INSTITUT POLYTECHNIQUE DE GRENOBLE

THESE

pour obtenir le grade de

DOCTEUR DE L'Institut Polytechnique de Grenoble

Spécialité : *Informatique*

préparée au Laboratoire Informatique de Grenoble
dans le cadre de **l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutnue publiquement

par

MUZAMMIL SHAHBAZ, Muhammad

le 12 décembre 2008

TITRE

Reverse Engineering Enhanced State Models of Black Box Software Components to support Integration Testing

DIRECTEUR DE THESE
Roland GROZ, Professeur à l'Institut Polytechnique de Grenoble

JURY

Acknowledgements

While I remain the only responsible of imprecision and omissions, there are lot of people to whom I am indebted.

“I would like to thank my PhD advisor” — This sentence is quite common one could find in acknowledgments. I also tried to write some more appropriate words for the sake of formality; but honestly, I failed. For a moment, I dabbled into the vast vocabulary of English, French and even Urdu (my mother tongue). But the words like *Thanks*, *Merci* and *Shukria* suddenly lost their quality —

Roland Groz, my PhD advisor, deserves the appreciation beyond my capabilities of expression. He is my mentor, not only for the disciplines of Software Engineering, but for the disciplines of life. Without his guidance, I would be lost.

Apart from the brainstorming meetings and discussions on the perplexed formal world of automata learning, I enjoyed with him hiking in summers, skiing in winters and theaters on fine evenings. Above all, I cannot forget those memorable two weeks which I spent at his home last summer in Lannion. I feel greatly obliged for the kindness his wife Bénédicte has shown upon me, and for the love of his cheerful children.

I am grateful to Dr. Keqin Li, *SAP Research*, with whom I have worked for two years. His constructive remarks on my work have certainly improved my skills for writing formal algorithms, proofs and research papers.

I am lucky to receive technical insights from Dr. Alexandre Petrenko, Director *CRIM*, with whom I have been in touch on and off during my PhD. His dedication to research has greatly inspired me, and I am always keen to learn something new from him. As a remembrance, I shall be keeping the notes in his writing which he prepared to teach me Reachability Graphs.

I am greatly indebted to the reviewers of my thesis, Prof. Ana Cavalli and Prof. Thierry Jéron for this painstaking work. Their useful comments and remarks have certainly improved the quality of the thesis. I am also grateful to Prof. Doron Peled for the fruitful discussions we had during his visits to *Verimag* time to time. Also, I am thankful to Prof. Khaled El-Fakih whose insights introduced me to the application of machine learning in specification based testing.

I am warmly thankful to Prof. Yves Ledru, head of *Vasco* team, Dr. Alexandre Lefebvre, my manager at *Orange Labs*, Dr. Benoît Parreaux and Dr. Wei Monin, my co-supervisors at Orange Labs for their administrative and technical support. Also, I am grateful to Dr. Yves-Marie Quemener who has provided me various opportunities to speak out at Orange Lab's platform.

I would like to express my deepest gratitude to my parents Mr. & Mrs. Shahbaz, whose prayers and continuous support is my gospel of encouragement. Its because of them I have attained this position. I dedicate this thesis to them.

PhD requires lot of energy, especially when one is miles away from his family and homeland. I received recharging and vitality for this research through the social aspect of my life. I owe my sincere thanks to my friends *Pakistanais* and *Grenoblois* for their loving support in low and high times.

This research is funded by France Telecom R&D / Orange Labs under the CIFRE scholarship program. I would like to convey my special thanks to the organization for its confidence in me.

Abstract

Component based software engineering has gained a strong momentum in software industry that facilitates the building of complex systems using prefabricated components, aka COTS. A challenging issue in this discipline is to deliver quality of service while integrating COTS from various sources. The system designers require specifications or models of the components to understand their possible behaviors in the system. When components come from third-party sources, the specifications are often absent or insufficient for their formal analysis. Such components are termed as black boxes in literature.

The thesis addresses the problem of uncovering the behaviors of black box software components to support testing and analysis of the integrated system composed of such components. Typically, we propose reverse engineering techniques to infer finite state models of the components and base the approach of testing and analyzing the system on the inferred models.

We start by studying the inference of Mealy machine models in the settings of active learning theory and propose improvements in the existing algorithm to bring down the learning complexity. Later, we provide a framework for testing and analyzing the integrated system of black box Mealy components using the inferred models.

The thesis also proposes solutions for learning enhanced state models to cope with the problem of modeling complex systems. Such systems contain components that exchange lots of input and output parameters from arbitrary domains. We propose a parameterized model and an algorithm to infer such models from a black box component.

We present our tool *RALT* that implements the reverse engineering techniques and the integration testing framework. The approach has been validated on various case studies in the domain of France Telecom that have produced encouraging results.

Contents

List of Figures

List of Tables

List of Algorithms

List of Definitions

Chapter 1

Introduction

"Before we trust a component, we must be able to determine, reliably and in advance, how it will behave." — *Computer, IEEE Computer Society Press 1999*

1.1 Overview

Component Based Software Engineering has gained a strong momentum in many sectors of the software industry. The main reason of its prevalence is that it reduces the cost of developing complex systems by reusing the prefabricated pieces of software, called components-off-the-shelf (COTS) or third-party components, instead of developing the systems from scratch. As a consequence, most large-scale systems such as telecom services, web-based applications, data acquisition modules etc, are now-a-days based on the integration of COTS.

One of the most open challenges in using COTS is to deliver the quality of service. The system designers require specifications or models of the components to understand their possible behaviors in the system. Precisely, they want to check the possible interactions between the components to avoid any unexpected situation in the later run of the integrated system. In general, Model Driven Engineering (MDE) approaches are applied as a major mode that provide rigorous techniques for specifying, designing, analyzing, testing and verifying the system. However, it is a quite common situation that where MDE techniques are desirable, the specifications or the formal models of those components are not available. Even if they are available, they do not provide enough information that could be useful to drive formal techniques. Moreover, maintaining the specifications of COTS is unrealistic because they evolve over time that quickly invalidates the original design sketches. The need of specifications or formal models as a prerequisite in using COTS is a daunting prospect to the designers of large-scale systems. In industry, they mostly rely on informal and incomplete information to evaluate the quality of the overall system.

Our research focuses on devising techniques to uncover the behaviors of components which lack specifications or formal models, and facilitate the analysis of the integrated system composed of such components. Typically, we extract the finite state models from the components using (active) learning techniques and provide a framework to test the integrated system using the extracted models. We also propose solutions of learning enhanced state models to cope with the problem of modeling complex systems. Our approach is validated on various case studies provided by France Telecom R&D, that have produced encouraging results.

This chapter is the introduction to the thesis. Section 1.2 provides the general concepts of the problem we have addressed. Section 1.3 presents a motivational example in France Telecom R&D in the context of the thesis. Section 1.4 describes the global approach, hypotheses and scope of the research. Finally, Section 1.6 outlines the organization of the thesis.

1.2 Component Based Software Engineering

The interest in Component Based Software Engineering (CBSE) is increasing in both academia and industry as witnessed by the escalation of devoted conferences [CPV03] [SCHS07], journal issues [Rav03] [CSSW05], books [Szy02][GTWJ03] and forums [OCA] [OMG], just to cite a few. Apart from these research platforms, many new technologies have been established to support the deployment and execution of component based systems, e.g., Java EE [Mic], .NET [.NE], CCM [CORa], OSGi [All].

CBSE is the process of integrating components and make them interacting as intended. It promotes rapid system development by facilitating component reuse, but components may need to be tuned according to the system requirements before they are actually used. In fact, the components come from different sources and have been developed in different environments. Therefore, the capability of components to be adapted and configured according to the desired environments is important for the success of the CBSE approach. In the following, we discuss the approach in more detail, by first giving the notion of a component and then an overview over the challenges in the integration of components.

1.2.1 Notion of a Component

It will be interesting to discuss the notion of a software component before understanding the challenges in the subject. The widely accepted definition of the component comes from Szyperski et al [Szy02].

"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. A component can be deployed independently and is subject to composition by third parties."

This definition covers the different peculiar aspects of a component, i.e., "unit", "interface" and "composition". A component can be plugged into a system as a unit, with its features accessible via its defined set of interfaces and it communicates with other components in the composition for its services to the integrated system. The above definition also points to the component source, i.e., third party. The component is actually deployed by the system integrating organizations different from the developing organizations.

Since the components come from different sources, it is quite normal that they are not provided with specifications and technical corpora. Even if some documentation is provided, it is not sufficient for the component users to understand its behaviors completely. The documentation usually provides a high level description of the component and limited to syntactic definitions of its interfaces. Moreover, components continuously evolve over time to incorporate additional requirements. They are inherently difficult to understand and maintain due to their size and complexity. As long as, their long evolution history hinders in keeping their specifications up-to-date.

The implementation of a component is typically not exposed, rather only textual abstractions are attached as its interface specifications so that it can be plugged in the system. The signature of the interfaces and the basic set of inputs to the components are provided, but the source code or complete functional behavioral descriptions are often unavailable. The user simply benefits from the component functionalities without knowing anything about its underlying implementation details. In literature, the terms black box, gray box and white box are used with reference to different levels of closure of the component internal essence. In particular, a black box component does not disclose anything about its internal design, structure and implementation, whereas its opposite side, a white box component is completely exposed to its user. In between, there may exist different levels of grey box components depending upon how much details are available.

1.2.2 Challenges in the Integration of COTS

We discuss some of the challenges in the integration of COTS that are related to our work, directly or indirectly. We refer to the works [Har00][CCC+02][GTWJ03][Ber07] for comprehensive discussions on the topic.

Understanding the Component Behaviors

The most common problem in using COTS is the unfamiliarity of their behaviors and functions. As a first principle, the designers seek for technical corpora for the components and consult the related documentations and manuals for their understanding. In order to apply more formal approaches and MDE techniques, they obtain specifications and/or source codes and then perform activities under white-box framework such as static analysis [ALSU06], program slicing [Tip95], invariant detection [ECGN01][HL02], model extraction [NE02][WBH06], unit testing [Corb][JUn], validation [GEH05] and verification [CGP00].

Most of these techniques become ineffective when black box components are under consideration. Understanding the behavior and testing of such components is a challenging task, due to the unavailability of their source codes, updated specifications or formal models. Consequently, the practice of behavior analysis in industry is presently conducted in an ad-hoc way, either relying on human insights or using heuristic strategies based on incomplete information. Normally, the designers test the system on few known scenarios and make a judgment about its quality by observing its global behavior and matching it with their informal requirements [CJ02][GTWJ03].

Analyzing the Component Interactions

When the system is built with several components, it is important to ensure that the components that are developed separately, work properly together. Since the components are developed under different environments and may not be specialized for the particular environment where they would be integrated, there could be many interactions between the components which are undesirable and which could affect the features of each other. Moreover, the system may suffer from compositional problems such as deadlocks, livelocks and other behavior incompatibility and interoperability issues due to the interactions. The consequences of such interactions can range from minor irritations to complete system failures. At this stage, the designers turn their focus towards integration testing to analyze interactions in the integrated systems. There is also a research area, called *Feature Interaction* [CKMRM03], that aims to find ways of detecting as well as resolving feature interactions in the system.

There are several frameworks (e.g., [OHR01] [Edw01] [WPC01] [WCO03] [MPW04] [CLW05]) which are proposed to handle the interactions between the components during the integration phase. The designers use tools (see [BfCE04]) that generate interaction graphs [WPC01], which are useful to analyze the interactions with MDE techniques. However, the problem is complex in the black box context where these techniques are not directly applicable due to unavailability of source code and specifications. Moreover, the interpretation of each interaction as "good" or “bad” is not trivial. The interactions can be checked automatically if the expected behaviors of the system are written formally. When expected behaviors are unspecified or specified in a way that does not allow automated checking, then the interactions are checked manually which is notoriously difficult and time-consuming. If some expected i/o behaviors in the system are given in terms of scenarios in which the designer expresses “should” or “should not” requirements [BB98], then the scenarios can be checked on the system through testing and violations can be detected. When no expected behaviors are available in a formal shape, the designers often rely on anomalies such as system crashes [MFS90] [KKS98], or uncaught exceptions [CS04] as symptoms for unexpected behaviors.

Validating the System

The validation of the integrated system requires conformance of the behaviors of the system according to the requirements. Formally, this phenomenon is referred as *Conformance Testing* or *Fault Detection Problem* in software validation community [LY96]. Ideally, we are given a complete specification and a system (composed of black box or white box components) and we test to determine whether the system is a correct implementation of the specification. Other faces of system validation include evaluating the robustness to stressful load conditions or malicious inputs, or measuring given attributes such as performance or usability, or estimating the operational reliability, and so on.

The validation of the integrated system is a complex problem because of the quite obvious fact that testing can never be exact [Dij70]. The challenges of test selection and applying adequate testing strategies to identify the potential faults in the system are still alive. In addition to the warning that even though many tests passed, the system can still be faulty, provides little guidance what we can conclude about the system after having applied those tests. Going even further, how we can dynamically tune up our testing strategy as we proceed with accumulating test results, taking into account what we observe from the system in the result of testing. Especially in the case of black box context, how we could guess that the tests have explored all the state space of the system and all possible behaviors have been checked. In theory and practice, we rely on some underlying assumptions, or system approximations, to conclude that the system is sufficiently tested. If we perform exhaustive testing according to the approximations, then from successfully completing the test campaign we could justifiably conclude that the system is correct. That is, we still know that the system could be faulty, but we also know that we have assumed to be true with respect to the approximations [Ber07].

1.3 DoCoMo: A Motivational Example in France Telecom

This PhD thesis was proposed by France Telecom R&D due to the known problems in CBSE which its designers have experienced in the past. We provide here a motivational example which was recently experienced before the proposal of the thesis was officially triggered.

In 2004, France Telecom acquired a gaming component from *NTT DoCoMo Inc.*, a Japanese mobile software components maker. The component was integrated into the mobile phones and distributed in the market. Later, a complaint was reported that the users of the phones occasionally experience credit loss.

An investigation was carried out by the test engineers who revealed after several experiments that it was the gaming component that was causing the credit loss. They found a new behavior in the component which was previously unknown to them. Actually, when a user starts a game session, the component connects the user to internet and communicates with the Docomo's gaming server. At the end of the session, the component uploads the user's scores to the server. When the phone is connected to the internet, the user is charged according to the service tariff.

This was a classic black box component integration problem that was emerged due to i) an unknown behavior of the component, and ii) an underlying interaction with the web service module of the phone. Since the behavior was unknown to the users, the deduction of credit was considered an anomaly in the phones. In fact, Docomo has implemented this behavior to keep record of its international users and their scores history. However, the implementation was not specified in the documentation provided with the component.

(courtesy: Benoît Parreaux, Orange Labs, Lannion)

There is a similar kind of problem concerning the media player of mobile phones, on which we have demonstrated how our approach of reverse engineering can tackle such problems. In the explanation of the case studies in the thesis, we shall discuss the specified problem and the illustration of the approach on the problem in details.

1.4 The Approach of Learning and Testing

One of the major promising approaches to the problem of understanding the black box components is to extract their models from their observations. There are various mechanisms to collect the observations, either passively, i.e., by monitoring the component while it is running, or actively, i.e., by stimulating the component through testing. Then, certain machine learning techniques [KV94] [CW98] are used to infer the models that are consistent with the observations. These models are then used in the MDE techniques to generate tests for validating the components. It is important to note that the application of testing in this context is not only meant to uncover faults but also to uncover the unknown component behaviors by applying testing techniques.

The approach of extracting models from observations is not so simple. Although the learning techniques provide models that are consistent with the observations, they do not always provide the precise models. In general, it is unknown what amount of observations would be sufficient to deduce the complete model of the black box component. Normally, several iterations are required in which tests are generated from the learned model, then the component is tested to compare its corresponding behaviors with the learned model. When new observations in the result of testing find a discrepancy in the behavior comparison, then the observations are used to refine the model, again by using the learning techniques. The general approach is illustrated in Figure 1.1.

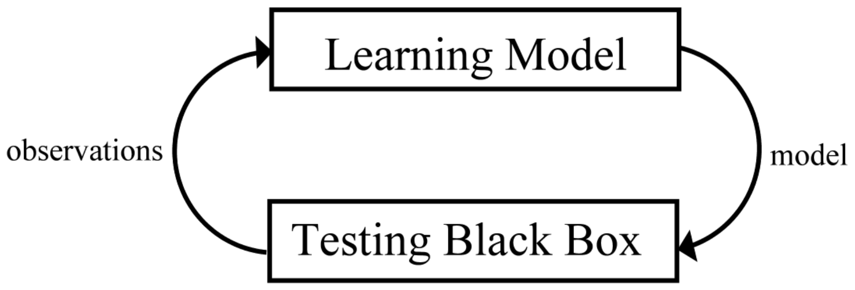


Figure 1.1: Learning and Testing Approach

It is a feedback loop between learning and testing in which the tests are generated from the learned models, then the learning techniques receive feedback from the tests, and then models are refined based upon new observations. In the next iteration, better tests are generated from the refined models and the loop starts again. The termination of the feedback loop could be based on different strategies, e.g., when certain properties on the models and components are satisfied, or when the assumptions on the model completeness are observed, or when the test adequacy criteria are achieved etc. Globally, this approach of learning and testing fulfills two purposes i) understanding of the behaviors of the components by deriving their approximate formal models *(reverse engineering)*, and ii) analyzing the components by testing them using their learned models *(validation)*. The main limit of this approach is the complexity of learning complete models [KV94] [PVY99]. In most cases, the learned model appears to be a partial representation of the complete behavior spectrum of the black box component. However, the partial models are generally acceptable to carry out the validation activities in practice [Ber07].

The rationale behind the feedback loop is that if the tests find discrepancies between the partial learned models and the actual component, it is likely that the tests exercise a new behavior of the component that was not uncovered so far [XN03]. Therefore, it greatly enhances the confidence when refined models are used to validate the components.

In our work, we exploit the use of learning and testing approach for component based systems. We propose solutions in two main directions: i) the components are learned as enhanced[1] finite state machines by adapting the existing learning algorithms, ii) the integrated system of those components is tested and analyzed using their learned models. Thus, we are providing a framework of testing and analyzing the integrated system of black box components using enhanced finite state models. In the following, we provide an overview of the two directions. The different parts of the approach in each direction are explained formally in their dedicated chapters (see Section 1.6 for the organization of the thesis).

1.4.1 Learning Finite State Machines

We extract the models of the components as finite state machines which can represent the precise descriptions of the component behaviors formally, and therefore, can be used in various MDE techniques. We propose algorithmic solutions of learning finite state machines, based on the existing works in grammatical inference [KV94], and propose improvements to tackle their complexity. In fact, the learned models are just the approximations of the real components. Our focus is to abstract the control part of the component and extract its structural and design information in a formal representation. We do not envision to deduce complete models of the components, which is impossible without having some strong assumptions on the hidden structure, such as knowing the upper bound on the number of states of the black box component [PVY99]. Apart from the assumptions, it is impossible in practice to find a precise model of the component which is usually far too large and, as results from the study of the application of the learning theory indicate [BJLS05], too time-consuming to obtain and to manage. It is also important to note that we are dealing with the integrated system in which components depend upon each other for their working. Normally, only some parts of the component functionalities are exercised in the system. So, we believe that learning complete models is not necessary in the integration framework. We aim to learn models that are approximate enough to represent the general behavioral spectrum of the black box components so that the system designer could depend upon objective data (the known set of observations from the systematic testing of the

[1]The term “enhanced” refers to a richer structure of Finite State Machines, in which transitions are labeled with inputs and outputs along with parameter values. The different enhanced models we are dealing with are presented in Chapter 2

components) rather than relying only on intuitions and experiences for satisfying their informal requirement specifications.

The other major focus of this direction is the learning of enhanced state machine models. Actually, the interactions between components consist of inter-component procedure calls (in a synchronous environment) or message passing (in an asynchronous environment). In both cases, the interactions are structured with a type (name of procedure or message) and parameters exchanged. It is observed that typical interoperability problems in the component integration framework emerge due to exchange of parameterized data from arbitrarily complex domains. The modeling of such components with simple finite state machines is inadequate which cannot capture the fine granularity of the component. Also, such components typically have formidable size of inputs, but they usually show similar behaviors on a subset of inputs. Learning such behaviors with fewer inputs and representing them in a compact model can tackle the complexity of the learning algorithm. Under this view, we propose the inference adapted to a representation of parameterized interactions of the component. This representation is referred as a parameterized finite state machine (PFSM) in the thesis.

1.4.2 Analyzing Integrated Systems using Learning and Testing

We use classical *divide et impera* strategy, i.e., the large and complex integrated system is disassembled into individual components, where each component is tested separately to learn its model and then the complete system is taken into account for integration testing using the learned models of all the components. The different steps of the learning and testing procedure are described as follows:

Step 1: All the components in the integrated system are learned in isolation so that the finite state models of each component are extracted.

Step 2: The learned models of each component are composed to make a product that is analyzed for compositional problems such as deadlocks and livelocks in the system. We can find a witness to such problems in the product of the learned models through validation techniques.

Since the product of the learned models is an approximation, the witness of the problem in the product may not actually exist in the actual system. In other words, it can be just an artifact due to the partiality of the models. Therefore, we confirm the problem on the system by simply experimenting the witness. If the experiment produces the problem in the system then we terminate the procedure by reporting the problem. Otherwise, the witness is a discrepancy between the product and the actual system. In this case, the witness is treated as a counterexample for the product.

Step 3: When a counterexample for the product of learned models is found, the counterexample is broken down for each component to identify the components whose partial models caused the discrepancy between the product and the actual system. Then, the identified components will be relearned using their relative counterexamples and their refined models will create a new product (following step 2).

Step 4: Once the product of the models is obtained that contains no compositional problems, thanks to the steps 2 and 3, the product serves as an input to MDE techniques for the purpose of analyzing, testing and validation of the integrated system. In fact, the product is a finite state machine, so several testing strategies from existing works can easily be used in our case. The generated tests may find more discrepancies between the product and the system, since the product may still not be a correct representation of the system.

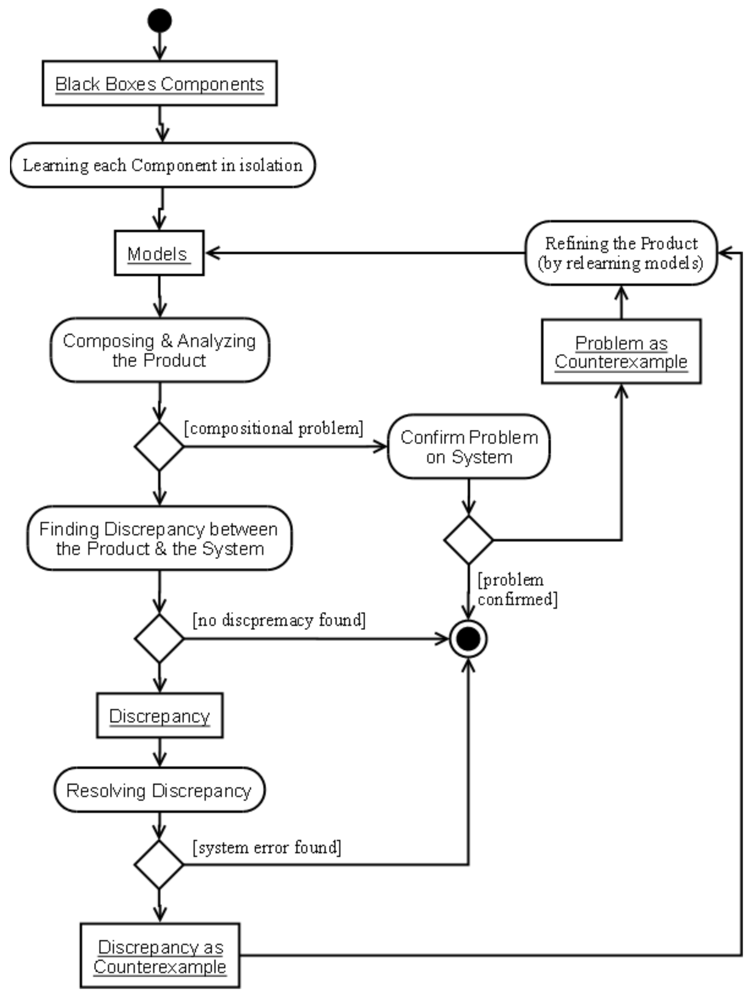
Step 5: We resolve the discrepancies found in the previous step by classifying it as an error in the system or an artifact of the learned models. The discrepancies may uncover system integration faults such as components behavioral compatibility issues, unexpected interactions, system errors due to crash or uncaught exceptions etc. In this case, we terminate the procedure by reporting the potential faults in the system. Otherwise, the discrepancy is an artifact, so we proceed for step 3 to refine the product by considering the discrepancy as a counterexample for the product.

The procedure of integration testing terminates when a compositional problem in the system is confirmed (step 2), or when the generated tests do not find any discrepancies between the product and the system (step 4), or when the real errors in the system are found (step 5). The flow of the procedure is given in Figure 1.2.

1.5 Hypotheses and Scope

The hypotheses and the scope of our work are given point wise as follows.

System of communicating components: The integrated system consists of several components which communicate with the system's environment and with each other via matching input and output symbols. Thus, a component has input/output behaviors, i.e., when receiving an input it produces the corresponding output to its environment. The input and output symbols are associated with parameters that represent the values exchanged



Black Boxes Components
Learning each Component in isolation
Models
Refining the Product (by relearning models)
Composing & Analyzing the Product
Problem as Counterexample
[compositional problem]
Confirm Problem on System
Finding Discrepancy between the Product & the System
[problem confirmed]
[no discpremacy found]
Discrepancy
Resolving Discrepancy
[system error found]
Discrepancy as Counterexample

Figure 1.2: Learning and Testing Approach for an Integrated System

during the component interactions. The typical examples of such systems are telecom services (e.g., call center solution), web-based applications, data acquisition modules and embedded system controllers. As a concrete example, suppose a traveler agent system that consists of a hotel database component and a user interface component. The database component receives a city name from the interface component and provides a list of hotels in the city in response. Then, the `city_name` is the input symbol and `(Paris)` is a possible input parameter value. Similarly, the `hotel_list` is the corresponding output symbol and `(Hilton, Sheraton, ...)` is a possible output parameter value.

System architecture is known: The architecture of the system is known, i.e., we know how the components are bound together through their interfaces. Moreover, the system can be disassembled and reintegrated whenever it is required. This assumption is obvious from our context where the designer knows how to integrate third-party components but does not know the possible interactions between the components and actual ordering of events etc, once the system is integrated. As an example, the designer knows how to bind the hotel database component with the user interface component, but unaware of the possible behaviors on different inputs to the traveler agent system.

Components are black boxes but their inputs are known: The components in the system may have different levels of exposure depending upon how much information about them is available. In our approach, we consider that all components are black boxes, i.e., their functional specifications and implementation details are not available. We do not assume a priori given behavior traces of the components or oracles, often presumed in traditional model learning. However, we assume that the input/output interfaces of the components are known and observable. This means we know the basic set of input symbols that can be given to a component through its input interfaces, and for each input, the corresponding output of the component can be observed through its output interfaces. The parameter domains for the input symbols are also known. For example, we know that the hotel database component receives input symbols such as `city_name`, `room_type`, `check-in_date`, `number_of_nights` etc. We also know that the parameter domain for `city_name` is the name of a city, e.g., `Paris, London, ...` etc, for `room_type` is the type of rooms, e.g., `single, double, ...` etc, for `check-in_date` is a value of type date, for `number_of_nights` is a positive integer.

Invalid inputs may be observed dynamically: According to the previous assumption, we know the set of possible input symbols to a component. However during the testing procedure, we may be able to observe which inputs are valid to be given at the current step and which are invalid. For example, the user interface of the travel agent system may consist of multiple web pages. At the first page, it receives only `city_name`; then it transfers to the second page for the next input and so on. In this case, all inputs except `city_name` are invalid on the first page. This knowledge can be used during the learning of the component.

Components as finite state machines: The component exhibits regular behaviors, i.e., the component can be modeled as a finite state machine. We intend to learn only the behaviors prescribed by the control structure of the finite state model. Moreover, we do not assume to know the upper bound on the number of states in the components. Instead of hunting for exact learning, we aim to learn approximate models that are expressive enough to provide powerful guidance for testing and to enhance the behavior understanding of the components, and thus, of the system. Another important prerequisite for this approach is the deterministic property of the component for reproducible and unambiguous interpretation of the test results.

Formal description expected behaviors are unavailable: We do not assume the provision of formally described expected behaviors by the designers. In practice, expected behaviors often do not exist for automatically checking the errors in the system. If some expected i/o behaviors are given in terms of scenarios, then scenarios can be checked on the learned models, or on the product of the learned models, and violations can be detected. However, even if scenarios are not available, then we rely on the detection of anomalies in the system, such as system crash or uncaught exceptions. In practice, such anomalies can be detected automatically.

Issue of output delays is overlooked: There may be some cases where the system's reaction to a given input consists of more than one outputs. Those outputs are produced with some delay and may occur after giving the next input to the system. Usually, functional testing does not care for exact timings and so delays do not matter much. But it is very important to identify the correct outputs of the input which triggered them. Thus, it is a common practice to wait after each input to collect all its corresponding outputs. Most often, appropriate timeouts are applied to ensure that the system has produced all outputs and settled in a "stable" state. In our settings, we overlook the issue of adding the delays in testing. In practice, this is considered as an implementation detail of the test drivers, which actually send the inputs from the environment to the system (or to the component) and send the corresponding outputs back to the environment.

Focus on functional aspects: We keep our focus on behavior learning and studying the interactions between the components and their functional aspects in the system. We are not dealing with other details, for instance, timing, performance and security issues in the system. However, we shall provide clues in the conclusion (Chapter 10) how to take such issues into consideration as a possible extension to our approach.

1.6 Thesis Organization

The organization of the thesis (excluding this chapter) is given as follows.

State Of The Art

- Chapter 2: provides the basic definitions and notations which are used globally in the manuscript. It describes the kinds of finite state models and the formal notion of their approximations, which will be used in the subsequent chapters. The different finite state models we are dealing with are:
  - Deterministic Finite Automata
  - Mealy Machines
  - Parameterized Finite State Machines
- Chapter 3 overviews the background work and surveys the state-of-the-art in the domain of learning and testing.
- Chapter 4 is the continuation of the state-of-the-art but dedicated to the inference of Deterministic Finite Automata.

Contributions

- Chapter 5 discusses the inference of Mealy Machines.
- Chapter 6 provides a framework for testing the integrated systems of Mealy components.
- Chapter 7 discusses the inference of Parameterized Finite State Machines.
- Chapter 8 describes the extended work towards learning functions in parameterized systems.
- Chapter 9 discusses our implemented tool and case studies.

Conclusion

- Chapter 10 summarizes and concludes the thesis by pointing out the future directions of the conducted research.

Chapter 2

Definitions, Notations and Models

This chapter provides the basic definitions and notations which are used globally in the manuscript. It describes the kinds of finite state models and the formal notion of their approximations, which will be used in the subsequent chapters.

2.1 Finite State Models

We are dealing with four kinds of finite state models in our work. The definition and the example of each model is given in the subsections.

2.1.1 Deterministic Finite Automaton

The formal definition of a Deterministic Finite Automaton is given as follows.

Definition 1 A *Deterministic Finite Automaton (DFA)* is a quintuple $(Q, \Sigma, \delta, F, q_0)$, where

- Q is the non-empty finite set of states
- Σ is the alphabet, i.e., the finite set of letters
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function.
- $F \subseteq Q$ is the set of final states
- $q_0 \in Q$ is the initial state

A string $\omega = i_1 \cdots i_k \in \Sigma^*$ is accepted by a DFA if there exists a sequence of transitions, labeled with each symbols in ω, starting from initial state and ending in an accepting state. We extend the transition function from symbols to strings in the standard way, i.e., $\delta(q_0, \omega) = \delta(\ldots \delta(\delta(q_0, i_1), i_2), \ldots, i_k)$. A string ω is accepted if and only if $\delta(q_0, \omega) \in F$, denoted by the *output function* $\Lambda(q_0, \omega) = 1$. Otherwise, ω is rejected, denoted by $\Lambda(q_0, \omega) = 0$. We define the *complete output function* when applying ω to DFA as follows: $\lambda(q_0, \omega) = \Lambda(q_0, i_1) \cdot \Lambda(q_0, i_1 \cdot i_2) \cdots \Lambda(q_0, i_1 \cdots i_k)$. An example of a DFA over the alphabet $\Sigma = \{a, b\}$ is shown in Figure 2.1. The final states are represented by double lined circles in the figure.

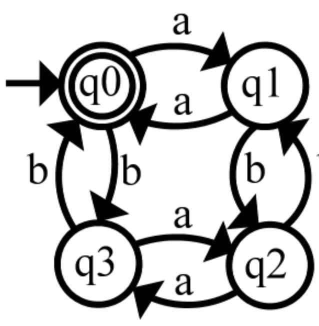


Figure 2.1: Example of a Deterministic Finite Automaton

2.1.2 Mealy Machine

The formal definition of a deterministic Mealy machine is given as follows.

Definition 2 A *Mealy Machine* is a sextuple $(Q, I, O, \delta, \lambda, q_0)$, where

- Q is the non-empty finite set of states
- $q_0 \in Q$ is the initial state
- I is the finite set of input symbols
- O is the finite set of output symbols
- $\delta : Q \times I \to Q$ is the transition function.
- $\lambda : Q \times I \to O$ is the output function $\square$

When a Mealy machine is in the current (source) state $q \in Q$ and receives $i \in I$, it moves to the target state specified by $\delta(q, i)$ and produces an output given by $\lambda(q, i)$. We extend the functions δ and λ from symbols to strings in the standard way as follows. For a state $q_1 \in Q$, an input string $\omega = i_1 \cdots i_k \in I^*$ takes the machine to the ending state by traversing each state $q_{j+1} = \delta(q_j, i_j), 1 \leq j \leq k$, and reaches to the final state $q_{k+1} = \delta(q_1, \omega)$, and produces an output string $\lambda(q_1, \omega) = o_1 \cdots o_k$, where $o_j = \lambda(q_j, i_j)$. We call $i_1/o_1 \ldots i_k/o_k$ an i/o trace of length k in state q_1.

We consider that the Mealy machines are input-enabled[1], i.e., $dom(\delta) = dom(\lambda) = Q \times I$. For a state where the given input is invalid, we add a loop-back transition on the state and add a special symbol Ω as the output for that input. So, we add Ω in O. For simplicity, we do not write Ω in the graphical representation of Mealy machines. We can depict Mealy machines as directed labeled graphs, where Q is the set of vertices. For each state $q \in Q$ and input symbol $i \in I$, there is an edge from q to $\delta(q, i)$ labeled by "i/o", where o is an output symbol given by $\lambda(q, i)$. An example of a Mealy machine over the sets $I = \{a, b\}$ and $O = \{x, y\}$ is shown in Figure 2.2.

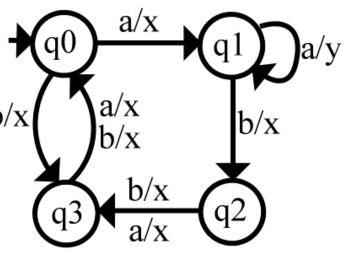


Figure 2.2: Example of a Mealy Machine

Definition 3 For every DFA $D = (Q_D, \Sigma, \delta_D, F, q_{0D})$, there is an equivalent Mealy Machine $M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$, that models the same language. The conversion is defined as follows:

- $Q_M = Q_D$
- $q_{0M} = q_{0D}$
- $I = \Sigma$
- $O = \{0, 1\}$
- $\delta_M(q, i) = \delta_D(q, i), \forall q \in Q_M, i \in I$
- $\lambda_M(q, i) = \begin{cases} 1 & , \delta_D(q, i) \in F, \forall q \in Q_M, i \in I \\ 0 & , otherwise \end{cases}$ □

[1]This is due to the implication of the learning algorithm to state the well-defined transition function. See Chapter 5, Section 5.2.1 for more on it.

2.1.3 Parameterized Finite State Machine

The formal definition of a Parameterized Finite State Machine is given as follows.

Definition 4 A *Parameterized Finite State Machine (PFSM)* is a septuple $(Q, I, O, D_I, D_O, \Gamma, q_0)$, where

- Q is the finite set of states
- I is the finite set of input symbols
- O is the finite set of output symbols
- D_I is the set of finite/infinite input parameter values
- D_O is the set of finite/infinite output parameter values
- q_0 is the initial state
- Γ is the set of transitions

A transition $t \in \Gamma$ is defined as $t = (q, q', i, o, p, f)$, where $q \in Q$ is a source state, $q' \in Q$ is a target state, $i \in I$ is an input symbol, $o \in O$ is an output symbol, $p \subseteq D_I$ is a predicate on input parameter values and $f : p \longrightarrow D_O$ is an output parameter function. The model is restricted with the following three properties:

Property 1 (Input Enabled) *The model is input enabled, i.e., for all $q \in Q$ and $i \in I$, there exists $t \in \Gamma$ such that $t = (q, q', i, o, p, f)$.*

Note that the property of input enabled is restrictively defined only on I, and not on I and D_I. That means, the model has an enabled transition for each state and each input symbol; in which the input symbol is associated with certain input parameter values, but may not be with all.

In order to make the model input-enabled, we add a loop-back transition on the state where the given input symbol is invalid and add a symbol Ω as the output. Similarly, there exists input symbols which do not take input parameter values at all. We add a symbol $\perp$ with the input symbol that expresses the absence of a parameter value. We add Ω in O and $\perp$ in D_I. For simplicity, we do not write these symbols in the graphical representation of PFSM models.

Property 2 (Input Deterministic) *The model is input deterministic, i.e., for $t_1, t_2 \in \Gamma$ such that $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2$ and $i_1 = i_2$ then $p_1 \cap p_2 = \phi$.*

Property 3 (Observable) *The model is observable, i.e., for $t_1, t_2 \in \Gamma$ such that $t_1 = (q_1, q_1', i_1, o_1, p_1, f_1)$, $t_2 = (q_2, q_2', i_2, o_2, p_2, f_2)$ and $t_1 \neq t_2$, if $q_1 = q_2$ and $i_1 = i_2$ then $o_1 \neq o_2$.*

For a given state, input and input parameter value, we determine the target state, output and output parameter value in a transition with the help of functions δ, λ and σ, respectively. The functions are defined as follows.

- $\delta : Q \times I \times D_I \longrightarrow Q$ is the transition function
- $\lambda : Q \times I \times D_I \longrightarrow O$ is the output function
- $\sigma : Q \times I \longrightarrow {D_O}^{D_I}$ is the output parameter function. ${D_O}^{D_I}$ is the set of all functions from D_I to D_O.

The properties 1 and 2 ensure that δ and λ are mappings. When a PFSM is in state $q \in Q$ and receives an input $i \in I$ along with a parameter value $x \in D_I$, then a transition (q, q', i, o, p, f) is enabled, in which $x \in p$, and the target state $q' = \delta(q, i, x)$, the output $o = \lambda(q, i, x)$ and the output parameter value $f(x) = \sigma(q, i)(x)$.

For $i \in I$ and $x \in D_I$, we write $i(x)$ the association of the input symbol i with the input parameter value x. For an input symbol string $\omega = i_1 \cdots i_k \in I^*$ and an input parameter value string $\alpha = x_1 \cdots x_k \in {D_I}^*$, we define a parameterized input string, i.e., the association of ω and α as $\omega \otimes \alpha = i_1(x_1) \cdots i_k(x_k)$, where $|\omega| = |\alpha|$. The association of an output symbol string and an output parameter value string is defined analogously. Then, for the state $q_1 \in Q$, when applying a parameterized input string $\omega \otimes \alpha$, the machine moves successively from q_1 to the states $q_{j+1} = \delta(q_j, i_j, x_j), 1 \leq j \leq k$. We extend the functions to parameterized input strings when applying $\omega \otimes \alpha$ on q_1 as $\delta(q_1, \omega, \alpha) = q_{k+1}$ to denote the final state, $\lambda(q_1, \omega, \alpha) = o_1 \cdots o_k$ to denote the complete output symbol string and $\sigma(q_1, \omega)(\alpha) = y_1 \cdots y_k$ to denote the complete output parameter value string, where each $o_j = \lambda(q_j, i_j, x_j)$ and $y_j = \sigma(q_j, i_j)(x_j)$.

An example of a PFSM model over the sets $I = \{a, b\}$, $O = \{s, t\}$, $D_I = D_O = \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers, is shown in Figure 2.3.

2.2 Quotient of Finite State Machines

A quotient of a state machine is an approximation with respect to a certain equivalence relation. In our learning and testing context, we consider the equivalence relation of states with respect to the observed traces of the machine, denoted by $\cong_\Phi$. The equivalence relation and the quotients of Mealy machines and PFSM models are defined in the subsections.

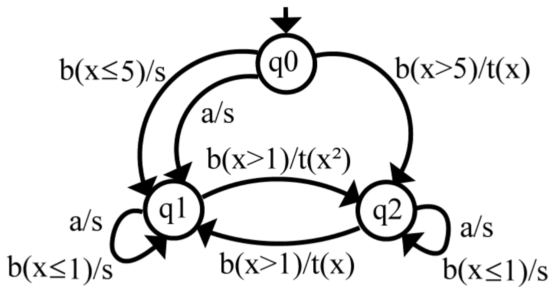


Figure 2.3: Example of a Parameterized Finite State Machine

2.2.1 Quotients for Mealy Machines

The state equivalence relation for a Mealy machine is defined as follows.

Definition 5 Let $(Q, I, O, \delta, \lambda, q_0)$ be a Mealy machine and $\Phi \subseteq I^*$ be a set of input strings, then the states $q, q' \in Q$ are Φ-*equivalent*, denoted by $q \cong_\Phi q'$, if and only if $\lambda(q, \omega) = \lambda(q', \omega)$, for all $\omega \in \Phi$. $\square$

A quotient of a Mealy machine is defined as follows.

Definition 6 Let $M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ and $\overline{M} = (Q_{\overline{M}}, I, O, \delta_{\overline{M}}, \lambda_{\overline{M}}, q_{0\overline{M}})$ be two Mealy machines and $\Phi \subseteq I^*$ be a set of input strings. Then, $\overline{M}$ is a Φ-*quotient* of M if and only if

1. $Q_{\overline{M}} \subset 2^{Q_M}$ such that $q_{0M} \in q_{0\overline{M}}$ and if $s \in q_{\overline{M}}$ and $t \in q'_{\overline{M}}$, for $q_{\overline{M}}, q'_{\overline{M}} \in Q_{\overline{M}}$ then $q_{\overline{M}} = q'_{\overline{M}}$ if and only if $s \cong_\Phi t$.
2. For all $q_{\overline{M}} \in Q_{\overline{M}}$ there exists $s \in q_{\overline{M}}$ such that for all $\omega \in \Phi$, $\lambda_M(s, \omega) = \lambda_{\overline{M}}(q_{\overline{M}}, \omega)$. $\square$

If Φ is the set of all the strings from I of certain length k, then the state equivalence relation for Mealy machines is called k-*equivalence*, and a quotient defined on such Φ is called k-*quotient*. The definitions are given as follows.

Definition 7 Let $(Q, I, O, \delta, \lambda, q_0)$ be a Mealy machine and $\Phi = I^k$ be the set of all the input strings from I of length k, then the states $q, q' \in Q$ are k-*equivalent* if and only if $\lambda(q, \omega) = \lambda(q', \omega)$, for all $\omega \in \Phi$. $\square$

A k-quotient of a Mealy machine is defined as follows.

Definition 8 Let $M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ and $\overline{M} = (Q_{\overline{M}}, I, O, \delta_{\overline{M}}, \lambda_{\overline{M}}, q_{0\overline{M}})$ be two Mealy machines and $\Phi = I^k$ be the set of all the input strings from I of length k. Then, $\overline{M}$ is a k-*quotient* of M if and only if

1. $Q_{\overline{M}} \subset 2^{Q_M}$ such that $q_{0M} \in q_{0\overline{M}}$ and if $s \in q_{\overline{M}}$ and $t \in q'_{\overline{M}}$, for $q_{\overline{M}}, q'_{\overline{M}} \in Q_{\overline{M}}$ then $q_{\overline{M}} = q'_{\overline{M}}$ if and only if s and t are *k-equivalent*.

2. For all $q_{\overline{M}} \in Q_{\overline{M}}$ there exists $s \in q_{\overline{M}}$ such that for all $\omega \in \Phi$, $\lambda_M(s, \omega) = \lambda_{\overline{M}}(q_{\overline{M}}, \omega)$. □

2.2.2 Quotient for PFSM models

The state equivalence relation for a PFSM model is defined as follows.

Definition 9 Let $(Q, I, O, D_I, D_O, \Gamma, q_0)$ be the PFSM model and $\Phi \subseteq \{\omega \otimes \alpha | \omega \in I^*, \alpha \in {D_I}^*, |\omega| = |\alpha|\}$ be a set of parameterized input strings, then the states $q, q' \in Q$ are Φ-*equivalent*, denoted by $q \cong_\Phi q'$, if and only if $\lambda(q, \omega, \alpha) = \lambda(q', \omega, \alpha)$ and $\sigma(q, \omega)(\alpha) = \sigma(q', \omega)(\alpha)$, for all $\omega \otimes \alpha \in \Phi$. □

The quotient of a PFSM model is defined as follows.

Definition 10 Let $P = (Q_P, I, O, D_{IP}, D_{OP}, \Gamma_P, q_{0P})$ and $\overline{P} = (Q_{\overline{P}}, I, O, D_{I\overline{P}}, D_{O\overline{P}}, \Gamma_{\overline{P}}, q_{0\overline{P}})$ be two PFSM models. Let $D_{I\overline{P}} \subseteq D_{IP}$, $D_{O\overline{P}} \subseteq D_{OP}$ and $\Phi \subseteq \{\omega \otimes \alpha | \omega \in I^*, \alpha \in {D_{I\overline{P}}}^*, |\omega| = |\alpha|\}$ be a set of parameterized input strings. Then, $\overline{P}$ is a Φ-*quotient* of P if and only if

1. $Q_{\overline{P}} \subset 2^{Q_P}$ such that $q_{0P} \in q_{0\overline{P}}$ and if $s \in q_{\overline{P}}$ and $t \in q'_{\overline{P}}$, for $q_{\overline{P}}, q'_{\overline{P}} \in Q_{\overline{P}}$ then $q_{\overline{P}} = q'_{\overline{P}}$ if and only if $s \cong_\Phi t$.

2. For all $q_{\overline{P}} \in Q_{\overline{P}}$ there exists $s \in q_{\overline{P}}$ such that for all $\omega \otimes \alpha \in \Phi$, $\lambda_P(s, \omega, \alpha) = \lambda_{\overline{P}}(q_{\overline{P}}, \omega, \alpha)$ and $\sigma_P(s, \omega)(\alpha) = \sigma_{\overline{P}}(q_{\overline{P}}, \omega)(\alpha)$.

2.3 General Notations

Some of the general notations which are used in the manuscript are listed below.

$\mathcal{L}(D)$ denotes the language of the DFA D.

$\otimes$ associates a string from one set with a string of another set. Let $\{a, b, c\}$ and $\{1, 2, 3\}$ be two sets, then the string $a \cdot b \cdot c$ is associated with the string $1 \cdot 2 \cdot 3$ as $a \cdot b \cdot c \otimes 1 \cdot 2 \cdot 3$, which is equal to $(a, 1) \cdot (b, 2) \cdot (c, 3)$. See Definition 4.

Ω denotes the absence of the output. If a finite state machine does not produce any output to a particular input, then the output to the input is Ω. See Definitions 2 and 4.

$\perp$ denotes the absence of the parameter value. In PFSM model, if an input symbol i does not take any input parameter, then i is associated with $\perp$, i.e., $i \otimes \perp$. See Definition 4.

Φ denotes a finite set of strings. See Definitions 5, 7 and 9.

$\cong_{\Phi}$ denotes the equivalence relation of the states of the finite state machines with respect to Φ. Let Φ be the set of input strings, then two states are equivalent with respect to Φ if they produce same output strings for all the input strings in Φ. See Definitions 5, 7 and 9.

$\mathit{pref}^k(\omega)$ denotes the prefix of the string ω of length k. Let $\omega = a \cdot b \cdot c \ldots z$ be a string, then $\mathit{pref}^3(\omega) = a \cdot b \cdot c$.

$\mathit{suff}^k(\omega)$ denotes the suffix of the string ω of length k. Let $\omega = a \cdot b \ldots x \cdot y \cdot z$ be a string, then $\mathit{suff}^3(\omega) = x \cdot y \cdot z$.

$IS(v)$ denotes the input symbol string from a (parameterized) input string v. This function is used in PFSM models. Let $I = \{a, b, c\}$ and $D_I = \{1, 2, 3\}$. If $v = a \cdot b \cdot c \otimes 1 \cdot 2 \cdot 3$, i.e., a parameterized input string, then $IS(v) = a \cdot b \cdot c$. If $v = a \cdot b \cdot c$, i.e., a non-parameterized input string, then $IS(v) = a \cdot b \cdot c$.

Chapter 3

Background and Related Work

This chapter overviews the background work and surveys the state-of-the-art in the domain of automata learning and testing. It presents the notable works in passive learning and testing, active learning and testing and learning enhanced state models. It finally discusses the potential subareas that are addressed in the thesis.

3.1 Introduction

The research in the behavior inference of software systems has extensive body of literature. Most of the works is related to *program understanding* in the case of the availability of source code. For example, Ernst et al. [ECGN01] developed a tool, called *Daikon*, to infer program invariants over a set of monitoring variables in the running program. Walkinshaw et al. [WBH06] developed a method to extract state transitions from source code and identify the statements which trigger specific transitions. When source code is not available, researchers look for partial specifications and build their understanding on the inference of component interactions by applying tests on the system that are derived from the specifications. For example, Abdurazik and Offutt [AO00] used UML collaboration diagrams for generating test suites that cover the interaction patterns in the system. Wu et al. [WPC01] derived *Component Interaction Graph (CIG)* that captures interactions and dependencies among components by probing their interfaces and consulting available information. When considering the black box components where no source code and formal specifications are available, most approaches rely on inference from the system execution and deriving the formal models from the observations. A promising approach for the behavioral inference of such systems is applying the automata learning techniques and combining it with automata based testing techniques. Our work mainly focuses on this approach and we survey the significant works in automata learning and testing domain in this chapter.

The theoretical research in automata learning has been vibrant for decades, but there are not many studies with respect to its practical orientation, even in its general context. Only in the past few years, there are some efforts for providing a framework for combining automata learning and testing techniques and dealing with real applications. This research is generally divided into two wide categories, i.e., passive and active. In *passive learning*, there is no control over the observations we receive to learn the model. In *active learning*, there is a liberty to experiment or query the black box machine to collect observations and then learn the model. In the following sections, we discuss the concepts and some notable works under these categories, and later we point out the potential subareas that are addressed in the thesis.

3.2 Passive Automata Learning and Testing

The concept of passive learning is to learn the model from a given set of observations. The essence of this concept is that we are bound to learn from what we are given. The observations could be randomly collected from the black box machine and then we can build an algorithm to estimate a model from these observations. In the following, we present the theoretical results of passive automata learning and then applications of this concept in learning and testing of the black box systems.

3.2.1 Theoretical Results

The research in automata learning finds its roots in the grammatical inference works. Gold [Gol72] was the first who laid down the theoretical framework for analyzing the grammatical inference problem. The problem can roughly be stated as to infer a grammar that represents an unknown regular language from a given set of samples. The samples are the strings from the alphabet which are the words of the language, called positive samples, and possibly not in the language, called negative samples. Such a grammar can be represented as a regular expression or as a Deterministic Finite Automaton (DFA). Therefore, the problem has been referred as a regular inference or an automata learning problem [dlH05]. Gold [Gol78] showed that finding a minimum DFA from a set of positive and negative samples is NP-Complete. The research was continued to find methods for approximate learnability that could learn concepts with a high probability but in low complexity. The variety of models like Probably Approximately Correct (PAC) model [Val84] and the mistake-bound models [Lit87], [HLW94] were proposed. But the results imply the hardness of the problem.

Other techniques have been proposed for the inference of finite state automata, some of them based on recurrent neural network architectures [Pol91] [GMC+92] and Markov models

[BRH04]. Although these methods may exhibit, from a conceptual standpoint, some advantages; the results [HG94] [CW98] show that they are not competitive in inference problems.

3.2.2 Applications

The above approaches were based on passive learning that relies on the given information to learn a model. There is much work on describing the computational complexities of various passive learning models (see surveys [AS83],[Pit89]). In the following years, the research evolved from studying the theoretical complexities to evaluating the practical performance of the learning methods. In the passive learning context, several algorithms based upon the state merging approach (see survey [BO05]) and the genetic algorithms (e.g., [Dup94],[LCJ06]) were proposed which were experimented on the standard DFA benchmarks. Cook and Wolf [CW98] compared the algorithmic and statistical methods of passive learning to infer the software processes from event based data. Despite the discouraging theoretical results of passive learning approach, there exists applications of this approach in the behavior inference of black box systems by mixing it with testing techniques. The intuition was to apply inference techniques on application domains and assess their practicality. In the following, we discuss some of the recent works under this context.

Mining Specifications

Ammons et al. [ABL02] proposed *mining specifications* approach for discovering the formal models of the black box systems. They observe the system execution and concisely summarize the frequent interaction patterns of components in the system as state machines that capture both temporal and data dependencies. The inferred models can be used to increase system understanding and can be applied for verification and debugging.

The mining approach consists of four parts: tracer, flow dependence annotator, scenario extractor and automaton learner. The tracer records the interactions of components how they interact with an Application Programming Interface (API) or Abstract Data Type (ADT) of each other. The tracer records only function calls and returns, although depending on the API/ADT, it allows tracing other events such as network messages in the system. The flow dependence annotates the traces with constraints how the interactions may be reordered and identify related interactions that could form scenarios. The scenario extractor extracts the scenarios, i.e., a small set of interdependent interactions from annotated traces, and prepares them for the automaton learner. The automaton learner analyzes the extracted scenarios and infers a probabilistic finite state automaton (PFSA) using a passive learning algorithm, called sk-string method [RP97]. A PFSA is a nondeterministic finite automaton (NFA) in which each edge is labeled by an abstract interaction and weighted by how often the edge is traced generated or accepting scenarios. Rarely-used edges correspond to infrequent behaviors, which are later on removed in post processing along with all weights, leaving a NFA.

Detecting Differences in System Versions

Mariani and Pezz [MP05] proposed a *behavior capture and testing* approach for detecting the differences in two system versions. The approach consists in first collecting the observations about component interactions during testing and field execution, and then monitoring the interactions for new component versions or for existing components in the new software system, to detect differences with respect to the previously observed behaviors. The observed differences provide information about both new behaviors that may correspond to new requirements, and misbehaviors that may correspond to unexpected erroneous interactions.

The observations are collected through object-flattening technique, which consists of identifying the set of methods (or APIs) to extract state information semi-automatically. Those observations are used to calculate two classes of invariants , i.e., I/O invariants and interaction invariants. I/O invariants describe the relation between data exchanged among components and are computed with Daikon [ECGN01]. Interaction invariants describe the interactions among components, and are computed by synthesizing finite state machines. They provide a passive learning algorithm to learn the state machine using the collected observations. The calculated invariants are then checked at runtime in the new system. The invariant violations are interpreted by the designers to analyze the behaviors of the new system.

System Verification

Bertolino et al. [BMP06] proposed an approach of verifying system architecture specifications (which describes the components, connectors, interfaces and ports) with the collected observations of the system (which specify the actual behaviors of the components). The approach works by first capturing the traces of executions and transferring the traces into a formal model. The model is then checked for compliance with the system architecture specifications, using a model checker.

The specifications of the system architecture are realized in terms of stereotyped UML 2.0 component diagrams. The observations are collected by instrumenting the middleware and monitoring the component interactions in the running system. The interactions are then realized into Message Sequence Charts (MSC). The verification is done by applying a tool, called Charmy [IMP05]. Charmy translates the specifications into Promela (language of the model checker SPIN) [Hol03]. Then, it translates the MSCs into Büchi Automata [Büc62] (the automata representation for temporal formula [Eme90], which describe properties to be verified). Finally, SPIN evaluates the properties validity with respect to the Promela code. If counterexamples are produced, an error is reported.

3.3 Active Automata Learning and Testing

The concept of active learning is to collect observations by asking queries and receiving responses. It then learns a model based upon those observations. This is different from passive learning in which the model is learned only based upon the given observations. In active learning, the ability to ask queries helps to elucidate conflicts in the inference results. In the following, we present the theoretical results of active automata learning and then applications of this concept in learning and testing of the black box systems.

3.3.1 Theoretical Results

Pitt and Warmuth [PW89] have shown that although passive learning approach is apparently intractable, the combination of active and passive learning is feasible. Kearns and Vazirani [KV94] is a good reference to have an overview on the computational complexities of the passive and active learning approaches. Active learning is considerably a better approach in which the black box system is explored (in a systematic way) in finite time and then an inference is done based on the observations from the system with interesting properties. In this framework, Angluin [Ang87], elaborated on the algorithm of Gold [Gol72], proved that if a learning algorithm uses queries to collect observations and obtains clues for the target model, then finite automata can be learned in polynomial time. In the settings of grammatical inference, Angluin provided an efficient algorithm to learn a minimum DFA that models an unknown regular language. Angluin's algorithm plays an important role in our work. We shall describe the complete algorithm with its complexity analysis in Chapter 4. Here we provide a brief sketch of the algorithm.

Angluin's algorithm asks *membership queries* over the known alphabet Σ of the language to check whether certain strings from Σ^* are accepted or rejected by the language. The result of each such query is recorded as an observation. These queries are asked iteratively until some conditions are satisfied on the collective observations. The algorithm estimates a DFA, called conjecture, based on the recorded observations. It then asks an *equivalence query* to a so called *oracle*, that knows the unknown language, to verify whether the conjecture is equivalent to the target DFA. The oracle validates the conjecture if it is correct or replies with a counterexample otherwise. The algorithm uses this counterexample to perform another run of asking membership queries until it constructs a "better" conjecture. It iterates in this fashion until it produces a correct conjecture that is isomorphic to the target DFA. Let $|\Sigma|$ be the size of the alphabet Σ, n be the total number of states in the target DFA and m be the length of the longest counterexample provided by the oracle, then the worst case complexity of Angluin's algorithm is $O(|\Sigma|mn^2)$.

Note that there is no polynomial time algorithm if we allow only membership queries and require that the target DFA be exactly learned [Ang81]. In other words, we must have a mechanism to check the equivalence of the conjecture and the target model. Angluin [Ang87] used the concept of the oracle that resolves this equivalence check. The oracle is a theoretical construction to make an idealization of a potentially hard problem, in order to provide a clean setup in regular inference. In reality, there exists no such oracle. The alternatives of the oracle assumption demand a compromise on precision and cost. There are several testing techniques that can be applied to check this equivalence and obtain counterexamples if the conjectured model is different from the target model. For example, one can perform a random walk over the strings of alphabet and test each string on the conjecture and on the black box to detect the differences [RS93] [SL07] [SHL08]. But this may provide long counterexamples that can influence the complexity of the algorithm negatively. Moreover, random testing does not provide a guarantee on the exact learning. Another way is to use the methods from conformance testing [LY96] that can provide a systematic way of achieving the answer of an equivalence query. Let d be the number of states in the conjecture and assume that we know some upper bound l on the number of states in the black box such that $l > d$, then by applying the tests in a conformance test suite by Vasilevskii and Chow algorithm (aka VC-algorithm or W-method [Vas73][Cho78]) to the black box, we shall find at least one test that can detect its difference with the conjecture. This test constitutes a counterexample as the answer of an equivalence query. The worst case complexity of this method is $O(d^2l|\Sigma|^{l-d+1})$, that is, exponential in the difference between the upper bound on the number of states of the black box and the conjecture. The other conformance testing techniques that can also be used are Wp [FBK$^+$91] and Z [LY96] methods, which use a smaller test suite compared to W-method.

3.3.2 Applications

Despite some practical problems in Angluin's algorithm, this is considered a remarkable work and has been applied in various domains, for instance, map learning [RS93], behavior modeling [MS01a] [HMS03] [RSM07], model checking [PVY99] [SL07], testing [HNS03] [SHL08] etc. The applications of Angluin's algorithm in real black box learning and testing problems, combined with its improvements and statistical analysis [BJLS05], paves the way toward reverse engineering complex software systems. In the following, we discuss some of the recent and significant works under this context.

Black Box Checking

Peled with other researchers has produced a series of papers that provide a solution of model checking black box systems. Black Box Checking (BBC) [PVY99] is the first and the pioneering paper, according to our knowledge, which proposed to combine active learning and model checking under one framework. The problem which is addressed is: a prerequisite for model checking is the provision of a model, which black box system does not provide. BBC proposes to learn the model first, then perform model checking on the learned model. The algorithm used for learning is Angluin's algorithm, and the equivalence check is performed by VC-algorithm. Therefore, it is assumed that the upper bound l on the number of states in the system is known.

BBC is an iterative procedure which is described as follows. First, the model of the system is learned through Angluin's algorithm. Once a model is learned, it is provided to a model checker for verifying the given property. This yields one of the following two possibilities.

1. If the model checker produces a counterexample, i.e., the model does not satisfy the property, then the counterexample is confirmed on the system if it is indeed a counterexample for the system. Recall that the model checking is performed on the approximation of the system, and not on the system directly.

    - If the system confirms the counterexample, i.e., the system does not satisfy the property, then the counterexample is reported and the procedure terminates.
    - If the system refutes the counterexample, then the counterexample is given to the learning algorithm to refine the model.

2. If the model checker does not produce a counterexample, i.e., the learned model satisfies the property, then the model is checked for equivalence with the system. For this purpose, VC-algorithm is applied to look for discrepancy between the current approximation and the system.

    - If VC-algorithm finds such a sequence that can distinguish the system with the current model, then the sequence is a counterexample for the model, which is refined by giving the counterexample to the learning algorithm.

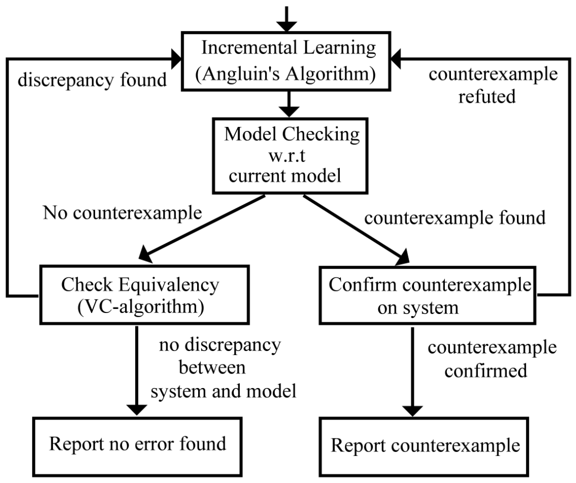

Incremental Learning (Angluin's Algorithm)
discrepancy found
counterexample refuted
Model Checking w.r.t current model
No counterexample
counterexample found
Check Equivalency (VC-algorithm)
Confirm counterexample on system
no discrepancy between system and model
counterexample confirmed
Report no error found
Report counterexample

Figure 3.1: The learning and model checking procedure of Black Box Checking

- If VC-algorithm finds no counterexample, i.e., the model is completely learned and trivially the system satisfies the property, then the procedure terminates.

The BBC procedure is summarized in Figure 3.1. The worst case of this procedure arrives when the system satisfies the property and the model is completely learned. Assuming that the total number of states in the system is n (unknown but smaller than the known upper bound l), then the worst case complexity of the BBC procedure is given as $O(n^3|\Sigma|^n + n^2l|\Sigma|^{l-n+1})$. Note that if we do not know the bound l, then the learning algorithm can run until time permits [PVY99].

The next paper of the same series is Adaptive Model Checking (AMC) [GPY02], in which the authors assume an existing model that may be inaccurate but not completely obsolete. The experiments of AMC showed that initializing the learning algorithm with the existing information expedite the performance of the algorithm. The last paper of this series is Grey Box Checking (GBC) [EGPQ06], in which the authors assume that some components in the system are known (white boxes), and the rest are unknown (black boxes), giving rise to a grey box system. The experiments of GBC showed that the speedup over BBC can be up to two order of magnitude. The experiments of AMC and GBC concluded that the average case complexity for model checking real systems can be improved depending upon how much information about the system is available. However, the exponential time complexity cannot be avoided in the worst case.

Protocol Learning and Testing

Shu and Lee [SL07] used active learning techniques to validate the security properties of the protocol whose implementation is unknown. Their work is inspired by the Black Box Checking work [PVY99] in which the model is learned through Angluin's algorithm, then the property is checked on the learned model, followed by applying a conformance testing method for the equivalence check. They assume that the protocol specification is given from which the security properties can be extracted and checked against the unknown implementation. However, the implementation may contain more behaviors than the given specification. Therefore, they also intend to learn the implementation in the iterative procedure and check the security property on the incrementally learned models. The protocol specification is modeled as a *Symbolic Parameterized Finite State Machine (SP-EFSM)* that is a compact representation of an equivalent Mealy machine. They modify Angluin's algorithm to learn directly the Mealy machine instead of a DFA model. For equivalence check, they implement a method that generates random checking sequences of the given length up to t times. We explain their learning and testing procedure as follows.

First, the black box protocol implementation is learned through Angluin's algorithm. Once a model is learned, it is checked for security violations against the specification. As in the BBC procedure, this yields one of the following two possibilities.

1. If a violation is found, i.e., there is a counterexample that leads the model to violation, then the counterexample is confirmed on the black box.

    - If the black box confirms the counterexample then the procedure terminates with result "FAIL".
    - If the black box refutes the counterexample, then the counterexample is given to the learning algorithm to refine the model.

2. If no violation is found, then the learned model is checked for equivalence using their method of generating random checking sequences.

    - If the equivalence method finds such a sequence that can distinguish the model with the black box, then the sequence is a counterexample for the model, which is refined by giving the counterexample to the learning algorithm.
    - If the equivalence method finds no counterexample, then the procedure terminates with result "PASS". Note that in this case, it does not necessarily mean that the last

learned model is equivalent to the black box; instead it means no further discrepancies can be found between the model and the black box by their equivalence check method.

Let $|I|$ be the size of the input set of the Mealy machine, n be the total number of states in the machine and t be the number of times random checking sequences are generated, then the total complexity of their protocol learning and testing procedure is $O(|I|n^4+tn^2+|I|n^3+nf(d))$, where $f(d)$ is the cost of validating the security for a conjecture with d states.

Web Application Exploration

Raffelt et al. [RSM07] used learning and testing approach in the exploration of a web application. They studied Mantis [Man07], an open-source online bug tracking system, and analyzed the user authentication module of the application. Their approach is very similar to the works described above, with a difference is that they do not check any property on the system; rather their intention is to understand the working of the web application. They learned how the application behaves if a user enters without login and what parts of the application are accessible if the user is authenticated.

They adapted Angluin's algorithm to learn Mealy machines of the system. Initially, the input set of the system is manually provided to the learning algorithm, but new inputs may be discovered during the learning process (in terms of links, forms, web pages etc). Therefore, they dynamically enhance the input set whenever new inputs are discovered, and relearn the model with the new input set. When no more inputs are discovered, then the equivalence check is performed on the last learned model by applying the Wp-method [FBK$^+$91]. If the learned model does not conform to the real system, a counterexample is returned, and thereafter, the model is refined and the iterative procedure starts again. The procedure terminates when the equivalence check does not provide a counterexample, which means that the model is completely learned.

Reverse Engineering by combining Passive and Active Approaches

Walkinshaw et al. [WBHS07] used a combination of passive and active learning approaches for reverse engineering state models. Their technique is based upon Dupont's Query-driven State Merging (QSM) inference algorithm [DLD$^+$08], which espouses the features of both approaches. The working of QSM algorithm is sketched as follows.

The QSM algorithm accepts a set of positive and negative samples as input. Then, it represents the samples as a prefix tree automaton, in which each state is either accepting or rejecting depending upon the corresponding sample. Then, it searches for candidate states to merge, followed by merging those states. The resulting automaton is actually a conjecture which may accept or reject more strings that are not handled by the previous automaton (initially the tree automaton). Therefore, the algorithm generates membership queries from the conjecture to check the validity of those strings. If all queries are validated, then the conjecture is returned, otherwise the samples are updated with the strings that are not validated and the QSM procedure is called again. The algorithm runs until no queries are generated.

The authors applied QSM technique in learning an open-source Java drawing tool, called *JHotDraw*[1]. The samples is this case are the traces of method invocations that are accepted or rejected by the application. First, they develop a set of mappings from the sequences of method invocations in the traces to abstract functions, such that, a low level dynamic trace can be lifted to a series of high-level functions. Then, each trace is made into a string of abstract functions and fed into the QSM algorithm. If the algorithm generates queries, they are asked to the application. For the queries which are not validated, their corresponding traces are fed back into the QSM algorithm and the process repeats until no further queries are generated.

Domain Specific Optimizations

Hungar et al. [HNS03][HMS03] studied the domain specific applications of Angluin's algorithm and evaluated its practicality. They noticed that Angluin's algorithm can be optimized if it learns prefix-closed languages. A language $\mathfrak{L}$ is prefix-closed if for every string ω in $\mathfrak{L}$, all prefixes of ω are in $\mathfrak{L}$. A DFA is called prefix-closed if its language is prefix-closed. So they proposed improvements in the algorithm for the problems that can be modeled as prefix-closed DFAs.

The optimizations are realized in terms of *filters* that parse the membership queries of Angluin's algorithm before asking to the real system. The filters select few queries among the generated queries to ask to the system. The rest of the queries can be answered automatically according to the result of the asked queries. For example, a prefix of an accepted string ω will always be accepted in a prefix-closed language. So, it does not need to ask the prefixes; instead asking ω is sufficient. Another example is of a string that is rejected by the language. So the suffixes of such a string will always be rejected. In this way, the number of membership queries can be reduced in the case of prefix-closed DFAs. Their experiments with the randomly generated prefix-closed DFAs achieved 20 % reduction of membership queries compared to the basic algorithm.

The authors also applied their improvements in the testing of a call center system. They were mostly concerned with the inference of the behaviors of the system on different call scenarios.

[1]http://www.jhotdraw.org

Originally, they modeled their problem as a Mealy machine but then transformed it into an equivalent prefix-closed DFA, so that Angluin's algorithm could be applied, incorporating their optimizations.

In the continuation of the same work, Margaria et al. [MNRS04] proposed to learn Mealy machines directly by adapting Angluin's algorithm and compared the results of the previous work of prefix-closed DFA learning with direct Mealy machine learning. Their results showed a further reduction in the membership queries when learning Mealy machines directly, compared to the previous results. The authors commented that this is due to the fact that Mealy machines model more naturally the i/o behavior of the system, which DFAs do not support the structure directly. DFAs require an encoding in terms of artificial transitory states which increases the model size in terms of the number of states. Since, the complexity of Angluin's algorithm is polynomial on the number of states, Mealy machine learning experiences quite less number of queries due to its reduced state size compared to the equivalent DFA.

3.4 Learning Enhanced State Models

The domain specific studies and the applications of Angluin's algorithm, for example in the learning and testing of telecom systems [HNS03] [HMS03] [MNRS04], in the exploration of web applications [RSM07] and in constructing the models of protocols [SL07] [BJR06], advocate for learning more enhanced state machine models than simple DFAs. The main reasons are: i) Basic DFA models do not capture the fine granularity of a complex system. They can be used in the context of model checking because it builds reachability graphs akin to DFA. On the other hand, when considering complex systems which have input and output behaviors or which are composed of components that exchange lots of parameterized data values from arbitrary domains (like protocol data unit), DFA modeling could result in a combinatorial blow up on the transition level. There is a need of rich structure that is more expressive and capable of modeling such systems without losing generality. ii) In order to mitigate the computational complexity of the algorithm, it is useful to enclose the (possibly infinite) domains of data values of the system into some sort of parameterized structure. The number of queries in the learning algorithm grows with the size of the data values. But usually such systems show similar behaviors on a subset of those values. If some values are irrelevant or never used, the learning algorithm may still work without taking them into account.

There have been several trials of learning enhanced models using the settings of Angluin's algorithm. The best part of this algorithm is that it can be easily adapted to learn Mealy machines. This adaptation has been successfully applied in many works, e.g., [MNRS04] [RSM07]

[SL07]. Actually, learning Mealy machines is an intermediary level between learning naive models and expressive models like *Extended Finite State Machines (EFSM)* [LY96]. For example, Shu and Lee [SL07] modeled their protocol specifications as SP-EFSM, but they rely on learning its equivalent Mealy machine. Recently, Berg et al. [BJR06] proposed a parameterized model which can be learned directly from a black box component using the original settings of Angluin's algorithm. This model preserves all the properties of DFA, plus incorporates parameters and predicates associated with the labels on transitions. However, the model does not contain outputs and output parameters. Also, it assumes only boolean space for the parameter values associated with the labels. Lorenzoli et al. [LMP06] proposed to learn *Finite State Automata with Parameters (FSAP)*, which is similar to the model of Berg et al. [BJR06]. But this is in passive learning settings in which they assume the provision of a given set of traces through which they can learn FSAP by applying a state merging algorithm, called *k-tail* [BF72]. The algorithm allows merging states in the traces which have same k future, i.e., that are followed by the same behaviors up to a depth of k steps.

3.5 Discussion

We have laid out the background for the research developed in this thesis. We summarize the bibliographic study of the domain and point out the potential subareas that are addressed in the thesis.

3.5.1 Summary

The problem of behavior inference is in general difficult. Many techniques have been proposed to derive formal models of the system behaviors under different contexts. When source code is available, the models are extracted by directly deeming into the implementation. When (partial) specifications are available, the interactions of the real components are modeled by deriving tests from the specifications. When considering the behavioral inference of the black box systems, the problem becomes quite challenging. Previously, passive learning approaches have been used for the purpose of inferring models and testing the properties on the systems. However, it is observed that active learning approaches provide quite better results than the passive ones. In this vein, Angluin's algorithm [Ang87] has been considered a remarkable work which has been applied to various domains for learning and testing real applications. It is evident that the theoretical complexity of Angluin's algorithm with the alternatives of its oracle assumption is still intractable [PVY99]. On the contrary, the experimental results of the application of the learning and testing approach using Angluin's settings evidence encouraging results where the

Works	**Objective**	**Original Model**	**Learned Model**
Peled et al. [PVY99]	Model Checking	DFA	DFA
Shu and Lee [SL07]	Inference and Testing	SP-EFSM	Mealy Machine
Raffelt et al. [RSM07]	Inference	Mealy Machine	Mealy Machine
Walkinshaw et al. [WBHS07]	Inference	DFA	DFA
Hungar et al. [HNS03]	Inference and Testing	Mealy Machine	DFA
Margaria et al. [MNRS04]	Inference	Mealy Machine	Mealy Machine
Berg et al. [BJR06]	Inference	Parameterized Machine	Parameterized Machine

Table 3.1: Summary of the works in Active Learning and Testing Approach

real world systems have been considered [HNS03] [HMS03] [BJLS05] [RSM07] [SL07] [SHL08]. Table 3.1 provides the list of works in the active learning and testing approach, in the order of their appearance in the chapter. The first column labels the works, the second column mentions the objective of the works, the third column mentions the original model in which the authors modeled their problem, the fourth column mentions the model which they actually learned.

3.5.2 Our Work in Learning and Testing

Our work is complementary to the active learning and testing approach that has been addressed previously, notably in black box checking [PVY99]. In fact, we consider a system of black box components that are communicating with each other, and then analyze the whole system using the learned models of the individual components. However, instead of applying system verification techniques to check the given user-defined properties, we test the system for finding the compositional problems such as deadlock and livelock. In the absence of formal requirements, we look for generic errors in the system that can cause failures during its execution. Moreover, the availability of observations before starting the learning procedure, as in [GPY02] [EGPQ06] [WBHS07], is not assumed in our work.

We exploit the use of Angluin's algorithm to learn each component in isolation and later compute the product of the learned models. However different from other approaches [PVY99] [SL07] [RSM07], we do not assume the upper bound on the number of states in the system to apply conformance testing methods as a replacement of the equivalence check. The accurate estimation of such a bound in the real system is hard which can explode the complexity of the learning procedure when applying conformance testing method for the equivalence check. In our work, we take benefit of the product of the learned models to derive tests for the equivalence check of the integrated system. These tests from the product most likely visit the unexplored parts of the components, since the product represents an approximation of the whole system, and therefore these tests can stimulate interactions between the components. This means the tests from the product can exercise those parts of the components which are relevant to the integrated system and might not be explored during their unit learning. We argue that this approach can alleviate the oracle problem, since the same product of models from which the tests are generated can act as an oracle. The problem is the partiality of the models which may not fully describe the internal structure of the components. However, partial models are accepted as a viable solution to oracle automation in the black box testing framework [Ber07]. Weyuker [Wey82] calls such models "pseudo-oracles" and notes that it is sometimes much easier to distinguish plausible from implausible results than to precisely distinguish correct from incorrect results. The challenge is to find the best trade off between precision and cost. In any case, the model learned from the component is an approximation of the real model. It is necessary to state the formal relation between what we learn from the component and what is the reality. In the case of state machines, we show that the approximated model can be described as a *quotient*[1] of the real model, which is defined according to some equivalence relation on states. So, we propose a framework of leaning and testing the integrated system of black box components, by learning the quotients of the components, and then deriving tests from the quotients to test the integrated system.

Learning Enhanced State Models

The other important area we addressed is the learning of enhanced state models. As discussed before, it is quite desirable to devise learning methods for such models. This problem is still not addressed adequately and needs more attention in terms of both theory and practice. In our work, we propose enhanced models and the algorithms for learning such models directly from the black box components.

First, we consider the adaptation of Angluin's algorithm for learning Mealy machines (like Margaria et al. [MNRS04] did for the comparison of DFA and Mealy machine learning). However, we have observed that even the adapted algorithm can be further improved such that the complexity of learning is significantly reduced. We propose our improvements and show with the help of complexity calculations and experiments that our improvements has a gain over the direct adaptation.

Later, we propose a *Parameterized Finite State Machine (PFSM)* model and the algorithm for learning such models. PFSM is more expressive compared to the models proposed in the previous works of automata inference (e.g., [BJLS05] [LMP06] [RSM07] [SL07]) in terms of parameterized inputs/outputs, infinite domain of parameter values, predicates on input parameters and observable nondeterminism when interacting with input parameter values. Compared

[1]See Chapter 2 for the definitions of quotients

to the usual EFSM model [LY96], we stop short of including variables in the model, because when we learn a black box, we cannot distinguish in its internal structure what would be encoded as (control) state and what would be encoded in variables. All state information in our model is encoded in the state machine structure.

Following the discussion on PFSM learning, it is important to mention how to select parameter values for testing during the learning procedure. DFA and Mealy machines have a finite set of inputs and the learning algorithm considers all inputs for their learning. The parameterized model of Berg et. al. [BJR06] consists of boolean parameters and therefore selecting the values is not a problem. In the FSAP model of Lorenzoli et al. [LMP06], the possible parameter values are already included in the given traces. In the case of PFSM models, the domain of parameter values can be infinite. Therefore, the selection of parameter values during the learning of a PFSM model is an issue. We assume to know only parameter types but it is unknown what concrete values the components must be given for its learning. This leads to a classic problem of test data selection in black box testing [Kor99]. There is an enormous body of literature on this specific problem and we do not refer to any specific strategy for parameter value selection. In our work, selecting the values is mostly intuitive or relies on simple techniques, e.g., random testing, bounded exhaustive testing, equivalence partitioning, boundary value analysis [MSBT04], to name a few.

3.5.3 Extended Work

We extend the work of learning PFSM models towards learning functions. In fact, the active learning of PFSM models involves in testing certain input parameter values and observing the corresponding output parameter values. Finally, it outputs a PFSM conjecture in which transitions are labeled with input/output parameter value pairs. However, it is possible that instead of labeling with just pairs, we learn meaningful relationships over the observed values and then label the transitions with those relations. Those relations are actually an approximations of the output parameter functions in the PFSM model. We extend the work of PFSM inference in this direction. In our first attempt, we propose to learn such relations as data invariants. We exploit the use of Daikon (the invariant detector) [ECGN01] by providing it the set of observations on the input/output parameter values and then inferring the data invariants over the values.

3.5.4 Main Contributions

In the light of the above discussion, we summarize the main contributions in the thesis in the order of their presentation in the manuscript.

- We have improved the Mealy machine adaptation of Angluin's algorithm and proved that our method has significantly reduced the complexity of the learning algorithm. The theoretical results are also verified on a workbench of finite state machines.
- We have proposed a framework of learning and testing integrated systems of black box components.
- We have proposed a Parameterized Finite State Machine model that can be learned using the original settings of Angluin's algorithm.
- We have extended the work of PFSM inference towards learning functions. We have proposed a method to infer output parameter functions in the PFSM model using the data invariant inference mechanism.
- We have validated our approach on the case studies from the industry in which real systems have been considered.

Chapter 4

Deterministic Finite Automaton Inference

This chapter is a continuation of the state-of-the-art, but dedicated to the inference of Deterministic Finite Automata through the active learning approach. It describes Angluin's algorithm, its complexity and its variants proposed in other works.

4.1 Learning Algorithm for DFA

A finite regular language is a subset of Σ^*, i.e., the set of finite strings of letters. A regular language can be modeled as a DFA (Definition 1), which accepts the strings from Σ^* those are included in the language and rejects all others. The regular inference problem can be seen as identifying the regular language modeled as a DFA. There are several frameworks for inferring a DFA from a black box machine which accepts a regular language. The most well-known in active learning approach, roughly called "learning from queries", was introduced by Angluin [Ang87]. She presented an algorithm, called L^*, for learning a minimum target DFA in polynomial time. The basic idea of the algorithm is to explore the system systematically by asking queries and collect the observations in the result of queries to build an automaton. The concept of learning from queries and a full literature on different types of queries is given by her continuation paper [Ang88], and the later framework paper by Watanabe [Wat94]. However, the two main assumptions in the concept that are also required by the algorithm L^* are as follows:

1. The basic alphabet Σ is known

2. The machine can be reset before each query

The algorithm asks two types of queries. A *membership query* is asked to test whether a string from Σ^* is contained in the target language. The result of each such query in terms of 1 (accepted) or 0 (rejected) is recorded as an observation. These queries are asked iteratively until some conditions are satisfied on the collective observations. L^* estimates the target DFA, called conjecture, based upon the recorded observations. It then asks an *equivalence query* to a so called *oracle* to verify the hypothetical conjecture. The oracle validates the conjecture if it is correct or replies with a counterexample otherwise. A counterexample is a string which is accepted by the target DFA but not by the conjecture, or vice versa. L^* uses this counterexample to perform another run of asking membership queries until it constructs a "better" conjecture. L^* iterates in this fashion until it produces a correct conjecture that is isomorphic to the target DFA. The basic set-up of the learning algorithm is presented in Figure 4.1.

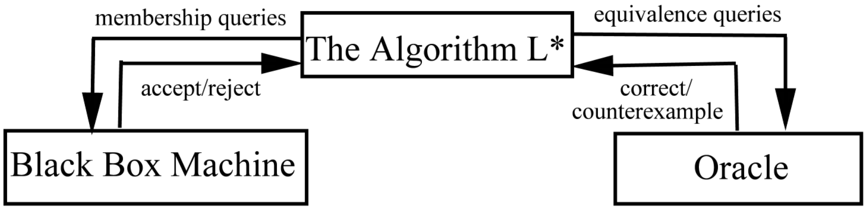


Figure 4.1: Concept of the Learning Algorithm L^*

We describe the complete algorithm L^* and its complexity in the following sections. We denote by $\mathcal{D} = (Q_\mathcal{D}, \Sigma, \delta_\mathcal{D}, F_\mathcal{D}, q_{0\mathcal{D}})$ the target unknown DFA that has a minimum number of states. The output function and the complete output function for $\mathcal{D}$ are denoted by $\Lambda_\mathcal{D}$ and $\lambda_\mathcal{D}$, respectively.

4.1.1 Observation Table

The algorithm L^* maintains a data structure, called observation table OT_D, to record the results of the queries. To describe the structure of the table, let S_D and E_D be the non-empty finite sets of finite strings over Σ. S_D is a prefix-closed set and E_D is a suffix-closed set of strings. Let T_D be a finite function that maps $(S_D \cup S_D \cdot \Sigma) \times E_D$ to $\{0, 1\}$. An observation table (S_D, E_D, T_D) can be visualized as a two-dimensional array with rows labeled by the elements of $S_D \cup S_D \cdot \Sigma$ and columns labeled by the elements of E_D, with the entry for a row $s \in S_D \cup S_D \cdot \Sigma$ and a column $e \in E_D$ equals to $T_D(s, e)$. Suppose $s, t \in S_D \cup S_D \cdot \Sigma$ are two rows, then s and t are equivalent, denoted by $s \cong_{E_D} t$, if and only if $T_D(s, e) = T_D(t, e)$, for all $e \in E_D$. We denote by $[s]$ the equivalence class of rows that also includes s.

Initially, S_D and E_D contain an empty string ϵ and augment as the algorithm runs. The membership queries are constructed from the table, where each $s \in S_D$ is a prefix and each $e \in E_D$ is a suffix of the queries. The interpretation of T_D is that $T_D(s, e)$ is 1 if $s \cdot e$ is accepted by $\mathcal{D}$, otherwise it is 0. Thus, $T_D(s, e) = \Lambda_{\mathcal{D}}(q_{0\mathcal{D}}, s \cdot e)$.

The algorithm L^* eventually uses the observation table to build a DFA conjecture. The strings or prefixes in S_D are the potential states of the conjecture. So, they are called "access" strings as they allow to access the states of the target DFA. The strings or suffixes in E_D distinguish these states from each other. So, they are called "distinguishing" strings. The strings in $S_D \cup S_D \cdot \Sigma$ are used to construct the transition function, such that for every state $s \in S_D$ there is a transition for each $i \in \Sigma$. An example of the observation table (S_D, E_D, T_D) for learning the DFA in Figure 2.1 is given in Table 4.1, where $\Sigma = \{a, b\}$.

		E_D
		ϵ
S_D	ϵ	1
$S_D \cdot \Sigma$	a	0
	b	0

Table 4.1: Example of the Observation Table (S_D, E_D, T_D)

To build a valid DFA conjecture from the observations, the table must satisfy two conditions. The first condition is that the table must be *closed*, that is, for each $t \in S_D \cdot \Sigma$, there exists an $s \in S_D$ such that $s \cong_{E_D} t$. If it is not closed, then there is a state $s \in S_D$ and $i \in \Sigma$ such that the transition function cannot be defined for s and i (see the definition of δ_D below).

The second condition is that the table must be *consistent*, that is, for each $s, t \in S_D$ such that $s \cong_{E_D} t$, it holds that $s \cdot i \cong_{E_D} t \cdot i$, for all $i \in \Sigma$. If it is not consistent then two seemingly equivalent states may point to different target states for the same letter in Σ.

When the observation table (S_D, E_D, T_D) is closed and consistent, then a DFA conjecture can be constructed as follows:

Definition 11 Let (S_D, E_D, T_D) be a closed and consistent observation table, then the DFA conjecture $M_D = (Q_D, \Sigma, \delta_D, F_D, q_{0D})$ is defined, where

- $Q_D = \{[s] | s \in S_D\}$
- $q_{0D} = [\epsilon]$
- $\delta_D([s], i) = [s \cdot i], \forall s \in S_D, i \in \Sigma$
- $F_D = \{[s] | s \in S_D \wedge T_D(s, \epsilon) = 1\}$

Angluin proved that this conjecture is well defined with respect to the observations recorded in the table (S_D, E_D, T_D). Theorem 1 claims the correctness of the conjecture.

Theorem 1 *If (S_D, E_D, T_D) is a closed and consistent observation table, then the DFA conjecture M_D is consistent with the finite function T_D. Any other DFA consistent with T_D but inequivalent to M_D must have more states.* □

Proof (Sketch) The following three lemmas further illustrate the theorem. We provide a sketch of the proof here and refer to the original paper [Ang87] for details.

Lemma 1 *Assume that (S_D, E_D, T_D) is a closed and consistent observation table and M_D is the conjecture from the table, then for every $s \in S_D \cup S_D \cdot \Sigma$, $\delta_D(q_{0D}, s) = [s]$.* □

This is proved by induction on the length of s. It is clearly true when the length is 0, i.e., $s = \epsilon$, since $q_{0D} = [\epsilon]$. Assuming that this is true for every $s \in S_D \cup S_D \cdot \Sigma$ of length k, let $t \in S_D \cup S_D \cdot \Sigma$ of length $k+1$, i.e., $t = s \cdot i$ for some $i \in \Sigma$. Since, S_D is **prefix-closed**, s must be in S_D, for either t is in S_D or t is in $S_D \cdot \Sigma$. Then, by the induction hypothesis, we have $\delta_D(q_{0D}, t) = \delta_D(\delta_D(q_{0D}, s), i) = \delta_D([s], i) = [s \cdot i] = [t]$.

Lemma 2 *Assume that (S_D, E_D, T_D) is a closed and consistent observation table, then the conjecture M_D is consistent with the function T_D. That is, for every $s \in S_D \cup S_D \cdot \Sigma$ and $e \in E_D$, $\delta_D(q_{0D}, s \cdot e)$ is in F if and only if $T_D(s, e) = 1$.* □

This is proved by induction on the length of e. When $e = \epsilon$, then for $s \in S_D \cup S_D \cdot \Sigma$, we know that $\delta_D(q_{0D}, s) = [s]$, by Lemma 1. By the definition of F, $[s]$ is in F if and only if $T_D(s, \epsilon) = 1$. Assuming that this is true for every $e \in E_D$ of length k, let $f \in E_D$ of length $k+1$. Since, E_D is **suffix-closed**, $f = i \cdot e$ for some $i \in \Sigma$. Let $s \in S_D \cdot \Sigma$, then since the table is closed, there exists $t \in S_D$ such that $s \cong_{E_D} t$. Then, we have $\delta_D(q_{0D}, s \cdot f) = \delta_D(q_{0D}, t \cdot i \cdot e)$. By the induction hypothesis on e, $\delta_D(q_{0D}, t \cdot i \cdot e)$ is in F if and only if $T_D(t, i \cdot e) = 1$. Since, $s \cong_{E_D} t$ and $f = i \cdot e$, $T_D(t, i \cdot e) = T_D(s, f)$. Hence, $\delta_D(q_{0D}, s \cdot f)$ is in F if and only if $T_D(s, f) = 1$ is claimed.

Lemma 3 *Assume that (S_D, E_D, T_D) is a closed and consistent observation table and the conjecture M_D has n states. Suppose $M'_D = (Q'_D, \Sigma', \delta'_D, F'_D, q'_{0D})$ is another DFA consistent with T_D that has n or fewer states, then M'_D is isomorphic to M_D.* □

Since M'_D is consistent with T_D, then for each $s \in S_D \cup S_D \cdot \Sigma$ and $e \in E_D$, $\delta'_D(q'_{0D}, s \cdot e)$ is in F' if and only if $T_D(s, e) = 1$, which means $\delta'_D(\delta'_D(q'_{0D}, s), e)$ is in F' if and only if $T_D(s, e) = 1$. So $\delta'_D(q'_{0D}, s)$ is equal to the row s in $S_D \cup S_D \cdot \Sigma$. Hence, as s ranges over all of S_D, $\delta'_D(q'_{0D}, s)$ ranges over all the elements of Q, so M'_D must have at least n states, i.e., it must have exactly n states. To complete isomorphism, Angluin also proved that for each $s \in S_D$, there is a unique $q' \in Q'$, such that $\delta'_D(q'_{0D}, s) = [s]$.

This concludes the proof of Theorem 1, since Lemma 2 shows that M_D is consistent with T_D and Lemma 3 shows that any other DFA consistent with T_D is either isomorphic to M_D or contains at least one more state. Thus, M_D is uniquely the minimum DFA consistent with T_D. ∎

4.1.2 The Algorithm L^*

The algorithm L^* starts by initializing the observation table (S_D, E_D, T_D) by $S_D = E_D = \{\epsilon\}$. To determine T_D, L^* asks membership queries constructed from the table. For each $s \in S_D \cup S_D \cdot \Sigma$ and $e \in E_D$, a membership query is constructed as $s \cdot e$. The result of each query is recorded in the table accordingly. After filling the table with the results of the queries, L^* checks if (S_D, E_D, T_D) is closed and consistent.

If (S_D, E_D, T_D) is not closed, then L^* finds $t \in S_D \cdot \Sigma$ such that $t \ncong_{E_D} s$, for all $s \in S_D$. Then, it moves t to S_D and extends $S_D \cdot \Sigma$ accordingly. The algorithm then asks membership queries for the new rows in the table.

If (S_D, E_D, T_D) is not consistent, then L^* finds $s, t \in S_D, e \in E_D$ and $i \in \Sigma$ such that $s \cong_{E_D} t$ but $T_D(s \cdot i, e) \neq T_D(t \cdot i, e)$. Then, it adds the string $i \cdot e$ to E_D and extends the table by asking membership queries for the missing elements.

These two operations are repeated until (S_D, E_D, T_D) is closed and consistent. Finally, L^* makes a DFA conjecture M_D from the table according to Definition 11.

4.1.3 Learning with oracle

The learning of the unknown DFA $\mathcal{D}$ by asking queries is an iterative step. A conjecture from a closed and consistent table after a run of the algorithm may still contain less number of states than the minimum $\mathcal{D}$. This is because the conjecture is an approximation drawn after finite number of experiments, which may not have explored all the states of the hidden model. Therefore, there must be some counterexample that can distinguish the conjecture and the hidden model to start another iteration of the learning algorithm with a quest to learn a better approximation.

Angluin uses a concept of an *oracle* that presumably knows the target language. L^* presents the conjecture to the oracle that acknowledges whether the conjecture is correct. This correctness check is called asking an *equivalence query* for the conjecture. The oracle replies either *yes*, signifying that the conjecture is correct, or with a counterexample. If the oracle replies *yes*, then L^* terminates by giving a final conjecture from a closed and consistent observation table. If the oracle replies with a counterexample, then L^* processes the counterexample in the observation table to refine the conjecture. The method of processing counterexample is given as follows.

Let $\nu \in \Sigma^+$ be a counterexample, then L^* processes ν in (S_D, E_D, T_D) by adding all the prefixes of ν in S_D. Then, the table is extended accordingly and the missing elements of the table are filled by asking membership queries. The algorithm then makes the table closed and consistent, and outputs the new conjecture.

This follows the asking of another equivalence query for the new conjecture. The process continues until the oracle accepts the conjecture and the algorithm terminates. Algorithm 1 summarizes the complete method for inferring the exact DFA of the unknown language.

```
Input: The alphabet $\Sigma$
Output: DFA conjecture $M_D$
begin
    initialize the observation table $(S_D, E_D, T_D)$ with the sets
    $S_D = E_D = \{\epsilon\}, S_D \cdot \Sigma = \{\epsilon \cdot i\}, \forall i \in \Sigma$ ;
    ask the membership queries from $(S_D, E_D, T_D)$ ;
    update $(S_D, E_D, T_D)$ with the results of the queries ;
    repeat
        while $(S_D, E_D, T_D)$ is not closed or not consistent do
            if $(S_D, E_D, T_D)$ is not closed then
                find $t \in S_D \cdot \Sigma$ such that $t \ncong_{E_D} s$, for all $s \in S_D$ ;
                move $t$ to $S_D$ ;
                ask membership queries for the extended table ;
            end
            if $(S_D, E_D, T_D)$ is not consistent then
                find $s, t \in S_D, e \in E_D, i \in \Sigma$ such that $s \cong_{E_D} t$,
                but $T_D(s \cdot i, e) \neq T_D(t \cdot i, e)$ ;
                add $i \cdot e$ to $E_D$ ;
                ask membership queries for the extended table ;
            end
        end
        make the conjecture $M_D$ from $(S_D, E_D, T_D)$ ;
        ask the equivalence query for $M_D$ ;
        if oracle replies with a counterexample $\nu$ then
            add all the prefixes of $\nu$ to $S_D$ ;
            ask membership queries for the extended table ;
        end
    until oracle replies yes to the conjecture $M_D$ ;
    return the conjecture $M_D$ from $(S_D, E_D, T_D)$ ;
end
```

Algorithm 1: The Algorithm L^*

4.1.4 Complexity

Angluin proved that the conjecture from a closed and consistent (S_D, E_D, T_D) can be constructed in polynomial time of factors

- $|\Sigma|$, i.e., the size of Σ
- n, i.e., the number of states of the minimum DFA $\mathcal{D}$
- m, i.e., the maximum length of any counterexample provided by the oracle

Initially, S_D contains one element. Each time (S_D, E_D, T_D) is found not closed, one element is added to S_D. This introduces a new row to S_D, so a new state in the conjecture. This can happen for at most $n-1$ times. For each counterexample of length at most m, there can be at most m strings that are added to S_D, and there can be at most $n-1$ counterexamples to distinguish n states. Thus, the size of S_D cannot exceed $n + m(n-1)$.

Initially, E_D contains one element. Each time (S_D, E_D, T_D) is found not consistent, one element is added to E_D. This can happen for at most $n-1$ times to distinguish n states. Thus, the size of E_D cannot exceed n.

Putting these together, the maximum size of $(S_D \cup S_D \cdot \Sigma) \times E_D$ in the worst case is

$$(\overbrace{n + m(n-1)}^{|S_D|} + \overbrace{(n + m(n-1))|\Sigma|}^{|S_D \cdot \Sigma|})\,(\overbrace{n}^{|E_D|}) = O(|\Sigma| m n^2)$$

The maximum number of membership queries is the worst case size of the observation table. Theorem 2 summarizes the results concerning L^*.

Theorem 2 *Given any minimally adequate oracle presenting an unknown regular language, the algorithm L^* eventually terminates and outputs a DFA conjecture isomorphic to the minimum DFA $\mathcal{D}$ modeling the language. Moreover, if n is the number of states of $\mathcal{D}$ and m is an upper bound on the length of any counterexample provided by the oracle, then the total running time of L^* is bounded by a polynomial in m and n.* $\square$

4.1.5 Example

We illustrate the algorithm L^* on the DFA $\mathcal{D}$, given in Figure 4.2, over the alphabet $\Sigma = \{a, b\}$. The DFA $\mathcal{D}$ accepts all the strings with an even (or zero) number of "a" and an even (or zero) number of "b".

In order to learn $\mathcal{D}$, the algorithm L^* initializes the observation table (S_D, E_D, T_D) with $S_D = E_D = \{\epsilon\}$ and $S_D \cdot \Sigma$ with $\{a, b\}$. It asks the membership queries for ϵ, a, and b. This gives the table ${T_D}^{(1)}$, as shown in Table 4.2.

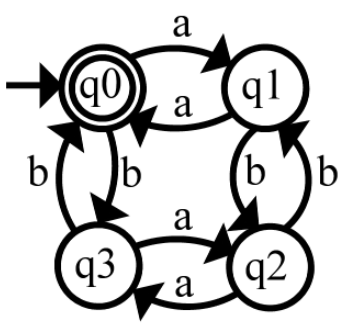


Figure 4.2: Example of a Deterministic Finite Automaton (repeated from Figure 2.1)

${T_D}^{(1)}$	ϵ
ϵ	1
$\boxed{a}$	0
b	0

Table 4.2: The Observation Table ${T_D}^{(1)}$. The box in the table shows the row which make the table not closed.

The table is consistent but not closed, since the row a in $S_D \cdot \Sigma$ is inequivalent to any row in S_D. So, the row a is moved[1] to S_D and the strings $a \cdot a$ and $a \cdot b$ are added to $S_D \cdot \Sigma$. The algorithm asks queries for the new rows to build the table ${T_D}^{(2)}$, as shown in Table 4.3.

${T_D}^{(2)}$	ϵ
ϵ	1
a	0
b	0
$a \cdot a$	1
$a \cdot b$	0

Table 4.3: The Observation Table ${T_D}^{(2)}$.

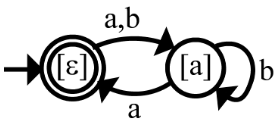


Figure 4.3: The conjecture ${M_D}^{(1)}$ from Table ${T_D}^{(2)}$.

The table ${T_D}^{(2)}$ is closed and consistent. So, the algorithm conjectures the machine ${M_D}^{(1)}$ from the table, as shown in Figure 4.3.

The next step is to ask an equivalence query for ${M_D}^{(1)}$. The answer from the oracle is the counterexample $b \cdot b$, since $\Lambda_{\mathcal{D}}({q0}_{\mathcal{D}}, b \cdot b) \neq \Lambda_D({q0}_{{M_D}^{(1)}}, b \cdot b)$. The algorithm adds all the prefixes of the counterexample, i.e., b and $b \cdot b$, to the upper part of the table. The lower part is extended

[1]Note that the row b in the table ${T_D}^{(1)}$ also makes the table not closed. Since it is equivalent to the row a, we move only one row to S_D at a time. In this case, we moved the row a

accordingly, i.e., the strings $b \cdot a$, $b \cdot b \cdot a$ and $b \cdot b \cdot b$ are added to $S_D \cdot \Sigma$. The membership queries are asked for newly added rows, resulting in the table ${T_D}^{(3)}$, as shown in Table 4.4.

${T_D}^{(3)}$	ϵ
ϵ	1
$\boxed{a}$	0
$\boxed{b}$	0
$b \cdot b$	1
$a \cdot a$	1
$a \cdot b$	0
$b \cdot a$	0
$b \cdot b \cdot a$	0
$b \cdot b \cdot b$	0

Table 4.4: The Observation Table ${T_D}^{(3)}$ after processing the counterexample $b \cdot b$. The boxes show the rows which make the table inconsistent.

The table ${T_D}^{(3)}$ is closed but not consistent, since $a \cong_{E_D} b$, but $T_D(a \cdot a, \epsilon) \neq T_D(b \cdot a, \epsilon)$. The algorithm adds the suffix a, which distinguishes the rows a and b, to E_D. The table is filled by asking membership queries for the missing entries to get the table ${T_D}^{(4)}$, as shown in Table 4.5.

${T_D}^{(4)}$	ϵ	a
ϵ	1	0
a	0	1
b	0	0
$b \cdot b$	1	0
$a \cdot a$	1	0
$a \cdot b$	0	0
$b \cdot a$	0	0
$b \cdot b \cdot a$	0	1
$b \cdot b \cdot b$	0	0

Table 4.5: The Observation Table ${T_D}^{(4)}$.

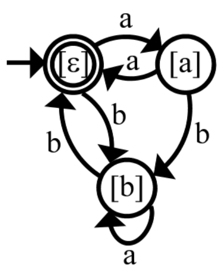


Figure 4.4: The conjecture ${M_D}^{(2)}$ from Table ${T_D}^{(4)}$.

The table ${T_D}^{(4)}$ is closed and consistent, so the algorithm makes the conjecture ${M_D}^{(2)}$, as shown in Figure 4.4, from the table.

Next, the algorithm asks an equivalence query for ${M_D}^{(2)}$. The response from the oracle is the counterexample $a \cdot b \cdot b$, since $\Lambda_{\mathfrak{D}}(q_{0_{\mathfrak{D}}}, a \cdot b \cdot b) \neq \Lambda_D(q_{0_{{M_D}^{(2)}}}, a \cdot b \cdot b)$. This follows the addition of all the prefixes of the counterexample to S_D. Since the prefix a is already in S_D, only the prefixes $a \cdot b$ and $a \cdot b \cdot b$ are added. The lower part of the table is then extended with strings $a \cdot b \cdot a$, $a \cdot b \cdot b \cdot a$, $a \cdot b \cdot b \cdot b$. The algorithm asks the membership queries for filling the missing entries to get the table ${T_D}^{(5)}$, as shown in Table 4.6.

${T_D}^{(5)}$	ϵ	a
ϵ	1	0
a	0	1
$\boxed{b}$	0	0
$b \cdot b$	1	0
$\boxed{a \cdot b}$	0	0
$a \cdot b \cdot b$	0	1
$a \cdot a$	1	0
$b \cdot a$	0	0
$b \cdot b \cdot a$	0	1
$b \cdot b \cdot b$	0	0
$a \cdot b \cdot a$	0	0
$a \cdot b \cdot b \cdot a$	1	0
$a \cdot b \cdot b \cdot b$	0	0

Table 4.6: The Observation Table ${T_D}^{(5)}$ after processing the counterexample $a \cdot b \cdot b$. The boxes show the rows which make the table inconsistent.

The table ${T_D}^{(5)}$ is closed but not consistent, since $b \cong_{E_D} a \cdot b$, but $T_D(b, b) \neq T_D(a \cdot b, b)$. The algorithm adds the suffix b to E_D and queries for the missing entries to get the table ${T_D}^{(6)}$, as shown in Table 4.7.

${T_D}^{(6)}$	ϵ	a	b
ϵ	1	0	0
a	0	1	0
b	0	0	1
$b \cdot b$	1	0	0
$a \cdot b$	0	0	0
$a \cdot b \cdot b$	0	1	0
$a \cdot a$	1	0	0
$b \cdot a$	0	0	0
$b \cdot b \cdot a$	0	1	0
$b \cdot b \cdot b$	0	0	1
$a \cdot b \cdot a$	0	0	1
$a \cdot b \cdot b \cdot a$	1	0	0
$a \cdot b \cdot b \cdot b$	0	0	0

Table 4.7: The Observation Table ${T_D}^{(6)}$.

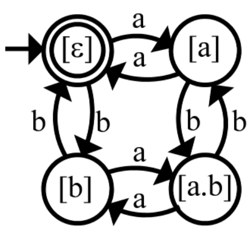


Figure 4.5: The conjecture ${M_D}^{(3)}$ from Table ${T_D}^{(6)}$.

The table ${T_D}^{(6)}$ is closed and consistent. The algorithm makes the corresponding conjecture ${M_D}^{(3)}$, as shown in Figure 4.5. The oracle replies *yes* in the final equivalence check, so the algorithm returns ${M_D}^{(3)}$ and terminates.

4.2 Variants of L^*

There are variety of methods that tend to improve the algorithm L^*. The three widely referred methods are listed below.

1. Rivest & Schapire [RS93] improved the algorithm by removing the consistency check[1]. They proposed a method for processing a counterexample in the observation table that consequently reduced the total number of membership queries to learn $\mathcal{D}$ completely. The upper bound on the number of queries in their method is given as $O(|\Sigma|n^2 + n\ log\ m)$.
2. Kearns & Vazirani [KV94] used a completely different data structure to record observations, i.e., a *binary discrimination tree* with labeled nodes. The upper bound on the number of queries in their method is given as $O(|\Sigma|n^3 + nm)$.
3. Hungar et al. [HNS03] suggested improvements for learning prefix-closed languages They showed with the help of experiments [HNS03][BJLS05] that L^* can be improved with domain specific optimizations. They achieved 20% reduction of membership queries on randomly generated prefix-closed DFAs by their method, compared to L^*.

The method of Kearns & Vazirani [KV94] has worst-case estimation quite close to L^*. The method of Hungar et al. [HNS03] is applied in a specific case, i.e., when the language is prefix-closed. In the following, we discuss the improvements proposed by Rivest & Schapire [RS93].

4.2.1 Proposition of Rivest & Schapire

Rivest & Schapire [RS93] noticed that Angluin's algorithm can be improved by removing consistency check. Consistency is checked only when two rows in S_D are found equivalent. Otherwise, the condition of consistency is always satisfied trivially. Therefore, the idea of keeping the size of S_D small was proposed such that it contains only inequivalent rows. They observed that inconsistency holds in S_D due to improper handling of counterexamples. A counterexample is an experiment that distinguishes two or more equivalent rows (or states) in the table and thereby causes an increase in the size of E_D. However, L^* does not follow this scheme directly, rather it adds a new row for each prefix of the counterexample in S_D assuming all are the potential states of the new conjecture. Later, the rows are filled with the help of membership queries

[1]The other improvement was removing the "reset" assumption by applying a homing sequence [LY96], so that the algorithm does not need to reset the machine before asking each membership query.

(no new column is added yet). This is where an inconsistency can occur in the table if two rows $s, t \in S_D$ are found equivalent, i.e., $s \cong_{E_D} t$, but their future behaviors are not equivalent, i.e., $s \cdot i \ncong_{E_D} t \cdot i$, for some $i \in \Sigma$. Thus, s and t must be distinguished in S_D by adding a distinguishing string in E_D.

Rivest & Schapire proposed a method of processing a counterexample which does not add the prefixes in S_D. Thus, the rows in S_D remain inequivalent during the whole learning procedure. They find a distinguishing string in the counterexample and directly add the string to E_D. But their method requires a relaxation on the prefix-closed and the suffix-closed properties of the table. In summary, the following two changes on the observation table (S_D, E_D, T_D) are imposed by their method:

1. S_D is not prefix-closed and E_D is not suffix-closed.

2. S_D contains only inequivalent rows, i.e., for all $s, t \in S_D$, $s \ncong_{E_D} t$.

Since, there cannot be two rows in S_D that represent a single state, there is no need to check consistency. In other words, there cannot occur an inconsistency in the observation table. The method of processing a counterexample is proposed as follows.

Let $\nu = i_0 \ldots i_{m-1}$ be the counterexample of length m. For $0 \leq k \leq m$, let u_k be the prefix of ν of length k and v_k be the corresponding suffix, i.e., $\nu = u_k \cdot v_k$; so that $u_0 = v_m = \epsilon$ and $v_0 = u_m = \nu$. Also, $\nu = u_k \cdot i_k \cdot v_{k+1}$, for $k < m$. Let $q_k = \delta_D([\epsilon], u_k)$ be the state of the conjecture that is reached by applying u_k, the initial state be $q_0 = [\epsilon]$ and q_{k+1} be the state reached by applying i_k on q_k, i.e., $q_{k+1} = \delta_D(q_k, i_k)$. The acceptance of ν by this conjecture is given by whether the ending state q_m is final.

Since the set of strings that are accepted from a state distinguishes the state from the others, we look upon suffix v_k as an experiment on the corresponding state q_k, and through membership queries we find out $\Lambda_{\mathfrak{D}}(q_k, v_k)$, or whether $\delta_{\mathfrak{D}}(q_k, v_k)$ is final. The fact that ν is a counterexample means that $\delta_{\mathfrak{D}}(q_{0\mathfrak{D}}, \nu)$ is final $\Longleftrightarrow$ q_m is not final. So consequently there must exist one or more breakpoint positions k such that $\delta_{\mathfrak{D}}(q_k, v_k)$ is final $\Longleftrightarrow$ $\delta_{\mathfrak{D}}(q_{k+1}, v_{k+1})$ is not final. Since $\delta_{\mathfrak{D}}(q_k, v_k) = \delta_{\mathfrak{D}}(\delta_{\mathfrak{D}}(q_k, i_k), v_{k+1})$, the suffix v_{k+1} is an experiment that distinguishes $\delta_{\mathfrak{D}}(q_k, i_k)$ from $q_{k+1} = \delta_D(q_k, i_k)$. It suffices now to add v_{k+1} to E_D . Rivest & Schapire show how a binary search finds a breakpoint with *log m* queries. For at most n counterexamples, the method requires *n log m* queries. The maximum size of S_D is n, since it contains all inequivalent rows which cannot exceed n. The maximum size of E_D is n, since it contains one distinguishing string for each counterexample. Therefore, the maximum size of $(S_D \cup S_D \cdot \Sigma) \times E_D$ is $(n + |\Sigma| n)n$. In total, the method of Rivest & Schapire uses at most $O(|\Sigma| n^2 + n\ log\ m)$ membership queries.

A point worth noting is the following. The observation table is kept small by not adding all the prefixes of a counterexample in S_D. Thus, the original property of S_D that it is prefix-closed has been altered. For processing each counterexample, only one string is added to E_D, where E_D may not contain its suffixes. Thus, the original property of E_D that it is suffix-closed has been altered. Consequently, the observation table is not prefix-closed and suffix-closed; the properties which are required to prove that the conjecture from the table is always consistent with the observations (see Theorem 1 and its proof). That means, there may exist a string $s \cdot e \in S_D \cup S_D \cdot \Sigma$ that has already been tested as a membership query but $\Lambda_{\mathfrak{D}}(q_{0\mathfrak{D}}, s \cdot e) \neq \Lambda_D(q_{0D}, s \cdot e)$. Thus, the method of Rivest & Schapire does not guarantee satisfying Theorem 1. Balcazar et al. [BDG97] argued that the new conjecture obtained from this method may still classify a previous counterexample incorrectly, so that the same counterexample can potentially be used to answer several equivalence queries. In addition, Berg & Raffelt [BR05] compiled the results from Balcazar et al. [BDG97] and explained the complete method of Rivest & Schapire.

4.3 Conclusion

This chapter discussed the inference of a Deterministic Finite Automaton (DFA) through an active learning approach. It described a well-known algorithm, called L^* [Ang87], with its complexity discussion and illustration on the example.

We learned that the complete automaton that models the hidden structure of a black box machine can be inferred using membership and equivalence queries in a polynomial time (assuming that the oracle provides counterexamples for the intermediate conjectures). Rivest & Schapire [RS93] improved the complexity of the algorithm by proposing a different method of processing counterexamples. However, the method relaxed the prefix-closed and suffix-closed properties of the table which are required to produce the conjecture consistent with the observations. We explained the method and its problems which are also indicated by Balcazar et al. [BDG97].

We note that the original idea of keeping only inequivalent rows can be encouraged if different methods of processing counterexamples are proposed such that they do not alter the above mentioned properties of the table. Moreover, we intend to extend this work for learning enhanced machines (see Chapters 5 and 7). In these extensions, we borrow the idea of Rivest & Schapire and present a new method of processing counterexamples such that the original properties of the observation table are preserved. The complete discussion on this topic is presented in Chapter 5.

Chapter 5

Mealy Machine Inference

This chapter covers the details of inferring a Mealy machine through the active learning approach. It describes the inference algorithm adapted from Angluin's algorithm and its complexity. Then, it presents our improvements on the algorithm and its complexity, followed by the formal relation of the Mealy conjecture with the actual model, and finally the application of the Mealy machine inference.

5.1 Motivation

Our application domain, i.e., telecom services, web-based applications, data acquisition modules, embedded system controllers, are considered as complex systems that characterize their behaviors in terms of input/output (i/o). Typically, these systems receive inputs from the environment, take decisions on internal transitions, perform computations and finally produce the corresponding outputs to the environment. Arguably, the more natural modeling of such systems is through Mealy machines (Definition 2), which DFAs do not support the structure directly. Moreover, it is observed that a DFA model normally contains far more states than a Mealy machine if they model the same problem [MNRS04] [Nei03]. Thus, efforts of learning Mealy machines are desirable also from this aspect.

Angluin's algorithm L^* cannot be used directly to learn Mealy machines because they produce output strings in response to the queries and do not distinguish states as final or non-final, as in the case of DFA models. However, it is observed that L^* can tackle Mealy machines through model transformation techniques. A simple way is to define a mapping from inputs and outputs of the machine to letters in a DFA's alphabet Σ. This can be done either by taking inputs and outputs as letters, i.e., $\Sigma = I \cup O$ [HNS03] or by considering couples of inputs and outputs as letters, i.e., $\Sigma = I \times O$ [MS01b]. But these methods exploit the size of Σ and thus raise complexity problems because the algorithm is polynomial on these factors. However, there is a straightforward implication of L^* on learning Mealy machines by slightly modifying the structure of the observation table. The idea is to record the behaviors of the system as output strings in the table instead of recording just “1” and “0”, as in the case of language inference. This adaptation of L^* to learn Mealy machines has been discussed in several works, formally [Nei03] [BGJ$^+$05] [SL07], and informally [MNRS04] [PO99].

In this chapter, we discuss the learning of Mealy machines using the settings from L^*. We can adapt the structure of the observation table and related concepts, such as making the table closed and consistent and making a conjecture from the table etc, for the algorithm of learning Mealy machines. For processing counterexamples in the table, we can also easily adapt the corresponding method from L^*. However, we propose improvements to the basic adaptation and show that our improvements significantly reduce the complexity of the algorithm. A short introduction of the improvements is given here.

Inspired by Rivest & Schapire [RS93], we have observed that the algorithm for inferring Mealy machines can be improved if the consistency check is removed. As discussed in Chapter 4, an inconsistency can occur due to the improper handling of counterexamples in the observation table. Our contribution in the inference of Mealy machines is a proposal of a new method for processing counterexamples that does not cause inconsistency in the table and consequently reduces the complexity of the algorithm. The complexity analysis shows that by using our method for processing counterexamples, the algorithm for learning Mealy machines requires quite less number of queries, compared to the adapted method.

The organization of the chapter is as follows. In Section 5.2, we describe the adapted algorithm to learn Mealy machines. In Section 5.3, we provide our improved algorithm and discuss the complexity comparison of the two algorithms. Furthermore, we formulate an approximation relation between a Mealy machine conjecture and the unknown machine in Section 5.4. In Section 5.5, we discuss the algorithm L^* according to the new method of processing counterexamples. Finally, an application of the Mealy machine inference and its results are given in Section 5.6. Section 5.7 concludes the chapter.

5.2 Learning Algorithm for Mealy Machines

The formal definition of a Mealy machine $(Q, I, O, \delta, \lambda, q_0)$ is given in Definition 2 (Chapter 2). In this section, we detail the learning of Mealy machines using the settings from Angluin's algorithm L^*, that has also been mentioned in the existing works. The basic idea is to explore the system systematically by asking queries and build the conjecture automaton from the observations that are collected in the result of the queries. As for DFA learning, the two main assumptions in learning Mealy machines are

1. The basic input set I is known
2. The machine can be reset before each query

The algorithm asks *output queries* [SL07] that are strings from I^+ and obtain the corresponding output strings from the machine. This is similar to the concept of membership queries in L^*. The difference is that instead of 1 or 0, the machine replies with the complete output string. Let $\omega \in I^+$, i.e., an input string of the query, then the machine replies to the query with $\lambda(q_0, \omega)$. The response to each query is recorded in the observation table. The queries are asked iteratively until the conditions on the observation table are satisfied. Finally, the algorithm builds a Mealy machine conjecture from the table. If a counterexample is provided for the conjecture, then the algorithm processes the counterexample in the table and refines the conjecture.

We describe the structure of the observation table in Section 5.2.1, then the algorithm for learning Mealy machines in Section 5.2.2 and its illustration on an example in Section 5.2.3. Later, we explain the method for processing counterexamples in the table in Section 5.2.4. The complexity of the learning algorithm is given in Section 5.2.5. Finally, the illustration of processing counterexamples is given in Section 5.2.6.

We denote by $\mathcal{M} = \{Q_{\mathcal{M}}, I, O, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}}, q_{0\mathcal{M}}\}$ the unknown Mealy machine model that has a minimum number of states. We assume that the input/output interfaces of the machines are accessible, i.e., the input interface from where an input can be sent and the output interface from where an output can be observed.

5.2.1 Observation Table

We denote by $L_M{}^*$ the learning algorithm for Mealy machines. At any given time, $L_M{}^*$ has information about a finite collection of input strings from I^+ and their corresponding output strings from O^+. This information is organized into an observation table, denoted by (S_M, E_M, T_M). The structure of the table is directly imported from the Angluin's algorithm L^*. Let S_M and E_M be the non-empty finite sets of finite strings over I. S_M is a prefix-closed set that always contains an empty string ϵ. E_M is a suffix-closed set ($\epsilon \notin E_M$). Let T_M be a finite function that maps $(S_M \cup S_M \cdot I) \times E_M$ to O^*. If $s \in S_M \cup S_M \cdot I$ and $e \in E_M$, then $T_M(s, e)$ contains an output string taken from $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e)$. Since, S_M is prefix-closed, we already know the result for the prefixes of s. Therefore, it is sufficient to record only the suffix of the output string which corresponds to e. Thus, $T_M(s, e) = \mathit{suff}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e))$. The rows of the table consist of the elements of $S_M \cup S_M \cdot I$ and the columns consist of the elements of E_M.

Since S_M and E_M are non-empty sets, the table is initialized by $S_M = \{\epsilon\}$ and $E_M = I$, i.e., every input symbol makes one column in the table, with the entry for a row $s \in S_M \cup S_M \cdot I$ and a column $e \in E_M$ equals to $T_M(s, e)$. The equivalence of rows is defined with respect to the strings in E_M. Suppose $s, t \in S_M \cup S_M \cdot I$ are two rows, then s and t are equivalent, denoted by $s \cong_{E_M} t$, if and only if $T_M(s, e) = T_M(t, e)$, for all $e \in E_M$. We denote by $[s]$ the equivalence class of rows that also includes s. An example of the observation table (S_M, E_M, T_M) over $I = \{a, b\}$ is given in Table 5.1.

		E_M	
		a	b
S_M	ϵ	x	x
$S_M \cdot I$	a	y	x
	b	x	x

Table 5.1: Example of the Observation Table (S_M, E_M, T_M)

The algorithm $L_M{}^*$ eventually uses the observation table (S_M, E_M, T_M) to build a Mealy machine conjecture. The strings or prefixes in S_M are the potential states of the conjecture, and the strings or suffixes in E_M distinguish these states from each other.

To build a valid Mealy machine conjecture from the observations, the table must be *closed* and *consistent*. The table is *closed* if for each $t \in S_M \cdot I$, there exists an $s \in S_M$, such that $s \cong_{E_M} t$. The table is *consistent* if for each $s, t \in S_M$ such that $s \cong_{E_M} t$, it holds that $s \cdot i \cong_{E_M} t \cdot i$, for all $i \in I$.

When the observation table (S_M, E_M, T_M) is closed and consistent, then a Mealy machine conjecture can be constructed as follows:

Definition 12 Let (S_M, E_M, T_M) be a closed and consistent observation table, then the Mealy machine conjecture $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ is defined, where

- $Q_M = \{[s] | s \in S_M\}$
- $q_{0M} = [\epsilon]$
- $\delta_M([s], i) = [s \cdot i], \forall s \in S_M, i \in I$
- $\lambda_M([s], i) = T_M(s, i), \forall i \in I$

To see that M_M is well defined, note that S_M is a non-empty prefix-closed set and it contains at least one row ϵ, hence Q_M and q_{0M} are well-defined. For every $s \in S_M$, there exists $s \cdot i$ in $S_M \cup S_M \cdot I$, for all $i \in I$, and since (S_M, E_M, T_M) is closed, $[s \cdot i] \in Q_M$; hence δ_M is well defined. For every $s \in S_M$ and $i \in I$, $T_M(s,i)$ always exists, since $E_M \supseteq I$. Hence, λ_M is well defined. Theorem 3 claims the correctness of the conjecture. Neise [Nei03] has given a formal proof of the correctness, which is a simple demonstration of the Angluin's algorithm, in which the range of the output function is replaced by O^+. Moreover, the conjecture is the minimum machine by construction.

Theorem 3 *If (S_M, E_M, T_M) is a closed and consistent observation table, then the Mealy machine conjecture M_M is consistent with the finite function T_M. That is, for every $s \in S_M \cup S_M \cdot I$ and $e \in E_M$, $\lambda_M(\delta_M(q_{0M}, s), e) = T_M(s,e)$. Any other Mealy machine consistent with T_M but inequivalent to M_M must have more states.* □

Note that the conjecture M_M is input-enabled, i.e., for each state in Q_M and for each input in I, it has an enabled transition. In practice, we may have a machine $\mathcal{M}$ for which we observe that after applying a certain prefix $s \in S_M$ to the machine, the given input $i \in E_M$ is invalid. Then, we would not ask an output query $s \cdot e$; instead, we directly add Ω[1] to $T_M(s,e)$. This is useful to tackle the query complexity in practice.

5.2.2 The Algorithm ${L_M}^*$

The algorithm ${L_M}^*$ starts by initializing (S_M, E_M, T_M) with $S_M = \{\epsilon\}$ and $E_M = I$. To determine T_M, it asks output queries constructed from the table. For each $s \in S_M \cup S_M \cdot I$ and $e \in E_M$, a query is constructed as $s \cdot e$. The corresponding output string of the machine is recorded with the help of function T_M, i.e., $T_M(s,e) = \mathit{suff}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e))$.

After filling the table with the result of the queries, ${L_M}^*$ checks if the table is closed and consistent. If it is not closed, then ${L_M}^*$ finds $t \in S_M \cdot I$ such that $t \ncong_{E_M} s$, for all $s \in S_M$. Then, it moves t to S_M and $T_M(t \cdot i, e)$ is determined for all $i \in I, e \in E_M$ in $S_M \cdot I$. If the table is not consistent, then ${L_M}^*$ finds $s, t \in S_M, e \in E_M$ and $i \in I$ such that $s \cong_{E_M} t$, but $T_M(s \cdot i, e) \neq T_M(t \cdot i, e)$. Then, it adds the string $i \cdot e$ to E_M and extends the table by asking output queries for the missing elements.

When the table is closed and consistent, ${L_M}^*$ makes a Mealy machine conjecture M_M from the table according to Definition 12.

[1]See Definition 2 for the usage of Ω

5.2.3 Example

We illustrate the algorithm ${L_M}^*$ on the Mealy machine $\mathcal{M}$ given in Figure 5.1. The algorithm initializes (S_M, E_M, T_M) with $S_M = \{\epsilon\}$ and $S_M \cdot I = E_M = \{a, b\}$. Then, it asks the output queries to fill the table, as shown in Table 5.1. When the table is filled, ${L_M}^*$ checks if it is closed and consistent.

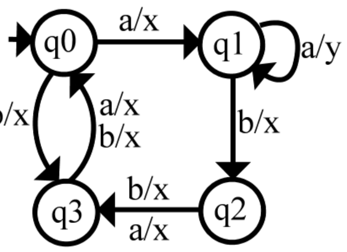


Figure 5.1: Example of a Mealy Machine (repeated from Figure 2.2)

Table 5.1 is not closed since the row a in $S_M \cdot I$ is not equivalent to any row in S_M. Therefore, the row a is moved to S_M and the table is extended accordingly. Then, ${L_M}^*$ asks the output queries for the missing elements of the table. Table 5.2 shows the resulting observation table.

The new table is closed and consistent, so ${L_M}^*$ terminates by making the conjecture ${M_M}^{(1)} = (Q_{{M_M}^{(1)}}, I, O, \delta_{{M_M}^{(1)}}, \lambda_{{M_M}^{(1)}}, {q_0}_{{M_M}^{(1)}})$ from Table 5.2. The conjecture ${M_M}^{(1)}$ is shown in Figure 5.2.

	a	b
ϵ	x	x
a	y	x
b	x	x
$a \cdot a$	y	x
$a \cdot b$	x	x

Table 5.2: Closed and Consistent Observation Table (S_M, E_M, T_M) for learning $\mathcal{M}$ in Figure 5.1

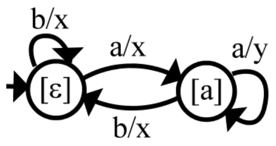


Figure 5.2: The conjecture ${M_M}^{(1)}$ from Table 5.2

The conjecture ${M_M}^{(1)}$ is not correct and can be refined with the help of a counterexample. The methods for processing counterexamples are discussed in the following sections. We shall illustrate the methods with the help of the same example. We provide here a counterexample that will be used in their illustrations.

Let $a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$ be a counterexample for ${M_M}^{(1)}$, since

- $\lambda_{M_M{}^{(1)}}(q_{0_{M_M{}^{(1)}}}, a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a) = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot y$ and

- $\lambda_{\mathcal{M}}(q_{0_{\mathcal{M}}}, a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a) = x \cdot x \cdot x \cdot x \cdot x \cdot x \cdot x$.

We choose a long counterexample to better illustrate the methods and to realize how they work when the counterexamples of arbitrary lengths are provided. In practice, it is not sure whether we obtain always the shortest counterexample.

5.2.4 Processing Counterexamples in ${L_M}^*$

Angluin's algorithm L^* provides a method for processing a counterexample in the observation table (S_D, E_D, T_D), so that the conjecture is refined with at least one more state. For the algorithm ${L_M}^*$, we can adapt the Angluin's method straightforwardly for processing counterexamples in the observation table (S_M, E_M, T_M). The adapted method is described as follows.

Directly Adapted Method from L^*

The adaptation of Angluin's method for processing a counterexample in (S_M, E_M, T_M) is simply adding the prefixes of the counterexample to S_M and then extending the table accordingly. The formal representation of the method is given below.

Let $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0_M})$ be the conjecture from a closed and consistent observation table (S_M, E_M, T_M) for learning the machine $\mathcal{M}$. Let ν be a string from I^+ as a counterexample such that $\lambda_M(q_{0_M}, \nu) \neq \lambda_{\mathcal{M}}(q_{0_{\mathcal{M}}}, \nu)$. Then, ${L_M}^*$ adds all the prefixes of ν to S_M and extends (S_M, E_M, T_M) accordingly. The algorithm makes another run of output queries until (S_M, E_M, T_M) is closed and consistent, followed by making a new conjecture.

Algorithm 2 summerizes the algorithm ${L_M}^*$ with the adapted method for processing counterexamples.

```
Input: The set of input symbols $I$
Output: Mealy machine conjecture $M_M$
begin
    initialize the observation table $(S_M, E_M, T_M)$ with the sets
    $S_M = \{\epsilon\}, E_M = I, S_M \cdot I = \{\epsilon \cdot i\}, \forall i \in I$ ;
    ask the output queries from $(S_M, E_M, T_M)$ ;
    update $(S_M, E_M, T_M)$ with the results of the queries ;
    while $(S_M, E_M, T_M)$ is not closed or not consistent do
        if $(S_M, E_M, T_M)$ is not closed then
            find $t \in S_M \cdot I$ such that $t \ncong_{E_M} s$, for all $s \in S_M$ ;
            move $t$ to $S_M$ ;
            ask output queries for the extended table ;
        end
        if $(S_M, E_M, T_M)$ is not consistent then
            find $s, t \in S_M, e \in E_M, i \in I$ such that $s \cong_{E_M} t$, but $T_M(s \cdot i, e) \neq T_M(t \cdot i, e)$ ;
            add $i \cdot e$ to $E_M$ ;
            ask output queries for the extended table ;
        end
        make the conjecture $M_M$ from $(S_M, E_M, T_M)$ ;
        if there is a counterexample $\nu$ for $M_M$ then
            add all the prefixes of $\nu$ to $S_M$ ;
            ask output queries for the extended table ;
        end
    end
    return the conjecture $M_M$ from $(S_M, E_M, T_M)$ ;
end
```

Algorithm 2: The Algorithm ${L_M}^*$

5.2.5 Complexity

We analyze the total number of output queries asked by ${L_M}^*$ in the worst case by the factors

- $|I|$, i.e., the size of I
- n, i.e., the number of states of the minimum machine $\mathcal{M}$
- m, i.e., the maximum length of any counterexample provided during the learning of $\mathcal{M}$

Initially, S_M contains one element. Each time (S_M, E_M, T_M) is found not closed, one element is added to S_M. This introduces a new row to S_M, so a new state in the conjecture. This can happen for at most $n-1$ times. For each counterexample of length at most m, there can be at most m strings that are added to S_M, and there can be at most $n-1$ counterexamples to distinguish n states. Thus, the size of S_M cannot exceed $n + m(n-1)$.

Initially, E_M contains $|I|$ elements. Each time (S_M, E_M, T_M) is found not consistent, one element is added to E_M. This can happen for at most $n-1$ times to distinguish n states. Thus, the size of E_M cannot exceed $|I| + n - 1$.

The maximum size of $(S_M \cup S_M \cdot I) \times E_M$ is given as

$$(\overbrace{n + m(n-1)}^{|S_M|} + \overbrace{|I|(n + m(n-1))}^{|S_M \cdot I|})\,(\overbrace{|I| + n - 1}^{|E_M|}) = O(|I|^2nm + |I|mn^2)$$

Thus, ${L_M}^*$ produces a correct conjecture by asking maximum $O(|I|^2nm + |I|mn^2)$ output queries.

5.2.6 Example

For the conjecture ${M_M}^{(1)}$ in Figure 5.2 for learning the Mealy machine $\mathcal{M}$ in Figure 5.1, we have a counterexample as $\nu = a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$. According to the adapted method for processing counterexample, ${L_M}^*$ adds all the prefixes of ν, i.e., a, $a \cdot b$, $a \cdot b \cdot a$, $a \cdot b \cdot a \cdot b$, $a \cdot b \cdot a \cdot b \cdot b$, $a \cdot b \cdot a \cdot b \cdot b \cdot a$, and $a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$ to S_M and extends $S_M \cdot I$ accordingly. The table is then filled with the missing elements by asking output queries. Table 5.3a shows the resulting observation table. Then, ${L_M}^*$ checks if the table is closed and consistent.

Table 5.3a is closed but not consistent since $\epsilon \cong_{E_M} a \cdot b$, but $T_M(\epsilon \cdot a, a) \neq T_M(a \cdot b \cdot a, a)$. To make the table consistent, the string $a \cdot a$ is added to E_M and the table is filled accordingly. Table 5.3b shows the resulting observation table, in which the rows ϵ and $a \cdot b$ have become different. Now, ${L_M}^*$ checks if Table 5.3b is closed and consistent.

Table 5.3b is closed but not consistent since $a \cdot b \cong_{E_M} a \cdot b \cdot a$ but $T_M(a \cdot b \cdot a, a \cdot a) \neq T_M(a \cdot b \cdot a \cdot a, a \cdot a)$. To make the table consistent, the string $a \cdot a \cdot a$ is added to E_M. For the same rows, the other reason for inconsistency is due to $T_M(a \cdot b \cdot b, a \cdot a) \neq T_M(a \cdot b \cdot a \cdot b, a \cdot a)$. Therefore, the string $b \cdot a \cdot a$ is also added to E_M and the table is filled accordingly. Table 5.3c shows the resulting observation table, in which the rows $a \cdot b$ and $a \cdot b \cdot a$ have become different.

Table 5.3c is closed and consistent, and thus ${L_M}^*$ terminates by making a conjecture isomorphic to $\mathcal{M}$ (Figure 5.1). The total number of output queries asked by ${L_M}^*$ is 85.

5.3 Improvements to Mealy Machine Inference

We propose improvements to the algorithm of learning Mealy machines by providing a new method for processing counterexamples in the observation table (S_M, E_M, T_M). The complexity calculations and the experimental results of our proposal evidence a significant reduction in the output queries that the algorithm asks during the learning procedure. We denote the

	a	b
$\boxed{\epsilon}$	x	x
a	y	x
$\boxed{a \cdot b}$	x	x
$a \cdot b \cdot a$	x	x
$a \cdot b \cdot a \cdot b$	x	x
$a \cdot b \cdot a \cdot b \cdot b$	x	x
$a \cdot b \cdot a \cdot b \cdot b \cdot a$	x	x
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$	y	x
b	x	x
$a \cdot a$	y	x
$a \cdot b \cdot b$	x	x
$a \cdot b \cdot a \cdot a$	x	x
$a \cdot b \cdot a \cdot b \cdot a$	y	x
$a \cdot b \cdot a \cdot b \cdot b \cdot b$	x	x
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot b$	x	x
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot a$	y	x
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot b$	x	x

(a) Adding the prefixes of the counterexample $a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$ to S_M

	a	b	$a \cdot a$
ϵ	x	x	$x \cdot y$
a	y	x	$y \cdot y$
$\boxed{a \cdot b}$	x	x	$x \cdot x$
$\boxed{a \cdot b \cdot a}$	x	x	$x \cdot x$
$a \cdot b \cdot a \cdot b$	x	x	$x \cdot y$
$a \cdot b \cdot a \cdot b \cdot b$	x	x	$x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot a$	x	x	$x \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$	y	x	$y \cdot y$
b	x	x	$x \cdot x$
$a \cdot a$	y	x	$y \cdot y$
$a \cdot b \cdot b$	x	x	$x \cdot x$
$a \cdot b \cdot a \cdot a$	x	x	$x \cdot y$
$a \cdot b \cdot a \cdot b \cdot a$	y	x	$y \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot b$	x	x	$x \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot b$	x	x	$x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot a$	y	x	$y \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot b$	x	x	$x \cdot x$

(b) Adding $a \cdot a$ to E_M

	a	b	$a \cdot a$	$a \cdot a \cdot a$	$b \cdot a \cdot a$
ϵ	x	x	$x \cdot y$	$x \cdot y \cdot y$	$x \cdot x \cdot x$
a	y	x	$y \cdot y$	$y \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot x$	$x \cdot x \cdot x$
$a \cdot b \cdot a$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot y$
$a \cdot b \cdot a \cdot b$	x	x	$x \cdot y$	$x \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot a$	x	x	$x \cdot y$	$x \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$	y	x	$y \cdot y$	$y \cdot y \cdot y$	$x \cdot x \cdot x$
b	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot y$
$a \cdot a$	y	x	$y \cdot y$	$y \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot y$
$a \cdot b \cdot a \cdot a$	x	x	$x \cdot y$	$x \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot a$	y	x	$y \cdot y$	$y \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot b$	x	x	$x \cdot y$	$x \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot y$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot a$	y	x	$y \cdot y$	$y \cdot y \cdot y$	$x \cdot x \cdot x$
$a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot x$	$x \cdot x \cdot x$

(c) Adding $a \cdot a \cdot a$ and $b \cdot a \cdot a$ to E_M

Table 5.3: The Observation Tables (S_M, E_M, T_M) for processing the counterexample $a{\cdot}b{\cdot}a{\cdot}b{\cdot}b{\cdot}a{\cdot}a$ for ${M_M}^{(1)}$ using the adapted method from L^*. The boxes in the tables show the rows which make the tables inconsistent.

algorithm with the improved method for processing counterexamples by ${L_M}^+$. In the following, we describe the idea of the improvement in Section 5.3.1, followed by the algorithm ${L_M}^+$ in Section 5.3.2 and the improved method for processing counterexamples in Section 5.3.3. We provide the correctness proof of the method in Section 5.3.4, complexity of ${L_M}^+$ in Section 5.3.5, illustration of the method in Section 5.3.6 and finally the discussion on the improvements in Section 5.3.7.

5.3.1 Motivation

To explain the main intuition behind the improvement, we recall the discussion on learning DFA in Chapter 4. As a suggestion to improve the algorithm L^*, Rivest & Schapire [RS93] argued to avoid consistency check by keeping only inequivalent rows in the first part of the observation table (S_D, E_D, T_D). This is how the size of the table can be reduced and so the total number of membership queries that are required to fill the table. They noticed that an inconsistency occurs due to improper handling of counterexamples in the table. They proposed a method for processing counterexamples, which cost less in terms of membership queries for learning DFA. However, their method relaxed the prefix-closed and suffix-closed properties of the table. Consequently, the new conjecture might not be consistent with the table (Theorem 1), and therefore, might still classify the previous counterexamples incorrectly. This means the same counterexample can potentially be used to answer several equivalence queries in L^* [BDG97].

Our improvement in the algorithm for learning Mealy machines is due to Rivest & Schapire's idea. We also suggest to keep only inequivalent rows in S_M so that inconsistencies can never occur. However, we propose a new method for processing counterexamples such that it does not import the same problem as in the case of Rivest & Schapire. Our method for processing counterexample keeps (S_M, E_M, T_M) prefix-closed and suffix-closed, and therefore, the new conjecture is always consistent with the observations in (S_M, E_M, T_M), according to Theorem 3.

5.3.2 The Algorithm ${L_M}^+$

In the algorithm ${L_M}^+$, the definition of the observation table (S_M, E_M, T_M), described in Section 5.2.1, and the basic flow of the algorithm, described in Section 5.2, remain unchanged. However, the additional property of (S_M, E_M, T_M) is that all the rows in S_M are inequivalent, i.e., for all $s, t \in S_M$, $s \not\cong_{E_M} t$. This means ${L_M}^+$ does not need to check for consistency because it always trivially holds. However, ${L_M}^+$ processes counterexamples according to the new method, which is described in the following.

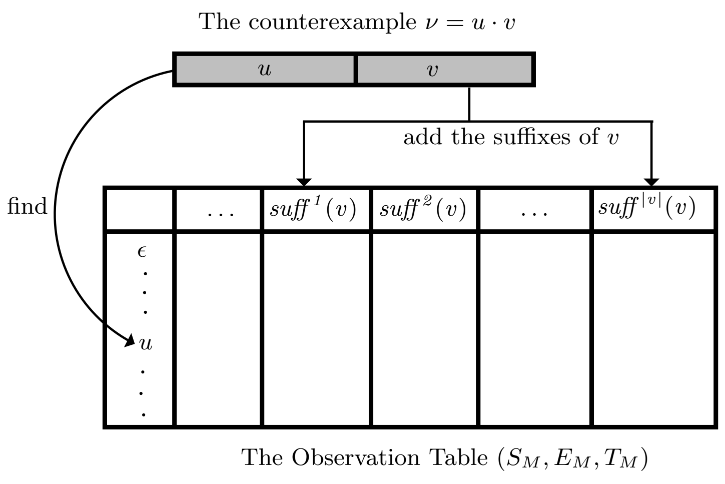


Figure 5.3: Conceptual view of the method for processing counterexamples in ${L_M}^+$

5.3.3 Processing Counterexamples in ${L_M}^+$

Let $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ be the conjecture from the closed (and consistent) observation table (S_M, E_M, T_M) for learning the machine $\mathcal{M}$. Let ν be a string from I^+ as a counterexample such that $\lambda_M(q_{0M}, \nu) \neq \lambda_{\mathcal{M}}(q_{0\mathcal{M}}, \nu)$. The main objective of a counterexample is to distinguish the conjecture from the black box machine. That means, the counterexample must contain a distinguishing sequence to distinguish at least two seemingly equivalent states of the conjecture; so that when applying the distinguishing sequence on these states, they become different. In our method, we look for the distinguishing sequence in the counterexample and directly add it to E_M, so that the two seemingly equivalent rows[1] in S_M become different. For that purpose, we divide ν into its appropriate prefix and suffix such that the suffix contains the distinguishing sequence. We divide ν by looking at its longest prefix in $S_M \cup S_M \cdot I$ and take the remaining string as the suffix. Let $\nu = u \cdot v$ such that $u \in S_M \cup S_M \cdot I$. If there exists $u' \in S_M \cup S_M \cdot I$ another prefix of ν then $|u| > |u'|$, i.e., u is the longest prefix of ν in $S_M \cup S_M \cdot I$. The idea of selecting u from the observation table is that u is the access string that is already known such that $\lambda_M(q_{0M}, u) = \lambda_{\mathcal{M}}(q_{0\mathcal{M}}, u)$. The fact that ν is a counterexample then $\lambda_M(q_{0M}, u \cdot v) \neq \lambda_{\mathcal{M}}(q_{0\mathcal{M}}, u \cdot v)$ must hold. That means, v contains the distinguishing sequence to distinguish two rows in S_M. So, it is sufficient to add v to E_M. In fact, we add all the suffixes of v such that E_M remains suffix-closed.

Figure 5.3 provides a conceptual view of the method for processing a counterexample ν. It shows that ν is divided into the prefix u and the suffix v, such that $u \in S_M \cup S_M \cdot I$. Then, ν is

[1]Recall that the rows in S_M represent the states of the conjecture

processed by adding all the suffixes of v to E_M. The correctness proof of the method is given in the following section.

5.3.4 Correctness

Let $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ be the conjecture from the closed (and consistent) observation table (S_M, E_M, T_M). Let $\nu = u \cdot i \cdot v$ be the counterexample for M_M such that $\lambda_M(q_{0M}, u \cdot i \cdot v) \neq \lambda_{\mathcal{M}}(q_{0\mathcal{M}}, u \cdot i \cdot v)$. Let $u \cdot i$ be the longest prefix of ν in $S_M \cup S_M \cdot I$ and v be the corresponding suffix of ν. If ν is a counterexample then it must distinguish $[u \cdot i]$ from a seemingly equivalent state, i.e., $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, u \cdot i \cdot v) \neq \lambda_{\mathcal{M}}(q_{0\mathcal{M}}, t \cdot v)$, for some $t \in S_M$ such that $[t] = [u \cdot i]$. Thus, v contains a distinguishing sequence for the rows $u \cdot i$ and t.

Suppose we process ν in (S_M, E_M, T_M) by adding all the suffixes of v to E_M. Lets name the table as (S'_M, E'_M, T'_M) after this addition. Later, we ask output queries to fill the missing elements of the table (S'_M, E'_M, T'_M). Then, E'_M contains the distinguishing sequence that distinguishes the rows t and $u \cdot i$ in (S'_M, E'_M, T'_M). That is, there must exist some experiment $e \in E'_M$ such that $T'_M(t, e) \neq T'_M(u \cdot i, e)$. This implies that $u \cdot i \ncong_{E'_M} t$. In fact, $u \cdot i \in S'_M \cdot I$, since $t \in S'_M$ and there cannot be two equivalent rows in S'_M. If $u \cdot i \in S'_M \cdot I$ then trivially $u \in S'_M$. Moreover, in the table (S_M, E_M, T_M), if $u \cdot i \ncong_{E_M} s$, for $s \in S_M$, then in the extended table (S'_M, E'_M, T'_M), $u \cdot i \ncong_{E'_M} s$ also holds, for $s \in S'_M$. Therefore, $u \cdot i$ is a row in (S'_M, E'_M, T'_M) that is inequivalent to any row in S'_M. This makes the table not closed. Thus, making the table closed will move $u \cdot i$ to S'_M. Since, u is already in S'_M, this operation keeps (S'_M, E'_M, T'_M) prefix-closed. Since, S'_M is extended by one row, the new conjecture M'_M from the closed (S'_M, E'_M, T'_M) will contain at least one more state than M_M.

It is simple to check whether (S'_M, E'_M, T'_M) is suffix-closed, since E'_M is extended from E_M, which is suffix-closed, and E'_M contains the suffixes of v. Thus, (S'_M, E'_M, T'_M) is suffix-closed.

This proves the correctness of the method, since (S'_M, E'_M, T'_M) is a closed (and consistent) observation table that is prefix-closed and suffix-closed and contains the prefix $u \cdot i$ and the suffix v of the counterexample ν. Therefore, the conjecture M'_M from (S'_M, E'_M, T'_M) will be consistent with the function T'_M (Theorem 3) that will find at least one more state.

We summarize the correctness results concerning the method in the following theorem.

Theorem 4 *Let (S_M, E_M, T_M) be a closed (and consistent) observation table and M_M be the conjecture from (S_M, E_M, T_M). Let $\nu = u \cdot i \cdot v$ be the counterexample for M_M, where $u \cdot i$ is in $S_M \cup S_M \cdot I$. Let the table be extended as (S'_M, E'_M, T'_M) by adding all the suffixes of v to E_M, then the closed (and consistent) observation table (S'_M, E'_M, T'_M) is prefix-closed and suffix-closed. The conjecture M'_M from (S'_M, E'_M, T'_M) will be consistent with T'_M and must have at least one more state than M_M.* □

Algorithm 3 summarizes the algorithm ${L_M}^+$.

```
Input: The set of input symbols $I$
Output: Mealy machine conjecture $M_M$
begin
    initialize the observation table $(S_M, E_M, T_M)$ with the sets
    $S_M = \{\epsilon\}, E_M = I, S_M \cdot I = \{\epsilon \cdot i\}, \forall i \in I$ ;
    ask the output queries from $(S_M, E_M, T_M)$ ;
    update $(S_M, E_M, T_M)$ with the results of the queries ;
    while $(S_M, E_M, T_M)$ is not closed do
        if $(S_M, E_M, T_M)$ is not closed then
            find $t \in S_M \cdot I$ such that $t \not\cong_{E_M} s$, for all $s \in S_M$ ;
            move $t$ to $S_M$ ;
            ask output queries for the extended table ;
        end
        make the conjecture $M_M$ from $(S_M, E_M, T_M)$ ;
        if there is a counterexample $\nu$ for $M_M$ then
            divide $\nu = u \cdot v$ such that $u$ is the longest prefix in $S_M \cup S_M \cdot I$ ;
            add all the suffixes of $v$ to $E_M$ ;
            ask output queries for the extended table ;
        end
    end
    return the conjecture $M_M$ from $(S_M, E_M, T_M)$ ;
end
```

Algorithm 3: The Algorithm ${L_M}^+$

5.3.5 Complexity

We analyze the total number of output queries asked by ${L_M}^+$ in the worst case by the factors

- $|I|$, i.e., the size of I
- n, i.e., the number of states of the minimum machine $\mathcal{M}$
- m, i.e., the maximum length of any counterexample provided during the learning of $\mathcal{M}$

The size of S_M increases monotonically up to the limit of n as the algorithm runs. The only operation that extends S_M is *closed*. Every time (S_M, E_M, T_M) is not closed, one element is added to S_M. This introduces a new row to S_M, so a new state in the conjecture. This can happen at most $n - 1$ times, since it keeps one element initially. Hence, the size of S_M is at most n.

E_M contains $|I|$ elements initially. If a counterexample is provided then at most m suffixes are added to E_M. There can be provided at most $n-1$ counterexamples to distinguish n states, thus the maximum size of E_M cannot exceed $|I| + m(n-1)$.

The maximum size of $(S_M \cup S_M \cdot I) \times E_M$ is given as

$$(\overbrace{n}^{|S_M|} + \overbrace{|I|n}^{|S_M \cdot I|})\ (\overbrace{|I| + m(n-1)}^{|E_M|}) = O(|I|^2 n + |I|mn^2)$$

Thus, ${L_M}^+$ produces a correct conjecture by asking maximum $O(|I|^2 n + |I|mn^2)$ output queries.

5.3.6 Example

We illustrate the algorithm ${L_M}^+$ on the Mealy machine $\mathcal{M}$ given in Figure 5.1. Since, ${L_M}^+$ is only different from ${L_M}^*$ with respect to the method for processing counterexamples, the initial run of ${L_M}^+$ is same as described in Section 5.2.3. So, ${L_M}^+$ finds a closed (and consistent) table as Table 5.2 and draws the conjecture ${M_M}^{(1)}$, shown in Figure 5.2, from Table 5.2. Here, we illustrate how ${L_M}^*$ processes counterexamples to refine the conjecture.

For the conjecture ${M_M}^{(1)}$, we have a counterexample as $\nu = a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$. According to the improved method for processing counterexample, ${L_M}^+$ finds the longest prefix u of the counterexample in $S_M \cup S_M \cdot I$ in Table 5.2. The prefix $u = a \cdot b$ is the longest prefix found, so the remaining suffix is $v = a \cdot b \cdot b \cdot a \cdot a$. The algorithm adds all the suffixes of v, i.e., a, $a \cdot a$, $b \cdot a \cdot a$, $b \cdot b \cdot a \cdot a$ and $a \cdot b \cdot b \cdot a \cdot a$ to E_M. The table is filled by asking output queries for the missing elements. Table 5.4a is the resulting observation table. Then, ${L_M}^+$ checks if the table is closed.

Table 5.4a is not closed since the rows b and $a \cdot b$ are not equivalent to any rows in S_M. Hence, the rows b and $a \cdot b$ are moved to S_M and the table is extended accordingly. The table is filled by asking output queries for the missing elements. Table 5.4b is the resulting observation table. Now, ${L_M}^+$ checks whether Table 5.4b is closed.

Table 5.4b is closed, and thus ${L_M}^+$ terminates by making a conjecture isomorphic to $\mathcal{M}$ (Figure 5.1). The total number of output queries asked by ${L_M}^+$ is 54.

5.3.7 Discussion

We have presented two algorithms for inferring Mealy machines, namely ${L_M}^*$ and ${L_M}^+$. The algorithm ${L_M}^*$ is a straightforward adaptation from the algorithm L^*. The algorithm ${L_M}^+$ is our proposal that contains a new method for processing counterexamples. Having the complexity calculations of the two algorithms, i.e,

	a	b	$a \cdot a$	$b \cdot a \cdot a$	$b \cdot b \cdot a \cdot a$	$a \cdot b \cdot b \cdot a \cdot a$
ϵ	x	x	$x \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$
a	y	x	$y \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x$	$y \cdot x \cdot x \cdot x \cdot x$
$\boxed{b}$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x \cdot y$
$a \cdot a$	y	x	$y \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x$	$y \cdot x \cdot x \cdot x \cdot x$
$\boxed{a \cdot b}$	x	x	$x \cdot x$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$

(a) Adding the suffixes of $v = a \cdot b \cdot b \cdot a \cdot a$ to E_M

	a	b	$a \cdot a$	$b \cdot a \cdot a$	$b \cdot b \cdot a \cdot a$	$a \cdot b \cdot b \cdot a \cdot a$
ϵ	x	x	$x \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$
a	y	x	$y \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x$	$y \cdot x \cdot x \cdot x \cdot x$
b	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x \cdot y$
$a \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$
$a \cdot a$	y	x	$y \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x$	$y \cdot x \cdot x \cdot x \cdot x$
$b \cdot a$	x	x	$x \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$
$b \cdot b$	x	x	$x \cdot y$	$x \cdot x \cdot x$	$x \cdot x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x \cdot x$
$a \cdot b \cdot a$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x \cdot y$
$a \cdot b \cdot b$	x	x	$x \cdot x$	$x \cdot x \cdot y$	$x \cdot x \cdot x \cdot x$	$x \cdot x \cdot x \cdot x \cdot y$

(b) Moving the rows b and $a \cdot b$ to S_M

Table 5.4: The Observation Tables (S_M, E_M, T_M) for processing the counterexample $a \cdot b \cdot a \cdot b \cdot b \cdot a \cdot a$ for ${M_M}^{(1)}$ using the improved method. The boxes in the tables show the rows which make the tables not closed.

- ${L_M}^* : O(|I|^2nm + |I|mn^2)$
- ${L_M}^+ : O(|I|^2n + |I|mn^2)$

it is observed that ${L_M}^+$ has a gain on the number of output queries compared to ${L_M}^*$. We discuss the different aspects of this gain in the following.

The crux of the complexity comparison comes from the fact that we are interested in systems that contain huge data sets. When these systems are learned as Mealy machines, the size of the input set I becomes large enough to cripple the learning procedure. In most cases, $|I|$ is a dominant factor over the number of the states n. Therefore, when we look on the parts of the complexity calculations which exhibit a difference, i.e.,

- $|I|^2nm$ for ${L_M}^*$ and
- $|I|^2n$ for ${L_M}^+$,

then it is obvious that ${L_M}^+$ has a clear gain over ${L_M}^*$ as $|I|$ grows.

Another aspect of the complexity gain of ${L_M}^+$ comes from the fact that it is not easy to obtain always "smart" counterexamples that are short and yet can find the difference between the black box machine and the conjecture. In practice, when the sources of getting counterexamples such as random testing or conformance testing methods (see discussion in Chapter 3) are applied, counterexamples of arbitrary lengths are obtained. They are usually long input strings that run over the same states of the black box machine many times to exhibit the difference. When ${L_M}^*$ processes such counterexamples in the observation table by adding all the prefixes of the counterexample to S_M; it adds unnecessarily as many states as the length of the counterexample. This follows the extension of the table due to $S_M \cdot I$. However, after filling the table with output queries, it is realized that only few prefixes in S_M are the potential states. On the contrary, the method for processing counterexample in ${L_M}^+$ consists in adding the suffixes of only a part of the counterexample to E_M. Then, ${L_M}^+$ finds the exact rows through output queries which must be the potential states and then moves the rows to S_M (see Section 5.3.4). So, the length of a counterexample m is less worrisome when applying ${L_M}^+$. As m becomes large, ${L_M}^+$ has more gain over ${L_M}^*$.

From the above discussion, we conclude that ${L_M}^+$ outperforms ${L_M}^*$, notably when the size of the input set I and the length of the counterexamples m are large. We have also confirmed the gain of ${L_M}^+$ over ${L_M}^*$ by experimentation on a case study: *CWB* [MS04], which is a workbench of synthetic finite state models of real world systems. We have measured the average case complexity of the two algorithms, where m, I and n are of different sizes. The experimental results are given in Chapter 9 [1].

As a reference to our contribution in making automata learning practical, we remark that the complexity of the learning algorithm can still be tackled further by playing on the factor $|I|$. In fact, a Mealy machine has a fixed set of inputs, so all inputs are taken into account for learning. Whereas, if the machine has a similar behavior for the subset of inputs, then it can be learned with few inputs, where each input belongs to different subsets. We notice that the issue of large input set can be tackled by considering PFSM models, which has a reduced $|I|$ (due to its parameterized structure), compared to the same problem modeled as a Mealy machine. In PFSM, we can consider few key inputs and consider the rest as parameters associated with the inputs which show similar behaviors. Thus, the complexity of the algorithm can be further reduced in practice. This topic is discussed in Chapter 7.

5.4 Relation of M_M with $\mathcal{M}$

A Mealy machine conjecture given by ${L_M}^+$ is an approximation of the unknown model. In the absence of an oracle, the procedure of learning cannot rely on the counterexamples which could

[1]Chapter 9 is dedicated for case studies. The result for this case study is also given in that chapter.

progressively lead to a correct conjecture. Therefore, we must have a formal relation between the conjecture and the unknown model which is known to be true at any given time. We prove that the Mealy machine conjecture M_M drawn from a closed (and consistent) observation table (S_M, E_M, T_M) is a Φ-*quotient* (Definition 6) of the unknown model, where $\Phi = E_M$. This is stated in the following theorem.

Theorem 5 *Let* $\mathcal{M} = \{Q_{\mathcal{M}}, I, O, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}}, q_{0\mathcal{M}}\}$ *be the unknown machine and* $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ *be the conjecture from the closed (and consistent) observation table* (S_M, E_M, T_M)*, then* M_M *is an* E_M*-quotient of* $\mathcal{M}$. □

Proof According to the definition of the Mealy machine quotient (Definition 6), the theorem can be proved in two parts.

1. Let M_M be a conjecture from the closed (and consistent) observation table (S_M, E_M, T_M), then for two states $q_M, q'_M \in Q_M$, there exists $s, t \in S_M$ such that $[s] = q_M$ and $[t] = q'_M$, according to Definition 12. We also know that $s \cong_{E_M} t$, if and only if $T_M(s, e) = T_M(t, e)$, for all $e \in E_M$. This is true for all the members of $[s]$ and $[t]$. Therefore, $[s] \cong_{E_M} [t]$ implies that $q_M = q'_M$. Otherwise, there exists $e \in E_M$ such that $T_M(s, e) \neq T_M(t, e)$ and hence $s \ncong_{E_M} t$. This means, $[s] \ncong_{E_M} [t]$, implies that $q_M \neq q'_M$. This proves the first part, since q_M and q'_M can only be equal when s and t are equivalent with respect to E_M, i.e., they produce same output on all the strings from E_M, and not otherwise.

2. For $q_M \in Q_M$, there exists $s \in S_M$ such that $[s] = q_M$ (Definition 12). We know that $T_M(s, e) = \mathit{suff}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e)) = \lambda_{\mathcal{M}}(\delta_{\mathcal{M}}(q_{0\mathcal{M}}, s), e)$, for all $e \in E_M$. Since, M_M is consistent with the observations in (S_M, E_M, T_M), then $\lambda_M(\delta_M(q_{0M}, s), e) = T_M(s, e)$ also holds (Theorem 3). This proves the second part, since $\lambda_{\mathcal{M}}(\delta_{\mathcal{M}}(q_{0\mathcal{M}}, s), e) = \lambda_M(\delta_M(q_{0M}, s), e)$. ■

5.5 Discussion on Processing Counterexamples in L^*

A question that naturally emerges from the improvement of the Mealy machine algorithm is the following: Why the method of processing counterexamples of ${L_M}^+$ cannot be used as an improvement to the basic DFA learning algorithm L^*?

The answer is stated in the following. The method of processing counterexamples that is used in the algorithm ${L_M}^+$ can also be used in the algorithm L^*. However, we have to take into account certain changes in the structure of the observation table (S_D, E_D, T_D) for this purpose. Due to those changes, the query complexity of the algorithm becomes greater than the complexity of the original algorithm L^*, given in Chapter 4, Section 4.1.4. The explanation is provided in the following. First we explain what changes in (S_D, E_D, T_D) we have to consider in order to use our method of processing counterexamples in L^* in Section 5.5.1. Then, we calculate the query complexity and compared to the original in Section 5.5.2

5.5.1 Required Changes in the Observation Table

Recall that for learning the DFA $\mathcal{D} = (Q_\mathcal{D}, \Sigma, \delta_\mathcal{D}, F_\mathcal{D}, q_{0\mathcal{D}})$ with the output function $\Lambda_\mathcal{D}$ and the complete output function $\lambda_\mathcal{D}$, we construct the observation table (S_D, E_D, T_D) to ask membership queries (with the help of $(S_D \cup S_D \cdot \Sigma) \times E_D)$ and to record the answers of the queries (with the help of T_D). The function T_D is defined as $(S_D \cup S_D \cdot \Sigma) \times E_D \longrightarrow \{0, 1\}$. That means, we record the answers as either 0 or 1 given by $\Lambda_\mathcal{D}$.

In order to process counterexamples in (S_D, E_D, T_D) according to our method, we shall add the suffixes of a counterexample to E_D. Recall that there must be at least one suffix that contains a distinguishing sequence to distinguish at least two rows in the table. If a counterexample is not the shortest one, then the suffix might end up with a letter that could not distinguish the rows. In this case, if we record only the information given by $\Lambda_\mathcal{D}$, then we can miss the vital information to distinguish rows. This information can be obtained through $\lambda_\mathcal{D}$, i.e., we must record the strings of $0s$ and $1s$ (given by $\lambda_\mathcal{D}$), instead of recording just 0 and 1 (given by $\Lambda_\mathcal{D}$). Precisely, we record only the suffix of the length of $e \in E_D$, likewise in the algorithm ${L_M}^+$. That is, for the query $s \cdot e$, $s \in S_D \cup S_D \cdot \Sigma, e \in E_D$, we record $\mathit{suff}^{|e|}(\lambda_\mathcal{D}(q_{0\mathcal{D}}, s \cdot e))$ in $T_D(s, e)$.

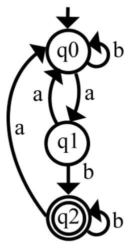


Figure 5.4: Example of a DFA for illustrating our method of processing counterexamples in L^*

	ϵ
ϵ	0
a	0
b	0

Table 5.5: Observation Table for learning the DFA in Figure 5.4

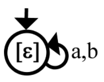


Figure 5.5: DFA Conjecture from Table 5.5

Let us explain this concept with the help of an example. Figure 5.4 shows a DFA defined on $\Sigma = \{a, b\}$. For learning this DFA, we construct a closed (and consistent) observation table, shown in Table 5.5, and draws the conjecture from the table, as shown in Figure 5.5.

Let $a \cdot b \cdot a$ be a counterexample for the conjecture. According to the method of processing counterexamples, we look for the longest prefix of the counterexample in $S_D \cup S_D \cdot \Sigma$ and add the suffixes of the remaining string to E_D. In this example, we find the remaining string as $b \cdot a$ and thus add the suffixes a and $b \cdot a$ to E_D. The table is then filled with the help of asking membership queries and recording the information given by $\Lambda_{\mathcal{D}}$, as shown in Table 5.6.

In this table, however the counterexample has been processed, we do not observe any rows that are distinguished from each other. This is due to the fact that the counterexample is not the shortest one and none of its suffix in E_D ends up on a letter that can distinguish the rows.

	ϵ	a	$b \cdot a$
ϵ	0	0	0
a	0	0	0
b	0	0	0

Table 5.6: Observation Table after processing the counterexample $a \cdot b \cdot a$ and filling the information given by $\Lambda_{\mathcal{D}}$

	ϵ	a	$b \cdot a$
ϵ	0	0	00
a	0	0	10
b	0	0	00

Table 5.7: Observation Table after processing the counterexample $a \cdot b \cdot a$ and filling the information given by $\lambda_{\mathcal{D}}$

Now, let us redefine the function T_D as $T'_D : (S_D \cup S_D \cdot \Sigma) \times E_D \longrightarrow \{0,1\}^*$, i.e., the function maps to the strings of $0s$ and $1s$. Then, we can construct the table to record the information given by $\lambda_{\mathcal{D}}$. For the same example, we construct the table (S_D, E_D, T'_D) as shown in Table 5.7. Now, we can clearly distinguish the rows ϵ and a due to the suffix $b \cdot a$ in E_D.

5.5.2 Complexity

We calculate the query complexity of the algorithm L^* with our method of processing counterexamples in the table (S_D, E_D, T'_D), with the help of factors

- $|\Sigma|$, i.e., the size of Σ
- n, i.e., the number of states of the minimum machine $\mathcal{D}$
- m, i.e., the maximum length of any counterexample provided during the learning of $\mathcal{D}$

Initially S_D contains only element, i.e., ϵ. The size augments when new rows are added for making the table closed. This can happen at most $n-1$ times. Thus, the maximum size of S_D cannot exceed n.

Initially E_D contains one element, i.e., ϵ. The size augments when new strings are added for processing counterexamples. If a counterexample is provided then at most m suffixes are added to E_D. There can be provided at most $n-1$ counterexamples to distinguish n states, thus the maximum size of E_D cannot exceed $m(n-1)$.

So, the maximum size of $(S_D \cup S_D \cdot \Sigma) \times E_D$ is given as

$$(\overbrace{n}^{|S_D|} + \overbrace{n|\Sigma|}^{|S_D \cdot \Sigma|})\,(\overbrace{m(n-1)}^{|E_D|}) = O(|\Sigma|mn^2)$$

In the original algorithm L^*, the number of membership queries is the maximum size of the table. This is because for each $s \in S_D \cup S_D \cdot \Sigma$, $e \in E_D$, the string $s \cdot e$ is considered as one query. There, we ask whether $s \cdot e$ is accepted or rejected and record 1 or 0 respectively.

In our case, we need to record the answer of each letter in e. Thus, the number of membership queries for getting the answer of $s \cdot e$ is actually $|e|$. This is true for all $e \in E_D$. If m is the maximum length of e, then the maximum number of membership queries in the worst case is $O(|\Sigma|mn^2) \cdot m = O(|\Sigma|m^2n^2)$.

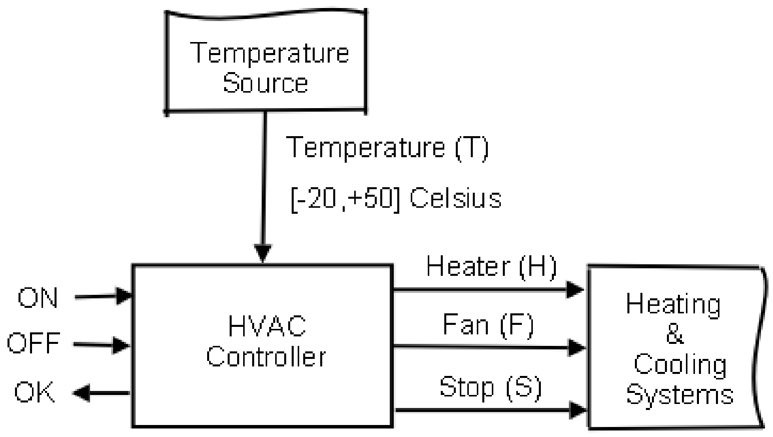
This complexity is greater than the worst time complexity of the original algorithm that is $O(|\Sigma|mn^2)$ (cf. Chapter 4, Section 4.1.4). Thus, our method of processing counterexamples in the algorithm ${L_M}^+$ does not claim improvements in the algorithm L^*.

5.6 Application: The HVAC controller

5.6.1 Description

We study the application of the Mealy machine inference on a simplified version of a real world example. The example is an HVAC (Heating-Ventilation-Air-Conditioning) controller that regulates the heating and cooling components in a building according to the specific temperature values. The specification of the controller is taken from the UPNP standardization[1]. Figure 5.6 presents the global HVAC system and the interactions of the controller with other components. There are various modes of the controller according to the specifications, however we focus on its very generic functionality, that is, controlling the heating and cooling components on the change of climate. The controller accepts inputs from its environment to control the components. It receives ON and OFF for starting and shutting down the components, and temperature T that changes its mode of controlling. The temperature values range from -20℃ to +50℃. The control modes are as follows. It turns on the heater H when temperature values are between -20℃ and 11℃, turns on the fan F when the temperature values are between 16℃ and 50℃. The controller stops S the components when the temperature value becomes out of range for the specific component or the command OFF is given from the environment. Each component can also be regulated on high h and low l speed levels depending upon the temperature intensity. The speed levels will be used in PFSM model inference and are not taking into account here.

[1]HVAC V1.0 Standardized DCP. http://www.upnp.org/standardizeddcps/hvac.asp



The working of the HVAC Controller on different temperature values is as follows.
For low temperature values, i.e., $[-20, 11]$, *it turns on the heater.*
For high temperature values, i.e., $[16, 50]$, *it turns on the fan.*
For medium temperature values, i.e., $[11, 16]$, *it stops the heater/fan.*

Figure 5.6: Global diagram of the HVAC system

5.6.2 Inference of the HVAC controller

The behavior of the HVAC controller can be modeled as a Mealy machine. We can use ${L_M}^+$ to explore the controller behaviors and its interactions with other components in the HVAC system. As a basic requirement of the algorithm, the input set of the controller is prepared which consists of inputs ON, OFF and the temperature values. In fact, there are many temperature values as inputs to the controller. We can consider each value as an input symbol to construct queries for learning. For example, a temperature value 12°C can be symbolized as $T12$ in the set. But this method yields many inconveniences.

- It enumerates all the data values with in range. In the example, there would be 71 different inputs only for the temperature values, i.e., [-20°C, +50°C], if we insist to include whole domain. It becomes more tricky when the values are from complex domains (e.g., floating-point, real etc).
- It would lead to unnecessary complexity of the model which cripples the practicality of the learning algorithm (as $|I|$ and n will increase).
- It will result in losing the control structure of the model, by mixing with the data values. In fact, the interest of learning is to extract the control structure or an abstraction which is relevant to investigate the functional behaviors of the system.

Due to these reasons, we design a restrictive input set for learning a Mealy machine model of the controller by selecting few temperature values from each interval. For example, we choose -5℃ and 5℃ as low temperatures from the interval [-20,11], 15℃ as a moderate temperature from the interval [12,15], 25℃ and 35℃ as high temperatures from the interval [16,50]. They are symbolized as T-5, T5, T15, T25 and T35 respectively. The complete input set for learning Mealy machine model of the controller in this experiment is given as $I = \{ON, OFF, T\text{-}5, T5, T15, T25, T35\}$. The observation table (S_M, E_M, T_M) is initialized and the algorithm ${L_M}^+$ is applied till the table is closed. Table 5.8 shows the final observation table (S_M, E_M, T_M). The outputs for the invalid inputs are recorded as Ω. For simplicity, we skip the rows in the table which contain Ω in all cells. Figure 5.7 shows the conjecture from Table 5.8. The total number of output queries to learn the conjecture is 140.

		E_M						
		ON	OFF	T-5	$T5$	$T15$	$T25$	$T35$
S_M	ϵ	OK	Ω	Ω	Ω	Ω	Ω	Ω
	ON	Ω	S	H	H	S	F	F
	$ON \cdot T5$	Ω	S	H	H	S	S	S
	$ON \cdot T25$	Ω	S	S	S	S	F	F
$S_M \cdot I$	$ON \cdot OFF$	OK	Ω	Ω	Ω	Ω	Ω	Ω
	$ON \cdot T$-5	Ω	S	H	H	S	S	S
	$ON \cdot T15$	Ω	S	H	H	S	F	F
	$ON \cdot T35$	Ω	S	S	S	S	F	F
	$ON \cdot T5 \cdot OFF$	OK	Ω	Ω	Ω	Ω	Ω	Ω
	$ON \cdot T5 \cdot T$-5	Ω	S	H	H	S	S	S
	$ON \cdot T5 \cdot T5$	Ω	S	H	H	S	S	S
	$ON \cdot T5 \cdot T15$	Ω	S	H	H	S	F	F
	$ON \cdot T5 \cdot T25$	Ω	S	H	H	S	F	F
	$ON \cdot T5 \cdot T35$	Ω	S	H	H	S	F	F
	$ON \cdot T25 \cdot OFF$	OK	Ω	Ω	Ω	Ω	Ω	Ω
	$ON \cdot T25 \cdot T$-5	Ω	S	H	H	S	F	F
	$ON \cdot T25 \cdot T5$	Ω	S	H	H	S	F	F
	$ON \cdot T25 \cdot T15$	Ω	S	H	H	S	F	F
	$ON \cdot T25 \cdot T25$	Ω	S	S	S	S	F	F
	$ON \cdot T25 \cdot T35$	Ω	S	S	S	S	F	F

Table 5.8: Closed (and Consistent) Observation Table for learning a Mealy machine model of the HVAC controller

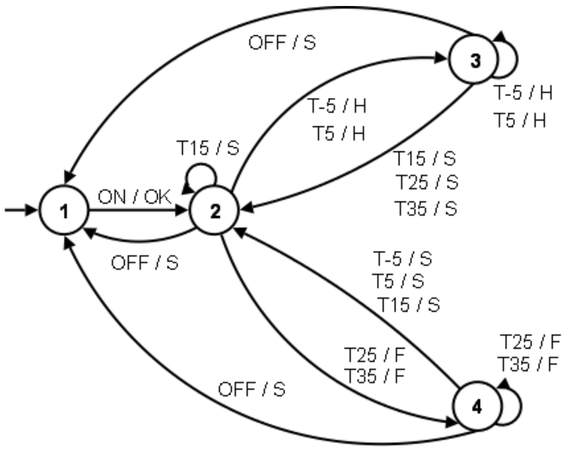


Figure 5.7: Mealy Machine conjecture of the HVAC Controller from Table 5.8

5.7 Conclusion

This chapter discussed the inference of Mealy machines from a black box machine. We studied the adaptation of Angluin's algorithm L^* to infer Mealy machines. We proposed improvements to the basic adaptation by providing a new method for processing counterexamples in the observation table. The complexity analysis showed that our improved algorithm has a significant gain over the adapted algorithm. The gain has also been confirmed on a case study for measuring the average case complexity of the algorithms.

Later, we discussed the approximation of the Mealy machine conjecture given by the learning algorithm with respect to the unknown model. We proved that a Mealy machine conjecture drawn from a closed (and consistent) observation table (S_M, E_M, T_M) is an E_M-*quotient* of the unknown model.

We also discussed why our proposed method of processing counterexamples does not improve complexity in the basic algorithm L^*. We explained the algorithm with the help of an example and calculated its worst case complexity with our method.

Finally, an application of the Mealy machine inference for the HVAC controller is given and its results are discussed.

We believe that learning Mealy machines is a step forward towards learning more enhanced models. From the discussions in this chapter – especially the complexity analysis of the algorithm and the inference of the HVAC controller – it is evident that learning enhanced models is strongly desirable, such that they could adequately model the structurally complex systems and the complexity of learning could be tackled. Therefore, we propose PFSM models (Definition 4) and provide an algorithm for leaning such models. This topic is discussed in Chapter 7.

Chapter 6

Integration Testing

This chapter provides a framework for testing integrated system of black box components using the partial models of the components. It describes the system architecture, the composition of the components and the integration testing procedure.

6.1 Motivation

The experience in component based software engineering is evident that the integration of high-quality components may not yield high-quality software systems. That is why integration testing came into place to assess the quality of the integrated system [GTWJ03]. It is not easy to evaluate all the possible interactions between the components in the system and uncover inter-component faults. It is even harder to assess the quality of the integrated system when the components are used without source code, specifications or other related information.

We propose the approach of combining learning and testing techniques to test the integrated system composed of black box components. In the previous chapters, we have explained how the components can be learned individually in order to extract their finite state models. At this stage, we use these models to evaluate the quality of the integrated system by applying integration testing techniques. Specifically, we want to understand how the components interact with each other in the system, or what conclusions can be inferred about the system resulting from the composition of the components. In fact, unit learning of components cannot confirm their reliable behavior in the system because i) the learned models are partial and thus their composition cannot portray the integrated system exactly, ii) the system may suffer from compositional problems such as deadlocks, livelocks or behavioral compatibility issues, which can only be detected with integration testing activities.

In this chapter, we propose an integration testing framework for the system of black box components using their partially learned models. Our goal is to automate integration testing by deriving tests from the learned models to trigger the potential interactions between the components in the system. The integration testing activity from partially learned models provides results in two dimensions:

- The models of the components can be refined iteratively whenever the derived tests find discrepancies between the behaviors of the models and the actual system. The refined models will provide better understanding of the system, and also more tests can be generated for the system.
- The compositional problems and potential errors in the system can be discovered during the course of testing.

Recall that we do not assume the provision of formal specifications so that tests could be generated for the system and errors could be uncovered. In this case, our approach can guarantee the integration testing based upon at least the quotients of black box components. Moreover, the models are refined iteratively which enhance confidence when tests are generated from the refined models.

The organization of the chapter is as follows. Section 6.2 describes the architecture of the integrated system and the composition of the components in the system. Section 6.3 explains the approach of learning and testing integrated systems. Section 6.4 concludes the chapter.

6.2 The Integrated System

We first describe the system architecture in Section 6.2.1, then provide a formal model of the system of Mealy components in Section 6.2.2. Later, we define the product of Mealy components in Section 6.2.3.

6.2.1 System Architecture

We consider an integrated system of black box components in which components are communicating asynchronously through their input/output interfaces with each other and with the system's environment. The components communicate by receiving inputs through their input interfaces and sending outputs through their output interfaces. In the system architecture, we distinguish two kinds of interfaces, namely, external and internal. The *external interfaces* of the system are the ones through which the components communicate with the environment. The *internal interfaces* of the system are the ones through which the components communicate with each other. We can distinguish external and internal inputs (outputs) of the system through the type of interfaces. We write simply inputs (outputs) to generalize external or internal; unless they are specified.

It is important to mention that the number of components and the way they are assembled in the system are known a priori. This means, we can easily distinguish the external and internal interfaces in the system. However, the notion of external and internal vary from one system architecture to the other. We may assemble the system in a different way so that the interfaces are distinguished differently. Here, we assume that once the architecture is built for integration, the external and internal interfaces of the system are not changed during the whole integration testing procedure. Moreover, we assume that both external and internal interfaces are observable, but only the external interfaces are controllable. This means that the communication between the components can be monitored but cannot be interrupted on the integrated level.

We assume that the system has a single message in transit, i.e., for each component and each input, only one output is produced. Furthermore, the components in the system are *deterministic* and *input-enabled*. This means when a component receives an input on any of its state, one and only one transition on the state is enabled and an observable output is produced. Recall that the observability can be handled, e.g., by considering Ω as the output, when the given input is invalid or the component responses with no outputs. The absence of outputs can be detected with the help of timeouts.

The system is in *stable* mode when no transition of any components is enabled. The system can receive external inputs only when it is in stable mode. The architecture of the system with n components is shown in Figure 6.1, in which the components communicate with the environment through the external interfaces and to each other through the internal interfaces of the system. The dotted line shows that the internal interfaces are observable to the environment.

6.2.2 Formal Model of a Mealy System

The architecture of the system composed of Mealy components (or shortly, Mealy system) is described as follows:

Assume a system of n components, i.e., $\{C_1, \ldots, C_n\}$, where for each $i \neq j, 1 \leq i, j \leq n$, $C_i = (Q_i, I_i, O_i, \delta_i, \lambda_i, q_{0i})$ and $C_j = (Q_j, I_j, O_j, \delta_j, \lambda_j, q_{0j})$ are Mealy components, such that $I_i \cap I_j = \emptyset$ and $O_i \cap O_j = \emptyset$. The components C_i and C_j communicate if $I_i \cap O_j \neq \emptyset$ or $O_i \cap I_j \neq \emptyset$. Let I be the union of all inputs, i.e., $I = \bigcup_{i=1}^{n} I_i$, and O be the union of all outputs, i.e., $O = \bigcup_{i=1}^{n} O_i$. The set $I_{ext} = I \backslash O$ contains the external inputs of the system that can be given from the environment and $O_{ext} = O \backslash I$ contains the external outputs of the system that can be given to the environment. We consider a system with at least one external input and one

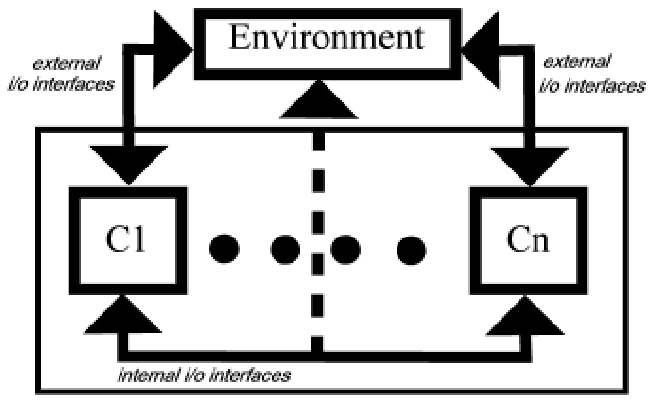


Figure 6.1: Architecture of the Integrated System of n Components. The dotted line shows that the internal interfaces are observable to the environment.

external output. For each component C_i it holds that if $a \in I_i$ then either $a \in I_{ext}$ or $a \in O_j$, and if $a \in O_i$ then $a \in O_{ext}$ or $a \in I_j$ for some C_j.

We assume that the system is working under slow environment that provides external inputs only when the system is in stable mode. Initially, the system is in stable mode and can be stimulated by providing an external input from I_{ext}. A component receives the input and produces either an external output to the environment or an internal input to another component. The other component receives the internal input and produces either an external output to the environment or an internal input to another component. Finally, the system produces an external output to the environment and stays in the stable mode until it receives another external input. An example of the Mealy system Sys that consists of two components M and N is given in Figure 6.2. We write the external inputs (outputs) in uppercase letters and internal inputs (outputs) conversely. The models $M = (Q_M, I_M, O_M, \delta_M, \lambda_M, q_{0M})$ and $N = (Q_N, I_N, O_N, \delta_N, \lambda_N, q_{0N})$ are given in Figure 6.3, in which $I_M = \{A, b, E\}$, $O_M = \{x, y, Z\}$, $I_N = \{x, y, t\}$ and $O_N = \{b, D, F\}$.

6.2.3 Product of Mealy Components

Given a Mealy system $\{C_1, \ldots, C_n\}$, the slow asynchronous product of the components in the system, denoted by $\prod$, is a Mealy machine $(Q, I, O, \delta, \lambda, q_0)$ such that

- $I = \bigcup_{i=1}^{n} I_i$ and $O = \bigcup_{i=1}^{n} O_i$ are the sets of all inputs and outputs respectively.

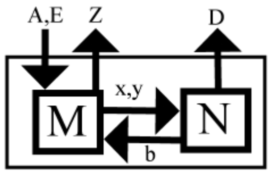
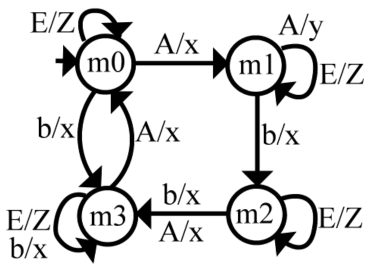
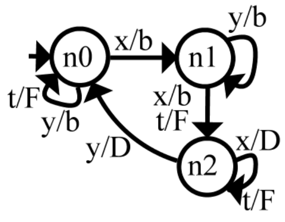


Figure 6.2: Example of the Mealy System Sys.



(a) Component M



(b) Component N

Figure 6.3: Components M and N of the System Sys in Figure 6.3.

- $Q \subseteq Q_1 \times \ldots Q_n \times (I \cup \{\epsilon\})$ is a subset of the Cartesian product of the states of the components and the union of their input sets. Since, the system has a single message in transit, a state in Q holds only one input. The symbol ϵ in the state represents no input, i.e., the system is in stable mode and the state is a stable state of the system waiting for an external input.
- $q_0 = (q_{0_1}, \ldots, q_{0_n},\ \epsilon) \in Q$ is the initial state

The functions δ, λ and the set Q are the smallest sets obtained by applying the following inference rules. For $1 \leq i \leq n$, let $C_i = (Q_i, I_i, O_i, \delta_i, \lambda_i, q_{0_i})$ and $q_i \in Q_i$ then for $a \in I$

1. If $a \in I_{ext} \cap I_i$ and $(q_1, \ldots, q_n,\ \epsilon) \in Q$ then $\lambda((q_1, \ldots, q_n,\ \epsilon), a) = \lambda_i(q_i, a)$ and
   - If $\lambda_i(q_i, a) \in O_{ext}$ then $\delta((q_1, \ldots, q_n,\ \epsilon), a) = (q_1, \ldots, \delta_i(q_i, a), \ldots, q_n,\ \epsilon)$
   - If $\lambda_i(q_i, a) \notin O_{ext}$ then $\delta((q_1, \ldots, q_n,\ \epsilon), a) = (q_1, \ldots, \delta_i(q_i, a), \ldots, q_n,\ \lambda_i(q_i, a))$
2. If $a \in I_i \backslash I_{ext}$ and $(q_1, \ldots, q_n,\ a) \in Q$ then $\lambda((q_1, \ldots, q_n,\ a), a) = \lambda_i(q_i, a)$ and
   - If $\lambda_i(q_i, a) \in O_{ext}$ then $\delta((q_1, \ldots, q_n, a), a) = (q_1, \ldots, \delta_i(q_i, a), \ldots q_n,\ \epsilon)$
   - If $\lambda_i(q_i, a) \notin O_{ext}$ then $\delta((q_1, \ldots, q_n, a), a) = (q_1, \ldots, \delta_i(q_i, a), \ldots, q_n,\ \lambda_i(q_i, a))$

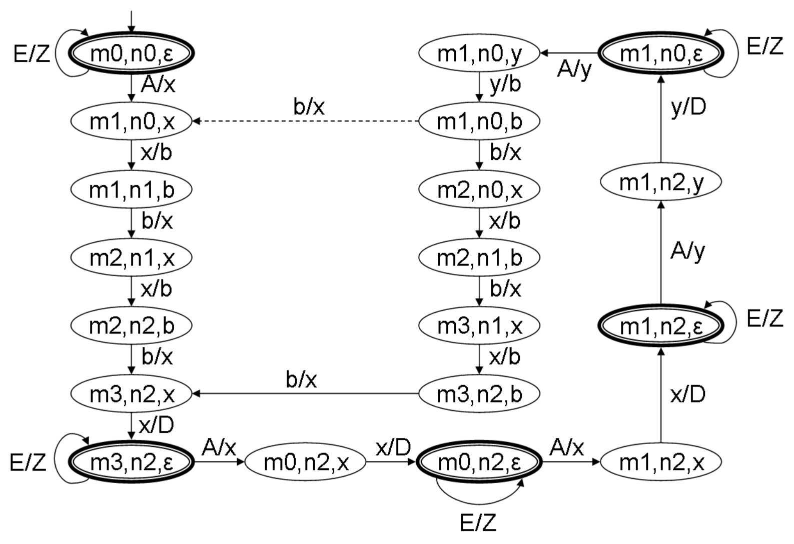


Figure 6.4: The product $\prod^{Sys}$ of components M and N. The stable states of the product are shown in double circles. The dotted arrow shows how the product can be minimized.

Note that the product $\prod$ is not input-enabled, i.e., for certain $q \in Q$ and $a \in I$, $\lambda(q, a)$ and $\delta(q, a)$ are undefined. However, all the stable states of the product have a transition for each external input, i.e., for all $(q_1, \ldots, q_n, \ \epsilon) \in Q$ and $a \in I_{ext}$, $\lambda((q_1, \ldots, q_n, \ \epsilon), a)$ and $\delta((q_1, \ldots, q_n, \ \epsilon), a)$ are defined.

As an example, the product of components M and N, denoted by $\prod^{Sys}$, is shown in Figure 6.4. The stable states of the product are shown in double circles.

6.3 The Approach of Learning and Integration Testing

We explain our approach of learning and integration testing of the system of black box components in five steps. We introduce each step here and then explain in the subsections. In Step 1, we learn the components in isolation using the learning algorithms described in the previous chapters. For each component, we consider a restrictive input set, contrary to considering all possible inputs for learning. In Step 2, we construct the product of the learned models obtained in the previous step (Step 2(a)). The models may partially represent the system, therefore, we analyze the product of the models to find compositional problems (Step 2(b)). The problem found in the product will be confirmed on the actual system (Step 2(c)). If confirmed, we terminate the procedure by reporting the problem. Otherwise, the problem is an artifact and we proceed for refining the product by considering the problem as a counterexample for the product. In Step 3, we refine the product by relearning one or more components using the learning algorithms and then repeat Step 2 on the refined product. In Step 4, we generate tests from the product of the learned models which contains no compositional problems thanks to Steps 2 and 3. The tests are executed on the real system, which may find discrepancies between the product and the system. In case a discrepancy is found, we proceed for Step 5. Otherwise, the procedure will terminate. In Step 5, we resolve the discrepancy if it is classified as a real error in the system or just an artifact due to the partiality of the product. In the former case, we terminate the procedure by reporting the error. In the latter case, we refine the product by proceeding for Step 3 and considering the discrepancy as a counterexample. The iterative procedure is repeated until any compositional problem in the system is found, a real error in the system is found or no discrepancy between the product of the learned models and the system is found.

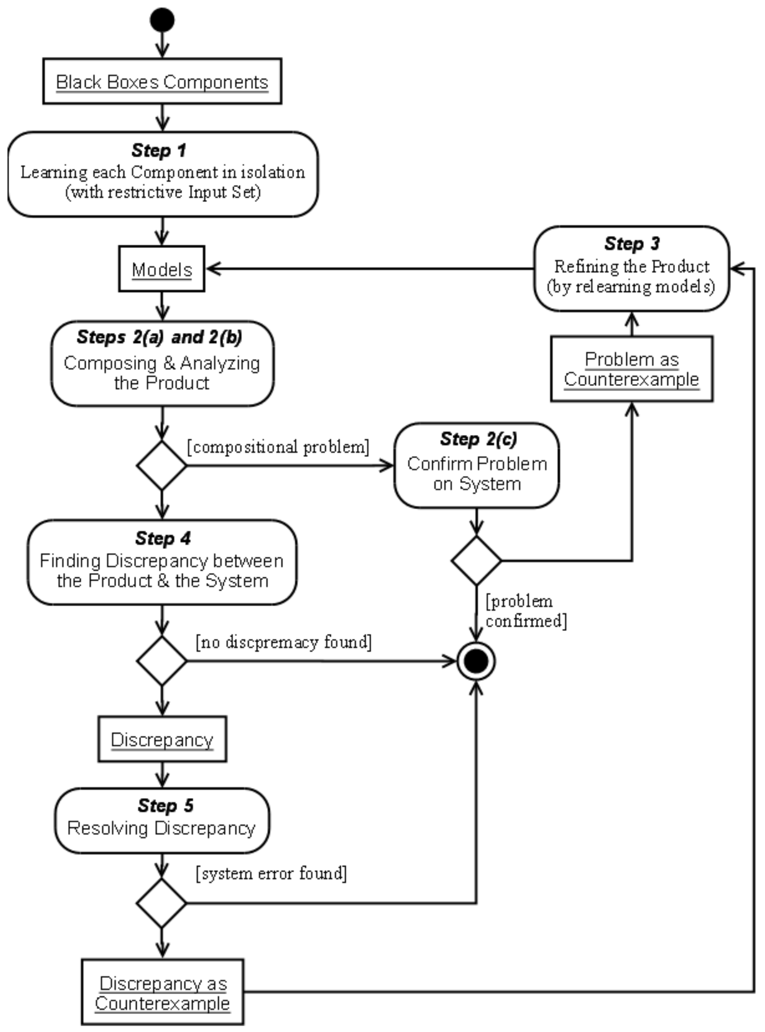

Black Boxes Components
Step 1
Learning each Component in isolation (with restrictive Input Set)
Models
Step 3
Refining the Product (by relearning models)
Steps 2(a) and 2(b)
Composing & Analyzing the Product
Problem as Counterexample
[compositional problem]
Step 2(c)
Confirm Problem on System
Step 4
Finding Discrepancy between the Product & the System
[problem confirmed]
[no discpremacy found]
Discrepancy
Step 5
Resolving Discrepancy
[system error found]
Discrepancy as Counterexample

Figure 6.5: Learning and Testing Approach for an Integrated System

The approach of learning and testing for integrated systems is illustrated in Figure 6.5. In the following, we explain each step of the approach in details and demonstrate it on Mealy systems with the help of the example Sys in Figure 6.2.

6.3.1 Step 1: Learning Components with Restrictive Input Sets

The first step is to learn the components in the system to extract their models. For each component, we need to construct its input alphabet in order to start its learning process. A straightforward method is to consider all possible inputs. Considering the complexity issues, this may not be a reasonable approach, since a subset of the component inputs is used in the integrated system normally. However, we can restrict the size of the input alphabets by knowing which inputs a component can receive from the other components in the system. In reality, this knowledge is usually not completely available beforehand and inputs to the components are discovered during testing or system execution.

In our approach, we construct the input alphabets of the components on-the-fly during learning. The idea is to select some components in the system and construct their input set manually. Then learn those components one by one and extract their models. The learning of the components reveals the outputs which would be given as the inputs to the other components in the system. This means we can learn the other components considering only those inputs which are possible in the system. In this way, we can restrict the input set of some components by observing what possible inputs the components can receive in the system.

In this approach, if the learning of a component C_k discovers new inputs for an already learned component C_j, then C_j will be relearned with the new set of inputs. This can be seen as a particular type of a counterexample for the component which has already been learned, but now its input set is augmented with new inputs by the learning of C_k. Hence, the learning of the individual components is an iterative process in which a component may be learned iteratively whenever its new inputs are discovered. The approach is useful when the input alphabets of the components are of formidable sizes. Fortunately, the input set of each component is finite, so the process of discovering new inputs for each component must converge to the subset of the inputs used by other components in their interactions.

The order of component learning is important in this approach and the system architecture should be taken in view for setting the priority of selecting components for learning. Initially, we select the components which have most interactions with the environment to learn them at first place because they are more independent than the components which totally rely on the other components for their execution. Recall that the system architecture is known and we know a priori which components in the system do or do not communicate with the environment.

In the example of system Sys, we select component M first to learn since it receives external inputs. Whereas, component N is learned after the inputs from M to N are discovered.

In the following, we discuss Step 1 for Mealy systems and illustrate it in the example.

Description on the Mealy System

In a Mealy system, we learn each component one by one using the algorithm ${L_M}^+$ (Algorithm 3). The procedure is described as follows.

For the system $\{C_1, \ldots, C_n\}$, let a component $C_i = (Q_i, I_i, O_i, \delta_i, \lambda_i, q_{0i})$ be selected to learn first. Let ${I_i}^{(1)} \subseteq I_i$ be the input set constructed to learn C_i and ${M_i}^{(1)} = ({Q_i}^{(1)}, {I_i}^{(1)}, {O_i}^{(1)}, {\delta_i}^{(1)}, {\lambda_i}^{(1)}, {q_{0i}}^{(1)})$ be its learned model.

Let a component $C_j = (Q_j, I_j, O_j, \delta_j, \lambda_j, q_{0j})$ such that ${O_i}^{(1)} \cap I_j \neq \emptyset$, then the input set of C_j is constructed as ${I_j}^{(1)} = {O_i}^{(1)} \cap I_j$. In fact, ${I_j}^{(1)} \subseteq I_j$, so we restrict the input set for C_j with respect to the composition of C_i and C_j. Similarly, the model ${M_j}^{(1)} = ({Q_j}^{(1)}, {I_j}^{(1)}, {O_j}^{(1)}, {\delta_j}^{(1)}, {\lambda_j}^{(1)}, {q_{0j}}^{(1)})$ will be learned for C_j and the set ${O_j}^{(1)}$ will restrict the input set of the other components in the system.

Assume there exists a component C_k whose learned model is given as ${M_k}^{(1)} = ({Q_k}^{(1)}, {I_k}^{(1)}, {O_k}^{(1)}, {\delta_k}^{(1)}, {\lambda_k}^{(1)}, {q_{0k}}^{(1)})$. Then, ${O_k}^{(1)}$ discovers new inputs for a previously learned component, say C_j, if ${O_k}^{(1)} \cap I_j \neq \emptyset$ and ${O_k}^{(1)} \cap {I_j}^{(1)} = \emptyset$. Let ${I_j}^{(2)} = {I_j}^{(1)} \cup ({O_k}^{(1)} \cap I_j)$, i.e, the set of the previous and the newly discovered inputs for C_j, then C_j will be relearned as ${M_j}^{(2)} = ({Q_j}^{(2)}, {I_j}^{(2)}, {O_j}^{(2)}, {\delta_j}^{(2)}, {\lambda_j}^{(2)}, {q_{0j}}^{(2)})$.

Example

In the example Sys, we select component M to learn first and construct its input set ${I_M}^{(1)} = I_M = \{A, b, E\}$. The algorithm ${L_M}^+$ is applied and the conjecture $M^{(1)} = ({Q_M}^{(1)}, {I_M}^{(1)}, {O_M}^{(1)}, {\delta_M}^{(1)}, {\lambda_M}^{(1)}, {q_{0M}}^{(1)})$, shown in Figure 6.6, is obtained from the closed observation table, shown in Table 6.1.

Next, we learn component N and construct its input set. We have obtained the output set of $M^{(1)}$ as ${O_M}^{(1)} = \{x, y, Z\}$. So, the input set for N is constructed as ${I_N}^{(1)} = I_N \cap {O_M}^{(1)} = \{x, y\}$. The learning algorithm obtains the conjecture $N^{(1)} = ({Q_N}^{(1)}, {I_N}^{(1)}, {O_N}^{(1)}, {\delta_N}^{(1)}, {\lambda_N}^{(1)}, {q_{0N}}^{(1)})$, shown in Figure 6.7, from the closed observation table, shown in Table 6.2.

6.3.2 Step 2: Computing and Analyzing the Product

This step is explained in three parts.

	A	b	E
ϵ	x	x	Z
A	y	x	Z
b	x	x	Z
E	x	x	Z
$A \cdot A$	y	x	Z
$A \cdot b$	x	x	Z
$A \cdot E$	y	x	Z

Table 6.1: Closed Observation Table for $M^{(1)}$.

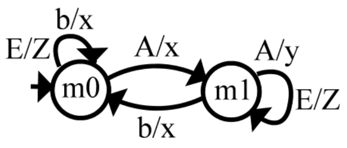


Figure 6.6: The model $M^{(1)}$ from Table 6.1.

	x	y
ϵ	b	b
x	b	b
y	b	b

Table 6.2: Closed Observation Table for $N^{(1)}$.

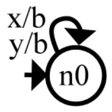


Figure 6.7: The model $N^{(1)}$ from Table 6.2.

Step 2(a): We compute the product of the models that have been obtained after learning in Step 1. Now, we analyze the product in Step 2(b).

Step 2(b): We analyze the product of the learned models for any compositional problems such as deadlocks and livelocks. Such problems can be detected via reachability analysis of the state space of the product. Moreover, user-defined properties on the system can be also be checked, for instance, by using a model checker [PVY99]. The detection of a problem in the product would produce an i/o trace as a *witness* to the problem in the product. The witness will be confirmed on the real system in Step 2(c). In case no problem is found, we proceed for Step 4.

Step 2(c): The composition problems of the product detected in Step 2(b) may also exist in the real system. But the witness to a problem in the product does not confirm the existence of the problem in the system because it may be just an artifact due to the partial learning of the individual components. Therefore, we have to confirm the problem in the product on the actual system. If the problem is confirmed then we terminate the procedure and report the problem. Otherwise, the problem in the model is proved to be an artifact and the corresponding behavior of the system upon the confirmation of the problem is considered to be a *counterexample* for the product. In this case, we proceed to the refinement of the product, which is Step 3 of the procedure.[1]

In the following, we discuss Step 2 for Mealy systems and illustrate it on the example.

Description on the Mealy System

- According to Step 2(a), we compute the product of the learned components in the Mealy system. For all learned models, ${M_1}^{(k)}, {M_2}^{(k)}, \cdots, {M_n}^{(k)}$ obtained after Step 1, we compute the product $\prod^{(k)}$ of the models as described in Section 6.2.3. We denote by $\prod$, the product of the actual components, or in other words, the real system.

- According to Step 2(b), we analyze the product $\prod^{(k)}$ for compositional problems. This is explained as follows.

  In the definition of the Mealy system, each component in our assumption is modeled as an input-enabled Mealy machine. This means that for every input, the component must produce an observable output. We also assume that the internal interfaces of the components in the system are observable and the communication between the components cannot be blocked. Thus, the system in this case cannot contain deadlocks. However, the system can contain compositional livelocks such that by inputing a specific sequence of external inputs, the system exhibits a cyclic behavior of the composed components and does not terminate by producing an external output. In our approach, we find a livelock in the system by analyzing the product of the learned components. If the product contains a cyclic path of internal inputs and outputs, i.e., there is a loop on a set of unstable states with no stable state in between, then it contains a livelock. The formal definition of the livelock in Mealy system is given below.

  Let $\prod^{(k)} = (Q^{(k)}, I^{(k)}, O^{(k)}, \delta^{(k)}, \lambda^{(k)}, {q_0}^{(k)})$ be the product of the learned models obtained after Step 1, $q^{(k)} \in Q^{(k)}$ be an unstable state of $\prod^{(k)}$. Then, $\prod^{(k)}$ contain a livelock if there exists a sequence of internal inputs, i.e., $u \in I^{(k)} \backslash I_{ext}$, such that $\delta^{(k)}(q^{(k)}, u) = q^{(k)}$. Let $u' \in I^{(k)}$ such that $\delta^{(k)}({q_0}^{(k)}, u') = q^{(k)}$, then $u' \cdot u$ is a witness to a livelock in $\prod^{(k)}$.

- According to Step 2(c), we confirm the livelock found in $\prod^{(k)}$ on the real system $\prod$. We confirm the livelock by applying the witness $u' \cdot u$ to $\prod$ and observe its corresponding behavior. In fact, we apply the external input projection of the witness, since only the external inputs can be applied to the integrated system. If $\prod$ contains a livelock, then it

[1] Note that a "witness" is an i/o trace that witnesses a problem in the product, and a "counterexample" is an i/o trace that proves the difference between the product and the model.

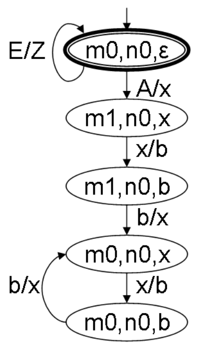


Figure 6.8: The product $\prod^{(1)}$ of the learned models $M^{(1)}$ and $N^{(1)}$.

will enter into a never ending loop exhibiting the cycles of internal inputs and outputs. What number of times a cycle should be observed to declare a livelock is an open question in black box testing [GLPS08]. Here, we assume that if the cycle is observed twice, then we stop[1] the system execution and declare a livelock. Under this assumption, let w be an i/o trace of $\prod^{(k)}$ when applying $u' \cdot u \cdot u$ on $q_0{}^{(k)}$. Let v be the i/o trace of $\prod$ when applying the external input projection of w, i.e., $w \downarrow_{I_{ext}}$. If $v = w$, then $\prod$ contains a livelock. We declare the livelock in $\prod$ and terminate the procedure. Otherwise, $\prod$ produces an external output and v is a counterexample for $\prod^{(k)}$.

Example

In the example Sys, we first compute the product $\prod^{(1)} = (Q^{(1)}, I^{(1)}, O^{(1)}, \delta^{(1)}, \lambda^{(1)}, q_0{}^{(1)})$ of the learned models $M^{(1)}$ and $N^{(1)}$ shown in Figure 6.8. Then, we analyze $\prod^{(1)}$ for compositional problems. The product $\prod^{(1)}$ contains a livelock, since from the unstable state $(m0, n0,\ x)$, there exists a sequence of internal inputs $x \cdot b$ such that $\delta^{(1)}((m0, n0,\ x), x \cdot b) = (m0, n0,\ x)$. Then, we obtain the i/o trace $w = A/x \cdot x/b \cdot b/x \cdot x/b \cdot b/x \cdot \mathbf{x/b} \cdot b/x$ that contains a witness to a livelock in $\prod^{(1)}$. Now, we confirm the livelock on the system. For that, we give the external input projection $w \downarrow_{I_{ext}} = A$ to the real system, i.e., $\prod^{Sys}$ in Figure 6.4. The corresponding behavior of $\prod^{Sys}$ is obtained as $v = A/x \cdot x/b \cdot b/x \cdot x/b \cdot b/x \cdot \mathbf{x/D}$. The behavior v ends by producing the external output D, which means $v \neq w$ and v is a counterexample for $\prod^{(1)}$.

6.3.3 Step 3: Refining the Product

If we find a counterexample v for the product of learned models, then the product is refined in this step. The counterexample is found due to the partiality of the models of one or more

[1]This logic can be implemented in the test driver that is actually sending the inputs to the real system and observing the corresponding outputs

components in the system. Therefore, we can identify those components whose models can be refined using v as a counterexample. In fact, v is a complete i/o trace of the system, so we need to project v on the input and output sets of each component to obtain its respective i/o trace. Then, each projected trace will be run on its component's model to identify the differences in the behaviors. When a component is identified, it is relearned using its projected trace as a counterexample. The refined model obtained from relearning the component will contain at least one new state (see Theorem 4 in Section 5.3.4 of Chapter 5). Therefore, the product of the refined models will also be refined. Moreover, the refined model of the component may discover new inputs for other components. This follows the procedure of learning components with restrictive input sets, described in Step 1, followed by the iterative process (Steps 2-5).

In the following, we discuss Step 3 for Mealy systems and illustrate it on the example.

Description on the Mealy System

Let v be a counterexample for the product of the learned models $\prod^{(k)}$ and $M_i{}^{(1)} = (Q_i{}^{(1)}, I_i{}^{(1)}, O_i{}^{(1)}, \delta_i{}^{(1)}, \lambda_i{}^{(1)}, q_{0i}{}^{(1)})$ be the learned model of a component C_i. To check whether v contains a counterexample for $M_i{}^{(1)}$, we obtain $v\downarrow_{I_i{}^{(1)}}$ and $v\downarrow_{O_i{}^{(1)}}$, i.e., the input and output projections of v on $I_i{}^{(1)}$ and $O_i{}^{(1)}$, respectively. If $\lambda_i{}^{(1)}(q_{0i}{}^{(1)}, v\downarrow_{I_i{}^{(1)}}) \neq v\downarrow_{O_i{}^{(1)}}$, then $v\downarrow_{I_i{}^{(1)}}$ is a counterexample for $M_i{}^{(1)}$. We provide the counterexample to $L_M{}^{+}$ to relearn $C_i{}^{(1)}$ and obtain $M_i{}^{(2)}$.

Note that if v is a counterexample for a product then it contains a counterexample for at least one component in the system. This is trivial because every transition in the product represents a transition of some component in the system. This means if a transition in the product shows a different behavior in the system then the corresponding transition in the component also behaves differently. Therefore, the projection of v on the input and output sets of each components will correctly find a counterexample for at least one component.

Example

In the example Sys, $v = A/x \cdot x/b \cdot b/x \cdot x/b \cdot b/x \cdot x/D$ is a counterexample obtained after Step 2 for the product $\prod^{(1)}$. We project v on the input and output sets of each component and check the behavior of the component against its model. we find that $\lambda_M{}^{(1)}(q_{0M}{}^{(1)}, v\downarrow_{I_M{}^{(1)}}) = v\downarrow_{O_M{}^{(1)}}$. However, $\lambda_N{}^{(1)}(q_{0N}{}^{(1)}, v\downarrow_{I_N{}^{(1)}}) \neq v\downarrow_{O_N{}^{(1)}}$. In fact, $v\downarrow_{I_N{}^{(1)}} = x \cdot x \cdot x$ and $v\downarrow_{O_N{}^{(1)}} = b \cdot b \cdot D$, but $\lambda_N{}^{(1)}(q_{0N}{}^{(1)}, v\downarrow_{I_N{}^{(1)}}) = b \cdot b \cdot b$. Thus, $x \cdot x \cdot x$ is a counterexample for $N^{(1)}$ and the refined model $N^{(2)}$, shown in Figure 6.9 along with its closed observation table in Table 6.3, is obtained.

This follows Step 2 in which we make the new product $\prod^{(2)}$ of $M^{(1)}$ and $N^{(2)}$, shown in Figure 6.10. The product $\prod^{(2)}$ does not contain any compositional problems and thus we proceed for Step 4.

	x	y	$x \cdot x$
ϵ	b	b	$b \cdot b$
x	b	b	$b \cdot D$
$x \cdot x$	D	D	$D \cdot D$
y	b	b	$b \cdot b$
$x \cdot y$	b	b	$b \cdot D$
$x \cdot x \cdot x$	D	D	$D \cdot D$
$x \cdot x \cdot y$	b	b	$b \cdot b$

Table 6.3: Closed Observation Table for $N^{(2)}$.

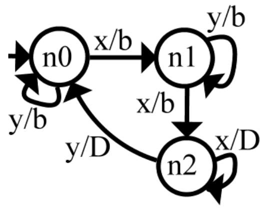


Figure 6.9: The model $N^{(2)}$ from Table 6.3.

6.3.4 Step 4: Finding Discrepancy between the Product and the System

At this stage, we obtain the product of the learned models that contains no compositional problems. But the product may still partially represent the system and there can be found discrepancies between the behavior of the system and the behavior depicted by the product. In previous works [PVY99][SL07][SHL08] of the similar framework, VC algorithm [Vas73] [Cho78] has been used to find the counterexamples as discrepancies between the partial model and the black box system. However, we must have an upper bound on the number of states in the system in order to apply the algorithm. In spite of this, we can use model based testing methods based on coverage criteria for test generation. There can be many different coverage criteria [PvBY96], in which case excessive number of tests of arbitrary lengths can be generated. In our approach, we do not fix a particular method for generating tests for finding discrepancies; rather we leave it as an open choice to the designer of the system.

For the sake of simplicity, we describe the integration testing approach using transition based coverage criteria to generate tests for the system. In fact, we generate tests by covering each transition in the stable states of the product at least once. We cover only transitions of the stable states because only the external inputs can be given to the system in the integration testing. Then, the behavior of the system in the result of testing can be compared with the behavior depicted by the product. If the behaviors are different, then a discrepancy is found. In the following, we describe the test generation method for Mealy systems.

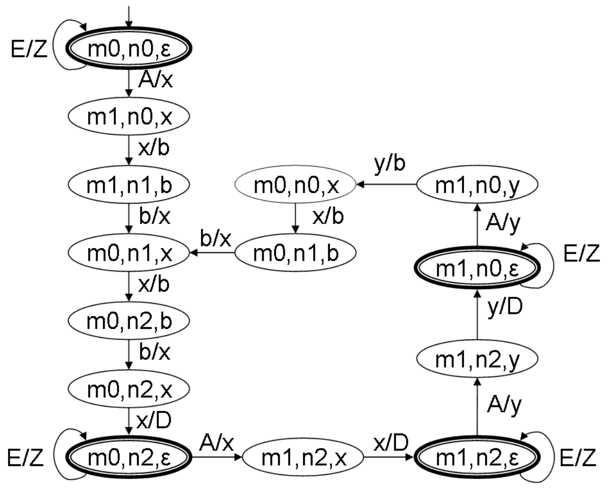


Figure 6.10: The product $\prod^{(2)}$ of the learned models $M^{(1)}$ and $N^{(2)}$.

Description on the Mealy System

We calculate the spanning tree of the product of the learned models $\prod^{(k)}$ and traverse the tree from the root to each leaf node in the breadth-first-search manner to obtain the i/o traces in the tree. The set of i/o traces are the tests for the integrated system. Let w be a test from the set of traces, then the external input projection of w, i.e., $w\downarrow_{I_{ext}}$, is the input sequence given to the system. For each test w, the behavior of the real system $\prod$ on $w\downarrow_{I_{ext}}$ can be compared with the corresponding inputs and outputs in w. Let the i/o trace obtained from $\prod$ be v. If $w \neq v$, then a discrepancy between $\prod^{(k)}$ and $\prod$ is detected.

Example

In the example Sys, we generate tests from the product $\prod^{(2)}$ using the method described above. Here, we show a test $w = A/x{\cdot}x/b{\cdot}b/x{\cdot}x/b{\cdot}b/x{\cdot}x/D{\cdot}A/x{\cdot}x/D{\cdot}\mathbf{A/y}{\cdot}y/D$ such that the behavior of $\prod^{Sys}$ on $w\downarrow_{I_{ext}} = A{\cdot}A{\cdot}A$ is observed as $v = A/x{\cdot}x/b{\cdot}b/x{\cdot}x/b{\cdot}b/x{\cdot}x/D{\cdot}A/x{\cdot}x/D{\cdot}\mathbf{A/x}{\cdot}x/D$. Thus, $w \neq v$ and v is a witness to a discrepancy between $\prod^{(2)}$ and $\prod^{Sys}$.

6.3.5 Step 5: Resolving Discrepancy

In the previous step, the actual behavior of the system v instead of the behavior w of the product reveals a discrepancy between the system and the product. Surely, v is a counterexample for the product which does not represent the system accurately. Then, we can refine the product with the help of this counterexample.

Anomalies	**Symbols**
System Crash	`SC`
Uncaught Exception	`Exp`
Out of Memory problem	`OutMem`

Table 6.4: List of anomalies for automatic checking during the integration testing procedure

At the same time, since the system has been exercised on the tests generated in Step 4, the tests might have revealed real errors in the system. In this case, although we know that the learned product is different from the system, we have also found an error in the system.

The functional errors in the system can be checked automatically if a partial specification or expected behaviors in the form of scenarios are provided [RG99] [RF07]. Otherwise, the designer can intervene to classify them manually. In our approach, we are checking the kind of anomalies in the system for which we do not need the provision of partial specifications to detect them automatically. Actually, we add this logic in the test driver that if the system exhibits any such anomalies during testing, then it highlights a special symbol against the anomaly in v. The list of such anomalies and their symbols according to our current implementation of the approach is given in table 6.4. The list is not exhaustive and can be extended to incorporate more general programming errors.

In this step, we resolve whether v exhibits an anomaly or we should go for the refinement of the product considering v as a counterexample. If v exhibits an anomaly, then v is reported as a witness to a real error in the system. In this case, we terminate the procedure. Otherwise, v is a counterexample for the product that shows that the product does not represent the system accurately. Here, an eyeball inspection would be useful to decide either to continue for the refinement step or to stop the iterative learning procedure. We choose to refine the product in quest of learning a more complete representation of the system. Moreover, this choice makes the approach more automatic.

Therefore, when a counterexample is found, we go for the refinement of the product (Step 3) and start again the procedure on the refined product. The description of the step is given on Mealy systems as follows.

Description on the Mealy System

Let v be the behavior of the real system $\prod$ obtained in Step 4, such that v is a witness to a discrepancy between $\prod$ and the product of the learned models $\prod^{(k)}$. Then, v is checked if it exhibits an anomaly. If an anomaly is found, v is reported as a witness to a real error in $\prod$. Otherwise, v is a counterexample for $\prod^{(k)}$, which follows Step 3 for the refinement of $\prod^{(k)}$.

Example

In the example Sys, we found a witness $v = A/x \cdot x/b \cdot b/x \cdot x/b \cdot b/x \cdot x/D \cdot A/x \cdot x/D \cdot A/x \cdot x/D$ after Step 4, that is a discrepancy between $\prod^{(2)}$ and $\prod^{Sys}$. In this step, we classify v as a counterexample for $\prod^{(2)}$ instead of an error in Sys. Therefore, we proceed for Step 3 to refine $\prod^{(2)}$.

In Step 3, we identify the exact component for refinement by making the projection of v on the input and output sets of $M^{(1)}$ and $N^{(2)}$. Then, we find that $v\downarrow_{I_{M^{(1)}}} = A \cdot b \cdot b \cdot A \cdot A$ and $v\downarrow_{O_{M^{(1)}}} = x \cdot x \cdot x \cdot x \cdot x$, but $\lambda_{M^{(1)}}({q_0}_M{}^{(1)}, v\downarrow_{I_{M^{(1)}}}) = x \cdot x \cdot x \cdot x \cdot y$. Hence, $A \cdot b \cdot b \cdot A \cdot A$ is a counterexample for $M^{(1)}$. We refine the model as $M^{(2)}$, shown in Figure 6.11 along with its closed observation table, shown in Table 6.5. For the model $N^{(2)}$, there is no counterexample found. Hence, we proceed for Step 2.

In Step 2, we make the new product $\prod^{(3)}$ of $M^{(2)}$ and $N^{(2)}$, shown in Figure 6.12. The new product does not contain compositional problems and therefore, we proceed for Step 4.

In Step 4, we do not find discrepancies between $\prod^{(3)}$ and $\prod^{Sys}$. Hence, we terminate the procedure.

	A	b	E	$A \cdot A$	$b \cdot A \cdot A$
ϵ	x	x	Z	$x \cdot y$	$x \cdot x \cdot x$
A	y	x	Z	$y \cdot y$	$x \cdot x \cdot x$
b	x	x	Z	$x \cdot x$	$x \cdot x \cdot x$
E	x	x	Z	$x \cdot y$	$x \cdot x \cdot x$
$A \cdot A$	y	x	Z	$y \cdot y$	$x \cdot x \cdot x$
$A \cdot b$	x	x	Z	$x \cdot x$	$x \cdot x \cdot x$
$A \cdot E$	y	x	Z	$y \cdot y$	$x \cdot x \cdot x$
$b \cdot A$	y	x	Z	$x \cdot y$	$x \cdot x \cdot x$
$b \cdot b$	y	x	Z	$x \cdot x$	$x \cdot x \cdot x$
$b \cdot E$	x	x	Z	$x \cdot x$	$x \cdot x \cdot x$

Table 6.5: Closed Observation Table for $M^{(2)}$.

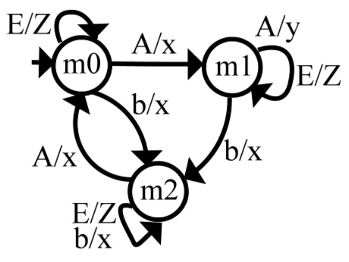


Figure 6.11: The model $M^{(2)}$ from Table 6.5.

6.3.6 Termination Criteria

We terminate the procedure of integration testing at three steps, 1) At Step 2, when a compositional problem is confirmed in the system, 2) At Step 4, when no discrepancies between the product and the system are found, 3) At Step 5, when we classify a discrepancy as a real error in the system.

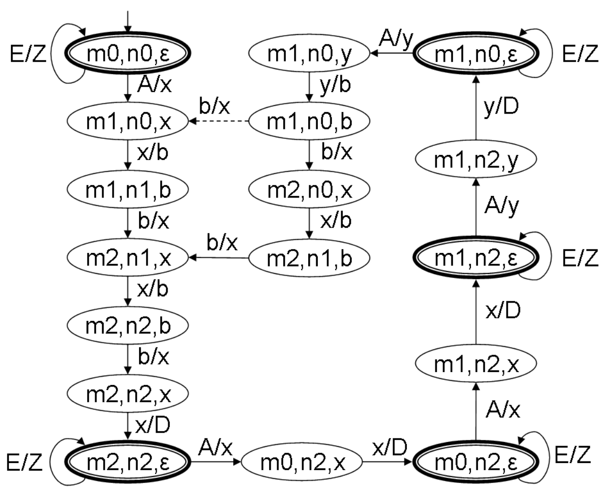


Figure 6.12: The product $\prod^{(3)}$ of learned models $M^{(2)}$ and $N^{(2)}$. The dotted arrow shows how the product can be minimized.

Note that the absence of discrepancies in Step 4 does not necessarily mean that the product has been learned completely. Instead, it means that no discrepancies can be found by the generated tests. Here, we can opt for a different testing strategy to generate more tests. Moreover, we can also count the number of iterations in the procedure and terminate if the number exceeds certain threshold. Such criteria can be used for customization and for making the approach more flexible.

6.4 Conclusion

We have given an approach for learning and testing integrated systems of black box components. The procedure consists of five steps. In the first step, we learn the components in isolation with their restrictive input sets. In Step 2, we compute the product of the learned models and analyze the product to find composition problems. For every problem found in the product, we confirm it on the system and report on confirmation and terminate the procedure. Otherwise, we consider the problem as an artifact and proceed for Step 3. In Step 3, we refine the product by relearning each component. In Step 4, we generate the tests from the product model with no compositional problems and find discrepancies between the product and the system. If no discrepancies are found, then we terminate the procedure. In Step 5, the discrepancies found in the previous step are resolved. If the discrepancies are classified as real errors then we terminate the procedure by reporting the errors. Otherwise, we proceed for refining the product and the iterative procedure starts again.

We have explained each step with the help of an example (Figure 6.2). It is worth noting that the final product $\prod^{(3)}$, shown in Figure 6.12, is actually same as the real system $\prod^{Sys}$, shown in Figure 6.4, in the sense that they are isomorphic if they are minimized[1]. Whereas, the final learned model of component M, i.e., $M^{(2)}$, is still not learned completely. This example justifies that we do not always need to learn complete models of the components in the integrated system. Moreover, the learned models serve as oracles in the integration testing procedure to find potential errors such as compositional problems and general kind of errors in the system. Also, the tests generated from the product find counterexamples for the partial models which are then refined in the iterative procedure.

As discussed in Chapter 5, the size of counterexamples is a key factor for analyzing the complexity of learning a model. In our procedure of integration testing, the counterexamples of the shortest length are not always produced, which is not the ideal case for learning the model in minimum number of queries. However, the worst case complexity of learning in our procedure remains polynomial as we have computed (cf. Chapter 5, Section 5.3.5).

The finding of counterexamples for the product depends upon the quality of tests that are generated through the testing strategy in our procedure. The comparison of different strategies has not been addressed in the scope of this work. We use the existing works of test generation from models in our approach, which are mentioned and discussed in Chapters 1 and 3.

[1]The minimization is shown with dotted arrows in both products

Chapter 7

Parameterized Machine Inference

This chapter covers the details of inferring a Parameterized Finite State Machine through the active learning approach. It describes the inferring algorithm and its complexity. The proofs of the related theorems and an application of the model are also given.

7.1 Motivation

It is argued that learning Mealy machines is still not sufficient because they do not model adequately the systems that exchange lots of parameterized data values from arbitrary complex domains. For example, the controller of HVAC system works on many different temperature values, user modes and status of external devices. The Mealy machine modeling of such system would create a formidable size of input set, and thus, would result in a combinatorial blow up on the transition level. Moreover, the complexity of the learning algorithm is greatly enhanced by taking all data values (as inputs) in the learning procedure. As a matter of fact, it is not obvious to test all values to identify the complete behavioral spectrum of the system. First, it is not possible to exhaustively test all possibilities. Second, the behavior of such systems is confined on a few subsets of values and differs only on minor details. For example, the behavior of the HVAC controller on all the temperature values less than 12℃ is the same, that is turning on the heater, but only the speed of the device may differ from one value to another. Thus, it will be more adequate if such a system could be modeled in a compact (finite state machine) representation that shows the global behaviors (as state transitions) and the minor details could be treated as parameter values (associated with inputs and outputs on transitions).

Due to these reasons, there is a good argument to model a system into an expressive form, that can detail the intended behaviors of the system in a compact representation and can encapsulate a large (or infinite) input set into few key inputs; such that the complexity of the learning algorithm can be catered. We have presented a parameterized model *PFSM* in

Definition 4 (Chapter 2). In this chapter, we discuss the learning of PFSM models using the original settings of Angluin's algorithm. Section 7.2 explains the learning algorithm for PFSM. Section 7.4 provides an illustration of learning HVAC controller using PFSM learning algorithm. Section 7.5 concludes the chapter.

We shall be using HVAC Controller example in the explanation of the PFSM learning algorithm. The complete description of the example can be seen in Chapter 5 (Section 5.6). Here, we recall the terms that we use in the example: ON to turn on the system, OFF to turn off the system, T for temperature, H for heater, F for fan, l for low speed, h for high speed and S to stop the fan/heater.

7.2 Learning Algorithm for PFSM models

Keeping in view the fundamentals of DFA and Mealy machine inference algorithms, we move towards the inference of PFSM models. We illustrate the motivation of enriching the structure of the observation table and adapting the previous definitions to accommodate input/output parameters. Consider a Mealy machine model of the HVAC controller when all 71 temperature values, i.e., $[-20, +50]$, are taken as different input symbols. The structure of the table will expand as the number of rows and columns augment with the size of the input set. If the observation table is adapted for parameterized inputs then the whole range of temperature values can be encapsulated into a single input symbol as T and $[-20, +50]$ can be considered as the domain of parameter values for T. Thus, the behaviors of different temperature values recorded in multiple rows and columns of the observation table (in case of Mealy machine inference) can be collapsed into a single cell as shown in Figure 7.1. Moreover, if the behaviors recorded in the multiple cells are the same, then the observation table for PFSM inference is extended only with one row contrary to multiple rows in the case of the observation table for Mealy machine inference.

The problem with learning PFSM models is that we can only test a finite number of input parameter values. That means, we may not be able to learn completely the i/o parameter domains, or precisely, the predicates on the input parameter values and the output parameter function on a transition. However, the number of transitions labeling different outputs from a given state can be learned if a parameter value from each of the predicates on these transitions is tested. The issue can be resolved under uniformity hypothesis [BGM91] [Pha94], i.e., the domain of parameter values is partitioned into equivalence classes such that the component behaves indifferently for all parameter values in a class of the partition. Then, it is sufficient to

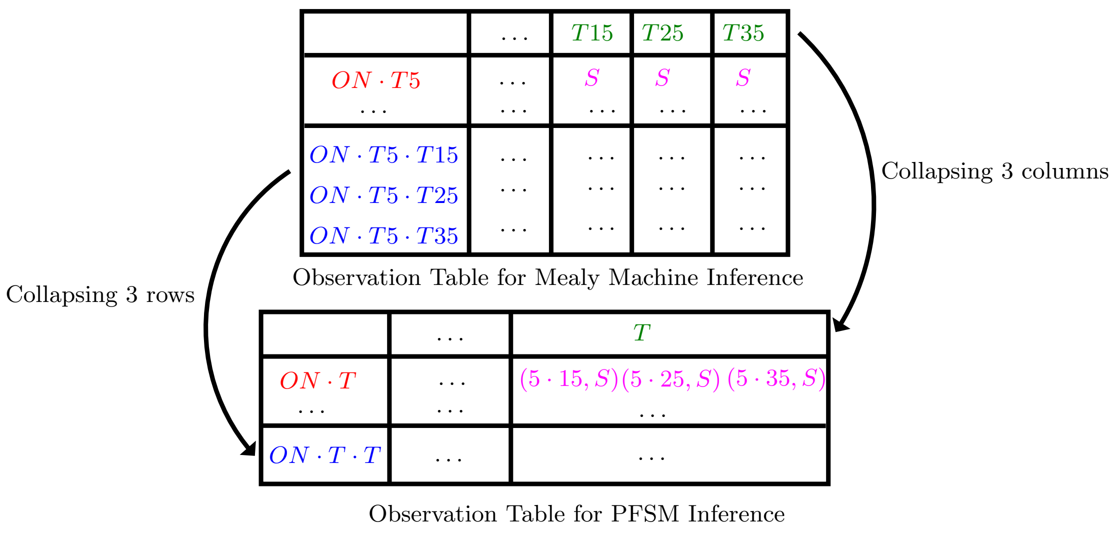


Figure 7.1: Idea of collapsing rows and columns in the Observation Table for PFSM inference

test only one value from each class. But such a hypothesis would take us back to the learning of Mealy machines, in which, we can create one input for all the elements of a class.

It turns out that the issue of selecting the parameter values in the learning procedure points to the classical problem of selecting test data in black box testing [Kor99]. In our case, we opt for a random selection of parameter values from the domain. For example, if the domain is $[-20, +50]$, then the learning algorithm selects randomly the values between the range and test the black box machine on those values. Other techniques, e.g., equivalence partitioning, boundary value analysis etc, and heuristics could also be applied based upon the available knowledge on the parameter domain.

For the description of the PFSM learning algorithm, let $\mathcal{P} = \{Q_{\mathcal{P}}, I, O, D_I, D_O, \Gamma_{\mathcal{P}}, q_{0\mathcal{P}}\}$ be the unknown PFSM model that has a minimum number of states. The transition function, the output function and the output parameter function for $\mathcal{P}$ are denoted by $\delta_{\mathcal{P}}, \lambda_{\mathcal{P}}$ and $\sigma_{\mathcal{P}}$, respectively. Following the active learning theory, the main assumptions for PFSM learning are

- The basic set of the input symbols I and the domain of the input parameter values D_I are known. In the example of the HVAC controller, we have $I = \{ON, OFF, T\}$ and $D_I = [-20, 50] \cup \{\perp\}$. The inputs ON and OFF do not take input parameters, so they will be associated with $\perp$ [1]. The input T will be associated with $[-20, 50]$.

- The machine can be reset before each query.

[1]See Definition 4 for the usage of $\perp$

The algorithm asks *parameterized queries* that are strings from I^* and ${D_I}^*$. Therefore, it requires an access to its interfaces, i.e., the input interface from where a parameterized input can be sent and the output interface from where a parameterized output can be observed.

In the following sections, we describe the structure and properties of the observation table and then the algorithm for PFSM inference. We use ω to denote an *input symbol string* from I^*, α to denote an *input parameter value string* from ${D_I}^*$, ϖ to denote an *output symbol string* from O^*, β to denote an *output parameter value string* from ${D_O}^*$. A string of the form $\omega \otimes \alpha$ is called an *input parameterized string.* A string of the form $\varpi \otimes \beta$ is called an *output parameterized string.* Moreover, we assume that $\perp \in D_I$ and $\Omega \in O$ always hold.

7.2.1 Observation Table

We denote by ${L_P}^*$ the learning algorithm for PFSM models. At any given time, ${L_P}^*$ maintains an observation table, denoted by (S_P, R, E_P, T_P), for constructing the parameterized queries, recording the corresponding observations in the result of queries and making the PFSM conjecture consistent with the observations. The examples of such a table are given in Table 7.1 and Table 7.2. In order to deal with the specific nature of PFSM models, we have to adapt the structure of the table accordingly. Lets have an overview what adaptations we require for inferring PFSM models before digging up into the formal definitions.

The basic structure of the table is similar to the structure of the observation table (S_M, E_M, T_M) in Mealy machine inference. That is, the rows given by $S_P \cup R$ and the columns given by E_P consist of input symbol strings. Table 7.1 is an example in which rows and columns are labeled with input symbol strings. In addition, (S_P, R, E_P, T_P) deals with a parameterized structure of the PFSM model. Therefore, the rows of the table may also consist of input parameterized strings in addition to the input symbol strings. Table 7.2 is an example in which some rows are labeled with input symbol strings and the others with input parameterized strings. Similarly, the table records the output parameterized strings in the cells in result of the parameterized queries. Table 7.1 and 7.2 both show the cells containing the output parameterized strings.

The other particularity of the PFSM model is that it produces different parameter output strings on different input parameter value strings. In the example of the HVAC controller, the parameter value 5 for the input T turns on the heater, but the parameter value 25 turns on the fan. The two different behaviors are recorded in the table in the same cell. This can be seen in Table 7.2, where the cell of the row labeled by ON and the column labeled by T contains two parameterized output strings $H \otimes l$ and $F \otimes l$ for the values 5 and 25 respectively. We call the rows that record multiple behaviors in the cells as *disputed rows*, since they yield multiple transitions in the conjecture, which are labeled with different output strings for the same input symbol strings but for the different input parameter strings. This requires further testing to determine the target states of the transitions. We refer to this special treatment as *treating the disputed rows.*

Finally, we need new definitions for the comparison of rows. The rows must contain the common input parameter values so that their corresponding output strings could be compared. In Table 7.1, the rows labeled by ϵ and ON contain different parameter values, namely 5 and 25, for the column labeled by T. The two rows are not directly comparable. In Table 7.2, the two rows contain both values, and thus are comparable. We call the operation of making the rows containing common input parameter values as *balancing* the rows. This follows the definition of row *equivalence*, that the rows are not only equivalent by the output symbol strings but also by the output parameter value strings. With the new definition of row equivalence, the definition of *closed* is also modified accordingly. We take advantage of our improvement in the Mealy machine inference, and keep the size of S_P in (S_P, R, E_P, T_P) reduced, i.e., all rows in S_P are inequivalent. Therefore, we do not need to check *consistency* in the table.

In the following, we describe the structure of the observation table and provide the related definitions with examples.

Basic Structure of the Table

Let $\mathcal{U}$ be a set of parameterized input strings, i.e., $\mathcal{U} = \{\omega \otimes \alpha | \omega \in I^+, \alpha \in {D_I}^+, |\omega| = |\alpha|\}$. Then, the structure of the observation table (S_P, R, E_P, T_P) is defined as follows.

$S_P \subseteq \mathcal{U} \cup I^*$ and $R \subseteq \mathcal{U} \cup I^*$ are the nonempty finite sets of parameterized input strings that make the rows of the table. S_P is a prefix-closed set that is used to identify potential states in the conjecture and R is used to satisfy properties of the table.

$E_P \subseteq I^+$ is the suffix-closed nonempty finite set of input symbol strings that makes the columns of the table and distinguishes the states of the conjecture from each other.

The elements of $(S_P \cup R) \times E_P$ are used to construct *parameterized queries.* The result of the query is recorded in the table with the help of the function T_P that records the pairs of input parameter value strings and parameterized output strings. Formally, T_P maps $(S_P \cup R) \times E_P$ to the power set of $\{(\alpha, \varpi \otimes \beta) | \alpha \in {D_I}^+, \varpi \in O^+, \beta \in {D_O}^+\}$.

Section 7.2.2 describes how to construct parameterized queries and record the corresponding observations in the table. Here an example is given. Let $s \in S_P \cup R, e \in E_P, \alpha \in {D_I}^+$, where $|\alpha| = |IS(s)| + |e|$, then a query is constructed as $IS(s) \cdot e \otimes \alpha$.[1] Let the corresponding parameterized output string be $\varpi \otimes \beta$, where $\varpi = \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, \omega, \alpha)$ and $\beta = \sigma_{\mathcal{P}}(q_{0\mathcal{P}}, \omega)(\alpha)$ (and $|\varpi| = |\beta|$). Then, the observation is recorded by adding the pair $(\alpha, \mathit{suff}^{|e|}(\varpi) \otimes \mathit{suff}^{|e|}(\beta))$ to

[1] $IS(s)$ is the input symbol string from s. See Chapter 2 for the definition.

$T_P(s, e)$. Note that we record only the suffixes of ϖ and β of length $|e|$, likewise the algorithm of Mealy machine inference. In Table 7.1, a query is constructed as $ON \cdot T \otimes \perp \cdot 25$, in which $s = ON \in S_P \cup R, e = T \in E_P$ and $\alpha = \perp \cdot 25$. The corresponding parameterized output string of the query is observed as $OK \cdot F \otimes \perp \cdot l$, in which $\varpi = OK \cdot F$ and $\beta = \perp \cdot l$. Then, the pair $(\perp \cdot 25, F \otimes l)$ is added to $T_P(ON, T)$.

		E_P		
		ON	OFF	T
S_P	ϵ	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(5, \Omega \otimes \perp)$
	ON	$(\perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp, S \otimes \perp)$	$(\perp \cdot 25, F \otimes l)$
R	OFF	$(\perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot 5, \Omega \otimes \perp)$
	T	$(25 \cdot \perp, \Omega \otimes \perp)$	$(25 \cdot \perp, \Omega \otimes \perp)$	$(5 \cdot 25, \Omega \otimes \perp)$

Table 7.1: Example of the Observation Table for learning a PFSM model of the HVAC controller

		E_P		
		ON	OFF	T
S_P	ϵ	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(5, \Omega \otimes \perp)$ $(25, \Omega \otimes \perp)$
	ON	$(\perp \cdot \perp, OK \otimes \perp)$	$(\perp \cdot \perp, S \otimes \perp)$	$(\perp \cdot 5, H \otimes l)$ $(\perp \cdot 25, F \otimes l)$
	OFF	$(\perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot 5, \Omega \otimes \perp)(\perp \cdot 25, \Omega \otimes \perp)$
R	$ON \cdot ON$	$(\perp \cdot \perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp \cdot 5, \Omega \otimes \perp)$ $(\perp \cdot \perp \cdot 25, \Omega \otimes \perp)$
	$ON \cdot OFF$	$(\perp \cdot \perp \cdot \perp, OK \otimes \perp)$	$(\perp \cdot \perp \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot \perp \cdot 5, \Omega \otimes \perp)$ $(\perp \cdot \perp \cdot 25, \Omega \otimes \perp)$
	$ON \cdot T \otimes \perp \cdot 5$	$(\perp \cdot 5 \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot 5 \cdot \perp, S \otimes \perp)$	$(\perp \cdot 5 \cdot 5, H \otimes l)$ $(\perp \cdot 5 \cdot 25, S \otimes \perp)$
	$ON \cdot T \otimes \perp \cdot 25$	$(\perp \cdot 25 \cdot \perp, \Omega \otimes \perp)$	$(\perp \cdot 25 \cdot \perp, S \otimes \perp)$	$(\perp \cdot 25 \cdot 5, S \otimes \perp)$ $(\perp \cdot 25 \cdot 25, F \otimes l)$

Table 7.2: Balanced and Dispute-free Observation Table for learning a PFSM model of the HVAC controller

Notations in the Table

We define the notations in (S_P, R, E_P, T_P) that are helpful to describe the learning algorithm.

Distinguishing Subsets:

In a PFSM model, different input parameter values can generate different output symbols and output parameter values. For example, the temperature value 5°C turns on the heater while the value 25°C turns on the fan. The two different behaviors will be represented as two different transitions. This can be seen in Table 7.2. For $ON \in S_P \cup R$ and $T \in E_P$, $T_P(ON, T)$ contains the pairs $(\perp \cdot 5, H \otimes l)$ and $(\perp \cdot 25, F \otimes l)$ indicating that the behavior on the query $ON \cdot T \otimes \perp \cdot 5$

is different from the query $ON \cdot T \otimes \perp \cdot 25$. Thus in (S_P, R, E_P, T_P), for any $s \in S_P \cup R$ and $e \in E_P$, there may exist $(\alpha_1, \varpi_1 \otimes \beta_1), (\alpha_2, \varpi_2 \otimes \beta_2) \in T_P(s, e)$ such that $\varpi_1 \neq \varpi_2$, i.e, $T_P(s, e)$ contains pairs in which the output strings are different. Then, we can divide $T_P(s, e)$ into *distinguishing subsets* such that the output strings in all the pairs in one subset are same and the output strings in two pairs from different subsets are different. The formal definition of the distinguishing subsets is given as follows.

Definition 13 Let $\eta(s, e)$ denote the number of different output strings contained by $T_P(s, e)$, then we can divide $T_P(s, e)$ into $\eta(s, e)$ *distinguishing subsets*, i.e., $T_P(s, e) = \bigcup_{k=1}^{\eta(s,e)} d_k(s, e)$, such that for all $(\alpha_1, \varpi_1 \otimes \beta_1), (\alpha_2, \varpi_2 \otimes \beta_2) \in d_k(s, e)$, $\varpi_1 = \varpi_2$. Moreover, if $d_1(s, e)$ and $d_2(s, e)$ are two distinguishing strings then for all $(\alpha_1, \varpi_1 \otimes \beta_1) \in d_1(s, e), (\alpha_2, \varpi_2 \otimes \beta_2) \in d_2(s, e)$, $\varpi_1 \neq \varpi_2$. □

In Table 7.2, $T_P(ON, T)$ contains two distinguishing subsets, namely $d_1(ON, T) = \{(\perp \cdot 5, H \otimes l)\}$ and $d_2(ON, T) = \{(\perp \cdot 25, F \otimes l)\}$, since $H \neq F$.

Distinguishing Input Parameter Value Strings:

In each pair in a distinguishing subset, e.g., $(\perp \cdot 5, H \otimes l) \in d_1(ON, T)$, the string $\perp \cdot 5$ is called a *distinguishing input parameter value string*. We create a set of such parameter value strings from all the distinguishing subsets in $T_P(s, e)$. The definition of the set is as follows.

Definition 14 Let a distinguishing subset $d_k(s, e) = \{({\alpha_1}^{(k)}, \varpi^{(k)} \otimes {\beta_1}^{(k)}), \ldots, ({\alpha_m}^{(k)}, \varpi^{(k)} \otimes {\beta_m}^{(k)})\} \subseteq T_P(s, e)$, with $m = |d_k(s, e)|$. Then, we call $\pi(d_k(s, e))$ the set of *distinguishing input parameter value string* from $d_k(s, e)$, i.e., $\pi(d_k(s, e)) = \{{\alpha_1}^{(k)}, \ldots, {\alpha_m}^{(k)}\}$. Moreover, for two distinguishing subsets $d_1(s, e)$ and $d_2(s, e)$, $\pi(d_1(s, e)) \cap \pi(d_2(s, e)) = \emptyset$, i.e., each distinguishing subset contains different distinguishing parameter value strings. We call $\pi(T_P(s, e))$ the set of all the distinguishing parameter value strings from $T_P(s, e)$, i.e., $\pi(T_P(s, e)) = \bigcup_{k=1}^{\eta(s,e)} \pi(d_k(s, e))$.□

Here are the examples for the above notations. In Table 7.2, $\pi(d_1(ON, T)) = \{\perp \cdot 5\}, \pi(d_2(ON, T)) = \{\perp \cdot 25\}$ and $\pi(T_P(ON, T)) = \{\perp \cdot 5, \perp \cdot 25\}$.

We denote by $\mathcal{D}(s, e, \alpha)$ a distinguishing subset $d_k(s, e)$ such that $\alpha \in \pi(d_k(s, e))$, $1 \leq k \leq \eta(s, e)$, for $s \in S_P \cup R$, $e \in E_P$. For example, $\mathcal{D}(ON, T, \perp \cdot 5) = \{(\perp \cdot 5, H \otimes l)\}$ in Table 7.2.

Output Functions in (S_P, R, E_P, T_P):

Each pair in a distinguishing subset contains an output symbol string and an output parameter value string. We refer the two as the functions OS and OPS respectively. The functions are defined as follows.

	T
$T \otimes 15$	$(15 \cdot 25, F \otimes l)$

If $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, T \cdot T, 15 \cdot 25) = S \cdot F$
and $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, T \cdot T)(15 \cdot 25) = \perp \cdot l$
then $OS(T, T, 15 \cdot 25) = F$
and $OPS(T, T, 15 \cdot 25) = l$

Figure 7.2: Illustration of Property 4

Definition 15 For $(\alpha, \varpi \otimes \beta) \in \mathcal{D}(s, e, \alpha)$, we refer to ϖ as the *output string* function $OS(s, e, \alpha)$ and β as the *output parameter string* function $OPS(s, e, \alpha)$. □

For example, for $\mathcal{D}(ON, T, \perp\cdot 5) = \{(\perp\cdot 5, H \otimes l)\}$, $OS(ON, T, \perp\cdot 5) = H$ and $OPS(ON, T, \perp\cdot 5) = l$. The relation of OS with $\lambda_{\mathcal{P}}$ and OPS with $\sigma_{\mathcal{P}}$ can be formalized as Property 4. Figure 7.2 illustrates Property 4 that when a parameterized input string is given as $T \cdot T \otimes 15 \cdot 25$ and the corresponding output symbol string is $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, T \cdot T, 15 \cdot 25) = S \cdot F$ and the output parameter value string is $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, T \cdot T)(15 \cdot 25) = \perp \cdot l$, then we have $OS(T, T, 15 \cdot 25) = F$ and $OPS(T, T, 15 \cdot 25) = l$. Following is the formal description of Property 4.

Property 4 *Assume that there exists* $s \in S_P \cup R, e \in E_P$, $\alpha \cdot \gamma \in \pi(T_P(s, e))$, *where* $|\alpha| = |IS(s)|, |\gamma| = |e|$, *then we have* $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot e, \alpha \cdot \gamma) = \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s), \alpha) \cdot OS(s, e, \alpha \cdot \gamma)$ *and* $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot e)(\alpha \cdot \gamma) = \sigma_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s))(\alpha) \cdot OPS(s, e, \alpha \cdot \gamma)$.

Disputed Rows:

For $s \in S_P$ and $e \in E_P$, a distinguishing subset in $T_P(s, e)$ represents a transition in the conjecture. If $T_P(s, e)$ contains multiple distinguishing subsets then we call such an s a *disputed row.* For example, the row ON in Table 7.2 is disputed since $T_P(ON, T)$ contains two distinguishing subsets, namely $d_1(ON, T)$ and $d_2(ON, T)$. The formal definition of a disputed row is given as follows.

Definition 16 The row $s \in S_P$ is *disputed* if and only if there exists $e \in E_P$, such that $\eta(s, e) > 1$, i.e., $T_P(s, e)$ contains more than one *distinguishing subsets.* □

In order to make a well-defined transition function, the disputed rows must be *treated.* This is because a disputed row $s \in S_P$ contains many distinguishing subsets. Then, for each distinguishing subset, there will be a transition in the conjecture. In order to determine the target state of the transition, we must have a row in the table for the distinguishing subset. If there is a row for each distinguishing subset of s, then s is called treated. For example, the disputed row ON in Table 7.2 is treated because for each distinguishing subset $d_1(ON, T) = \{(\perp \cdot 5, H \otimes l)\}$ and $d_2(ON, T) = \{(\perp \cdot 25, F \otimes l)\}$, we have $ON \cdot T \otimes \perp \cdot 5$ and $ON \cdot T \otimes \perp \cdot 25$ in R, where $\perp \cdot 5 \in \pi(d_1(ON, T))$ and $\perp \cdot 25 \in \pi(d_2(ON, T))$. The formal definition is given as follows.

Definition 17 Let $s \in S_P$ be a disputed row in the observation table (S_P, R, E_P, T_P), then s is *treated* if and only if for every distinguishing subset $d_k(s,e) \subset T_P(s,e), e \in E_P, 1 \leq k \leq \eta(s,e)$, there exists $t \in S_P \cup R$ such that $t = IS(s) \cdot e \otimes \alpha$, where $\alpha \in \pi(d_k(s,e))$. □

Single Notation for types of rows:

Each $s \in S_P \cup R$ represents a row of the table. There could be two types of rows according to the definition. That is,

Type I: s can be an input symbol string, i.e., $s \in I^*$.

Type II: s can be a parameterized input string, i.e., $s \in \mathcal{U}$.

In Table 7.1, all the rows consist of input symbol strings (*Type I*), e.g., T in $S_P \cup R$. In Table 7.2, there are two rows that consist of parameterized input strings (*Type II*), namely $ON \cdot T \otimes \perp \cdot 5$ and $ON \cdot T \otimes \perp \cdot 25$ in $S_P \cup R$. The latter is in fact the result of the disputed row treatments. We unify the two types with a single notation as follows.

Definition 18 For certain $\omega \in I^*, \alpha \in {D_I}^*$ such that $|\omega| = |\alpha|$, we define ω_α to identify $s \in S_P \cup R$ such that either $s = \omega$ (*Type I*) or $s = \omega \otimes \alpha$ (*Type II*), and there exists $e \in E_P$ and $\gamma \in {D_I}^*$ such that $\alpha \cdot \gamma \in \pi(T_P(s,e))$. For ω, the two types cannot exist at the same time in $S_P \cup R$. □

In Table 7.1, T_5 means the row $T \in S_P \cup R$ since there exists $5 \cdot 25 \in \pi(T_P(T,T))$. Here $s = T$, $e = T$, $\alpha = 5$ and $\gamma = 25$. In Table 7.2, $(ON \cdot T)_{(\perp \cdot 25)}$ means the row $ON \cdot T \otimes \perp \cdot 25 \in S_P \cup R$ since there exists $\perp \cdot 25 \cdot 5 \in \pi(T_P(ON \cdot T \otimes \perp \cdot 25, T))$. Here $s = ON \cdot T \otimes \perp \cdot 25$, $e = T$, $\alpha = \perp \cdot 25$ and $\gamma = 5$. Additionally, in Table 7.1, $T_5 = T_{25}$. In Table 7.2, $(ON \cdot T)_{(\perp \cdot 5)} \neq (ON \cdot T)_{(\perp \cdot 25)}$.

Comparison of Rows

The rows $s_1, s_2 \in S_P \cup R$ are comparable with the help of the following definitions.

The rows s_1 and s_2 are called *balanced*, if they contain the same input parameter values for all $e \in E_P$. In Table 7.1, for ϵ and $ON \in S_P \cup R$ and $T \in E_P$, $T_P(\epsilon, T)$ contains only one pair in which T is associated with the input parameter value 5. Whereas, $T_P(ON, T)$ contains only one pair in which T is associated with the input parameter value 25. This creates an unbalance in the rows, since the observations are recorded for different input parameter values and the rows ϵ and ON cannot be compared directly. On the other hand, the rows ϵ and ON in Table 7.2 are balanced since both $T_P(\epsilon, T)$ and $T_P(ON, T)$ contain values 5 and 25. The formal definition is given below.

Definition 19 Let s_1 and s_2 be two two rows in (S_P, R, E_P, T_P), then s_1 and s_2 are *balanced* if and only if for all $\alpha_1 \in \pi(T_P(s_1, e))$ there exists $\alpha_2 \in \pi(T_P(s_2, e))$ such that $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$, and for all $\alpha_2 \in \pi(T_P(s_2, e))$ there exists $\alpha_1 \in \pi(T_P(s_1, e))$ such that $\mathit{suff}^{|e|}(\alpha_2) = \mathit{suff}^{|e|}(\alpha_1)$, for all $e \in E_P$. $\square$

The rows s_1 and s_2 are *equivalent* if they are balanced and they contain same parameterized output strings for the same parameterized input strings. Let $\Phi_P = \{e \otimes \mathit{suff}^{|e|}(\alpha), \forall \alpha \in \pi(T_P(s, e)), s \in S_P, e \in E_P\}$ be the set of parameterized strings from (S_P, R, E_P, T_P). For example in Table 7.2, $\Phi_P = \{ON \otimes \perp, OFF \otimes \perp, T \otimes 5, T \otimes 25\}$. Then, the rows in (S_P, R, E_P, T_P) are equivalent if for each parameterized input string in Φ_P, they produce the same parameterized output string. In Table 7.2, the rows ϵ and ON are not equivalent because for $T \otimes 5 \in \Phi_P$, they contain different output string, namely $\Omega \otimes \perp$ and $H \otimes l$. On the other hand, the rows ϵ and $ON \cdot OFF$ are equivalent. The formal definition is given below.

Definition 20 Let (S_P, R, E_P, T_P) be a balanced observation table and $\Phi_P = \{e \otimes \mathit{suff}^{|e|}(\alpha), \forall \alpha \in \pi(T_P(s, e)), s \in S_P, e \in E_P\}$ be the set of parameterized strings from the table, then for all $e \otimes \gamma \in \Phi_P$, there exists $\alpha_1 \in \pi(T_P(s_1, e))$ and $\alpha_2 \in \pi(T_P(s_2, e))$ such that $\gamma = \mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$, for all $s_1, s_2 \in S_P \cup R, e \in E_P$. Then, s_1 and s_2 are *equivalent*, denoted by $s_1 \cong_{\Phi_P} s_2$, if and only if $OS(s_1, e, \alpha_1) = OS(s_2, e, \alpha_2)$ and $OPS(s_1, e, \alpha_1) = OPS(s_2, e, \alpha_2)$. $\square$

For $s_1, s_2 \in S_P \cup R$, if $s_1 \cong_{\Phi_P} s_2$ then s_1 is in the equivalence class of s_2. We denote by $[s]$, $s \in S_P \cup R$, the equivalence class of rows that also includes s. The following lemma claims that $\cong_{\Phi_P}$ is an equivalence relationship.

Lemma 4 *Assume that (S_P, R, E_P, T_P) is a balanced observation table, then $\cong_{\Phi_P}$ is an equivalence relationship.* $\square$

Proof The lemma is proved on the properties symmetry, reflexivity and transitivity as follows.

Symmetry:

The property of symmetry for $\cong_{\Phi_P}$ is obvious.

Reflexivity:

The property of reflexivity implies that if $s_1 \cong_{\Phi_P} s_2$ then $s_2 \cong_{\Phi_P} s_1$ also holds, for all $s_1, s_2 \in S_P \cup R$. This is easy to prove from Definition 19. If $s_1 \cong_{\Phi_P} s_2$ then s_1 and s_2 are balanced, i.e., for all $\alpha_1 \in \pi(T_P(s_1, e))$, there exists $\alpha_2 \in \pi(T_P(s_2, e))$ such that $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$, for all $e \in E_P$. Conversely, for all $\alpha_2 \in \pi(T_P(s_2, e))$, there exists $\alpha_1 \in \pi(T_P(s_1, e))$ such that $\mathit{suff}^{|e|}(\alpha_2) = \mathit{suff}^{|e|}(\alpha_1)$, for all $e \in E_P$. If $s_1 \cong_{\Phi_P} s_2$, then $OS(s_1, e, \alpha_1) = OS(s_2, e, \alpha_2)$ and $OPS(s_1, e, \alpha_1) = OPS(s_2, e, \alpha_2)$ must hold. If $s_2 \cong_{\Phi_P} s_1$, then $OS(s_2, e, \alpha_2) = OS(s_1, e, \alpha_1)$

and $OPS(s_2, e, \alpha_2) = OPS(s_1, e, \alpha_1)$ must hold. This implies that when $s_1 \cong_{\Phi_P} s_2$ then $s_2 \cong_{\Phi_P} s_1$ also holds.

Transitivity:

The property of transitivity implies that if $s_1 \cong_{\Phi_P} s_2$ and $s_2 \cong_{\Phi_P} s_3$, then $s_1 \cong_{\Phi_P} s_3$ also holds, for all $s_1, s_2, s_3 \in S_P \cup R$.

Assume that s_1 and s_3 are not equivalent, i.e., $s_1 \ncong_{\Phi_P} s_3$. Since $s_1 \cong_{\Phi_P} s_2$, we know that s_1 and s_2 are balanced, and for all $\alpha_1 \in \pi(T_P(s_1, e))$ and $\alpha_2 \in \pi(T_P(s_2, e))$ such that $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$, for all $e \in E_P$, $OS(s_1, e, \alpha_1) = OS(s_2, e, \alpha_2)$ and $OPS(s_1, e, \alpha_1) = OPS(s_2, e, \alpha_2)$ holds.

Similarly, since $s_2 \cong_{\Phi_P} s_3$, we know that s_2 and s_3 are balanced, and for all $\alpha_2 \in \pi(T_P(s_2, e))$ and $\alpha_3 \in \pi(T_P(s_3, e))$ such that $\mathit{suff}^{|e|}(\alpha_2) = \mathit{suff}^{|e|}(\alpha_3)$, for all $e \in E_P$, $OS(s_2, e, \alpha_2) = OS(s_3, e, \alpha_3)$ and $OPS(s_2, e, \alpha_2) = OPS(s_3, e, \alpha_3)$ holds.

If $s_1 \ncong_{\Phi_P} s_3$, then there must exist $\alpha_1 \in \pi(T_P(s_1, e))$ and $\alpha_3 \in \pi(T_P(s_3, e))$ such that $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_3)$, for all $e \in E_P$, and $OS(s_1, e, \alpha_1) \neq OS(s_3, e, \alpha_3)$. Since the table is balanced, there exists $\alpha_2 \in \pi(T_P(s_2, e))$ such that $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2) = \mathit{suff}^{|e|}(\alpha_3)$. Since $s_1 \cong_{\Phi_P} s_2$, $OS(s_1, e, \alpha_1) = OS(s_2, e, \alpha_2)$ and $OPS(s_1, e, \alpha_1) = OPS(s_2, e, \alpha_2)$ holds. Since $s_2 \cong_{\Phi_P} s_3$, $OS(s_2, e, \alpha_2) = OS(s_3, e, \alpha_3)$ and $OPS(s_2, e, \alpha_2) = OPS(s_3, e, \alpha_3)$ holds. This means, $OS(s_1, e, \alpha_1) = OS(s_3, e, \alpha_3)$ and $OPS(s_1, e, \alpha_1) = OPS(s_3, e, \alpha_3)$. Therefore, $s_1 \cong_{\Phi_P} s_3$, which is a contradiction to the supposition. Hence, if $s_1 \cong_{\Phi_P} s_2$ and $s_2 \cong_{\Phi_P} s_3$ then $s_1 \cong_{\Phi_P} s_3$ also holds, for all $s_1, s_2, s_3 \in S_P \cup R$. ■

Constructing the PFSM conjecture

A PFSM conjecture from the table (S_P, R, E_P, T_P) can be constructed if the table fulfills the following three properties.

The table (S_P, R, E_P, T_P) must be *balanced*, i.e., for all $s, t \in S_P \cup R$, s and t are balanced. The table must be *dispute-free*, i.e., for all disputed rows $s \in S_P$, s is treated. The table must be *closed*, i.e., for every $t \in R$, there exists $s \in S_P$ such that $s \cong_{\Phi_P} t$.

When the observation table (S_P, R, E_P, T_P) is balanced, dispute-free and closed, the PFSM conjecture can be constructed as follows.

Definition 21 Let (S_P, R, E_P, T_P) be a balanced, dispute-free and closed observation table, then the PFSM conjecture $M_P = (Q_P, I, O, D_I, D_O, \Gamma_P, q_{0P})$ with the transition function δ_P, output function λ_P and output parameter function σ_P is defined, where

- $Q_P = \{[s] | s \in S_P\}$
- $q_{0P} = [\epsilon]$
- Γ_P : The set of transitions for the conjecture is defined as follows.

  For each $s \in S_P, i \in I$, there exists $\eta(s, i)$ transitions. Then, each distinguishing subset $d_k(s, i) \subseteq T_P(s, i), 1 \leq k \leq \eta(s, i)$, defines a transition $\{s, s', i, o, p, f\}$. Let $x = \mathit{suff}^{\,1}(\alpha)$, for all $\alpha \in \pi(d_k(s, i))$, then

  - p is the set of all such x
  - $s' = \delta_P([s], i, x) = [(IS(s) \cdot i)_\gamma], \gamma \in \pi(d_k(s, i))$
  - $o = \lambda_P([s], i, x) = OS(s, i, \alpha)$
  - $f(x) = \sigma_P([s], i)(x) = OPS(s, i, \alpha)$

To see that M_P is well-defined, note that S_P is a non-empty set and always contains at least one row ϵ. Also, $\cong_{\Phi_P}$ is an equivalence relationship (Lemma 4). Hence Q_P and q_0 are well-defined. For all $s \in S_P, i \in I, \alpha \in \pi(d_k(s, i)), 1 \leq k \leq \eta(s, i)$, there exists only one row $(IS(s) \cdot i)_\gamma \in S_P \cup R$ such that $\gamma \in \pi(d_k(s, i))$ for $x = \mathit{suff}^{\,1}(\alpha)$, since the table is balanced, dispute-free and closed. Hence, $[(IS(s) \cdot i)_\gamma] \in Q_P$ exists and therefore, δ_P is well-defined. Similarly, for $d_k(s, i)$, α and x, $OS(s, i, \alpha)$ and $OPS(s, i, \alpha)$ always exist. Hence, λ_P and σ_P are well-defined. We refer to Section 7.2.4 for details regarding the correctness of the conjecture.

7.2.2 The Algorithm $L_P{}^*$

The algorithm $L_P{}^*$ starts by initializing (S_P, R, E_P, T_P) with $S_P = \{\epsilon\}, E_P = I$ and $R = \{\epsilon \cdot i\}$, for all $i \in I$. Then, it asks parameterized queries constructed from the table and records the corresponding result of the queries in the table. Then, it checks properties and makes the table balanced, dispute-free and closed. Finally, it outputs a PFSM conjecture from the table. Each step of the algorithm is explained below.

Constructing the parameterized queries

The parameterized queries are constructed by the elements of $(S_P \cup R) \times E_P$. Since $S_P \cup R$ contains input symbol strings as well as parameterized input strings, the queries in each case are constructed as follows:

- Let $\omega \in S_P \cup R$ and $e \in E_P$, where $\omega \in I^*$ is an input symbol string. Then the query is constructed as $\omega \cdot e \otimes \alpha_1 \cdot \alpha_2$, where α_1 and α_2 are selected[1] from ${D_I}^+$ such that $|\omega| = |\alpha_1|$ and $|e| = |\alpha_2|$. For example, in Table 7.1, for $ON \in S_P \cup R$, $T \in E_P$, the query is constructed as $ON \cdot T \otimes \perp \cdot 25$, where $\perp \cdot 25 \in {D_I}^+$.

- Let $\omega \otimes \alpha_1 \in S_P \cup R$ and $e \in E_P$, where $\omega \otimes \alpha_1 \in \mathfrak{U}$ is a parameterized input string. Then the query is constructed as $\omega \cdot e \otimes \alpha_1 \cdot \alpha_2$, where α_2 is selected from ${D_I}^+$ such that $|e| = |\alpha_2|$. For example, in Table 7.2, for $ON \cdot T \otimes \perp \cdot 25 \in S_P \cup R$, $T \in E_P$, the query is constructed as $ON \cdot T \cdot T \otimes \perp \cdot 25 \cdot 5$, where $5 \in D_I$.

Recording the observations

The observation of each query is recorded in the table. There can be multiple rows which are updated correspondingly. Let $\omega \otimes \alpha$ be a query that generates a parameterized output string $\varpi \otimes \beta$, where $\varpi = \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, \omega, \alpha), \beta = \sigma_{\mathcal{P}}(q_{0\mathcal{P}}, \omega)(\alpha)$. Then, the string $\varpi \otimes \beta$ is recorded in the table as follows:

Let $s \in S_P \cup R, e \in E_P$ such that $IS(s) \cdot e$ is a prefix of ω. Let $\alpha' = \mathit{pref}^{|IS(s) \cdot e|}(\alpha)$, $\varpi' = \mathit{pref}^{|IS(s) \cdot e|}(\varpi)$, $\beta' = \mathit{pref}^{|IS(s) \cdot e|}(\beta)$ be the prefixes of α, ϖ and β respectively of the length of $IS(s) \cdot e$, then the pair

$$(\alpha', \mathit{suff}^{|e|}(\varpi') \otimes \mathit{suff}^{|e|}(\beta')) \text{ is added to } T_P(s, e), \text{ if it does not exist already.}$$

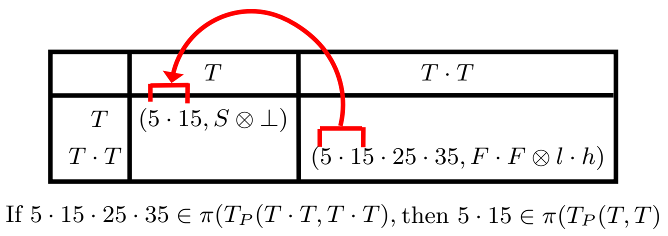
For example, for the query $ON \cdot T \cdot T \otimes \perp \cdot 25 \cdot 25$, we have the observation $OK \cdot F \cdot F \otimes \perp \cdot l \cdot l$. Then, in Table 7.2, for $\epsilon \in S_P \cup R$, $ON \in E_P$, we add $(\perp, OK \otimes \perp)$ to $T_P(\epsilon, ON)$, for $ON \in S_P \cup R$, $T \in E_P$, we add $(\perp \cdot 25, F \otimes l)$ to $T_P(ON, T)$ and for $ON \cdot T \otimes \perp \cdot 25 \in S_P \cup R$, $T \in E_P$, we add $(\perp \cdot 25 \cdot 25, F \otimes l)$ to $T_P(ON \cdot T \otimes \perp \cdot 25, T)$.

From the observation recording rule, we have Property 5 and Property 6, explained in Figure 7.3a and 7.3b respectively. Figure 7.3a explains that if $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T \cdot T, T \cdot T))$ then $5 \cdot 15 \in \pi(T_P(T, T))$ also holds. Figure 7.3b explains that if $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T, T \cdot T \cdot T))$ then $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T \cdot T, T \cdot T))$ also holds. The properties are defined formally as follows.

Property 5 *Assume that there exists $s, t \in S_P \cup R$ and $i \in I$ such that $IS(t) = IS(s) \cdot i$. Then, for all $e \in E_P$ and $\alpha \cdot \gamma \in \pi(T_P(t, e))$, where $|\alpha| = |IS(t)|$, α must be in $\pi(T_P(s, i))$.*

Property 6 *Assume that there exists $s \in S_P, t \in S_P \cup R, i \in I$ and $e, f \in E_P$, such that $f = i \cdot e$ and $IS(t) = IS(s) \cdot i$. Then, for all $\alpha \cdot x \cdot \gamma \in \pi(T_P(s, f))$, where $|\alpha| = |IS(s)|, |x| = |i|, |\gamma| = |e|$, $\alpha \cdot x \cdot \gamma$ must be in $\pi(T_P(t, e))$.*

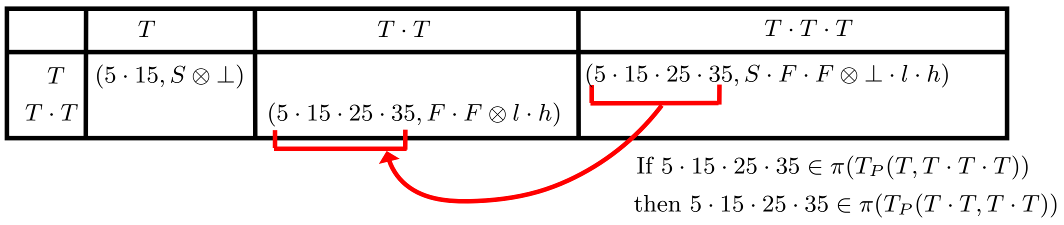
[1] We opt for random selection of parameter values. See discussion in Section 7.2



	T	$T \cdot T$
T	$(5 \cdot 15, S \otimes \bot)$	
$T \cdot T$		$(5 \cdot 15 \cdot 25 \cdot 35, F \cdot F \otimes l \cdot h)$

If $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T \cdot T, T \cdot T)$, then $5 \cdot 15 \in \pi(T_P(T, T)$

(a) Illustration of Property 5



	T	$T \cdot T$	$T \cdot T \cdot T$
T	$(5 \cdot 15, S \otimes \bot)$		$(5 \cdot 15 \cdot 25 \cdot 35, S \cdot F \cdot F \otimes \bot \cdot l \cdot h)$
$T \cdot T$		$(5 \cdot 15 \cdot 25 \cdot 35, F \cdot F \otimes l \cdot h)$	

If $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T, T \cdot T \cdot T))$
then $5 \cdot 15 \cdot 25 \cdot 35 \in \pi(T_P(T \cdot T, T \cdot T))$

(b) Illustration of Property 6

Figure 7.3: Illustrations of Property 5 and Property 6

Checking the properties of the table

After recording the results of the queries, the table is made balanced, dispute-free and closed in the following order.

Balanced: The table is made balanced after each query. Whenever it is not balanced, $L_P{}^*$ finds $s_1, s_2 \in S_P \cup R, e \in E_P, \alpha_1 \in \pi(T_P(s_1, e))$ such that there does not exist $\alpha_2 \in \pi(T_P(s_2, e))$ where $\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$. Then, $L_P{}^*$ balances s_1 and s_2 by constructing the query

$$IS(s_2) \cdot e \otimes \mathit{pref}^{|IS(s2)|}(\alpha'_2) \cdot \mathit{suff}^{|e|}(\alpha_1), \text{ where } \alpha'_2 \in \pi(T_P(s_2, e)),$$

and records the observations accordingly.

Dispute-Free: The table is made dispute-free after balancing. Let $s \in S_P$ be disputed then $L_P{}^*$ finds $e \in E_P$ such that $\eta(s, e) > 1$. Then, for every distinguishing subset $d_k(s, e) \subset T_P(s, e), 1 \leq k \leq \eta(s, e)$, the string

$$IS(s) \cdot e \otimes \alpha \text{ is added to } R, \text{ where } \alpha \in \pi(d_k(s, e)).$$

$L_P{}^*$ removes the original row $IS(s) \cdot e \in S_P \cup R$, and constructs additional queries for the new rows in the table.

Closed: When the table is made balanced and dispute-free, ${L_P}^*$ checks if it is closed. Whenever it is not closed, ${L_P}^*$ finds $t \in R$ such that for all $s \in S_P$, $s \not\cong_{\Phi_P} t$, and moves t to S_P. Then, R is extended as follows.

- If $t = \omega \in I^+$ an input symbol string, then the string $\omega \cdot i$ is added to R, for all $i \in I$.
- If $t = \omega \otimes \alpha \in \mathcal{U}$ an input parameterized string, then let $x = \mathit{suff}^{\,1}(\alpha')$ and $\alpha' \in \pi(T_P(t,i))$, then the string $\omega \cdot i \otimes \alpha \cdot x$ is added to R, for all $i \in I$.

Constructing the PFSM conjecture

When the observation table (S_P, R, E_P, T_P) is balanced, dispute-free and closed, the PFSM conjecture can be constructed according to Definition 21.

7.2.3 Processing Counterexamples in ${L_P}^*$

Let $\nu = \omega \otimes \alpha, \omega \in I^+, \alpha \in {D_I}^+$ be a parameterized string. In PFSM inference, we distinguish two types of strings that can be provided through external sources. The first type of strings may consist in parameter values that have not been tested in the learning procedure. If these parameter values do not bring any structural changes in the model, i.e., their processing in the table does not consequently create new transitions or states in the conjecture, then we call such strings *parameter augment examples.* If ν is a parameter augment example then processing ν in (S_P, R, E_P, T_P) consists in recording $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, \omega, \alpha)$ and $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, \omega)(\alpha)$ in the table accordingly, followed by making the table balanced. Since there are no structural changes; balancing the rows does not create any more disputed rows or make the two rows inequivalent that are previously equivalent. Therefore, the properties dispute-free and closed in the table shall hold trivially.

The second type of strings are those which bring structural changes in the model, i.e., their processing in the table consequently creates new transitions or states in the conjecture, then we call such strings *counterexamples.* If ν is a counterexample then processing ν is the reflection of the method that we have described in the Mealy machine inference (Chapter 5, Section 5.3). The method for processing counterexamples in (S_P, R, E_P, T_P) is described as follow.

Let $\nu = \omega \otimes \alpha$ be a counterexample then ${L_P}^*$ processes ν by searching $u \in S_P \cup R$ such that $IS(u)$ is the longest prefix of ω in $S_P \cup R$. Let v be the corresponding suffix of ω, i.e., $\omega = IS(u) \cdot v$. Then it adds all the suffixes of v to E_P and records $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, \omega, \alpha)$ and $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, \omega)(\alpha)$ in the table accordingly. Then (S_P, R, E_P, T_P) is made balanced, dispute-free and closed to conjecture a new PFSM model consistent with the observations.

Algorithm 4 summarizes the algorithm ${L_P}^*$.

Input: The set of input symbols I and the set of input parameter values D_I
Output: PFSM conjecture M_P

```
1  begin
2      initialize the observation table $(S_P, R, E_P, T_P)$ with the sets
3      $S_P = \{\epsilon\}, E_P = I, R = \{\epsilon \cdot i\}, \forall i \in I$ ;
4      ask the parameterized queries from $(S_P, R, E_P, T_P)$ ;
5      update $(S_P, R, E_P, T_P)$ with the results of the queries ;
6      while $(S_P, R, E_P, T_P)$ is not balanced, dispute-free or closed do
7          make the table balanced, i.e., for all $s, t \in S_P \cup R$, $s$ and $t$ are balanced ;
8          make the table dispute-free, i.e., for all $s \in S_P, e \in E$, such that $\eta(s, e) > 1$, $s$ is
           treated ;
9          make the table closed, i.e., for every $t \in R$, there exists $s \in S_P$ such that $s \cong_{\Phi_P} t$ ;
10     end
11     make the conjecture $M_P$ from $(S_P, R, E_P, T_P)$ ;
12     if there is a parameter augment example $\omega \otimes \alpha$ for $M_P$ then
13         update $(S_P, R, E_P, T_P)$ with $\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, \omega, \alpha)$ and $\sigma_{\mathcal{P}}(q_{0\mathcal{P}}, \omega)(\alpha)$ accordingly ;
14     end
15     if there is a counterexample $\omega \otimes \alpha$ for $M_P$ then
16         divide $\omega = IS(u) \cdot v$ such that $u \in S_P \cup R$ and $IS(u)$ is the longest prefix of $\omega$ in
           $S_P \cup R$ ;
17         add all the suffixes of $v$ to $E_P$ ;
18         ask the parameterized queries for the extended table ;
19     end
20     return the conjecture $M_P$ from $(S_P, R, E_P, T_P)$ ;
21 end
```

Algorithm 4: The Algorithm $L_P{}^*$

7.2.4 Correctness

Let (S_P, R, E_P, T_P) be a balanced, closed and dispute-free observation table and $M_P = (Q_P, I, O, D_I, D_O, \Gamma_P, q_{0P})$ be the conjecture from the table, then the correctness of the conjecture is claimed by Theorem 6 and Theorem 7. The proofs of the theorems are given in appendix A with illustrations.

Theorem 6 *If (S_P, R, E_P, T_P) is a balanced, dispute-free and closed observation table, then the conjecture M_P is consistent with the finite function T_P.* $\square$

Theorem 7 *If (S_P, R, E_P, T_P) is a balanced, dispute-free and closed observation table and the conjecture M_P has n states. Let another PFSM M'_P that accepts exactly the same parameter* *values as of M_P, i.e., for all $s \in S_P$, $i \in E_P$, $\alpha \in \pi(T_P(s,i))$, M'_P accepts only the values $suff^1(\alpha)$ for i. If M'_P is consistent with T_P but inequivalent to M_P, then M'_P must have more states.* □

7.2.5 Termination

The termination of ${L_P}^*$ is guaranteed by the finite space of states and transitions in $\mathcal{P}$. The operations which keep the algorithm running are making the table balanced, dispute-free and closed. Only the operations dispute-free and closed extend the table by adding rows. We show by analyzing each operation that ${L_P}^*$ will always terminate and produce a PFSM conjecture M_P.

Suppose a row s is added to S_P because the table is not closed. Then, by definition, s is not equivalent to all rows t in S_P, before S_P is augmented. So the number of rows in S_P is increased by one when s is added. Since, rows represent the states in the conjecture and S_P contains only inequivalent rows, the table will be found not closed for at most $n-1$ times, as there is initially one row in S_P already. Hence, ${L_P}^*$ will always eventually find a closed table.

Suppose a row t is added to R because the table is not dispute-free. Then, by definition, there exists $s \in S_P, e \in E_P$ such that $\eta(s,e) > 1$. The treatment of the disputed row s adds $\eta(s,e)$ rows in R. Let $\eta(s)$ denote the total number of distinguishing subsets in s, i.e., $\eta(s) = \eta(s,e_1) + \ldots + \eta(s,e_{|E_P|})$, for all $e_k \in E_P$, $1 \leq k \leq |E_P|$. Suppose w is the maximum number of total distinguishing subsets that can be found in the table, i.e., $w = \eta(s)$ such that $\eta(s) \geq \eta(t)$, for all $t \in S_P$. Then, for at most n rows in S_P, there can be added at most nw rows in R. Hence, ${L_P}^*$ will always eventually find a dispute-free table.

The operation of making the table balanced does not add a row in the table. Balancing is required when there are input parameter values in a row which are not found in the other rows. The rows are added in the table when the table is found not closed or not dispute-free. Both operations add finite number of rows in the table, i.e., at most $n + nw$. For each row, there can be tested finite number of parameter values. Thus, balancing finite number of rows with finite number of parameter values require finite number of queries to balance each row. Hence, ${L_P}^*$ will always eventually find a balanced table.

As far as parameter augment examples are concerned, the table is required to balance for new parameter values in the examples. This does not bring any structural change in the model. Thus, balancing the table in this case is no more than updating the transitions of the conjecture with new parameter values.

As far as counterexamples are concerned, the number of new states that can be discovered are at most n. So there can be at most $n-1$ counterexamples to distinguish n states, since there is one state initially. For transitions, there can be at most w transitions that can be discovered for a state. Both cases yield finite number of new rows in the table, when the table is found not closed or not dispute-free.

Hence, ${L_P}^*$ will always eventually find a balanced, dispute-free and closed observation table and terminate by producing a PFSM conjecture in finite number of operations.

7.2.6 Complexity

We analyze the total number of parameterized queries asked by ${L_P}^*$ in the worst case by the factors

- $|I|$, i.e., the size of I
- n, i.e., the minimum number of states in $\mathcal{P}$
- m, i.e., the maximum length of any counterexample provided during the learning of $\mathcal{P}$
- w, i.e., the maximum number of total distinguishing subsets $\eta(s)$ such that $\eta(s) \geq \eta(t)$, for all $t \in S_P$.
- p, i.e., the maximum number of input parameter value strings that are recorded in a cell of the table.

S_P contains initially one row. Every time the table is not closed, one element is added to S_P. This can happen at most $n-1$ times. Hence, the size of S_P is at most n.

R contains initially $|I|$ rows. Every time S_P is augmented (when the table is not closed), $|I|$ rows are added. This happens for $n-1$ times, so the number of rows that can be added to R in this case cannot exceed $n|I|$.

Every time the table is not dispute-free, at most w elements are added to R. For at most n rows, the number of rows that can be added to R in this case cannot exceed nw. Recall that $w = \eta(s)$ and $\eta(s) = \eta(s, e_1) + \ldots + \eta(s, e_{|E_P|})$ for all $e_k \in E_P$, $1 \leq k \leq |E_P|$. Since, $E_P \supseteq I$, therefore, nw includes $n|I|$ number of rows in the case the table is not closed. Hence, the size of R is at most nw.

E_P contains $|I|$ elements initially. If a counterexample is provided then at most m suffixes are added to E_P. There can be provided at most $n-1$ counterexamples to distinguish at most n states, thus the maximum size of E_P cannot exceed $|I| + m(n-1)$.

The number of parameterized queries are the queries required to balance the rows according to the number of parameter values recorded in the table. The row $s \in S_P \cup R$ is balanced when for all $t \in S_P \cup R$, $e \in E_P$, $\alpha_1 \in \pi(T_P(t,e))$, there exists $\alpha_2 \in \pi(T_P(s,e))$ such that

$\mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$. Otherwise, $\mathit{suff}^{|e|}(\alpha_1)$ is a parameter value string that is required to balance s for such e.

Let p be the maximum number of input parameter value strings that are required to balance any $s \in S_P \cup R$ for any $e \in E_P$, then there will be maximum $p(|S_P \cup R| \times |E_P|)$ parameterized queries required to balance the whole table.

Thus, the maximum parameterized queries required to output a correct PFSM conjecture is

$$p(\overbrace{n}^{|S_P|} + \overbrace{nw}^{|R|})\,(\overbrace{|I| + m(n-1)}^{|E_P|}) = O(p|I|nw + mn^2wp)$$

7.3 Relation of M_P with $\mathcal{P}$

We prove the relation M_P with $\mathcal{P}$ with respect to the set Φ_P. The relation is stated in the following Theorem.

Theorem 8 *Let $\mathcal{P} = \{Q_\mathcal{P}, I, O, D_I, D_O, \Gamma_\mathcal{P}, q_{0\mathcal{P}}\}$ be the unknown PFSM model and $M_P = (Q_P, I, O, D_I, D_O, \Gamma_P, q_{0P}\}$ be the PFSM conjecture from the balanced, dispute-free and closed observation table (S_P, R, E_P, T_P) and $\Phi_P = \{e \otimes \mathit{suff}^{|e|}(\alpha), \forall \alpha \in \pi(T_P(s, e)), s \in S_P, e \in E_P\}$ be the set of parameterized strings from (S_P, R, E_P, T_P), then M_P is a Φ_P-quotient of $\mathcal{P}$.*$\square$

Proof According to the definition of a PFSM quotient (Definition 10), the theorem can be proved in two parts.

1. Let M_P be a conjecture from a balanced, dispute-free and closed observation table (S_P, R, E_P, T_P), then for two states $q_P, q'_P \in Q_P$, there exists $s, t \in S_P$ such that $[s] = q_P$ and $[t] = q'_P$, according to Definition 21. We also know that $s \cong_{\Phi_P} t$, if and only if for all $e \otimes \gamma \in \Phi_P$, there exists $e \in E_P$, $\alpha_1 \in \pi(T_P(s, e))$ and $\alpha_2 \in \pi(T_P(t, e))$ such that $\gamma = \mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$ and $OS(s, e, \alpha_1) = OS(t, e, \alpha_2)$ and $OPS(s, e, \alpha_1) = OPS(t, e, \alpha_2)$. Since the table is balanced, such α_1 and α_2 must exist. This is true for all the elements of $[s]$ and $[t]$. Therefore, $[s] \cong_{\Phi_P} [t]$ implies that $q_P = q'_P$. Otherwise, there exists $e \otimes \gamma \in \Phi_P$, $e \in E_P$, $\alpha_1 \in \pi(T_P(s, e))$ and $\alpha_2 \in \pi(T_P(t, e))$ such that $\gamma = \mathit{suff}^{|e|}(\alpha_1) = \mathit{suff}^{|e|}(\alpha_2)$ but the condition $OS(s, e, \alpha_1) = OS(t, e, \alpha_2)$ and $OPS(s, e, \alpha_1) = OPS(t, e, \alpha_2)$ does not hold. Hence, $s \ncong_{\Phi_P} t$ in this case. This means $[s] \ncong_{\Phi_P} [t]$, implies that $q_P \neq q'_P$. This proves the first part, since q_P and q'_P can only be equal when s and t are equivalent with respect to Φ_P, i.e., they produce same parameterized output on all the strings from Φ_P, and not otherwise.

2. For $q_P \in Q_P$, there exists $s \in S_P$ such that $[s] = q_P$ (Definition 21). We know that for all $e \otimes \gamma \in \Phi_P$, $\lambda_\mathcal{P}(\delta_\mathcal{P}(q_{0\mathcal{P}}, IS(s), \alpha), e, \gamma) = OS(s, e, \alpha \cdot \gamma)$ and $\sigma_\mathcal{P}(\delta_\mathcal{P}(q_{0\mathcal{P}}, IS(s), \alpha), e)(\gamma) =$

$OPS(s, e, \alpha \cdot \gamma)$, where $\alpha \cdot \gamma \in \pi(T_P(s, e))$. Since, M_P is consistent with the observations in (S_P, R, E_P, T_P), then $\lambda_P(\delta_P(q_{0P}, s, \alpha), e, \gamma) = OS(s, e, \alpha \cdot \gamma)$ and $\sigma_P(\delta_P(q_{0P}, s, \alpha), e)(\gamma) = OPS(s, e, \alpha \cdot \gamma)$ also holds (Theorem 6). This proves the second part, since
$\lambda_{\mathcal{P}}(\delta_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s), \alpha), e, \gamma) = \lambda_P(\delta_P(q_{0P}, s, \alpha), e, \gamma)$ and
$\sigma_{\mathcal{P}}(\delta_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s), \alpha), e)(\gamma) = \sigma_P(\delta_P(q_{0P}, s, \alpha), e)(\gamma)$. ■

7.4 Application: The HVAC Controller

We have applied the algorithm $L_P{}^*$ for the inference of the HVAC controller described in Chapter 5 (Section 5.6). The discussion on the inference results is given in the following.

7.4.1 Inference of the HVAC Controller

In order to infer a PFSM model of the HVAC controller, the input set for the algorithm can be taken exactly as shown in Figure 5.6, without any particular adaptation like we did in the Mealy machine inference. Therefore, the input set for $L_P{}^*$ is constructed as $I = \{ON, OFF, T\}$ with $D_I = [-20, +50] \cup \{\perp\}$ as the input parameter domain. The algorithm is run until a balanced, dispute-free and closed observation table is found, shown in Table 7.3. The temperature values that are tested in the PFSM inference are the same as tested in the Mealy machine inference, i.e., -5,5,15,25,35 centigrades. The PFSM conjecture from the table is given in Figure 7.4.

7.4.2 Comparison of Mealy and PFSM learning

When a system can be modeled as a PFSM with the finite domain of i/o parameter values, there is an equivalent modeling as a Mealy machine. The correspondence between the PFSM model $(Q_P, I_P, O_P, D_{IP}, D_{OP}, \Gamma_P, q_{0P})$ and the Mealy machine model $(Q_M, I_M, O_M, \delta_M, \lambda_M, q_{0M})$ is the classical unfolding in extended state machines, i.e., $Q_M = Q_P$, $q_{0M} = q_{0P}$, $I_M = I_P \times D_{IP}$, $O_M = O_P \times D_{OP}$. The states remain the same, and transitions are replicated for each value of the parameter. Actually, the conjectures built by the algorithm $L_P{}^*$ always correspond to PFSM models with finite domains, since we test only finite number of values from the input parameter domains. The next chapter will address the issue of extending the parameter domain; but up to that point, we can consider for this comparison just PFSM models with finite domains.

Suppose that the system can be represented in both models, then we have the observation tables (S_M, E_M, T_M) and (S_P, R, E_P, T_P) for learning the Mealy machine and the PFSM model of the system respectively.

In Mealy machine learning, suppose there are k inputs $i_1, \ldots, i_k \in I_M$, such that for certain $s \in S_M$, the behaviors for all such inputs on the states $[s] \in Q_M$ are the same, i.e., $T_M(s, i_1) =$

		E_P		
		ON	OFF	T
S_P	ϵ	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(-5, \Omega \otimes \perp), (5, \Omega \otimes \perp), (15, \Omega \otimes \perp), (25, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
	ON	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$
	$ON \cdot T \otimes \perp \cdot 5$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, S \otimes \perp), (35, S \otimes \perp)$
	$ON \cdot T \otimes \perp \cdot 25$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, S \otimes \perp), (5, S \otimes \perp), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$
R	$ON \cdot OFF$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(-5, \Omega \otimes \perp), (5, \Omega \otimes \perp), (15, \Omega \otimes \perp), (25, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
	$ON \cdot T \otimes \perp \cdot 15$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$
	$ON \cdot T \cdot OFF \otimes \perp \cdot 5 \cdot \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(-5, \Omega \otimes \perp), (5, \Omega \otimes \perp), (15, \Omega \otimes \perp), (25, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
	$ON \cdot T \cdot OFF \otimes \perp \cdot 25 \cdot \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(-5, \Omega \otimes \perp), (5, \Omega \otimes \perp), (15, \Omega \otimes \perp), (25, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
	$ON \cdot T \cdot T \otimes \perp \cdot 5 \cdot 5$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, S \otimes \perp), (35, S \otimes \perp)$
	$ON \cdot T \cdot T \otimes \perp \cdot 5 \cdot 25$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$
	$ON \cdot T \cdot T \otimes \perp \cdot 25 \cdot 5$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, H \otimes h), (5, H \otimes l), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$
	$ON \cdot T \cdot T \otimes \perp \cdot 25 \cdot 25$	$(\perp, \Omega \otimes \perp)$	$(\perp, S \otimes \perp)$	$(-5, S \otimes \perp), (5, S \otimes \perp), (15, S \otimes \perp), (25, F \otimes l), (35, F \otimes h)$

Table 7.3: Balanced, Dispute-free and Closed Observation Table for learning PFSM model of HVAC controller

$\ldots = T_M(s, i_k)$. According to the structure of the table, we have additionally k rows in the table, i.e., $s \cdot i_1, \ldots, s \cdot i_k \in S_M \cup S_M \cdot I_M$.

Now, suppose that in PFSM learning, we have an input $i \in I_P$ that is associated with parameter values $i_1 \ldots i_k \in D_I$. According to the assumption that all the parameter values produce same behaviors, then we have $t \in S_P$ such that $OS(t, i, \alpha_1) = \ldots = OS(t, i, \alpha_k)$, where $\alpha_1, \ldots, \alpha_k \in \pi(T_P(t, i))$ and $i_1 = \mathit{suff}^{\,1}(\alpha_1), \ldots, i_k = \mathit{suff}^{\,1}(\alpha_k)$. Since, $\eta(t, i) = 1$, we have additionally only one row in the table, i.e., $t \cdot i \in S_P \cup R$. Then the table (S_P, R, E_P, T_P) has a gain of $k - 1$ rows compared to the table (S_M, E_M, T_M).

So the reduction in complexity achieved by using the PFSM learning algorithm over the Mealy machine learning algorithm is directly linked to the reduction in the size of the models

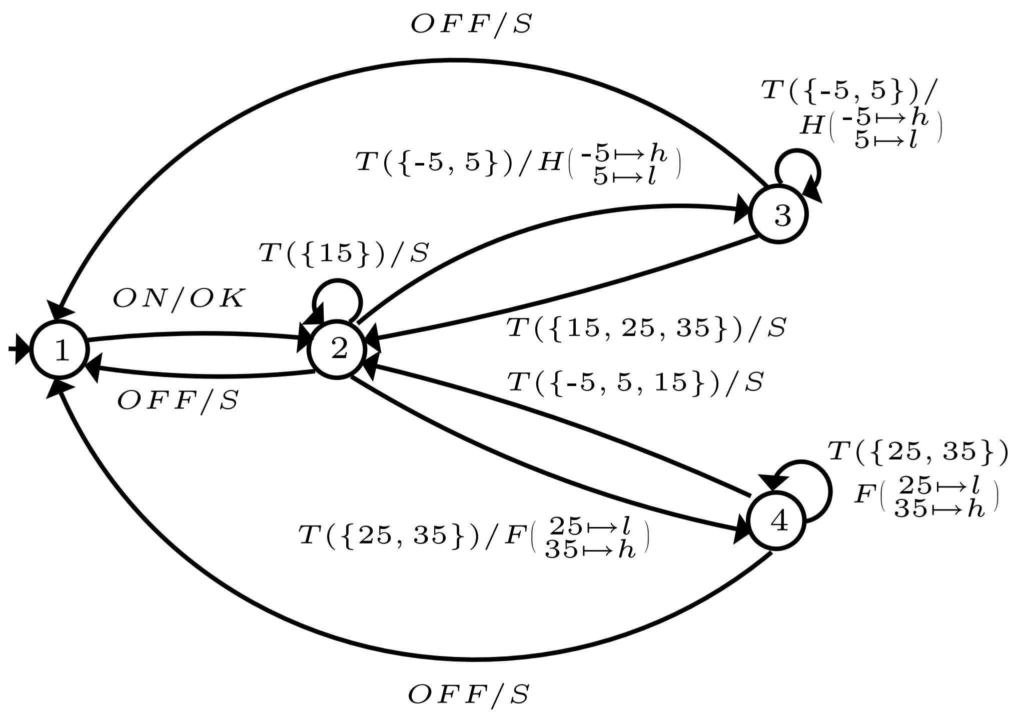


Figure 7.4: PFSM conjecture of the HVAC Controller from Table 7.3

that can be achieved with the use of PFSM modeling.

Actually, this crude analysis has to be refined by taking into account the level of information we may have on the equivalence of input parameter values; so the partitions induced by the predicates in the transitions of the PFSM model. In the above analysis, we assume that the Mealy machine splits the input parameter domain with one input for each value of the domain. This would be the case if we had no knowledge about the equivalence of parameter values to determine the behaviors of the machine.

If, however, we are testing with a perfect knowledge of the partition defined on the input parameter domain into equivalence classes, such that the system exhibits a similar behavior on all the values of a class, then we can perfectly construct our input set for learning the Mealy machine, in which each input symbol represents one class in the partition. As in the example of the HVAC controller, the values -5℃ and 5℃ both turn on the heater. Ideally, there should be one input that represents the whole interval $[-20, 11]$, since the behavior of the system on all the values within the interval is the same. In the Mealy machine inference of the example (Chapter 5, Section 5.6), we did not assume this knowledge beforehand and used two separate inputs, namely T-5 and T5, which actually represent the interval $[-20, 11]$. Thereby, the learning algorithm for Mealy machine ${L_M}^+$ constructed separate rows for each input in (S_M, E_M, T_M), shown in Table 5.8. On the contrary, the values -5°C and 5°C were considered as a single input when the PFSM model of the controller was inferred, because they exhibited the same behavior in (S_P, R, E_P, T_P), shown in Table 7.3.

Moreover, even with a perfect knowledge of the partition, there is still a gain with the PFSM learning algorithm, because rows would be split only in the case when there are different behaviors induced during testing. However, this is not the case in Mealy machines learning. There, we enumerate all values despite that some of them might have shown similar behaviors in some rows. In reality, what we should distinguish on one state of the system may not be true for the other states. This can be observed in the Mealy machine model of the HVAC controller, shown in Figure 5.7. In state 3, we only have to distinguish the interval $[-20, 11]$ from the other intervals, i.e., $[12, 15]$ and $[16, 50]$. Whereas in state 2, we really have to distinguish three intervals. And in state 1, we do not have to distinguish any interval, since the input is not valid.

To summarize the results of the HVAC controller example, let us take a closer look on Table 7.3. It is observed that the number of rows and columns in Table 7.3 are reduced (and so the number of queries), compared to Table 5.8. The total number of parameterized queries in PFSM learning for the HVAC controller is 84, compared to 140 output queries for learning Mealy machine (cf. Chapter 5, Section 5.6). The obvious reason is that each temperature value in the Mealy machine learning is considered as an input to the controller which contributed in its expansion. Whereas, the behavior of the controller remained mostly the same for different values (e.g., -5°C and 5°C).

Therefore, the main gain in the case of PFSM learning is that the algorithm efficiently keeps the size of the table reduced and only expands when it is required. This remains true whatever information we may have or learn about equivalent values.

Another benefit we get from PFSM learning, thanks to the parameterized structure of the model, is that, we were also able to obtain additional information about the HVAC system in terms of the output parameters. The output parameters indicate the speed of the appliances (heater/fan). That is, the heater is regulated with high h and low l speed levels on temperature values -5 and 5 centigrades respectively. The speed of the fan on the temperature values 25 and 35 centigrades is regulated analogously.

7.5 Conclusion

This chapter discussed the inference of parameterized black box systems. It is observed that the Mealy machine modeling is not sufficient because they do not adequately model the system that exchange lots of parameterized data values from arbitrary complex domains. The Mealy machine modeling of such systems would create a formidable size of the input set. As a consequence, it often results in a combinatorial blow up on the transition level. Moreover, the complexity of the learning algorithm is greatly enhanced by taking all data values as inputs. However, the systems usually manifest similar behaviors on a subset of values and differ only on minor details. Thus, it will be more adequate if such a system could be modeled in a compact (finite state machine) representation that shows the global behaviors (as state transitions) and the minor details could be treated as parameter values (associated with inputs and outputs on transitions).

We have proposed a parameterized finite state machine (PFSM) for modeling such systems. Taking the building blocks of the (active) learning algorithms for simple state machines, we have proposed a new algorithm $L_P{}^*$ that can learn a PFSM model in a polynomial time from a black box machine given the set of input symbols and the knowledge of the (infinite) parameter domain of the input symbols. If there is provided a counterexample then the algorithm refines the conjecture by processing the counterexample.

Finally, $L_P{}^*$ is applied on the inference of the HVAC Controller that is given in Chapter 5 for the application of Mealy machine inference algorithm $L_M{}^*$. The comparison of the two algorithms in terms of the size of the observation table, number of queries for inference and the amount of observations obtained in the two algorithms is also discussed.

The limitation of PFSM inference is that we learn a restrictive form of output parameter functions on the transitions. The functions are restricted with respect to the input parameter values we have tested in the learning procedure. In our case, we just have a simple mapping of input parameter values to their corresponding output parameter values. However, we have studied that if we assume certain restrictions on the functions, i.e., the functions are computable given the observations on input and output parameter values, then we can learn the functions and annotate the transitions with the functions instead of simple mappings. We are working on the application of the invariant detector *Daikon* [ECGN01] to detect such functions (as data invariants) using the observations in the table. We have tested on the machines that contain simple functions, like linear relationships (e.g., $y = 2x + 1$), ordering (e.g., $x \leq y$), etc. This work is detailed in Chapter 8.

Chapter 8

Parameter Function Inference in PFSM models

This chapter presents our extended work in PFSM inference for learning parameter functions. It proposes an idea of using invariant detection mechanism that can learn functions as data invariants over the observed parameter values during the learning procedure. The idea is explained with the help of an example. However, the results are preliminary at this stage and require further investigation from both theory and practice point of view.

8.1 Motivation

We have presented a parameterized model *PFSM* (Definition 4), and provided the algorithm $L_P{}^*$ (Algorithm 4) for inferring the PFSM models. A PFSM model contains transitions in which input and output symbols are associated with parameter values. The output parameter value for a given input parameter value is determined by an output parameter function in a transition. In the PFSM inference, we are limited to test certain input parameter values and observe the corresponding output parameter values, and finally produce a conjecture in which transitions are labeled with the input/output parameter value mappings. This means, we do not actually learn completely the output parameter functions on the transitions that have determined the output parameter values.

We extend the PFSM inference work in the direction of learning output parameter functions. We noticed that the observations on the input/output parameter values during the initial run of the algorithm can be used to infer the functions that have determined the output parameter values on the given input parameter values. Assuming that the input/output parameter domain is numeric, we can use external tools to learn arithmetic relations between the provided numeric parameter values. An example of such tools is invariant detectors.

In this chapter, we discuss the approach of using the invariant detection mechanism to learn functions (as invariants) using the observations of input and output parameter values. Section 8.2 discusses the approach in general. Section 8.3 introduces the invariant detector Daikon [ECGN01]. Section 8.4 illustrates the approach with the help of an example. Section 8.5 concludes the chapter.

8.2 Approach

We have observed that the output parameter functions in the PFSM model can be learned if certain restrictions on the functions are assumed, i.e., the functions are computable given the observations on the input and output parameter values. We see an application of invariant detection mechanism such that the functions can be learned as data invariants, using the observations on input and output parameter values on each transition as data values.

The key idea is the following. Once the learning algorithm $L_P{}^*$ terminates and a PFSM conjecture is obtained, then for each transition in the conjecture, we collect the mappings of input and output parameter values on the transition and provide the mappings to an invariant detector. The detector would learn the function as a data invariant over the mappings. The invariant is actually an approximation of the real output parameter function that has determined the output parameter values during the learning of the PFSM conjecture. Finally, the learned function will be annotated on the transition, replacing the original input and output parameter value mappings.

We have used the invariant detector *Daikon* [ECGN01] in our initial experiments. In the following, we introduce the tool shortly and then illustrate the approach of learning functions with the help of an example.

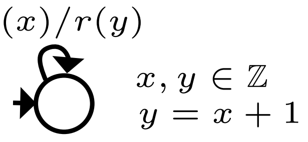
8.3 The Daikon Invariant Detector

Daikon [ECGN01] is an implementation of dynamic invariant detection that infers invariants over scalar and structured variables from program execution. The essential idea is to run the program over a test suite, collect traces from the program execution and use a generate-and-check algorithm to test a set of potential invariants against the traces. The algorithm initially assumes that all potential invariants over the variables of interest are true and incrementally tests each invariant against the observed values of these variables from the traces. At each step, the invariant is discarded if it is violated by the values to obtain a set of positive invariants. The remaining invariants at the end of the process are reported describing the relations as invariants on the set of values observed in the program behavior.

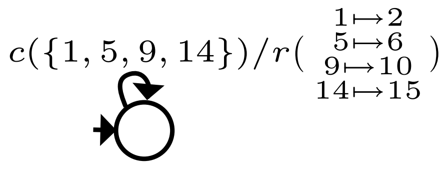
Daikon collects all program executions in a large file called data trace file '.dtrace'. It does not use source code at all when inferring invariants. The inference engine reads data trace files and runs the invariant detection algorithm on the values collected in the files. This functionality exactly matches our requirements. We have the mappings of input and output parameter values on each transition of the conjecture. The mappings can be written in a data trace file such that there will be one file for each transition. Then, Daikon can learn the invariant over the given values in the file. The learned invariant is actually the approximation of the parameter function on the transition of the system which has determined the output parameter values during the learning of the PFSM conjecture.

Daikon is enhanced with a number of optimizations that allows it to scale to large numbers of invariants. Currently, it checks over 70 invariants and the list is extendible by the users to accommodate their own domain-specific invariants and derived variables. Some of the invariants it detects are as follows: constant value ($x = a$), small value set ($x \in \{a, b, c\}$), range limits ($x \leq a$), non-zero ($x \neq 0$), modulus ($x \equiv a \pmod b$), comparisons ($x > y, x = y, ...$), linear relationships ($y = ax + b$), functional relationships ($y = f(x)$) and polynomial relationships ($z = ax + by + c$).

8.4 Example



(a) PFSM model of the counter


$c(\{1, 5, 9, 14\})/r(\begin{smallmatrix}1\mapsto 2\\5\mapsto 6\\9\mapsto 10\\14\mapsto 15\end{smallmatrix})$

(b) PFSM conjecture of the counter from Table 8.1

Figure 8.1: PFSM model of the counter and its conjecture

Let us illustrate the approach with the help of a simple example. Consider a counter which increments a given numeric value and returns the result back to the environment. A PFSM model of the counter is given in Figure 8.1a. The input symbol is c with the input parameter x and the output symbol is r with the output parameter y. The possible values for both parameters lie with in the range of integers.

The learning algorithm $L_P{}^*$ is applied to the counter and the balanced, dispute-free and closed observation table (S_P, R, E_P, T_P), shown in Table 8.1, is obtained. The input parameter

		E_P
		c
S_P	ϵ	$(1, r \otimes 2)$ $(5, r \otimes 6)$ $(9, r \otimes 10)$ $(14, r \otimes 15)$
R	c	$(1 \cdot 1, r \otimes 2)$ $(1 \cdot 5, r \otimes 6)$ $(14 \cdot 9, r \otimes 10)$ $(5 \cdot 14, r \otimes 15)$

Table 8.1: Balanced, Dispute-free and Closed Observation Table for the counter

```
Daikon version 4.3.1, released August 2, 2007;
http://pag.csail.mit.edu/daikon.
Processing trace data; reading 1 dtrace file:
[15:14:39]: Finished reading counter.dtrace
===================================================
Counter.counter():::ENTER
===================================================
Counter.counter():::EXIT
x one of { 1, 5, 9, 14 }
y one of { 2, 6, 10, 15 }
x == orig(x)
x - y + 1 == 0
Exiting Daikon.
```

Figure 8.2: Daikon output for the observed counter values recorded in Table 8.1. The invariant is inferred as: $x - y + 1 == 0$.

values that have been tested during the learning are $1, 5, 9, 14$ (random selection). The corresponding output parameter values are observed as $2, 6, 10, 15$ respectively. The conjecture is obtained from the table as shown in Figure 8.1b. The transition in the conjecture is labeled with the input and output parameter values observed in the learning procedure.

We write the mappings of the input and output parameter values as a pair (x, y), i.e., $(1, 2), (5, 6), (9, 10), (14, 15)$ in a data trace file and feed the file to Daikon. Daikon analyzes the values in the file and infers the invariant as $x - y + 1 == 0$, i.e., the value of y is always one greater than the value of x. The output of Daikon is shown in Figure 8.2. So, we replace the original mappings of input and output parameter values on the transition of the conjecture in Figure 8.1b, by the learned invariant. Finally, we achieve the PFSM model as shown in Figure 8.1a.

8.5 Conclusion

We have extended our work of inferring PFSM models and presented an idea of learning output parameter functions in the conjecture. The idea is to supply the observed mappings of input and output parameter values during the learning procedure to an invariant detector. The detector learns the output parameter function (as invariants) which has determined the output parameter values during the learning of the PFSM model.

The work of learning functions using invariant detection mechanism is currently in progress. We have performed additional experiments with Daikon and found it efficient for numeric applications. However, it has limitations for use in large sized applications where complex formulas are used for computations. Moreover, it generates few invariants unless it is supplied with very large data. We continue to further investigate in this direction with the intention to find solutions to overcome the limitations in learning functions.

Chapter 9

Tool and Case Studies

This chapter presents the implementation work carried out in the thesis. It introduces our tool RALT that implements our approach of learning and testing. Then, the number of case studies are presented that are performed with the tool.

9.1 RALT: Rich Automata Learning and Testing

The tool Rich Automata Learning and Testing (RALT) has been developed that implements our approach of learning and testing. RALT is developed under Java 2 Platform Standard Edition 5.0 and has been tested under Windows XP and Linux (Fedora) environments. The current version is 4.0 that consists of 70+ classes and 3 MB of source code. RALT has various modules and libraries for learning and testing black box systems. The architecture of RALT 4.0 with main modules is given in Figure 9.1. We refer to the technical report [NS08] for its complete design with UML diagrams of the classes.

Once the actual system of black box components is connected, RALT is a push button solution that finally outputs the models of the components, traces for compositional problems and generic errors in the system. The working of the modules shown in Figure 9.1 is given as follows.

Test Drivers: Test Drivers are responsible for connecting the actual system with RALT. The main functions include providing the input set of each component in the system to RALT, converting the abstract queries from RALT to actual tests for the system/components, executing the tests on the system/components and converting the concrete test results back to RALT's abstract format. The drivers are specific for each system. RALT provides an *Abstract Test Driver* class which the system specific test drivers inherit to communicate with RALT.

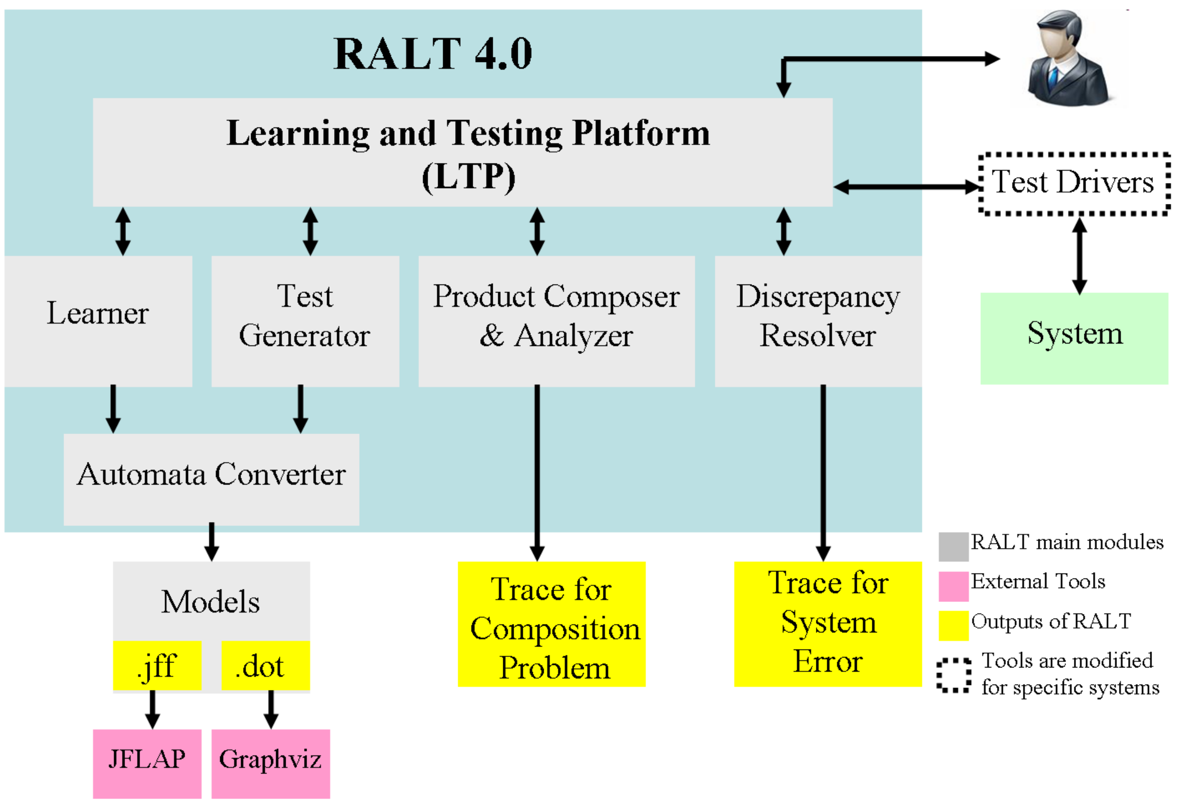


Figure 9.1: Global Architecture of RALT 4.0

Learning and Testing Platform (LTP): This is the main module with which the user interacts and accesses other modules. It contains the Abstract Test Driver and basic methods for the access of internal data structures. All other modules communicate through this module.

Learner: This module performs all learning operations using the learning algorithms library. It asks queries and get responses for the algorithms and finally outputs the conjectures. The different models that can be learned through this module are DFA, Mealy machines and PFSM. The learning algorithms that have been implemented are L^* (Algorithm 1), ${L_M}^*$ (Algorithm 2), ${L_M}^+$ (Algorithm 3) and ${L_P}^*$ (Algorithm 4). The learning algorithms library is shown in Figure 9.2. This module is also responsible for refining the models by receiving counterexamples.

Note: The module is the implementation for Step 1 and Step 3 of our approach (cf. Chapter 6, Section 6.3.1 and Section 6.3.3).

Product Composer and Analyzer: This module has a composer that receives (Mealy machine) models and computes their product. It has an analyzer which analyzes the product for compositional problems. It also confirms the problem on the system through the LTP module. It outputs the trace for the problem if detected or acknowledges if there is no problem.

Note: The module is the implementation for Step 2 of our approach (cf. Chapter 6, Section 6.3.2).

Test Generator: This module receives the product of (Mealy machine) models, generates tests from the product and executes the tests on the system through the LTP module. It uses the test generation methods library for the test generation. Then, it checks the discrepancies between the product and the system. Finally, it outputs a trace for a discrepancy or the product/models if no discrepancy is found. The test generation methods library is shown in Figure 9.2. Currently, it implements the method of test generation from transition coverage. We are enhancing it to incorporate other methods, e.g., W-method [Vas73][Cho78] and Wp-method [FBK$^+$91].

Note: The module is the implementation for Step 4 of our approach (cf. Chapter 6, Section 6.3.4).

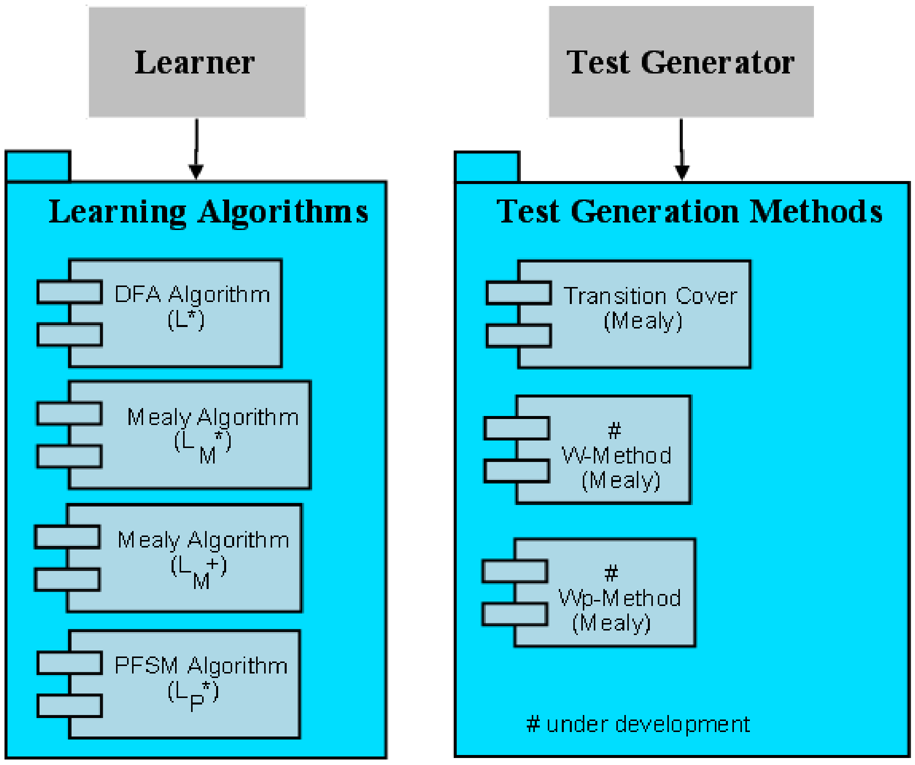


Figure 9.2: Libraries for Learner and Test Generator

Discrepancy Resolver: This module checks whether the discrepancy highlights an error in the system. Currently, it checks for system crash `SC`, uncaught exception `Exp` and out of memory problems `OutMem`. If any of such errors are found, it outputs the discrepancy trace for the error. Otherwise, it returns the discrepancy for the product refinement.

Note: The module is the implementation for Step 5 of our approach (cf. Chapter 6, Section 6.3.5).

Automata Converter: This module converts the conjectured models produced by other modules into the visual formats. Currently, it outputs ".jff" and ".dot" files for the models which can be seen using external tools JFLAP [Rod06] [JFL] and Graphviz [Gra] respectively.

When the test drivers for the system are written and the system is attached with RALT through the drivers, the user interacts with the LTP module to initialize the learning and testing procedure. The LTP module gets the input sets of the components from the test drivers and calls other modules for the specific operations. The modules are run according to the approach (explained in Chapter 6). However, the user can interact with each module separately on her choice. At any point, the user can output the learned models to one of the visual formats to check the intermediate learning step. Debugging option is also provided which outputs the information related to observation tables and test generation during the procedure on the user console. RALT is equipped with number of examples and short applications to help the user to understand how the modules work.

9.2 Case Studies

This section presents the case studies that has been performed with RALT. The number of case studies and their purpose are described below.

Edinburgh Concurrency Workbench: This case study is performed to assess the gain of our proposed algorithm ${L_M}^+$ (Algorithm 3) over the adapted algorithm ${L_M}^*$ (Algorithm 2) for learning Mealy machines. The purpose was to conduct experiments on a workbench of finite state machines so that we could evaluate the average case complexity of the algorithms. The results show that ${L_M}^+$ outperforms in terms of number of output queries when compared to ${L_M}^*$.

Air Gourmet: This case study is performed to evaluate our learning and testing approach explained in Chapter 6. This is an online flight reservation application that has special features for managing passenger meals on-board. The application consists of several components communicating to each other. Our experiment achieved good results and identified many implementation faults in the application.

Nokia 6131: This is our *"star"* case study in which we have discovered the unspecified behaviors of real mobile phones. We have conducted experiments on the Media Player which is a built-in component in the phones. We have experimented on three phone models, i.e., Nokia 6131, Nokia N93 and Sony Ericsson W300i, and identified that Nokia 6131 has different behaviors compared to the other phones. Those behaviors are not specified by its vendor. The purpose of this case study was to apply our reverse engineering technique on the real systems to extract their models (e.g., through the PFSM learning algorithm ${L_P}^*$) and to uncover the hidden behaviors.

Domotics: This is the largest case study we have taken in our work. Domotics is a system for home automation that consists of several consumer appliances. The experiments have been conducted on a prototype of Domotics deployed at Orange Labs. The prototype consists of real devices, i.e., Light, Media Renderer and TV. We have derived Mealy machine models of the devices to discover their behaviors and also learned their interactions in the prototype.

9.2.1 Edinburgh Concurrency Workbench

Introduction

The purpose of this case study is to compare the adaptation of Angluin's algorithm for learning Mealy machines ${L_M}^*$ (Algorithm 2) with our improved algorithm ${L_M}^+$ (Algorithm 3). As discussed in Chapter 5, the improvement on the adapted algorithm is in fact the proposal of a new method of processing counterexamples in the observation table. The worst case theoretical complexity analysis has shown that ${L_M}^+$ outperforms ${L_M}^*$ in terms of number of output queries (see Chapter 5, Section 5.3.5). It is interesting to evaluate the average case complexity of the algorithms when the input sets, the number of states and the length of counterexamples are of arbitrary sizes.

The examples in the case study are the synthetic finite state models of real world systems (e.g., Vending machine, ATM and ABP protocols, Mailing Systems etc) that are shipped with *Edinburgh Concurrency Workbench (CWB)* [MS04]. CWB is a tool for manipulating, analyzing and verifying concurrent systems. The examples in the workbench have also been used to investigate the applicability of Angluin's algorithm in learning reactive systems [BJLS05]. These examples were originally modeled as Non-Deterministic Finite Automata (NFA), with partial transition relations, in which every state is a final state. Therefore, we have transferred first each example to its corresponding DFA. The resulting DFA contains every state as final, plus one non-final (sink) state which loops itself for all inputs. All inputs from a state that are invalid (missing transitions in the original NFA) are directed to the sink state. According to Definition 3, a DFA $\mathcal{D} = (Q_{\mathcal{D}}, \Sigma, F_{\mathcal{D}}, q_{0\mathcal{D}}, \delta_{\mathcal{D}})$ can be seen as an equivalent Mealy machine $\mathcal{M} = (Q_{\mathcal{M}}, I, O, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}}, q_{0\mathcal{M}})$. Here, we learn the Mealy machines models of the CWB examples.

RALT has the implementation of both algorithms ${L_M}^*$ and ${L_M}^+$. However, we had to simulate an oracle in order to obtain counterexamples for CWB examples. This was achieved by using the simple principles of union and integration of finite state machines. The simulation of the oracle is described as follows.

Let $\mathcal{D}$ be an unknown DFA and C be a conjecture DFA then the language $\mathcal{L}(\overline{\mathcal{D}} \cap C)$ accepts those strings which are accepted by C but not by $\mathcal{D}$. The language $\mathcal{L}(\mathcal{D} \cap \overline{C})$ is analogously defined. Thus, we construct a DFA $\mathcal{Z}$ such that

$$\mathcal{L}(\mathcal{Z}) = \mathcal{L}(\overline{\mathcal{D}} \cap C) \cup \mathcal{L}(\mathcal{D} \cap \overline{C})$$

That is, the language $\mathcal{L}(\mathcal{Z})$ accepts strings which are accepted either by C or by $\mathcal{D}$ but not by both. Therefore, any string accepted by $\mathcal{L}(\mathcal{Z})$ is a counterexample for C. If $\mathcal{L}(\mathcal{Z}) = \emptyset$, then $\mathcal{L}(\mathcal{D}) = \mathcal{L}(C)$.

Note that the principle of finding counterexamples is explained for DFAs. In our case, we are learning Mealy machines. This means that every time a Mealy machine conjecture is learned, we have to convert it into its equivalent DFA in order to get counterexamples. Since there are only two outputs, i.e., $O = \{0, 1\}$, the transformation from a Mealy machine to an equivalent DFA is always possible [HU90]. This transformation is also done by the oracle which is explained in the implementation details.

Implementation Details

The action plan for the CWB case study is as follows. First we obtain the NFA of an example in CWB and convert it into its equivalent DFA $\mathcal{D}$. Then, we learn the Mealy machine of the example through RALT. The execution of output queries is performed by the test driver (details of the test driver are given below). Once the Mealy machine conjecture is obtained, the oracle converts the conjecture to its equivalent DFA. Then it calculates a counterexample by performing operations on the converted DFA and $\mathcal{D}$ (details of the oracle are given below). If a counterexample is produced, it is given back to RALT for refining the conjecture. Otherwise, the oracle says "yes" and RALT outputs the Mealy machine conjecture. The procedure is repeated for each algorithm.

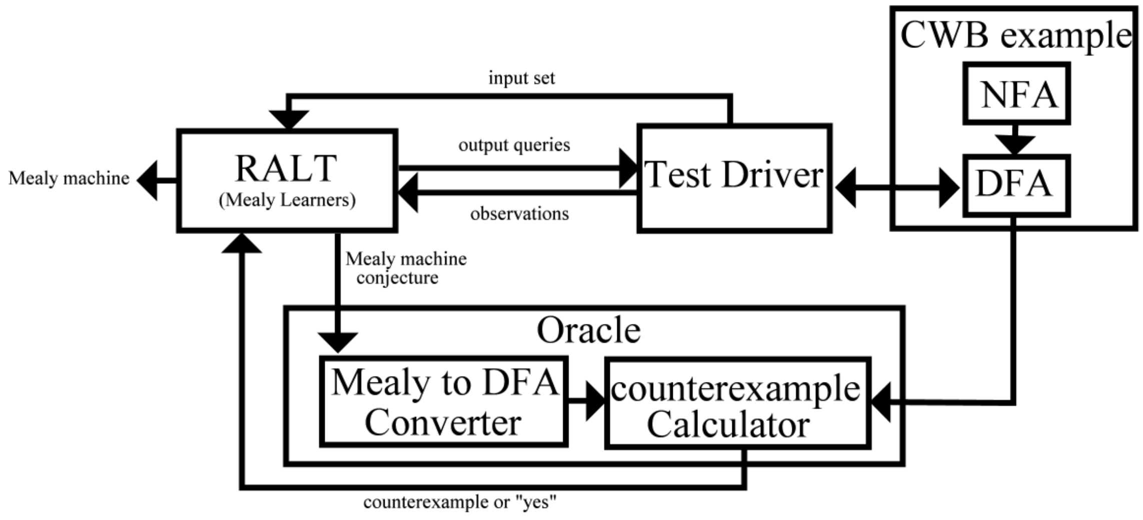


Figure 9.3: Settings for learning CWB examples with RALT

The settings of RALT for learning CWB examples is given in Figure 9.3. In this case study, we have used only the *Learner* module that implements ${L_M}^+$ and ${L_M}^*$ algorithms. The implementation details for the test driver and the oracle are described in the following.

Test Driver: The test driver obtains the DFA example $\mathcal{D}$ to run output queries from RALT. For each query, it starts from the initial state of $\mathcal{D}$ and runs the inputs in the query over the states of $\mathcal{D}$ progressively. For each input, it records the observation as "1" or "0", depending the target state is final or not. Finally, the test driver returns the collective observations to RALT. The pseudo-code of the test driver for one query is given below.

```
begin
    Let $\mathcal{D} = (Q_{\mathcal{D}}, \Sigma, F_{\mathcal{D}}, q_{0\mathcal{D}}, \delta_{\mathcal{D}})$ be the DFA example ;
    Let $\omega = i_1 \ldots i_m$ be a query ;
    Let $obs$ be a boolean array of size $m$ that is initialized with zeros ;
    Let $q = q_{0\mathcal{D}}$ be the initial state of $\mathcal{D}$ ;
    for $j = 1 \ldots m$ do
        $q = \delta_{\mathcal{D}}(q, i_j)$ ;
        if $q \in F_{\mathcal{D}}$ then $obs[j] = 1$;
    end
    return $obs$ ;
end
```

Oracle: The oracle obtains the DFA example $\mathcal{D}$ and the Mealy conjecture M. It then converts M into its equivalent DFA conjecture C. A Mealy machine does not produce output on the initial state. On the contrary, a DFA can accept an empty string if its initial state is final. Therefore, if the initial state of $\mathcal{D}$ is final, then the initial state of C is also made final. Then, it calculates the DFA $\mathcal{Z}$, i.e., the difference between $\mathcal{D}$ and C (as described above). Then, it traverses $\mathcal{Z}$ in the breadth-first-search manner and searches for a final state in $\mathcal{Z}$. If a final state is found, it returns a string from the initial state to the final state to RALT. The string is a counterexample for C, and thus for M. If no final state is found, it returns "yes". The pseudo-code of the oracle is given below.

```
begin
    Let $\mathcal{D}$ be the DFA example and $M$ be the Mealy machine conjecture ;
    Convert $M$ into its equivalent DFA conjecture $C$ ;
    Calculate $\mathcal{Z} = (\overline{\mathcal{D}} \cap C) \cup (\mathcal{D} \cap \overline{C})$ ;
    Starting from the initial state, traverse $\mathcal{Z}$ in the breath-first-search manner ;
    foreach state $q$ in $\mathcal{Z}$ do
        if $q$ is final then                    /* counterexample is found */
            Find a string from the initial state to $q$ ;
            return the string as a counterexample ;
        end
    end
    return "yes"                            /* no counterexample is found */ ;
end
```

If RALT receives a counterexample then it reruns the specific learning algorithm for processing the counterexample. This follows another iteration of making a refined conjecture. The process goes on until RALT receives "yes" for the conjecture. Then, RALT returns the conjecture and terminates the procedure.

Experimental Results

Each example in CWB is learned through RALT by applying both algorithms, i.e., ${L_M}^+$ and ${L_M}^*$, one by one, until their complete models are learned. The number of output queries required by the algorithms are given in Table 9.1. The first column labels the example. The second column shows the size of the input set I. The third column shows the minimum number of states in the example when modeled as DFA and Mealy machines. The fourth and fifth columns show the number of output queries asked by ${L_M}^*$ and ${L_M}^+$, respectively. The last column shows the reduction factor in queries asked by ${L_M}^+$ against ${L_M}^*$, i.e., $\frac{\textit{no. of output queries in } {L_M}^*}{\textit{no. of output queries in } {L_M}^+} - 1$.

Examples	$\lvert I\rvert$	No. of States	No. of Output Queries		Reduction Factor
		DFA / Mealy (min)	${L_M}^*$	${L_M}^+$	
R	3	5	48	48	0
ABP-Lossy	3	11	754	340	1.22
Peterson2	3	11	910	374	1.43
Small	5	11	462	392	0.18
VM	5	11	836	392	1.13
Buff3	3	12	580	259	1.24
Shed2	6	13	824	790	0.04
ABP-Safe	3	19	2336	754	2.1
TMR1	5	19	1396	1728	-0.2
VMnew	4	29	2595	1404	0.85
CSPROT	5	44	4864	3094	0.57

Table 9.1: Comparison of ${L_M}^*$ with ${L_M}^+$ on the examples of CWB workbench. The examples are listed in ascending order with respect to the number of states.

The experiments have been conducted on 11 CWB examples. All examples are of different sizes in terms of number of states and input set size. The results show that ${L_M}^+$ outperformed ${L_M}^*$ in almost all the examples. The greatest reduction factor we achieved is 2.1 on the example *ABP-Safe* which consists of 19 states and 3 inputs. The largest example in the workbench is *CSPROT* that consists of 44 states and 5 inputs. The reduction factor of 0.57 on this example is also significant. There is only one example *TMR1* in which ${L_M}^+$ has negatively performed.

As we know from our worst case complexity analysis (cf. Chapter 5.3, Section 5.3.5) that ${L_M}^+$ performs better than ${L_M}^*$. From the results of these experiments, we can also conclude that the average case complexity of ${L_M}^+$ is less than ${L_M}^*$. We have also conducted experiments on randomly generated DFAs. The result drawn from the CWB case study is also confirmed on those experiments.

9.2.2 Air Gourmet

Introduction

Air Gourmet [Sch01] is a sample Java application for flight reservations that has special features for managing passenger meals on-board. There are two main functions in this application. The first function is to reserve a flight. A passenger reserves a flight and provides her choices for meals that would be served on-board. She can also request quality attributes for meals. The second function is to provide a procedure for welcoming a passenger on-board and managing the meals which she has requested while reserving the flight. This is an open source application and consists of several components. We describe here the main components which took part in our case study.

Flight Reservation: The component registers a passenger for a specific flight and manages the functions related to meal choices.

Passenger Check-in: The component provides a procedure for checking-in a passenger.

Meals Manager: The component manages all functions related to the meals on-board and provides a procedure to select the meal for the specific passenger. It also manages the quality attributes the passenger might have requested during the flight reservation.

Each component can be accessed in the system separately to use its specific functions. However, the components also communicate with each other because of their dependencies. For example, Passenger Check-in uses Flight Reservation to obtain flight records and passenger data for checking the passenger. Similarly, Meals Manager uses Flight Reservation for checking the meals options which the passenger has requested. Figure 9.4 provides a conceptual view of the Air Gourmet application. It depicts how the components are communicating to each other and to the environment.

Implementation Details

We considered each component as a Mealy machine and applied our approach of learning and testing (explained in Chapter 6) to the Air Gourmet system. In order to conduct experiments, the system was connected to RALT with the help of test drivers. Since the source code was available, writing the test driver was an easy task. There were four test drivers in total, i.e., one for each component and one for the global system. The test driver for a single component was required for learning the component separately. The test driver for the global system was used to communicate with the integrated system as a whole. The communication of the components

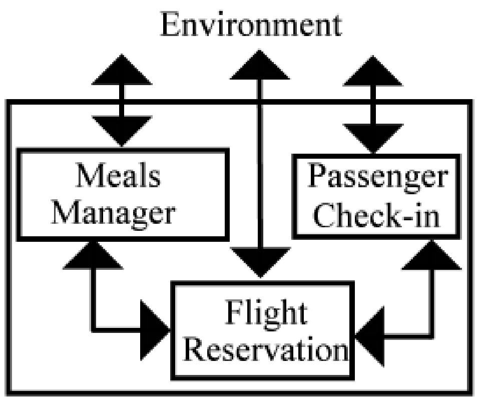


Figure 9.4: Conceptual view of the Air Gourmet System

were held by method calls. So, the implementation of the test drivers was a simple translation of abstract inputs from RALT to their corresponding method calls. Similarly, the test drivers translated each output, i.e., the return value of the called method, into an abstract symbol and sent back to RALT.

In order to initialize learning algorithms, we constructed input sets for each component. Apart from the normal inputs that each component accepts, we also added invalid inputs in the set to infer the behavior of the system on such inputs. Then, we implemented the notification of the invalid inputs in the test drivers. That is, the test drivers raised an `InvalidInputException` if the components returned a valid result on the invalid input.

Experimental Results

We followed the procedure of the learning and testing approach by first learning the components in isolation through their respective test drivers (Step 1). RALT learned each component and derived its Mealy machine model. The models obtained after learning were partial; except for the Passenger Check-in component that was learned completely.

Then, RALT composed the learned model to make a product analyzed for detecting livelocks (Step 2). There were no livelocks detected in the product. Hence, the refinement step (Step 3) was skipped and it proceeded for the test generation step.

RALT generated tests from the product and ran the tests on the integrated system through the system test driver (Step 4). The tests discovered several discrepancies between the product and the behavior of the system.

Then, RALT resolved the discrepancies if they were classified as real errors in the system (Step 5). Sometimes, a discrepancy revealed a real error in the system. In this case, RALT terminated the procedure by reporting the error. Otherwise, it refined the product by re-learning components using discrepancy as a counterexample (Step 3) and followed the iterative procedure.

Whenever a real error was found, we corrected the error in the implementation and reran RALT for the corrected system. This procedure went on until no further discrepancies were found and no further errors were detected.

The whole procedure was terminated in six iterations (including the rerunning of RALT after corrections in the implementation). Finally, we achieved the learned model of each component. As noted before, the model for Passenger Check-in was learned completely after Step 1. Moreover, the model for Flight Reservation was also learned completely after the third iteration. However, the model for Meals Manager was not learned completely. We spotted in the code of the component that few functions could not be learned by RALT. As far errors are concerned, we found several errors in the system, especially related to uncaught exceptions, e.g., Java `NullPointerException`. We also noted that the system accepts invalid inputs. For example, Flight Reservation accepted the reservation date as *14 July 1789* [1] in our experiments.

The results of the case study are provided in Table 9.2. The first column shows the component, the second column shows the number of inputs (including the invalid inputs), the third column shows the number of states in the learned model, the fourth column shows the number of errors that were detected in the component and the fifth column shows the type of errors.

Components	$\|I\|$	No. of States	No. of Errors	Error Types
Passenger Check-in	3	4	0	-
Flight Reservation	5	7	3	NPE, IIE, Date Parsing Exception
Meals Manager	4	8	4	NPE, IIE, IAE

NPE: `NullPointerException`, IIE: `InvalidInputException`, IAE: `IllegalArgumentException`

Table 9.2: Results of the Air Gourmet System.

As a whole, our experiments achieved good results on the case study in terms of learning models and identifying implementation faults in the application. The case study is small, but it enhances confidence that the approach is workable for behavioral exploration and for uncovering implementation errors in the systems. We intend to extend our experiments on large scale systems to assess the approach. Domotics is an example of such a system.

[1] révolution française

9.2.3 Nokia 6131

Introduction

Today's mobile phones provide a heterogeneous platform for adding new components that provide custom services to their users. The new components usually access the built-in mobile components to activate specific functions. For example, a game component accesses the internal media player to enable sound on phone's speakers. These components benefit from Mobile APIs that are provided by phone manufacturers, as it allows to use basic phone functions and to adapt, e.g., to change in quality of service, connectivity, and to integrate new functions. The general working of the built-in components and the design of APIs must adhere to some defined specifications, e.g., Java Specifications Requests (JSR) [Pro] for Java enabled mobile phones; so that, the same components can be added across the mobile platforms. However, it is observed that the built-in components contain far more behaviors than the specifications. This is because the specifications only describe the high level functions of the components and miss the details how the functions should be implemented. For example, the Mobile Media API (MMAPI) specifies how the basic media playback function can be accessed, but does not describe how actually the media processing happens and how the media player reacts with different playback settings. These implementation details depend upon the specific hardware and operating systems the phone manufacturers use. As a result, the implementation of the built-in components usually vary from one mobile platform to another.

Mobile Media Player

We have studied **Nokia 6131** (S40 series) and discovered the unspecified behaviors of its built-in components. Specifically, we studied the media player component how it behaves when other components interact to use its playback functions. The life cycle of the media player according to the standard multimedia specifications, i.e., JSR-135[1], is shown in Figure 9.5.

It consists of five states, namely, UNREALIZED, REALIZED, PREFETCHED, STARTED and CLOSED. Initially, the player is in the UNREALIZED state. Calling realize() moves it to the REALIZED state and initializes the information the player needs to acquire media resources. Calling prefetch() moves it to the PREFETCHED state, establishes network connections for streaming data, and performs other initialization tasks. Calling deallocate() returns the player to the REALIZED state, thereby releasing all the resources that it would have acquired. Calling start() causes a transition from the PREFETCHED state to the STARTED state, where the player can process data. When it finishes processing (reaches the end of a media stream) or

[1]http://java.sun.com/javame/reference/apis/jsr135

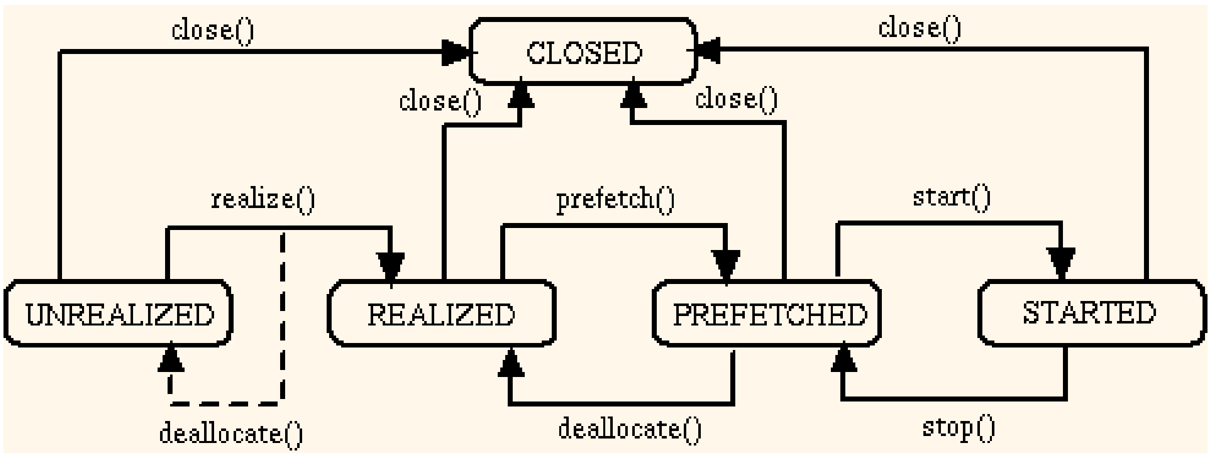


Figure 9.5: Mobile Media Player Life Cycle

when stop() is called, it returns to the PREFETCHED state. Finally, calling close() moves the player to the CLOSED state.

Inference Objectives

Figure 9.5 demonstrates the life-cycle for playing one media file on the phone. However, it is unspecified how the media player behaves and the changeover on the states takes place when more than one files are played simultaneously. Our objective was to learn the behaviors of the media player under this context for Nokia 6131. We learned the behaviors by connecting the mobile phone with RALT, and applied the algorithm $L_P{}^*$ (Algorithm 4) for learning a PFSM model of the media player. The first step in learning is to construct the input set for the component. Since, we were interested in checking the behaviors on two media files, we created two instances of the media player, namely $p1$ and $p2$. In order to play the files, we had to call realize(), prefetch() and start() methods in a particular sequence. However, the specifications say that if start() is called when the player is in the UNREALIZED or REALIZED state, it implicitly calls prefetch(). Thus, we used directly start() method for $p1$ and $p2$ instead of going through intermediate steps. The other two methods we considered are stop() and close(). Thus, the input set for learning the behaviors of the media player for two media files was $I = \{p1.start(), p2.start(), p2.stop(), p2.stop(), p1.close(), p2.close()\}$. The parameter values for $p1.start()$ and $p2.start()$ were the media files, $p1.wav$ and $p2.wav$ respectively, which we played on the player. The other inputs do not take parameter values; so the domain of parameter values was $D_I = \{p1.wav, p2.wav\}$.

Settings	Details
Java Micro Edition	Sun Java Wireless Toolkit 2.5.1
Connected Limited Device Configurations (CLDC)	version 1.0
Mobile Information Device Profile (MIDP)	version 2.0
Mobile Media API (MMAPI) specifications	JSR-135
Package	javax.microedition.media

Table 9.3: Settings for Media Player Test Driver

Details for the Test Driver

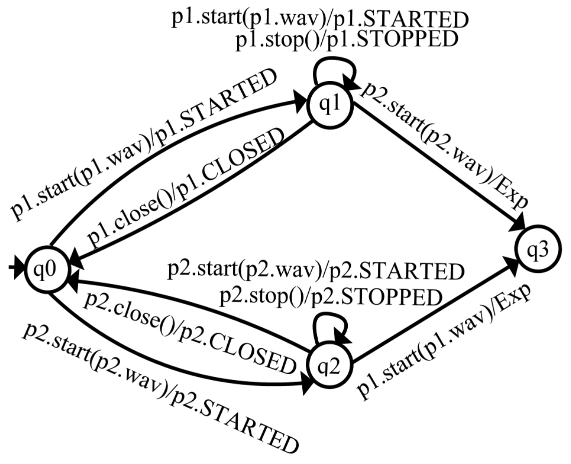
The test driver for connecting RALT with the mobile's media player was developed under Java Micro Edition framework. The detail settings for the media player test driver is given in Table 9.3.

The behaviors of the media player can be observed by listening to the player events. Player events deliver the information about the player's state changes and other relevant information from the player's controls. For example, when the start() method is called and the media file is started playing, the event STARTED is delivered. Similarly, the events STOPPED and CLOSED are delivered when the media file is stopped or closed respectively. The MMAPI provides a listener through which the events can be captured. The test driver kept track of events after calling each input method and recorded the corresponding observations through the listener. It recorded the events STARTED, STOPPED and CLOSED, as well as any exception *Exp* raised by the player during the testing procedure. The events are guaranteed to be delivered in the order that the methods representing the events are called. That means the events STARTED, STOPPED and CLOSED are delivered in the same order as the methods start(), stop() and close() are called. Therefore, the test driver always got deterministic behaviors of the player. Summarizing the observation recording, the output set of the player was constructed as $\{p1.STARTED, p2.STARTED, p1.STOPPED, p2.STOPPED, p1.CLOSED, p2.CLOSED, Exp\}$. The outputs do not contain parameters; so, the domain of output parameter values was $D_O = \emptyset$.

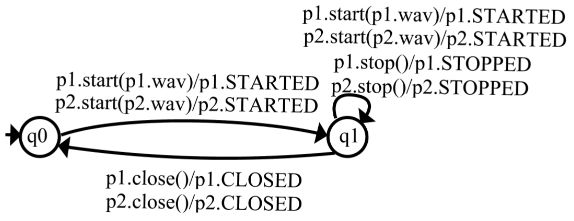
Inference Results

RALT ran the algorithm $L_P{}^*$ to learn a PFSM model of the media player of Nokia 6131 on the given inputs and parameter values. The learned model is shown in Figure 9.6a. The transitions with invalid method invocations are not shown in the model.

The model shows that when the player is in the initial state q_0, calling either $p1.start()$ or $p2.start()$ results in playing the given media files. However, if one of the files are being played



(a) PFSM model for the Media Player of Nokia 6131


p1.start(p1.wav)/p1.STARTED
p2.start(p2.wav)/p2.STARTED
p1.stop()/p1.STOPPED
p2.stop()/p2.STOPPED
p1.start(p1.wav)/p1.STARTED
p2.start(p2.wav)/p2.STARTED
q0
q1
p1.close()/p1.CLOSED
p2.close()/p2.CLOSED

(b) PFSM model for the Media Player of Nokia N93 and Sony Ericsson W300i

Figure 9.6: PFSM model for the Media Player of Nokia N93 and Sony Ericsson W300i

or the file is stopped, calling the methods to play the other media file raised the exception. This takes the player in the state q_3 from where no input is accepted. However, if the player is on state $q1$ or $q2$ and the respective files are closed, then this takes the player to the initial state where any of the files can be played. Thus, the model depicts that the two media files cannot be played at the same time. Even if the files are stopped, the player throws the exception if the other file has already been started. Therefore, one file has to be closed before starting the other.

We have performed the same experiments on **Nokia N93** (S60 series) and **Sony Ericsson W300i** mobile phones. Both have shown different behaviors from Nokia 6131. Their model is shown in Figure 9.6b, which depicts that the two files can be played simultaneously on these phones.

Application of the Case Study

The case study has successfully applied our reverse engineering technique on real mobile phones and uncovered hidden behaviors. It points out to a real benefit of the technique in the context of industry where understanding the system of third-party components is an issue. The designers manually conduct tests on the components using scattered pieces of incomplete information, combining with the knowledge from domain expertise, to uncover the unspecified behaviors of the system.

The case study is acknowledged by Orange Labs that has faced similar problems of unspecified mobile phone components in the past. We have given the example of *Docomo* in Chapter 1 (Section 1.3), which explained the problem of gaming components due to its unknown behaviors. The Nokia 6131 case study can be seen as a proof-of-concept for the Docomo example in the sense that the same kind of experiments can be performed to learn the model of the gaming component to reveal its hidden behaviors.

From this case study, Orange Labs has gain a benefit to understand Nokia 6131's media player behaviors. A problem might occur if a phone user starts playing a media file and then switches to third-party, e.g., a game component, which also plays its own file, then the two files cannot be played at the same time. Moreover, the exception raised in this scenario could halt the phone if it is not handled properly in the game component.

9.2.4 Domotics

Introduction

Domotics or Home Automation is a system for building automation, specializing in the specific automation requirements of private homes and in the application of automation techniques for the comfort and security of its residents. Although many techniques used in building automation (such as light and climate control, control of doors and window shutters, security and surveillance systems, etc.) are also used in home automation, additional functions in home automation include the control of multimedia home entertainment systems, automatic plant watering and pet feeding, and automatic scenes for dinners and parties.

ArchiteCture for Smart Environment (ACSE) [Gr08] is a running project in Orange Labs aiming to provide a smart home service-oriented-architecture and related services. We took a case study from this project to experiment with real devices that provide services under home automation context. The number of devices we have considered are:

Light Control System: involves aspects related to controlling electric lights that include

- Extinguished general of all the lights of the house
- Automatization of switched off / ignition in every point of light
- Regulation of the illumination according to the level of ambient luminosity.

Media Renderer: involves in rendering Audio/Video contents from the home network. It exposes a set of rendering controls for various features such as brightness, contrast, volume, etc.

TV: involves in playing the audio or video contents from the Media Renderer. It also displays the messages and events notified by other devices.

Prototype at Orange Labs

Orange Labs has deployed a prototype of Domotics that consists of abovementioned devices. There are several technologies for device installation that allows devices to connect seamlessly in home and corporate environments for the purpose of data sharing, communications, and entertainment etc. The two well-known technologies are

UPnP: The UPnP architecture [MNTW01] allows peer-to-peer networking of PCs, consumer appliances, and wireless devices. It is a distributed, open architecture based on established standards. The UPnP devices are compliant with the specifications provided by UPnP

Device	Vendor	Product Name	Technology
Light System	ProSyst	-	X10
Media Renderer	Philips	Streamium400i	UPnP
TV	Acer	AT3705-MGW	UPnP (compliant)

Table 9.4: The details of the devices in the Domotics prototype

Forum [UPn]. The specifications detail the input actions on which the devices operate; as well as, the methods to access their functions and to observe their status. However, the implementation may vary from vendor to vendor which is usually not exposed.

X10: X10 [Pac05] is an international and open industry standard for communication among electronic devices used for home automation. It primarily uses power line wiring for signaling and control, where the signals involve brief radio frequency bursts representing digital information.

The details of the device in our prototype are given in Table 9.4.

Implementation Details

OSGi [All] is a standardized Java-based lightweight service framework, which offers an access to a network of UPnP devices and services. *Apache Felix* [Apa] is a platform that provides an implementation for the OSGi framework and supporting the UPnP technology. The X10 devices can be accessed over the network through X10 protocol device drivers.

So, we have a network of UPnP and X10 devices that are accessible from Felix through OSGi framework. Felix contains *bundles*(jar files) to execute control on the devices. In order to connect the network with RALT, we have written test drivers for the devices which are actually the bundles deployed in Felix. Those bundles communicate with the devices and perform operations like running queries and sending back the results to RALT. We have also implemented a system test driver for Domotics that manages bundles in Felix. Figure 9.7 is a conceptual view of the settings of the Domotics system with RALT.

Experimental Results

We have conducted experiments on the prototype of Domotics that consists of devices described in Table 9.4. As a first step, we learned the Light System LS and Media Renderer MR separately and derived their Mealy machine models. Thereafter, we learned the interactions of all the devices and derived the Mealy machine model of the whole system. We explain the learning of LS and MR in the following, and then provide details of the whole system.

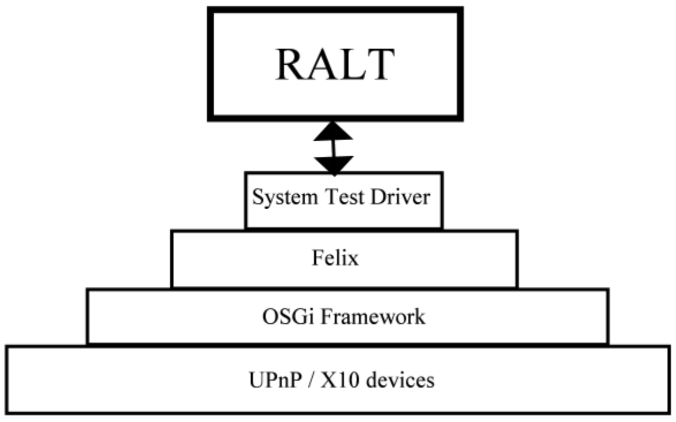


Figure 9.7: Conceptual view of the settings of the Domotics system with RALT

LS has four basic functions, i.e., ON, OFF, $BRIGHT$ and DIM. For the inputs $BRIGHT$ and DIM, it changes the lightning intensity according to predefined levels. In the device configuration, we used levels from 1 to 4, where 1 is the dimmest and 4 is the brightest intensity level. When the test driver receives the inputs $BRIGHT$ or DIM from RALT, it sends the inputs to the device and provides back the current intensity level.

We learned the model of the device using the input set described above. It took about four minutes and a five state Mealy machine model was derived as shown in Figure B.1 (Appendix B). It is observed that the LS uses the brightest intensity level when it is lit up. It decreases one level when it receives DIM and increases one level when it receives $BRIGHT$.

MR has five basic controls, i.e., ON, OFF, $PLAY$, $PAUSE$, $STOP$. We set a media file in the device configuration, so that it could perform specific actions on the file when it is given the specific inputs. For example, it could play the file on receiving the input $PLAY$.

We learned the model of the devices using the input set described above. It took about sixteen minutes and a four state Mealy machine model was derived as shown in Figure B.2 (Appendix B).

We have also prepared an experimental setup so that the devices could interact each other during their execution. The setup of the system is described as follows. The devices LS and MR can receive inputs from the environment and operate accordingly. Moreover, when LR changes its status, then TV displays the corresponding notification on its screen. We have written a kind of a test driver, called TV_Ctrl, which listens to the interactions of the devices and notifies to the environment. TV_Ctrl has the following responsibilities:

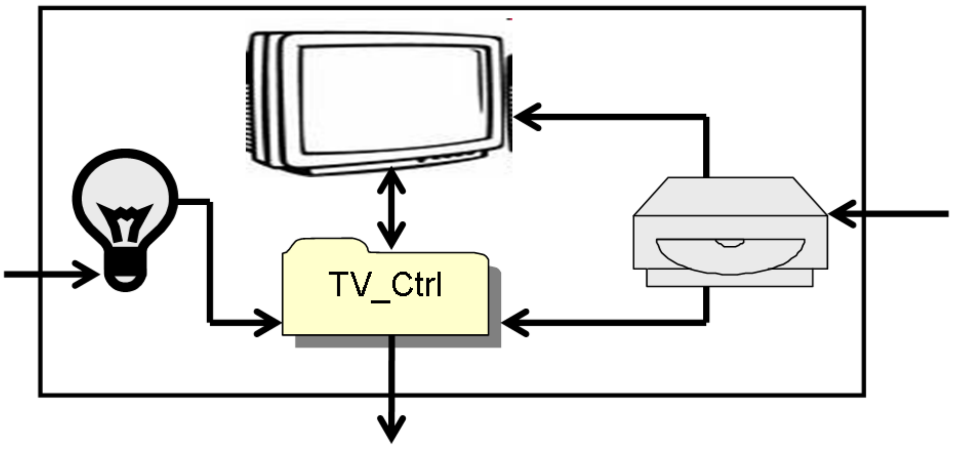


Figure 9.8: Setup of the Domotics System

- It receives a status change notification from LS and displays the notification on the TV screen. It also sends the notification as an external output.
- It listens to an action performed by MR and notifies the action by sending an external output.

The setup of the Domotics system is given in Figure 9.8. The interaction model of the system is learned in about thirty minutes. The Mealy machine model comprises of 141 states is given in Figure B.3 (Appendix B).

Our initial experiments on Domotics learned simple and custom made device interactions. However, we have learned the models of the real devices that are structurally complex and require great deal of attention for constructing their input sets. We have explained this case study very succinctly in this section and ignored the details of the device configurations and implementation of the test drivers. The complete illustration and experimental results of this case study can be seen in the technical report [NS08]. The work on this case study has been started since August 2008. Orange Labs is extending this prototype with more complicated scenarios, which follows the further expansion of our experimental results.

9.3 Conclusion

This chapter presented the implementation work carried out in the thesis. The details of the tool RALT 4.0 are given which we have developed to implement our approach of learning and testing. We have experimented with four case studies and presented their results in the chapter.

The first case study was *Edinburgh Concurrency Workbench* that aimed to evaluate the practical results of our proposed algorithm ${L_M}^+$ compared to the adapted algorithm ${L_M}^*$ for learning Mealy machines. The results showed that ${L_M}^+$ has outperformed ${L_M}^*$.

The second case study was *Air Gourmet* that aimed to evaluate our approach of learning and testing of integrated systems. The experiments achieved good results in terms of learning models of the components and uncovering implementation faults.

The third case study was Nokia 6131 that aimed to evaluate our reverse engineering techniques on the real systems. We conducted experiments on three mobile phones (Nokia 6131, Nokia N93, Sony Ericsson W300i) and learned the models of their media player component through the PFSM algorithm ${L_P}^*$. The model revealed unspecified behaviors of the component in Nokia 6131.

The fourth case study is Domotics that aimed to evaluate the application of our approach of learning structurally complex real systems. Domotics is a home automation system that consists of various communicating UPnP/X10 devices. We have performed initial experiments on the system of Light, Media Renderer and TV and learned their interactions. We intend to further extend this case study with more complex device interactions.

The case studies have produced encouraging results for the application of our approach of learning and testing. We have conducted experiments on a variety of systems that include small examples, like a workbench of synthetic finite state machines (CWB) and an open-source java application (Air Gourmet), as well as comparatively larger and difficult examples, like mobile phones and Domotics. Moreover, we have applied our approach on real black box systems and uncovered their unknown behaviors. Nokia 6131 is an example of such claim. This gives us a confidence that our approach of learning and testing, in general, is applicable in real world systems. Domotics is an example of such a challenging system because of its heterogeneous environment. We believe that RALT is capable of dealing with complex systems, however extensions are needed to cater with domain specific requirements and for writing test drivers for real systems.

Chapter 10

Conclusion and Perspectives

This chapter concludes the work of the thesis. It provides the summary of the work detailed in the previous chapters. Then, a short note on the publications achieved from this work is given, the future extensions and on going work are pointed out, followed by some lessons that are learned from this research.

10.1 Summary of the Thesis

Component Based Software Engineering has gained a strong momentum in many sectors of the software industry. The main reason of its prevalence is that it reduces the cost of developing complex systems by reusing Components-Of-The-Shelf (COTS) or third-party components, instead of developing the systems from scratch. However, delivering a quality of service in COTS is a challenge. The system designers require specifications or models of the components to understand their possible behaviors in the system. However, the specifications are often absent or insufficient for their formal analysis. Such components are termed as black boxes in literature.

The thesis addressed the problem of uncovering the behaviors of black box components to support the testing and the analysis of the integrated system composed of such components. We surveyed the state of the art in this domain and found that the combination of automata learning and testing techniques are useful in the behavioral exploration of the components and for the application of Model Driven Engineering (MDE) techniques. We realized that the passive learning techniques are less effective than the active learning techniques. In this vein, Angluin's algorithm has been considered a remarkable work for active learning finite state machines in polynomial time. Despite some practical problems (the equivalence check by assuming an oracle), the algorithm has been applied in many works of learning and testing in the last decade.

We used the building blocks of Angluin's algorithm and proposed a framework for learning enhanced state models to support integration testing. The contributions in the thesis can be summarized as follows.

- We studied the adaptation of Angluin's algorithm for learning Mealy machines and observed a room for improvements in the algorithm. We have proposed a new method of processing counterexamples in the algorithm such that the number of queries in the algorithm is significantly reduced when applying our method. The gain over queries is also confirmed by experimenting the number of examples in the concurrency workbench CWB [MS04]. (cf. Chapter 5)

- We have proposed a framework to test and analyze the integrated system of black box Mealy components. The framework consists in learning the components in isolation and then using the learned models for finding compositional problems and other generic errors, such as system crash, uncaught exceptions, out of memory etc, in the system. The equivalence check in the algorithm is replaced with a model based testing technique that comprises of deriving tests from the learned model and stimulate the interactions between the components such that they visit the parts of components which might have been unexplored previously. This can benefit in refining the models whenever discrepancies are found between the behaviors of the learned models and the actual components, and also in testing the integrated system for potential errors. (cf. Chapter 6)

- The learning of enhanced models has been advocated in many recent works in which real systems were considered. We have proposed a model, Parameterized Finite State Machine (PFSM), and an algorithm to learn PFSM models using the original settings of Angluin's algorithm. (cf. Chapter 7)

- We have extended the work of inferring PFSM models for learning parameter functions. The proposal was to use data invariant inference tools to learn functions as invariants over the observed parameter values. We have used Daikon and studied simple examples. (cf. Chapter 8)

- We have validated our approach of learning and testing of enhanced state of models on a number of case studies provided by France Telecom R&D. These applications lie in different telecom domains, namely cellular phones, web services and domotics. The experiments were carried out with RALT 4.0 (Rich Automata Learning and Testing), the tool developed during the thesis that implements our approach of learning and testing. The experimental results encourage to apply our approach in more sophisticated systems of black box components. RALT is now being deployed on the company's site where its designers shall use it for real world applications. (cf. Chapter 9)

10.2 Note on Publications

The number of publications achieved during the course of thesis (duration Nov. 2005 - Oct. 2008) is listed here in chronological order. A short introduction to each publication is given below.

1. **TAIC PART 2006 *(Doctoral Symposium)*:**

   This paper [Sha06] states the problem globally and provides a sketch of the approach presented in the thesis.

2. **TAIC PART 2006:**

   This paper [LGS06a] presents the algorithm ${L_M}^*$ (Algorithm 2) for learning Mealy machines. It also proposes an integration testing technique that uses Mealy models and detects incompatibilities between black box components. The approach was applied on a small example of hotel reservation system consists of two components.

3. **FORTE 2006:**

   This paper [LGS06b] presents a parameterized model and an algorithm for learning such model. The model is a simple extension of Mealy machines with parameters. This was an intermediate step in proposing enhanced models, and therefore, not mentioned in the thesis.

4. **TestCom 2007:**

   This paper [SLG07a] presents the PFSM model (Definition 4) and the algorithm ${L_P}^*$ (Algorithm 4) for learning PFSM models.

5. **AFADL 2007:**

   This paper [GSL07] (en français) presents our works in learning enhanced models. It includes the learning of DFA, Mealy machines and PFSM models.

6. **COMPSAC 2007:**

   This paper [SLG07b] covers the background of learning PFSM models (Testcom 2007) and proposes an integration testing framework that uses PFSM models and refines the models iteratively.

7. **ICFI 2007:**

   This paper [SPK07] presents an approach of detecting feature interactions in black box systems using model inference approach. A mobile phone system that integrates third-party components was studied and unknown interactions between the components were detected through the approach. This work was carried out to study an application of our learning and testing approach for detecting feature interactions. The work is not presented in the thesis.

8. **ISoLA 2007:**

   This paper [SG07] presents the approach of combining state machine inference with data invariant inference mechanism. It details the problem of learning output parameter functions in PFSM models, then provides the solution of using data invariant inference and the experimental results with Daikon.

9. **TestCom 2008:**

   This paper [GLPS08] presents an approach of detecting sporadic errors in a black box integrated system. Such errors occur due to several interleavings in response to a given input sequence, if that sequence is applied several times. This happens especially when the components in the system have multiple outputs that can invoke many other components. The paper proposes to combine inference, testing and reachability analysis to detect such errors in the systems.

   The approach presented in the paper is similar to the iterative approach of learning and testing presented in Chapter 6. That is, we learn the component models and test the system using the learned models in an iterative fashion. However, the key points of the approach presented in the paper which create differences with the approach presented in the thesis are

   - Instead of learning components one by one, we take the integrated system as a whole and learn its model.
   - We use a different model for learning systems, i.e., *Input Output Label Transitions System (IOTS)* with multiple outputs.
   - We learn the model of a system by inferring its *k-quotient*. An algorithm for this purpose is presented in the paper.
   - We apply reachability analysis on the learned model to detect errors such as

- unspecified receptions, i.e, one component sends an (internal) input but no other component can consume it.
- compositional livelocks, i.e, there is a loop of internal inputs in the system.
- races, i.e, given an input sequence, there could be several interleavings which could cause nondeterminism in the system's behavior.

10. **TSI 2008** ***(Journal Paper)*****:**

    This paper [GSL08] (en français) is a journal version of the AFADL 2007 paper, with the extensions of the experimental results on CWB workbench [MS04] and the complexity discussions covered in the thesis.

11. **IS 2008** ***(Journal Paper) (submitted)*****:**

    This journal paper [SLG08] covers the learning of enhanced finite state machines (DFA, Mealy and PFSM algorithms), the approach of learning parameter functions and the experimental results of the case study CWB [MS04].

10.3 Future Directions

There are many directions we foresee to extend this work. We have already started working on some of them, and others are still in our wish list. A brief overview of these directions is given below.

10.3.1 Learning Variable Approximations

The previous studies in the automata learning have concluded that it is unrealistic to assume that components could be modeled with a perfect abstraction in a finite, compact representation. However, inferring approximated models of components in a given modular system appears to be more realistic [HHNS02].

Our learning methods derive approximate models, precisely quotients, of the unknown models. It is interesting to note that the computational complexity and precision in the algorithms can be controlled by learning *k-quotients* (Definition 8). Here, the variable k bounds the exploration of the system such that the states of the quotient are *k-equivalent* (Definition 7). Thus, the complexity of learning and the precision of quotients can be controlled by k.

Our recent work [GLPS08] is a step forward towards learning variable approximations. Currently, the experiments are underway in which we are using *IF platform* [Boz04] for expressing automata systems. We have implemented the algorithm for inferring k-quotients. Now, we start working on case-studies for the practical analysis of the approach.

10.3.2 Learning Nondeterministic Machines

We are looking into the problem of learning nondeterministic finite state machines [Yok94]. The difficulty in learning such machines is that they produce multiple outputs on the same input sequences when applying several times. One possibility to resolve this problem is to make a *complete testing assumption*, i.e., by applying a given input sequence ω a finite number of times to a given non-deterministic black box machine, we exercise all possible transitions of the machine that can be traversed by ω [FvB92]. This assumption is similar to the one of so-called "all-weather conditions" for nondeterministic systems of Milner [Mil82]. Then, we can use the settings of Angluin's algorithm and record multiple outputs in the cells of the observation table. This follows the splitting of rows containing multiple outputs in the cells, similar to the PFSM algorithm $L_P{}^*$ (Algorithm 4). Finally, the transitions of the conjecture can be labeled with nondeterministic choices of outputs recorded in the observation table.

10.3.3 Test Generation Methods for Model Refinements

It has been observed that the equivalence check in Angluin's algorithm can be replaced by the application of model based testing techniques [HHNS02] [HNS03]. The essence is to validate the model if it is equivalent to the target model, as well as analyzing the system in more depth in our case. We believe that model based techniques, like the one proposed in the thesis, can exercise the system with respect to revealing maximum interactions of the components. We require a good experience on deriving tests from the partially learned models such that the stimulation of interactions could detect discrepancies and may lead to the refinement model step to start another iteration. A research on devising more efficient technique or proposing the combination of test generation strategies is included in our work items. This follows the extension of RALT with the implementation of new testing techniques.

10.3.4 Integration Framework for PFSM Components

We have proposed a framework for analyzing an integrated system of Mealy components. The framework can be easily adapted for the system of PFSM components. However, it is necessary to evaluate the test generation techniques for PFSM components. So far, our results are preliminary and selection of parameter values in testing is an issue. By considering hypotheses such as *uniformity* and *regularity* [BGM91] [Pha94], we believe that the issue could be resolved.

10.3.5 Testing Security Violations

System security testing includes testing the system to make sure that it behaves correctly in the presence of malicious attacks [Mcg06]. Typical outcomes of these attacks lead to system crash or confidentiality violations. We observe an application of our approach for testing the system's security and reliability. There are two areas in this context which we have started investigating.

1. The models of the system can be extracted with respect to detecting security violations. This can be done by designing the input set with invalid inputs and initialize the algorithm with such inputs. Then the model contains the transitions which are labeled with the invalid inputs and their corresponding outputs depict the response of the system on those inputs.

2. The learned models of the system can be used in fuzz testing, which is an effective approach to uncover security flaws in the system. Fuzz testing consists in finding implementation errors using malformed or semi-malformed data injection in an automated fashion [OWS]. It is usually conducted in an ad-hoc manner with input selected either randomly or manually. However, generating fuzz tests from the learned models has proven to be more effective than the traditional technique [SHL08]. The idea is to derive a prefix sequence from the model which takes the system to a certain state and then replaces the suffix with a fuzz function that will generate malformed inputs. We intend to collaborate with Orange Labs to apply this technique on testing *WiFi 802.11* drivers.

10.3.6 Experiments with Complex Systems

The experiments with case studies have produced encouraging results and motivated us to apply our approach in more complex systems. Most of the applications studied in the thesis lie in the domain of France Telecom. Our dream application is *Orange LiveBox* that is an ADSL wireless router and provides telecom and web-based services. As far specifications are concerned, Orange LiveBox is regarded as a black box system that was purchased by France Telecom with little and informal information of its underlying features from its providers, i.e., *Sagem* and *Inventel*. In order to uncover its features into formal means and to test the integration with services provided by France Telecom, we require meticulous details for designing the input sets and choosing the testing techniques in the integration framework.

10.4 Lessons Learned

Here are some lessons taught by this research that may be helpful to others in their work. Many of the lessons are not new – their successful application has been reflected in other tools and results – even if they have not always been explicitly stated.

Exact Learning is difficult but not required in most cases: It has been pointed out many times in the thesis that exact learning is not possible without having very strong assumptions on the hidden model. At the same time, it is realized that exact learning is not required in most cases. Our experiments justify that a reasonable approximation gives a satisfactory knowledge about the internal working of the components. Moreover, the interactions of components may not concern in learning complete models as they use a part of each other functionalities. It is also noted that most severe errors show up on short sequences, thus the learned models are quite useful in detecting those errors [HHNS02]. Mixing the plausible results of inference with human insights, we conclude that model learning and testing is indeed feasible in practice. Of course, we must look for other means of learning closer approximations such that their complexity does not hinder the practicality of the approach.

Designing input sets requires domain knowledge: A tricky point in our approach is to design an input set of a component to initialize the algorithms. The design of the input set of a real system is not easy and often requires domain expertise for the correct identification of the possible inputs. In the case of a parameterized system, designers have to choose respective parameters which could lead the system to the desirable state. Also considering the complexity of the algorithm on the number of inputs, it is recommended to consider only interesting inputs. Dynamic discovery of inputs in the integration framework is also not so simple. Because we learn components in isolation; sometimes, it requires great deal of attention to determine that the output produced by one component is indeed an input to the other component, so that the input can be added in the set of the other component for its learning.

Matching abstract with concrete data is not trivial: A real system has to be connected with an abstract environment to perform experiments. The system works on many implementation details, different timing notions, tags, identifiers and other data fields. One has to abstract from all these details to deal with the formal world of algorithms. Similarly, the abstract queries have to be concretized for their execution on real systems. The collection of the system responses in the result of queries is usually handled by observing the displays of the system. The meaning of "display" is quite versatile. Ideally, there should be provided inspectors, i.e., APIs that return the internal state information or outputs of the system without making any change. The system might also be equipped with a screen that displays the information or corresponding outputs of the system. There are other means, such as LEDs or speakers, that might be enabled or disabled on a particular query. They also provide valuable observation points and it is necessary to abstract these observations to provide to the algorithms. It is not trivial to connect with all points such that the test drivers could automatically gather observations.

RALT can detect errors using (lightweight) specifications: Initially, this work was not subjected for detecting specification based errors in the system, since such specifications for checking the system conformance were not assumed. We have been mostly relying on detecting compositional problems and generic errors in the system. However, we have noted that our tool RALT can detect errors if some lightweight specifications are provided. A simple form is writing the expected scenarios of the system [BB98]. A scenario could be an intuition or a vague idea that the designer tries to accomplish with the system. The scenarios in terms of inputs and outputs traces are easy to produce, which express "should" or "should not" requirements. Such scenarios can be written in RALT using the provided API. The tool can verify the scenarios on the learned models by simply executing the inputs in the scenarios and matching the corresponding outputs with the model. An error is produced when there is a mismatch.

Human involvement is unavoidable for best results We have tried to accomplish an automatic approach for learning and testing. However, we found that human involvement is unavoidable for producing best results. At different steps of our approach (Chapter 6), there are choices to opt, where human guidance can ease the situation. For example, the designer can evaluate whether a newly discovered input during the component learning phase should be taken into account for learning. At another point, an eye-ball inspection of the discrepancy between the learned models and the system may help in the easy identification of a potential error in the system.

10.5 Ending Note

Despite the progress we have made in this research, there are many future directions aiming at improving the effectiveness of automated software learning and testing. Our experiments were mainly conducted on real but small-scale problems (even if structurally complex) where no specifications were assumed, and focused on the behavior exploration. The two main directions that naturally emerge from this research are i) reducing the complexity of learning, and ii) devising testing techniques for detecting errors and model refinements. There is much room for improvement from both theoretical and practical aspects in these directions.

"After all these years, I do not know what I may appear to the world, but to myself, I seem to have been only a boy playing on the sea-shore and diverting myself in, now and then, finding a smoother pebble or a prettier shell than ordinary whilst the great ocean of truth lay all undiscovered before me." — *Isaac Newton*

Appendix A

Proofs of the PFSM Algorithm

Proof of Theorem 6

The theorem can be proved with the help of Lemma 5, 6 and 7 as follows.

Introduction of Lemma 5

In the observation table (S_P, R, E_P, T_P), the information obtained from the functions OS and OPS is consistent with respect to the information recorded in the table in different cells. This can be realized in Figure A.1a, where we have two rows s and t and three columns i, e and $i \cdot e$, then for the common prefixes of the input parameter value string $\alpha \cdot x \cdot \gamma$ recorded in the three different cells of the table, we have common outputs recorded in those cells. Or in other words, we have $OS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OS(s, i, \alpha \cdot x) \cdot OS(t, e, \alpha \cdot x \cdot \gamma)$ and $OPS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OPS(s, i, \alpha \cdot x) \cdot OPS(t, e, \alpha \cdot x \cdot \gamma)$. This is formalized in Lemma 5. Figure A.1b illustrates Lemma 5 that for the common prefixes of an input parameter value string, e.g., $\pi(T_P(T, T)) = 5 \cdot 15$, $\pi(T_P(T \cdot T, \cdot T \cdot T)) = 5 \cdot 15 \cdot 25 \cdot 35$ and $\pi(T_P(T, T \cdot T \cdot T)) = 5 \cdot 15 \cdot 25 \cdot 35$ recorded in the table, we have $OS(T, T \cdot T \cdot T, 5 \cdot 15 \cdot 25 \cdot 35) = OS(T, T, 5 \cdot 15) \cdot OS(T \cdot T, T \cdot T, 5 \cdot 15 \cdot 25 \cdot 35)$.

Now, we state Lemma 5 formally, followed by its proof.

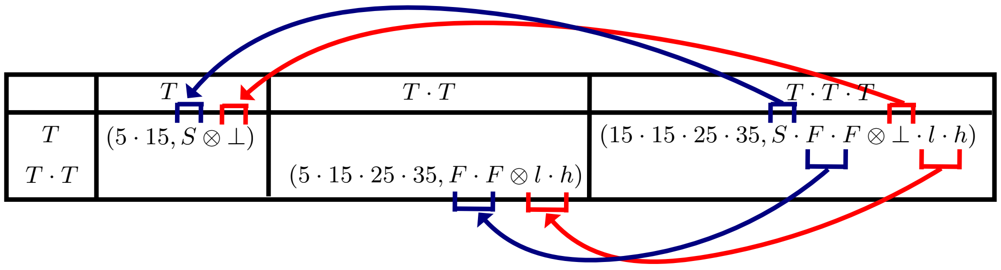
Lemma 5 *In an observation table (S_P, R, E_P, T_P), for all $s, t \in S_P \cup R, e \in E_P, i \in I$ such that $IS(t) = IS(s) \cdot i$ and $i \cdot e \in E_P$, then if there exists common input parameter value strings $\alpha \cdot x \in \pi(T_P(s, i)), \alpha \cdot x \cdot \gamma \in \pi(T_P(t, e))$ and $\alpha \cdot x \cdot \gamma \in \pi(T_P(s, i \cdot e))$, where $|\alpha| = |IS(s)|, |x| = |i|, |\gamma| = |e|$, then we have the output symbol string and the output parameter value string for s and $i \cdot e$ as $OS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OS(s, i, \alpha \cdot x) \cdot OS(t, e, \alpha \cdot x \cdot \gamma)$ and $OPS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OPS(s, i, \alpha \cdot x) \cdot OPS(t, e, \alpha \cdot x \cdot \gamma)$ respectively.* □

Proof The lemma can be proved by exhibiting Property 4.

Since, we know that

	i	e	$i \cdot e$
s	$(\alpha \cdot x, \varpi_1 \otimes \beta_1)$		$(\alpha \cdot x \cdot \gamma, \varpi_1 \cdot \varpi_2 \otimes \beta_1 \cdot \beta_2)$
t		$(\alpha \cdot x \cdot \gamma, \varpi_2 \otimes \beta_2)$	

(a) Realization of Lemma 5. $OS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OS(s, i, \alpha \cdot x) \cdot OS(t, e, \alpha \cdot x \cdot \gamma)$ and $OPS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OPS(s, i, \alpha \cdot x) \cdot OPS(t, e, \alpha \cdot x \cdot \gamma)$



T
$T \cdot T$
$T \cdot T \cdot T$
T
$T \cdot T$
$(5 \cdot 15, S \otimes \perp)$
$(5 \cdot 15 \cdot 25 \cdot 35, F \cdot F \otimes l \cdot h)$
$(15 \cdot 15 \cdot 25 \cdot 35, S \cdot F \cdot F \otimes \perp \cdot l \cdot h)$

(b) Illustration of Lemma 5

Figure A.1: Explanation of Lemma 5

$$\lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot i \cdot e, \alpha \cdot x \cdot \gamma) = \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s), \alpha) \cdot OS(s, i \cdot e, \alpha \cdot x \cdot \gamma) \tag{A.1}$$

also we can write

$$\begin{aligned} \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot i \cdot e, \alpha \cdot x \cdot \gamma) \quad &= \quad \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot i, \alpha \cdot x) \cdot \\ &\quad\quad OS(t, e, \alpha \cdot x \cdot \gamma), \text{ since } IS(t) = IS(s) \cdot i \end{aligned}$$

which can further be broken down as

$$\begin{aligned} \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s) \cdot i \cdot e, \alpha \cdot x \cdot \gamma) \quad &= \quad \lambda_{\mathcal{P}}(q_{0\mathcal{P}}, IS(s), \alpha) \cdot \\ &\quad\quad OS(s, i, \alpha \cdot x) \cdot \\ &\quad\quad OS(t, e, \alpha \cdot x \cdot \gamma) \end{aligned} \tag{A.2}$$

Thus, from equations (A.1) and (A.2), we have

$$OS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OS(s, i, \alpha \cdot x) \cdot OS(t, e, \alpha \cdot x \cdot \gamma)$$

Similarly, the following part of the lemma can be proved as above

$$OPS(s, i \cdot e, \alpha \cdot x \cdot \gamma) = OPS(s, i, \alpha \cdot x) \cdot OPS(t, e, \alpha \cdot x \cdot \gamma) \qquad \blacksquare$$

Lemma 6 *Assume that (S_P, R, E_P, T_P) is a balanced, dispute-free and closed observation table. For the conjecture M_P and for every $s \in S_P \cup R$, $e \in E_P$, $\alpha \cdot \alpha' \in \pi(T_P(s,e))$ such that $|\alpha| = |IS(s)|$, we have $\delta_P(q_{0P}, IS(s), \alpha) = [(IS(s))_\alpha]$.* □

Proof The lemma can be proved by the induction on the length of $IS(s)$. If $|IS(s)| = 0$, i.e., $IS(s) = s = \epsilon$, then we know that $[\epsilon] = q_0$ and ϵ is associated with no parameters (or $\perp$), hence trivially $\delta_P(q_0, \epsilon, \perp) = [\epsilon]$.

If $|IS(s)| = 1$ and therefore $|\alpha| = 1$, then there exists $i \in I$ such that $\epsilon \cdot i = IS(s)$. If $\alpha \cdot \alpha' \in \pi(T_P(\epsilon \cdot i, e))$ then according to Property 5, $\alpha \in \pi(T_P(\epsilon, i))$ also holds, since $E \supseteq I$. Then,

$$\begin{aligned}
\delta_P(q_{0P}, IS(s), \alpha) &= \delta_P(q_{0P}, \epsilon \cdot i, \alpha) \\
&= \delta_P(\delta_P(q_{0P}, \epsilon, \perp), i, \alpha) \\
&= \delta_P([\epsilon], i, \alpha), \text{from above} \\
&= [(\epsilon \cdot i)_\alpha], \text{by the def. of } \delta_P\text{, since } \alpha \in \pi(T_P(\epsilon, i)) \text{ (Property 5)} \\
&= [(IS(s))_\alpha]
\end{aligned}$$

Assume that the equation is true for every $s \in S_P \cup R$ where $IS(s)$ is of length k. Let $t \in S_P \cup R$ such that $IS(t) = IS(s) \cdot i$ for some $i \in I$. That means $IS(t)$ is of length $k+1$. Suppose $\alpha \cdot x \cdot \gamma \in \pi(T_P(t,e))$ for any $e \in E_P$ and $|\alpha| = |IS(s)|, |x| = |i|, |\gamma| = |e|$. From Property 5, $\alpha \cdot x$ is also in $\pi(T_P(s,i))$. This can be realized in Figure A.2. Moreover, s must be in S_P, for either $t \in S_P$ because S_P is prefix-closed, or $t \in R$, since the table is closed, and therefore $s \in S_P$. Then,

$$\begin{aligned}
\delta_P(q_0, IS(t), \alpha \cdot x) &= \delta_P(q_0, IS(s) \cdot i, \alpha \cdot x) \\
&= \delta_P(\delta_P(q_0, IS(s), \alpha), i, x) \\
&= \delta_P([(IS(s))_\alpha], i, x), \text{by the ind. hyp.} \\
&= [(IS(s) \cdot i)_{\alpha \cdot x}], \text{by the def. of } \delta_P\text{, since } \alpha \cdot x \in \pi(T_P(s,i)) \text{ (Property 5)} \\
&= [(IS(t))_{\alpha \cdot x}]
\end{aligned}$$

■

Lemma 7 *Assume that (S_P, R, E_P, T_P) is a balanced, dispute-free and closed observation table. Then, the conjecture M_P is consistent with the finite function T_P. That is, for every $s \in S_P \cup R, e \in E_P, \alpha \cdot \alpha' \in \pi(T_P(s,e)$, where $|\alpha| = |IS(s)|$ and $|\alpha'| = |e|$, then $\lambda_P(\delta_P(q_{0P}, IS(s), \alpha), e, \alpha') = OS(s, e, \alpha \cdot \alpha')$ and $\sigma_P(\delta_P(q_{0P}, IS(s), \alpha), e)(\alpha') = OPS(s, e, \alpha \cdot \alpha')$.* □

	i	e	$f = i \cdot e$
s	$\alpha \cdot x \in \pi(T_P(s,i))$		$\alpha \cdot x \cdot \gamma \in \pi(T_P(s,f))$
$t = s \cdot i$		$\alpha \cdot x \cdot \gamma \in \pi(T_P(t,e))$	

Figure A.2: Realization of Lemma 6 and Lemma 7

Proof The lemma can be proved by the induction on the length of e. In fact, $|e| > 0$, since E_P is initialized by the elements of I. If $|e| = 1$ and so $|\alpha'| = 1$, then

$$\begin{aligned}\lambda_P(\delta_P(q_0, IS(s), \alpha), e, \alpha') &= \lambda_P([(IS(s))_\alpha], e, \alpha'), \text{by Lemma 6}\\ &= OS(s, e, \alpha \cdot \alpha'), \text{by the def. of } \lambda_P\end{aligned}$$

Assume that the equation is true for all $e \in E_P$ of length k. Let $f \in E_P$ of length $k+1, k > 0$. Since, E_P is suffix-closed, $f = i \cdot e$ for some $i \in I$ and some $e \in E_P$ of length k. For $s \in S_P \cup R, \alpha \cdot \alpha' \in \pi(T_P(s,f))$, where $|\alpha| = |IS(s)|, |\alpha'| = |f|$, we can write $\alpha' = x \cdot \gamma$, where $|x| = |i|, |\gamma| = |e|$. Also, for s and i, let $t \in S_P \cup R$ such that $IS(t) = IS(s) \cdot i$. According to Property 6, $\alpha \cdot x \cdot \gamma \in \pi(T_P(t,e))$. This can be realized in Figure A.2. Then,

$$\begin{aligned}\lambda_P(\delta_P(q_{0P}, IS(s), \alpha), f, \alpha') &= \lambda_P(\delta_P(q_{0P}, IS(s), \alpha), i \cdot e, x \cdot \gamma)\\ &= \lambda_P([(IS(s))_\alpha], i \cdot e, x \cdot \gamma), \text{by Lemma 6}\\ &= \lambda_P([(IS(s))_\alpha], i, x) \cdot \lambda_P([(IS(s) \cdot i)_{\alpha \cdot x}], e, \gamma)\\ &= \lambda_P([(IS(s))_\alpha], i, x) \cdot \lambda_P([(IS(t))_{\alpha \cdot x}], e, \gamma)\\ &= OS(s, e, \alpha \cdot x) \cdot OS(t, e, \alpha \cdot x \cdot \gamma), \text{by the def. of } \lambda_P\\ &= OS(s, i \cdot e, \alpha \cdot x \cdot \gamma), \text{by Lemma 5}\\ &= OS(s, f, \alpha \cdot \alpha')\end{aligned}$$

The proof of $\sigma_P(\delta_P(q_{0P}, IS(s), \alpha), e)(\alpha') = OPS(s, e, \alpha \cdot \alpha')$ is similar to above. This concludes the proof of Theorem 6, since Lemma 7 shows that M_P is consistent with T_P. ■

Proof of Theorem 7

Proof Let $M'_P = \{Q'_P, I, O, D_I, D_O, \Gamma'_P, q'_{0P}\}$ in which for each state $q' \in Q', i \in I, x \in D_I$, the target state, the output and the output parameter value is determined by δ'_P, λ'_P and σ'_P respectively. Note that if M'_P accepts exactly the same parameter values as of M_P and is consistent with T_P then for all $s \in S_P \cup R, e \in E_P, \alpha \cdot \gamma \in \pi(T_P(s,e)), |\alpha| = |IS(s)|, |\gamma| = |e|$, $OS(s,e,\alpha \cdot \gamma) = \lambda_P(\delta_P(q_{0P}, IS(s), \alpha), e, \gamma) = \lambda'_P(\delta'_P(q'_{0P}, IS(s), \alpha), e, \gamma)$ holds. Also, $OPS(s,e,\alpha \cdot \gamma) = \sigma_P(\delta_P(q_{0P}, IS(s), \alpha), e)(\gamma) = \sigma'_P(\delta'_P(q'_{0P}, IS(s), \alpha), e)(\gamma)$ holds.

For all $s_1, s_2 \in S_P$, we know that $s_1 \not\cong_{\Phi_P} s_2$, i.e., there exists $e \in E_P$, $\alpha_1 \cdot \gamma \in \pi(T_P(s_1,e))$, $\alpha_2 \cdot \gamma \in \pi(T_P(s_2,e))$ such that $|\alpha_1| = |IS(s_1)|$, $|\alpha_2| = |IS(s_2)|$ and $|\gamma| = |e|$, and $OS(s_1,e,\alpha_1 \cdot \gamma) \neq OS(s_2,e,\alpha_2 \cdot \gamma)$. This means $\delta_P(q_{0P}, IS(s_1), \alpha_1) \neq \delta_P(q_{0P}, IS(s_2), \alpha_2)$. If M'_P is consistent with T_P, then $\delta'_P(q'_{0P}, IS(s_1), \alpha_1) \neq \delta'_P(q'_{0P}, IS(s_2), \alpha_2)$ also holds. So for all $s \in S_P, e \in E_P, \alpha \cdot \gamma \in \pi(T_P(s,e))$ such that $|\alpha| = |IS(s)|$ and $|\gamma| = |e|$, when s ranges over all of S_P, $\delta'_P(q'_{0P}, IS(s), \alpha)$ ranges over all the elements of Q_P. Hence, M'_P has at least n states, i.e., it must have exactly n states.

We define a mapping $\phi : Q_P \rightarrow Q'_P$. That means, for all $s \in S_P$ or $[s] \in Q_P$, $e \in E_P$ and $\alpha \cdot \gamma \in \pi(T_P(s,e))$, such that $|\alpha| = |IS(s)|, |\gamma| = |e|$, there is a corresponding state $\delta'_P(q'_{0P}, IS(s), \alpha) = q'_P \in Q'_P$ such that $\phi([s]) = q'_P$. This mapping is bijection. We must verify that it carries q_{0P} to q'_{0P}, i.e., $\phi(q_{0P}) = q'_{0P}$. This is as follows:

$$\begin{aligned}
\phi(q_{0P}) &= \phi([\epsilon]), \text{by the def. of } q_{0P} \\
&= \delta'_P(q'_{0P}, \epsilon, \perp), \text{by the def. of } \phi \\
&= q'_{0P}
\end{aligned}$$

For $s \in S_P, i \in I$, let $t \in S_P \cup R$ such that $IS(t) = IS(s) \cdot i$, for all $e \in E_P$, and $\alpha \cdot x \cdot \gamma \in \pi(T_P(t,e)$ such that $|\alpha| = |IS(s)|$, $|x| = |i|$ and $|\gamma| = |e|$, then according to Property 5, $\alpha \cdot x \in \pi(T_P(s,i)$. Then,

$$\begin{aligned}
\phi(\delta_P([s], i, x)) &= \phi([(IS(s) \cdot i)_{\alpha \cdot x}]), \text{by the def. of } \delta_P \\
&= \delta'_P(q'_{0P}, IS(s) \cdot i, \alpha \cdot x), \text{by the def. of } \phi
\end{aligned} \tag{A.3}$$

Also,

$$\begin{aligned}
\delta'_P(\phi([s]), i, x) &= \delta'_P(\delta'_P(q'_{0P}, IS(s), \alpha), i, x), \text{by the def. of } \phi \\
&= \delta'_P(q'_{0P}, IS(s) \cdot i, \alpha \cdot x)
\end{aligned} \tag{A.4}$$

From equations A.3 and A.4, we conclude that $\phi(\delta_P([s], i, x)) = \delta'_P(\phi([s]), i, x)$. We also need to check that for all $s \in S_P, i \in I, \alpha \cdot x \in \pi(T_P(s,i)), |\alpha| = |IS(s)|$, $\lambda_P([s], i, x) = \lambda'_P(\phi([s]), i, x)$. But we know that $\phi([s]) = \delta'_P(q'_{0P}, IS(s), \alpha)$. Since, M'_P is consistent with T_P, then for all $i \in I$,

$$\lambda'_P(\delta'_P(q'_{0P}, IS(s), \alpha), i, x) = \lambda'_P(\phi([s]), i, x) = OS(s, i, \alpha \cdot x) \tag{A.5}$$

Also, according to the definition,

$$\lambda_P([s], i, x) = OS(s, i, \alpha \cdot x) \tag{A.6}$$

From equations A.5 and A.6, we have $\lambda_P([s], i, x) = \lambda'_P(\phi([s]), i, x)$. Similarly, for σ_P and σ'_P, we must check that $\sigma_P([s], i)(x) = \sigma'_P(\phi([s]), i)(x)$. This is analogy to that of λ_P and λ'_P.

This concludes the proof of Theorem 7, since any other PFSM M'_P that accepts exactly the same parameter values and is consistent with T_P must have at least n states. If M'_P is inequivalent to M_P than it trivially has more states. ■

Appendix B

Models of the Domotics Case Study

The Mealy machine models of the devices has been generated by RALT 4.0 in JFLAP [JFL] format.

Model of the Light System *(ProSyst)*

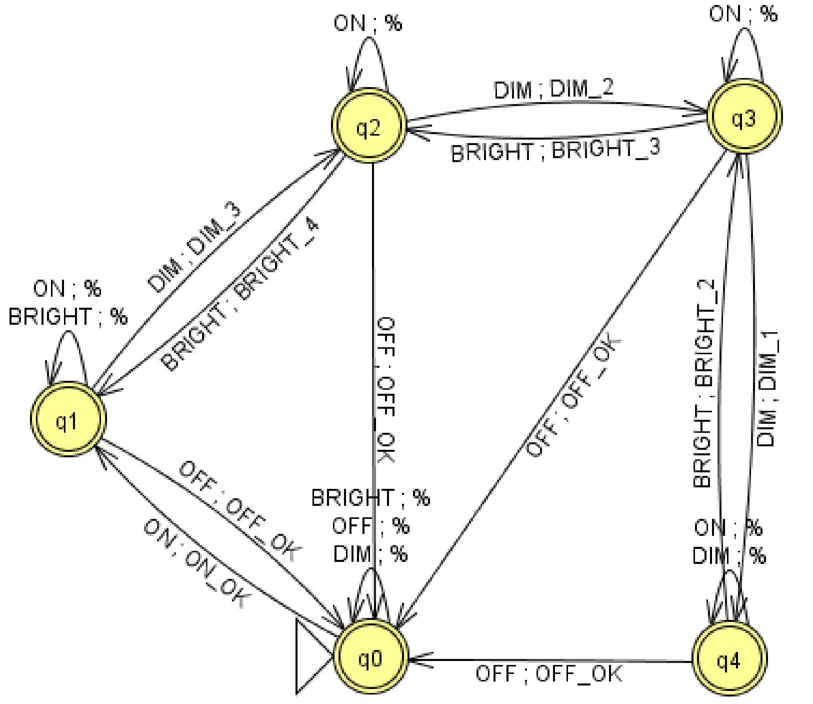


Figure B.1: Light System *(ProSyst)* (X10 Device)

Model of the Media Renderer *(Philips Streamium400i)*

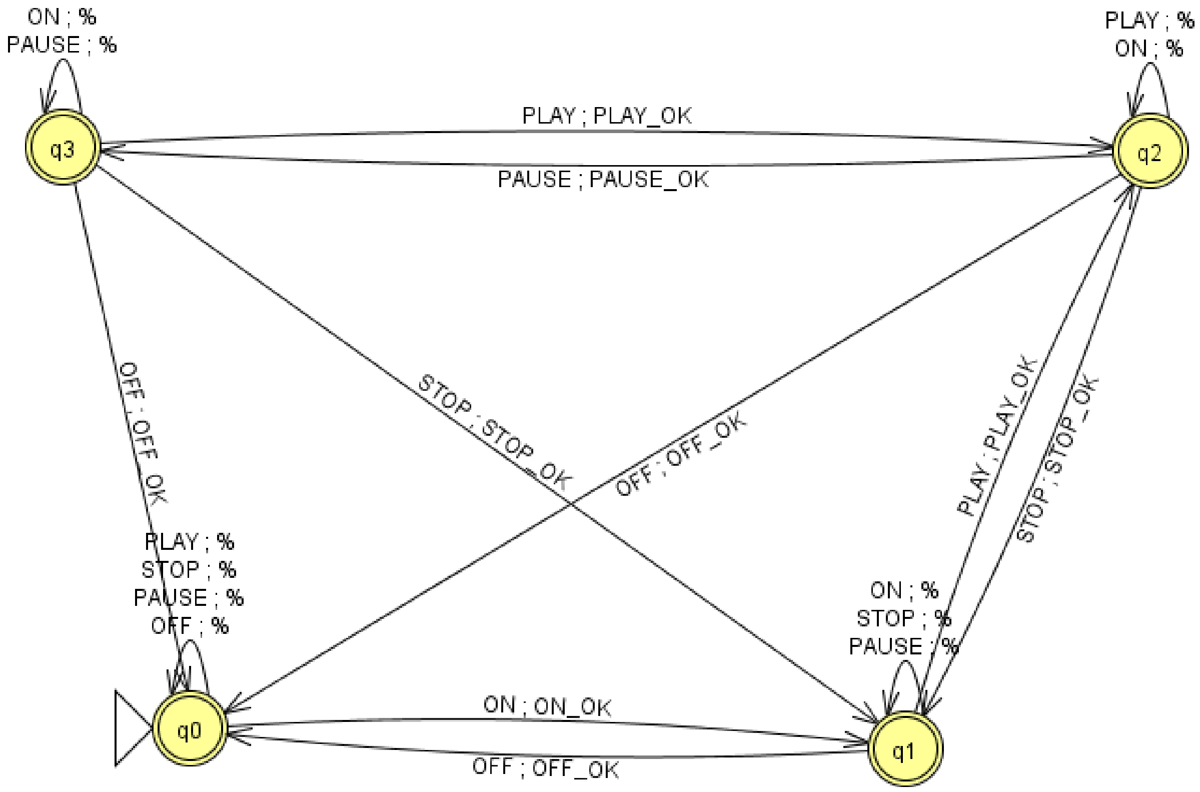


Figure B.2: Media Renderer *Streamium*400*i* (UPnP Device)

Model of the Domotics System

Figure B.3 is the interaction model of the devices in the Domotics system given in Figure 9.8. The devices in the system are: Light System *(ProSyst)*, Media Renderer *(Philips Streamium400i)* and TV *(Acer AT3705-MGW)*.

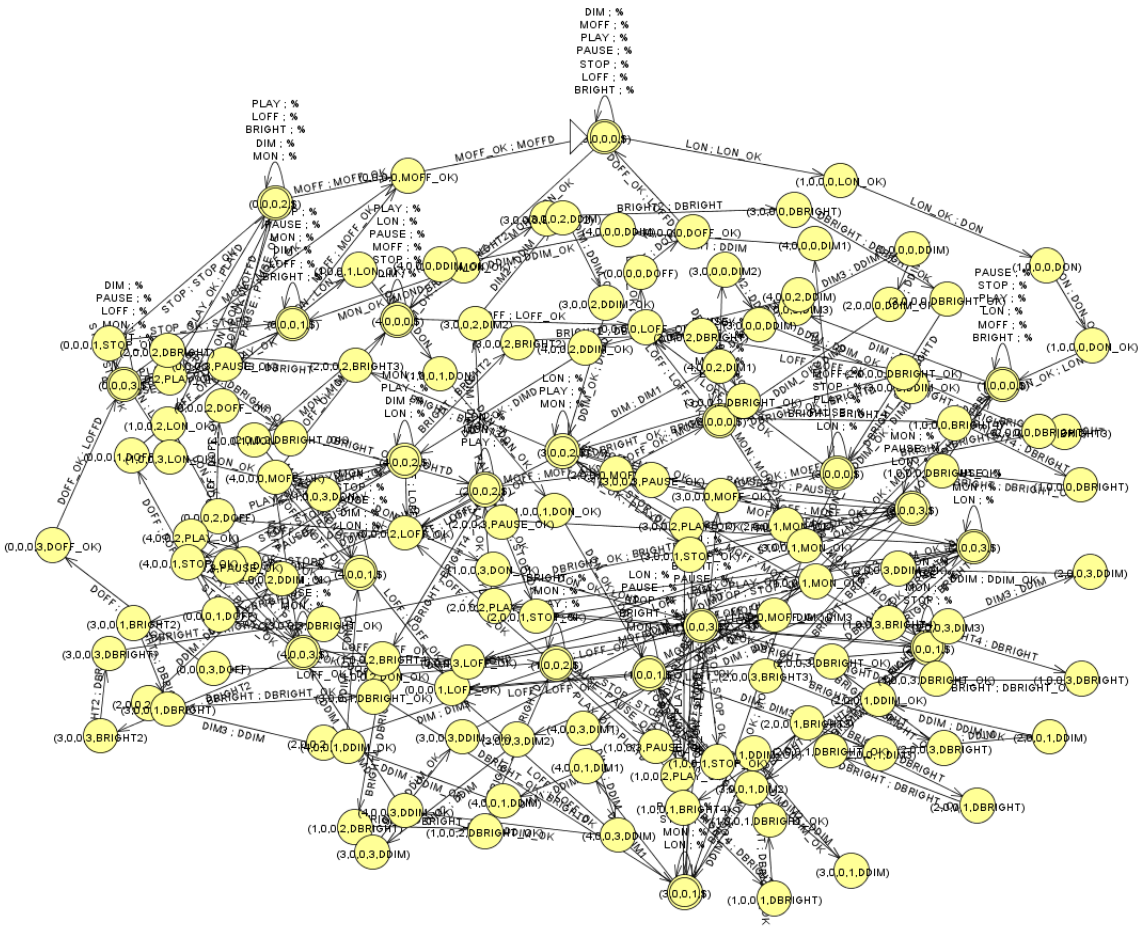


Figure B.3: Interaction model of the devices in the Domotics System

Bibliography

[ABL02] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, 2002. 25

[All] OSGi Alliance. OSGi Service Platform. http://www.osgi.org. 2, 147

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, August 2006. 4

[Ang81] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981. 28

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2:87–106, 1987. 27, 28, 35, 41, 44, 53

[Ang88] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988. 41

[AO00] Aynur Abdurazik and A. Jefferson Offutt. Using uml collaboration diagrams for static checking and test generation. In *UML*, pages 383–395, 2000. 23

[Apa] Apache. Apache Felix. http://felix.apache.org. 147

[AS83] D. Angluin and C.H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys (CSUR)*, 15(3):237–269, 1983. 25

[BB98] P. Bengtsson and J. Bosch. Scenario-based software architecture reengineering. *Proceedings of Fifth International Conference on Software Reuse*, pages 308–317, 1998. 5, 159

[BDG97] Jose L. Balcazar, Josep Diaz, and Ricard Gavalda. Algorithms for learning finite automata from queries: A unified view. In *Advances in Algorithms, Languages, and Complexity*, pages 53–72, 1997. 53, 65

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007. 3, 5, 7, 37

[BF72] A. Biermann and J. Feldman. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Computers*, 21(6):592–597, 1972. 35

[BfCE04] Mireille Blay-fornarino, Anis Charfi, and David Emsellem. Software interactions. *Journal of Object Technology*, 3(10), 2004. 4

[BGJ$^+$05] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen. On the correspondence between conformance testing and regular inference. In *FASE*, pages 175–189, 2005. 56

[BGM91] Gilles Bernot, Marie-Claude Gaudel, and Bruno Marre. A formal approach to software testing. In *AMAST*, pages 243–253, 1991. 100, 156

[BJLS05] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena. Insights to angluin's learning. *Electr. Notes Theor. Comput. Sci.*, 118:3–18, 2005. 8, 29, 36, 37, 51, 134

[BJR06] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *LNCS*, pages 107–121. Springer, 2006. 34, 35, 36, 38

[BMP06] Antonia Bertolino, Henry Muccini, and Andrea Polini. Architectural verification of black-box component-based systems. In *RISE*, pages 98–113, 2006. 26

[BO05] Miguel Bugalho and Arlindo L. Oliveira. Inference of regular languages using state merging algorithms with search. *Pattern Recognition*, 38(9):1457–1467, 2005. 25

[Boz04] *The IF Toolset*, volume 3185 of *LNCS*. Springer, 2004. 155

[BR05] Therese Berg and Harald Raffelt. Model checking. In *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*, pages 557–603. Springer, 2005. 53

[BRH04] James F. Bowring, James M. Rehg, and Mary Jean Harrold. Active learning for automatic classification of software behavior. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 195–205. ACM, 2004. 25

[Büc62] J. Richard Büchi. On a decision method in restricted second order arithmetic. *Proc. Int. Congress on Logic, Methodology and Philosophy of Science 1960*, pages 1–11, 1962. 26

[CCC$^+$02] Ivica Crnkovic, Ivica Crnkovic, Ivica Crnkovic, Magnus Larsson, and Magnus Larsson. Challenges of component-based development. *Journal of Systems and Software*, 61:201–212, 2002. 3

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000. 4

[Cho78] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. on Software Engineeing*, 4(3):178–187, 1978. 28, 92, 131

[CJ02] Rick D. Craig and Stefan P. Jaskiel. *Systematic Software Testing*. Artech House, Inc., Norwood, MA, USA, 2002. 4

[CKMRM03] Muffy Calder, Mario Kolberg, Evan H. Magill, and Stephan Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Netw.*, 41(1):115–141, 2003. 4

[CLW05] X. Cai, M.R. Lyu, , and K.F. Wong. A generic environment for cots testing and quality prediction. *Testing Commercial-off-the-shelf Components and Systems*, pages 315–347, 2005. 4

[CORa] CORBA Component Model. www.omg.org/technology/documents/formal/components.htm. 2

[Corb] ParaSoft Corporation. JTest. http://www.parasoft.com/. 4

[CPV03] Alejandra Cechich, Mario Piattini, and Antonio Vallecillo, editors. *Component-Based Software Quality - Methods and Techniques*, volume 2693 of *LNCS*. Springer, 2003. 2

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004. 5

[CSSW05] Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, and Kurt C. Wallnau. Automated component-based software engineering. *Journal of Systems and Software*, 74(1):1–3, 2005. 2

[CW98] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Trans. Softw. Eng. Methodol.*, 7(3):215–249, 1998. 6, 25

[Dij70] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, 1970. 5

[DLD+08] P. Dupont, B. Lambeau, C. Damas, A. Van Lamsweerde, and Place Sainte Barbe. The qsm algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22:77–115, 2008. 32

[dlH05] Colin de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005. 24

[Dup94] Pierre Dupont. Regular grammatical inference from positive and negative samples by genetic search: The GIG Method. In *ICGI '94: Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 236–245, London, UK, 1994. Springer-Verlag. 25

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, Feb 2001. 4, 23, 26, 38, 122, 124

[Edw01] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Softw. Test., Verif. Reliab.*, 11(2):97–111, 2001. 4

[EGPQ06] Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE*, pages 420–435, 2006. 30, 36

[Eme90] E. Allen Emerson. *Temporal and modal logic.* MIT Press, Cambridge, MA, USA, 1990. 27

[FBK+91] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17:591–603, 1991. 28, 32, 131

[FvB92] Susumu Fujiwara and Gregor von Bochmann. Testing non-deterministic state machines with fault coverage. In *Proceedings of the IFIP TC6/WG6.1 Fourth International Workshop on Protocol Test Systems IV*, pages 267–280. North-Holland Publishing Co., 1992. 156

[GEH05] Jerry Gao, Raquel Espinoza, and Jingsha He. Testing coverage analysis for software component validation. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, volume 1, pages 463–470. IEEE Computer Society, 2005. 4

[GLPS08] Roland Groz, Keqin Li, Alexandre Petrenko, and Muzammil Shahbaz. Modular system verification by inference, testing and reachability analysis. In *TestCom/FATES*, pages 216–233, 2008. 90, 154, 155

[GMC$^+$92] C. L. Giles, C. B. Miller, D. Chen, H. H. Chen, G. Z. Sun, and Y. C. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992. 24

[Gol72] E. Gold. System identification via state characterization. *Automatica*, 8:621–636, 1972. 24, 27

[Gol78] E. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978. 24

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 357–370, 2002. 30, 36

[Gr08] L. Grgen. State of the art of sensor data management. ACSE Project, WP2-Architecture. Technical report, Orange Labs, France Telecom Group, 2008. 146

[Gra] Graphviz - Graph Visualization Software. http://www.graphviz.org. 132

[GSL07] Roland Groz, Muzammil Shahbaz, and Keqin Li. Une approche incrémentale de test par extraction de modèles. In *AFADL*, June 2007. 153

[GSL08] Roland Groz, Muzammil Shahbaz, and Keqin Li. Extraction de modèles paramétrés au cours du test de composants logiciels. *Technique et science informatiques*, 27(8):977–1006, 2008. 155

[GTWJ03] Jerry Zayu Gao, Jacob Tsao, Ye Wu, and Taso H.-S. Jacob. *Testing and Quality Assurance for Component-Based Software.* Artech House, Inc., Norwood, MA, USA, 2003. 2, 3, 4, 79

[Har00] Mary Jean Harrold. Testing: A Roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72. ACM, 2000. 3

[HG94] Bill G. Horne and C. Lee Giles. An experimental comparison of recurrent neural networks. In *NIPS*, pages 697–704, 1994. 25

[HHNS02] Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Fundamental Approaches to Software Engineering*, pages 80–95, 2002. 155, 156, 158

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301. ACM, 2002. 4

[HLW94] D. Haussler, N. Littlestone, and M. K. Warmuth. Predicting 0,1-functions on randomly drawn points. *Inf. Comput.*, 115(2):248–292, 1994. 24

[HMS03] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. In *ITC*, pages 971–980, 2003. 28, 33, 34, 36

[HNS03] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *LNCS*, pages 315–327. Springer, 2003. 28, 33, 34, 36, 51, 55, 156

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual.* Addison-Wesley Professional, September 2003. 26

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990. 135

[IMP05] Paola Inverardi, Henry Muccini, and Patrizio Pelliccione. Charmy: an extensible tool for architectural analysis. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 111–114. ACM, 2005. 26

[JFL] JFLAP. http://www.cs.duke.edu/csed/jflap/. 132, 167

[JUn] JUnit.org. JUnit. http://www.junit.org/. 4

[KKS98] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *FTCS '98*, page 230. IEEE Computer Society, 1998. 5

[Kor99] Bogdan Korel. Black-box understanding of cots components. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 92. IEEE Computer Society, 1999. 38, 101

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory.* MIT Press, Cambridge, MA, USA, 1994. 6, 7, 8, 27, 51

[LCJ06] Zhifeng Lai, S. C. Cheung, and Yunfei Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. In *QSIC '06: Proceedings of the Sixth International Conference on Quality Software*, pages 410–417, Washington, DC, USA, 2006. IEEE Computer Society. 25

[LGS06a] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006. 153

[LGS06b] Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *LNCS*, pages 436–450. Springer, 2006. 153

[Lit87] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987. 24

[LMP06] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Inferring state-based behavior models. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 25–32. ACM Press, 2006. 35, 37, 38

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996. 5, 28, 35, 38, 51

[Man07] Mantis bug tracker. http://www.mantisbt.org/, June 2007. 32

[Mcg06] Gary Mcgraw. *Software Security: Building Security In.* Addison-Wesley Professional, 2006. 157

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990. 5

[Mic] Sun Microsystems. Java Platform, Enterprise Edition. http://java.sun.com/javaee/. 2

[Mil82] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. 156

[MNRS04] T. Margaria, O. Niese, H. Raffelt, and B. Steffen. Efficient test-based model generation for legacy reactive systems. In *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop*, pages 95–100. IEEE Computer Society, 2004. 34, 36, 37, 55, 56

[MNTW01] B.A. Miller, T. Nixon, C. Tai, and M.D. Wood. Home networking with universal plug and play. *Communications Magazine, IEEE*, 39(12):104–109, Dec 2001. 146

[MP05] Leonardo Mariani and Mauro Pezzè. Behavior capture and test: Automated analysis of component integration. *ICECCS*, 0:292–301, 2005. 26

[MPW04] Leonardo Mariani, Mauro Pezzè, and David Willmor. Generation of integration tests for self-testing components. In *FORTE Workshops*, pages 337–350, 2004. 4

[MS01a] Erkki Makinen and Tarja Systa. Mas - an interactive synthesizer to support behavioral modelling in uml. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society. 28

[MS01b] Erkki Mäkinen and Tarja Systä. Mas - an interactive synthesizer to support behavioral modelling in uml. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24. IEEE Computer Society, 2001. 55

[MS04] Faron Moller and Perdita Stevens. *Edinburgh Concurrency Workbench User Manual (Version 7.1)*, 2004. Available from http://homepages.inf.ed.ac.uk/perdita/cwb/. 71, 134, 152, 155

[MSBT04] Glenford J. Myers, Corey Sandler, Tom Badgett, and Todd M. Thomas. *The Art of Software Testing, Second Edition.* Wiley, June 2004. 38

[.NE] Microsoft .NET Framework 3.0. http://netfx3.com/. 2

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 229–239, New York, NY, USA, 2002. ACM Press. 4

[Nei03] Oliver Neise. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, 2003. 55, 56, 59

[NS08] Phong Thien Nguyen and Muzammil Shahbaz. Reverse Engineering Real Devices. Domotics – A Case Study. Technical report, Orange Labs, France, 2008. 129, 149

[OCA] Open Communications Architecture Forum (OCAF) Focus Group. http://www.itu.int/ITU-T/ocaf/. 2

[OHR01] Alessandro Orso, Mary Jean Harrold, and David S. Rosenblum. Component metadata for software engineering tasks. In *EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects*, pages 129–144. Springer-Verlag, 2001. 4

[OMG] The Object Management Group (OMG). http://www.omg.org/. 2

[OWS] Open Web Application Security Project. http://www.owasp.org. 157

[Pac05] Technica Pacifica. *Easy X10 Projects for Creating a Smart Home.* Indy-Tech Publishing, 2005. 147

[Pha94] Marc Phalippou. *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties.* PhD thesis, Université de Bordeaux 1, 1994. 100, 156

[Pit89] Leonard Pitt. Inductive inference, dfas, and computational complexity. In *AII '89: Proceedings of the International Workshop on Analogical and Inductive Inference*, pages 18–44, London, UK, 1989. Springer-Verlag. 25

[PO99] Jorge M. Pena and Arlindo L. Oliveira. A new algorithm for exact reduction of incompletely specified finite state machines. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(11):1619–1632, 1999. 56

[Pol91] Jordan B. Pollack. The induction of dynamical recognizers. *Mach. Learn.*, 7(2-3):227–252, 1991. 24

[Pro] Java Community Process. Java Specification Requests. http://jcp.org/en/jsr/all. 141

[PvBY96] A. Petrenko, G. v. Bochmann, and M. Yao. On fault coverage of tests for finite state specifications. *Comput. Netw. ISDN Syst.*, 29(1):81–106, 1996. 92

[PVY99] D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings of FORTE'99*, Beijing, China, 1999. 7, 8, 28, 29, 30, 31, 35, 36, 88, 92

[PW89] L. Pitt and M. K. Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. In *STOC: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 421–432. ACM Press, 1989. 27

[Rav03] T. Ravichandran. Special issue on component-based software development. *SIGMIS Database*, 34(4):45–46, 2003. 2

[RF07] O.R. Ribeiro and J.M. Fernandes. Validation of reactive software from scenario-based models. *QUATIC: International Conference on Quality of Information and Communications Technology*, pages 213–217, 2007. 94

[RG99] J. Ryser and M. Glinz. A scenario-based approach to validating and testing software systems using statecharts. *Proceedings of 12th International Conference on Software and Systems Engineering and their Applications*, 1999. 94

[Rod06] Susan Rodger. *JFLAP-an Interactive Formal Languages and Automata Package.* Jones and Bartlett, Boston, 2006. 132

[RP97] A. Raman and J. Patrick. The sk-strings method for inferring pfsa, 1997. 25

[RS93] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Machine Learning: From Theory to Applications*, pages 51–73, 1993. 28, 51, 53, 56, 65

[RSM07] Harald Raffelt, Bernhard Steffen, and Tiziana Margaria. Dynamic testing via automata learning. In *Haifa Verification Conference*, pages 136–152, 2007. 28, 32, 34, 36, 37

[Sch01] Stephen R. Schach. *Object-Oriented and Classical Software Engineering.* McGraw-Hill Pub. Co., 2001. 138

[SCHS07] Heinz W. Schmidt, Ivica Crnkovic, George T. Heineman, and Judith A. Stafford, editors. *Component-Based Software Engineering, 10th International Symposium, CBSE 2007, Medford, MA, USA, July 9-11, 2007, Proceedings*, volume 4608 of *LNCS*. Springer, 2007. 2

[SG07] Muzammil Shahbaz and Roland Groz. Using invariant detection mechanism in black box inference. In *ISoLA Workshop on Leveraging Applications of Formal Methods*, 2007. 154

[Sha06] Muzammil Shahbaz. Incremental inference of black-box components to support integration testing. In *TAIC PART*, pages 71–74, 2006. 153

[SHL08] Guoqiang Shu, Yating Hsu, and David Lee. Detecting communication protocol security flaws by formal fuzz testing and machine learning. In *FORTE*, pages 299–304, 2008. 28, 36, 92, 157

[SL07] Guoqiang Shu and David Lee. Testing security properties of protocol implementations - a machine learning based approach. In *ICDCS '07: Proceedings of the 27th International Conference on Distributed Computing Systems*, page 25, Washington, DC, USA, 2007. IEEE Computer Society. 28, 31, 34, 35, 36, 37, 56, 57, 92

[SLG07a] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning and integration of parameterized components through testing. In *TestCom/FATES*, pages 319–334, 2007. 153

[SLG07b] Muzammil Shahbaz, Keqin Li, and Roland Groz. Learning parameterized state machine model for integration testing. In *COMPSAC*, pages 755–760, 2007. 153

[SLG08] Muzammil Shahbaz, Keqin Li, and Roland Groz. Reverse engineering enhanced state machine models for black box software. *Information Sciences. Special issue on Computational Intelligence and Machine Learning*, 2008. Submitted. 155

[SPK07] Muzammil Shahbaz, Benoît Parreaux, and Francis Klay. Model inference approach for detecting feature interactions in integrated systems. In *ICFI*, pages 161–171, 2007. 154

[Szy02] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 2

[Tip95] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995. 4

[UPn] UPnP Forum. http://www.upnp.org. 147

[Val84] L. G. Valiant. A theory of the learnable. In *STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 436–445, New York, NY, USA, 1984. ACM Press. 24

[Vas73] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics and Systems Analysis*, 9:653–665, 1973. 28, 92, 131

[Wat94] Osamu Watanabe. A framework for polynomial-time query learnability. *Math. Syst. Theory*, 27(3):211–229, 1994. 41

[WBH06] Neil Walkinshaw, Kirill Bogdanov, and Mike Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006. 4, 23

[WBHS07] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 209–218. IEEE Computer Society, 2007. 32, 36

[WCO03] Ye Wu, Mei-Hwa Chen, and Jeff Offutt. Uml-based integration testing for component-based software. In *ICCBSS '03: Proceedings of the Second International Conference on COTS-Based Software Systems*, pages 251–260, London, UK, 2003. Springer-Verlag. 4

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982. 37

[WPC01] Ye Wu, Dai Pan, and Mei-Hwa Chen. Techniques for testing component-based software. *iceccs*, 00:0222, 2001. 4, 23

[XN03] Tao Xie and David Notkin. Exploiting synergy between testing and inferred partial specifications. In *In WODA*, pages 17–20, 2003. 8

[Yok94] T. Yokomori. Learning non-deterministic finite automata from queries and counterexamples. In *Machine Intelligence*, pages 169–189. Oxford University Press, Inc., 1994. 156