mona. <http://www.brics.dk/mona/>, 2004.
- [84] Kupferman, Vardi, and Wolper. An automata-theoretic approach to branching-time model checking. *JACM: Journal of the ACM*, 47, 2000.

- [85] Orna Kupferman and Moshe Vardi. From complementation to certification. In *Proceedings of the 10th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 591–606, Barcelona, Spain, 2004. Springer.
- [86] A. Kurzanski and P. Varaiya. Ellipsoidal techniques for reachability analysis. In *Hybrid Systems: Computation and Control, Third International Workshop*, pages 202–214, 2000.
- [87] LASH. The liege automata-based symbolic handler. <http://www.montefiore.ulg.ac.be/~boigelot/research/lash/>, 2004.
- [88] D. Lee and M. Yannakakis. Online minimization of transition systems (extended abstract). In ACM, editor, *Proceedings of the 24th annual ACM Symposium on Theory of Computing*, pages 264–274, New York, NY, USA, 1992. ACM Press.
- [89] J. Leroux and G. Sutre. On flatness for 2-dimensional vector addition systems with states. In *Proc. 15th Int. Conf. Concurrency Theory (CONCUR'04), London, UK, Aug.-Sep. 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2004.
- [90] Jerome Leroux and Gregoire Sutre. Flat counter automata almost everywhere! In *Automated Technology for Verification and Analysis, Taipei, Taiwan, 2005*.
- [91] LEVER. Learning to verify tool. <http://osl.cs.uiuc.edu/~vardhan/lever.html>, 2004.
- [92] Alexey Loginov, Thomas W. Reps, and Shmuel Sagiv. Abstraction refinement via inductive learning. In *CAV*, pages 519–533, 2005.
- [93] LSE. Linux scalability effort. <http://lse.sourceforge.net>, 2001.
- [94] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer-Verlag, 1992.
- [95] Zohar Manna and Amir Pnueli. Temporal verification diagrams. In *Theoretical Aspects of Computer Software*, pages 726–765, 1994.
- [96] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
- [97] Paul E. McKenney. RCU vs. locking performance on different CPUs. In *linux.conf.au*, Adelaide, Australia, January 2004. Available: <http://www.linux.org.au/conf/2004/abstracts.html#90> <http://www.rdrop.com/users/paulmck/rclock/lockperf.2004.01.17a.pdf>.
- [98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.
- [99] J. Meseguer, M. Palomino, and N. Martí-Oliet. Equational abstractions. In *Procs. of CADE'03*, LNCS. Springer, 2003.
- [100] J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to analysis of cryptographic protocols. In *In Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science*, 2004.
- [101] Jos'e Meseguer. Software specification and verification in rewriting logic. 2003.
- [102] Anders Møller. <http://www.brics.dk/~amoeller/automaton/>, 2004.
- [103] O. Burkart D. Caucal F. Moller and B. Steffen. Verification over infinite states. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 545–623. Elsevier, 2001.

- [104] David Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 47(2–3):177–202, May/June 2003.
- [105] M. Nilson. Regular model checking. Master’s thesis, Uppsala University, Department of Information Technology, 2000.
- [106] Marcus Nilsson. <http://www.regularmodelchecking.com>, 2004.
- [107] O. Lichtensteinstate system n and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New York, January 1985. ACM.
- [108] J. Oncina and P. Garcia. Inferring regular languages in polynomial update time. In *Pattern Recognition and Image Analysis*, volume 1 of *Series in Machine Perception and Artificial Intelligence*, pages 49–61. World Scientific, Singapore, 1992.
- [109] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, 1995.
- [110] R. Parekh and V. Honavar. DFA learning from simple examples. *Machine Learning*, 44:9–35, 2001.
- [111] Lawrence C. Paulson. Introduction to Isabelle. Technical Report UCAM-CL-TR-280, University of Cambridge, Computer Laboratory, January 1993.
- [112] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *FORTE/PSTV, Beijing, China*, 1999.
- [113] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *CAV’00*, 2000.
- [114] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. *Inform. Comput.*, 103(2):299–347, April 1993.
- [115] Tatiana Rybina and Andrei Voronkov. Brain : Backward reachability analysis with integers. In *AMAST*, pages 489–494, 2002.
- [116] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Proc. 9th International Conference on Computer Aided Verification (CAV’97)*, volume 1254, pages 72–83. Springer Verlag, 1997.
- [117] SMV. Symbolic model verifier. <http://www-2.cs.cmu.edu/~modelcheck/smv.html>, 2001.
- [118] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. In *7th. Symposium of Logics in Computer Science*, pages 394–406, Santa-Cruz, California, 1992. IEEE Computer Society Press.
- [119] T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, LNCS 3091, pages 119–133, Valencia, Spain, June 3-5, 2004. Springer.
- [120] A. S. Tanenbaum. *Computer Networks, 2nd Ed.* Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [121] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Mathematics*, 5(2):285–309, June 1955.
- [122] Gregory Tasey. The economic impacts of inadequate infrastructure for software testing. <http://www.nist.gov/director/prog-ofc/report02-3.pdf>, 2002.
- [123] P. Thati and J. Meseguer. Complete symbolic reachability analysis using back-and-forth narrowing. In *International Conference on Algebra and Coalgebra in Computer Science*, 2005.

- [124] W. Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier, Amsterdam, 1990.
- [125] T. Touili. Regular model checking using widening techniques. In *ENTCS*, volume 50. Elsevier, 2001.
- [126] B. A. Trakhenbrot and Y. M. Barzdin. *Finite Automata: Behavior and Synthesis*. North Holland, 1973.
- [127] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Actively learning to verify safety for FIFO automata. In *LNCS 3328, Proc. of FSTTCS'04, Chennai, India*, pages 494–505, 2004.
- [128] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Learning to verify safety properties. In *LNCS 3308, Proc. of ICFEM'04, Seattle, USA*, pages 274–288, 2004.
- [129] Abhay Vardhan, Koushik Sen, Mahesh Viswanathan, and Gul Agha. Using language inference to verify omega-regular properties. In *Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60, Edinburgh, UK, April 2005. Springer.
- [130] Abhay Vardhan and Mahesh Viswanathan. Learning to verify branching time properties. In *Proc. of the Twentieth IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA*, 2005.
- [131] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Logic in Computer Science '86*, pages 332–344, 1986.
- [132] Moshe Vardi. Verification of concurrent programs: The automata-theoretic framework. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 167–176, 1987.

Vita

Abhay Vardhan was born in Roorkee, India in 1974 to Smt. Shanta Gupta and Dr. Satya Prakash. His school years were spent in Roorkee and in 1991 he went to New Delhi to study engineering at the Indian Institute of Technology, Delhi. After graduating from IIT Delhi with a Bachelor of Technology in Mechanical Engineering in 1995, he came to the University of Illinois at Urbana Champaign to pursue graduate studies. He was awarded a Masters of Science degree in Mechanical Engineering in 1997. In the middle of 1997, he got very interested in Computer Science and decided to work towards a Masters degree in Computer Science as well, which he obtained in 1998.

Abhay started working in Motorola Urbana Design Center from August 1998 as a software developer. Starting in 2000, he also started pursuing a PhD in Computer Science while continuing his work at Motorola.