HAL
open science

Analyse et optimisation d'algorithmes pour l'inférence de modèles de composants logiciels

Muhammad Naeem Irfan

To cite this version:

Muhammad Naeem Irfan. Analyse et optimisation d'algorithmes pour l'inférence de modèles de composants logiciels. Apprentissage [cs.LG]. Université de Grenoble, 2012. Français. NNT : 2012GRENM072 . tel-00767894

HAL Id: tel-00767894

https://theses.hal.science/tel-00767894v1

Submitted on 20 Dec 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Muhammad Naeem IRFAN

Thèse dirigée par **Professeur Roland GROZ**
et co-encadrée par **Docteur Catherine ORIAT**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analysis and optimization of software model inference algorithms

Thèse soutenue publiquement le **19 Septembre 2012**
devant le jury composé de :

Mr Eric GAUSSIER
Professeur, Université Joseph Fourier Grenoble, Président
Mr Alexandre PETRENKO
Chercheur principal CRIM (Canada), Rapporteur
Mr Colin DE LA HIGUERA
Professeur, Université de Nantes, Rapporteur
Mr Frédéric DADEAU
MCF, Université de Franche-Comté Besançon, Examinateur
Mr Roland GROZ
Professeur, Grenoble INP, Directeur de thèse
Mme Catherine ORIAT
MCF, Grenoble INP, Co-encadrante de thèse

Acknowledgments

During my thesis, I continuously benefited from the guidance and expertise of my thesis advisor Professor Roland Groz. I owe my deepest gratitude to him for his critical feedback on my research work. His clear research vision helped me to achieve my research objectives. I am grateful to him for his moral and technical support. He was always there to assist me in all fields of life. I cannot forget the love and affection that I received from his very caring wife Bénédicte and lovely children. They invited me every summer to their home at Lannion to enjoy a part of my vacations with them.

I am also grateful to my co-advisor, Dr. Catherine Oriat, for her nice, very polite and constructive feedback. It has been a real pleasure to work with her and learning things in a very friendly environment.

I am fortunate to have worked with such advisors who were always a continuous source of energy and guidance. It was not possible to complete my PhD task without their support.

I am greatly indebted to my thesis reporters especially Professor Alexandre Petrenko, team director at CRIM Canada, for the useful comments that helped me to improve my research work presentation. The brain storming sessions and discussions with him helped me to improve my understanding of the finite state machine inference algorithms.

Résumé

Les Components-Off-The-Shelf (COTS) sont utilisés pour le développement rapide et efficace de logiciels tout en limitant le coût. Il est important de tester le fonctionnement des composants dans le nouvel environnement. Pour les logiciels tiers, le code source des composants, les spécifications et les modèles complets ne sont pas disponibles. Dans la littérature de tels systèmes sont appelés composants "boîte noire". Nous pouvons vérifier leur fonctionnement avec des tests en boîte noire tels que le test de non-régression, le test aléatoire ou le test à partir de modèles. Pour ce dernier, un modèle qui représente le comportement attendu du système sous test (SUT) est nécessaire. Ce modèle contient un ensemble d'entrées, le comportement du SUT après stimulation par ces entrées et l'état dans lequel le système se trouve.

Pour les systèmes en boîte noire, les modèles peuvent être extraits à partir des traces d'exécutions, des caractéristiques disponibles ou encore des connaissances des experts. Ces modèles permettent ensuite d'orienter le test de ces systèmes. Les techniques d'inférence de modèles permettent d'extraire une information structurelle et comportementale d'une application et de la présenter sous forme d'un modèle formel. Le modèle abstrait appris est donc cohérent avec le comportement du logiciel. Cependant, les modèles appris sont rarement complets et il est difficile de calculer le nombre de tests nécessaires pour apprendre de façon complète et précise un modèle.

Cette thèse propose une analyse et des améliorations de la version Mealy de l'algorithme d'inférence L* [Angluin 87]. Elle vise à réduire le nombre de tests nécessaires pour apprendre des modèles. La version Mealy de L* nécessite d'utiliser deux types de test. Le premier type consiste à construire les modèles à partir des sorties du système, tandis que le second est utilisé pour tester l'exactitude des modèles obtenus. L'algorithme utilise ce que l'on appelle une table d'observation pour enregistrer les réponses du système.

Le traitement d'un contre-exemple peut exiger d'envoyer un nombre conséquent de requêtes au système. Cette thèse aborde ce problème et propose une technique qui traite les contre-exemples de façon efficace. Nous observons aussi que l'apprentissage d'un modèle ne nécessite pas de devoir remplir complètement ces tables. Nous proposons donc un algorithme d'apprentissage qui évite de demander ces requêtes superflues.

Dans certains cas, pour apprendre un modèle, la recherche de contre-exemples peut coûter cher. Nous proposons une méthode qui apprend des modèles sans demander et traiter des contre-exemples. Cela peut ajouter de nombreuses colonnes à la table d'observation mais au final, nous n'avons pas besoin d'envoyer toutes les requêtes. Cette technique ne demande que les requêtes nécessaires.

Ces contributions réduisent le nombre de tests nécessaires pour apprendre des modèles de logiciels, améliorant ainsi la complexité dans le pire cas. Nous présentons les extensions que nous avons apportées à l'outil RALT pour mettre en oeuvre ces algorithmes. Elles sont ensuite validées avec des exemples tels que les tampons, les distributeurs automatiques, les protocoles d'exclusion mutuelle et les planificateurs.

Abstract

Components-Off-The-Shelf (COTS) are used for rapid and cost effective development of software systems. It is important to test the correct functioning of COTS in new environment. For third party software components source code, complete specifications and models are not available. In literature such systems are referred as black box software components. Their proper functioning in new environment can be tested with black box testing techniques like, comparison testing, fuzz testing, Model based testing. For Model based software testing, software models are required, which represent the desired behavior of a system under test (SUT). A software model shows that a certain set of inputs are applicable to the SUT and how it behaves when these inputs are applied under different circumstances.

For software black box systems, models can be learned from behavioral traces, available specifications, knowledge of experts and other such sources. The software models steer the testing of software systems. The model inference algorithms extract structural and design information of a software system and present it as a formal model. The learned abstract software model is consistent with the behavior of the particular software system. However, the learned models are rarely complete and it is difficult to calculate the number of tests required to learn precise and complete model of a software system.

The thesis provides analysis and improvements on the Mealy adaptation of the model inference algorithm L* [Angluin 87]. It targets at reducing the number of tests required to learn models of software systems. The Mealy adaptation of the algorithm L* requires learning models by asking two types of tests. First type of tests are asked to construct models i.e. *output queries*, whereas the second type is used to test the correctness of these models i.e. *counterexamples*. The algorithm uses an observation table to record the answers of output queries.

Processing a counterexample may require a lot of output queries. The thesis addresses this problem and proposes a technique which processes the counterexamples efficiently. We observe that while learning the models of software systems asking output queries for all of the observation table rows and columns is not required. We propose a learning algorithm that avoids asking output queries for such observation table rows and columns.

In some cases to learn a software model, searching for counterexamples may go very expensive. We have presented a technique which learns the software models without asking and processing counterexamples. But this may add many columns to the observation table and in reality we may not require to ask output queries for all of the table cells. This technique asks output queries by targeting to avoid asking output queries for such cells.

These contributions reduce the number of tests required to learn software models, thus improving the worst case learning complexity. We present the tool RALT which implements our techniques and the techniques are validated by inferring the examples like buffers, vending machines, mutual exclusion protocols and schedulers.

Table of Contents

1 Introduction **1**
1.1 Software Testing . . . 1
1.2 Software Models from Implementations . . . 2
1.2.1 Poor Equivalence Oracle . . . 4
1.2.2 Large Input Set . . . 5
1.2.3 Useless Queries . . . 5
1.3 Thesis Outline . . . 5

2 Definitions and Notations **7**
2.1 General Notations . . . 7
2.2 Finite State Machines . . . 8
2.2.1 Deterministic Finite Automaton . . . 8
2.2.2 Mealy Machine . . . 9

3 State of the Art **11**
3.1 Model Inference . . . 12
3.2 Passive Learning . . . 13
3.2.1 Passive Learning Approaches in General . . . 13
3.2.2 Inferring Models of Software Processes . . . 13
3.2.3 Generating Software Behavioral Models . . . 14
3.2.4 Exact *DFA* Identification . . . 15
3.2.5 Framework for Evaluating Passive Learning Techniques . . . 16
3.3 Active Learning . . . 17
3.3.1 Overview of the Learning Algorithm L^* . . . 18
3.3.2 Searching for Counterexamples . . . 19
3.3.3 Processing Counterexamples . . . 19
3.3.4 Executing new Experiments in Canonic Order . . . 20
3.3.5 Tables with Holes . . . 21
3.3.6 Optimized Learning with the Algorithm L^* . . . 23
3.3.7 Learning from Membership Queries Alone . . . 25
3.3.8 Learning from Counterexamples Alone . . . 25
3.3.9 Mealy Adaptation of the Algorithm L^* . . . 26
3.3.10 Optimized Mealy Inference . . . 26
3.3.11 Learning NFA Models . . . 27
3.3.12 Framework for Evaluating Active Learning Techniques . . . 28
3.4 Applications of Model Inference . . . 29
3.4.1 Specification Mining . . . 30
3.4.2 Integration Testing . . . 31
3.4.3 Dynamic Testing . . . 33
3.4.4 Communication Protocol Entities Models . . . 34

3.4.5 Security Testing . . . 35
3.5 Problem Statement . . . 36
3.6 Contributions . . . 38
3.6.1 Optimized Counterexample Processing Method . . . 38
3.6.2 Optimized Mealy Inference Algorithm . . . 39
3.6.3 Organizing Output Queries Calculation . . . 39
3.6.4 Mealy Inference Without Using Counterexamples . . . 40

4 Model Inference with the Algorithm L^* 43
4.1 The Learning Algorithm L^* . . . 43
4.1.1 Observation Table for Learning Algorithm L^* . . . 44
4.1.2 The Algorithm L^* . . . 46
4.1.3 Complexity of L^* . . . 47
4.2 *DFA* Inference of Mealy Machines and Possible Optimizations . . . 47
4.2.1 Prefix Closure . . . 48
4.2.2 Input Determinism . . . 48
4.2.3 Independence of Events . . . 49
4.3 Mealy Inference . . . 50
4.3.1 The Mealy Inference Algorithm ${L_M}^*$ . . . 51
4.4 Conclusion . . . 55

5 Searching and Processing Counterexamples 57
5.1 Searching for Counterexamples . . . 58
5.1.1 Counterexamples Search by Random Sampling . . . 59
5.1.2 Howar Algorithm for Counterexample Search . . . 59
5.1.3 Balle Algorithm for Counterexamples Search . . . 60
5.2 Processing Counterexamples . . . 60
5.2.1 Counterexample Processing Algorithm by Angluin . . . 61
5.2.2 Counterexample Processing Algorithm by Rivest and Schapire 63
5.2.3 Counterexample Processing Algorithm by Maler and Pnueli . 65
5.2.4 Counterexample Processing Algorithm by Shahbaz and Groz 68
5.2.5 Issue with Rivest and Schapire Algorithm . . . 69
5.3 The Improved Counterexample Processing Algorithm . . . 72
5.3.1 Motivation for Improved Counterexample Processing Algorithm 72
5.3.2 Counterexample Processing Algorithm Suffix1by1 . . . 73
5.3.3 Example for Processing Counterexamples with Suffix1by1 Algorithm . . . 73
5.3.4 Complexity . . . 74
5.3.5 Complexity Comparison . . . 75
5.4 Experiments to Analyze Practical Complexity of Suffix1by1 . . . 76
5.4.1 CWB Examples . . . 76
5.4.2 Random Machines . . . 78
5.5 Conclusion . . . 80

6 Improved Model Inference **83**
6.1 Motivation for the L_1 Algorithm . . . 83
6.2 Improved Mealy Inference Algorithm . . . 84
6.2.1 Observation Table . . . 85
6.2.2 The L_1 Algorithm . . . 87
6.2.3 Example for learning with L_1 . . . 88
6.2.4 Complexity of the L_1 Algorithm . . . 90
6.3 Inferring the HVAC controller . . . 90
6.3.1 Description of the HVAC controller . . . 91
6.3.2 Inference of the HVAC controller with the $L_M{}^*$ Algorithm . . . 91
6.3.3 Inference of the HVAC controller with the L_1 Algorithm . . . 92
6.4 Experiments to Analyze Practical Complexity of L_1 . . . 97
6.5 Conclusion . . . 99

7 Mealy Inference without Using Counterexamples **101**
7.1 Organization of Output Queries . . . 102
7.1.1 Output Queries and Dictionaries . . . 102
7.1.2 Motivation . . . 103
7.1.3 Improved Heuristic . . . 103
7.2 Motivation . . . 103
7.3 Learning without Counterexamples . . . 104
7.3.1 The GoodSplit Algorithm . . . 104
7.3.2 Termination Criteria for the GoodSplit Algorithm . . . 106
7.3.3 Greedy Choice for the GoodSplit Algorithm . . . 107
7.4 Mealy Adaptation of the GoodSplit Algorithm . . . 107
7.4.1 Observation Table . . . 108
7.4.2 The $L_{M\text{-}GS}$ Algorithm . . . 110
7.4.3 Example for learning with $L_{M\text{-}GS}$ Algorithm . . . 112
7.4.4 Complexity . . . 117
7.5 Discussion . . . 117
7.6 Conclusion . . . 118

8 Tool and Case Studies **119**
8.1 RALT . . . 119
8.1.1 Test Drivers . . . 120
8.1.2 Learning Platform . . . 120
8.1.3 Learner . . . 120
8.2 Case Studies . . . 122
8.2.1 Random Machine Generator . . . 122
8.2.2 Edinburgh Concurrency Workbench (CWB Examples) . . . 122
8.2.3 HVAC controller . . . 122
8.2.4 Coffee Machine . . . 123
8.3 Conclusion . . . 129

9 Conclusion and Perspectives **131**
9.1 Summary 131
9.2 Publications 132
9.3 Perspectives 135
9.3.1 Optimizing and extending the $L_{M\text{-}GS}$ Algorithm 135
9.3.2 Generic Test driver for RALT 136
9.3.3 Learning non-deterministic Machines 136
9.3.4 Addressing non-deterministic values 136
9.3.5 Improving Partial Models of Implementations 136

Bibliography **139**

List of Figures

1.1 Learning Framework 4

2.1 Deterministic Finite Automaton 8
2.2 Mealy Machine 9

3.1 Learning with the *Depth-M** algorithm 20
3.2 Learning with the *Breadth-M** algorithm 21
3.3 The conjecture from Table 3.4 22
3.4 The conjecture from Table 3.5 23
3.5 Initial *DFAs* M_1^1 and M_1^0 for $\Sigma = \{0, 1\}$. State q_ϵ is accepting in M_1^1 and non-accepting in M_1^0. 25
3.6 ZULU challenge results. 29
3.7 Learning and testing approach for integrated systems. 32

4.1 Mealy machine. 50
4.2 The Mealy machine conjecture $M_M{}'$ from Table 4.2 54

5.1 Conjectured Mealy machine. 63
5.2 Mealy machine with $I = \{a, b\}$. 70
5.3 The Mealy machine conjecture $Conj'$ from Table 5.5 71
5.4 The Mealy machine conjecture $Conj''$ from Table 5.6b 72
5.5 $|I|\in\{2, 3 \ldots 8\}$,$|O|$=7 and $|Q| = 40$ 79
5.6 $|I|$=2,$|O|$=2 and $|Q|\in\{3, 4 \ldots 40\}$ 80

6.1 The Mealy machine conjecture $Conj_1$ from Table 6.1 89
6.2 HVAC controller 91
6.3 Mealy Machine conjecture of the HVAC Controller 93
6.4 The Mealy machine conjecture $Conj_{hvac_1}$ from Table 6.4 94
6.5 The Mealy machine conjecture $Conj_{hvac_2}$ from Table 6.5c 95
6.6 $|I|\in\{2, 3, \ldots 10\}$ and $|O|$=5,n=40 98
6.7 $|I|$=5,$|O|$=7 and $n\in\{3, 4 \ldots 40\}$ 99

8.1 Rich Automata Learning and Testing Framework 120
8.2 Detailed Learner Framework 121
8.3 Mealy Machine conjecture of the Coffee Machine 123
8.4 Mealy Machine conjecture of the Coffee Machine 124
8.5 The Mealy machine conjecture $Conj_{coffee_1}$ from Table 8.2 125
8.6 The Mealy machine conjecture $Conj_{coffee_2}$ from Table 8.3d 127

List of Tables

3.1 5 problems to solve in each cell 17
3.2 Cell winners and best challengers 17
3.3 An observation table for a machine with $\Sigma = \{a, b\}$ 18
3.4 An incomplete table 22
3.5 By moving the row aba to S 23

4.1 Example for the observation table (S, E, T) 45
4.2 Initial observation table for the Mealy machine in Figure 4.1 52

5.1 The observation table after adding prefixes of CE 62
(a) The observation table after adding prefixes of CE. 62
(b) To make the observation table compatible ab is added to E_M. 62
5.2 The observation table after adding the distinguishing suffix $v_{j+1} = ab$ 66
(a) Observation table after adding the distinguishing suffix $v_{j+1} = ab$. 66
(b) Make the observation table closed by moving the row a to S_M. 66
(c) Make the observation table closed by moving aa to S_M. 66
5.3 The observation table after adding suffixes of CE 67
(a) Observation table after adding CE and its suffixes. 67
(b) Make the observation table closed by moving the row a to S_M. 67
(c) Make the observation table closed by moving aa to S_M. 67
5.4 The observation table after adding suffixes of v 70
(a) Observation table after adding suffixes of v. 70
(b) Make the observation table closed by moving the row a to S_M. 70
(c) Make the observation table closed by moving aa to S_M. 70
5.5 Initial Observation Table for the Mealy machine in Figure 5.2 71
5.6 The underlined rows are the rows which make the table not closed. The observation table in Table 5.6b is closed. 72
(a) Observation Table after adding distinguishing string $aaab$ . . . 72
(b) To make the table closed moving the row a to S_M. 72
5.7 The observation table after adding suffixes of CE 75
(a) The observation table after adding the suffix ab. 75
(b) Make the observation table closed by moving the row a to S_M. 75
(c) Make the table closed by moving the row aa to S_M. 75
5.8 Number of output queries required 77
5.9 Number of counterexamples required 78

6.1 Initial table for mealy inference with L_1 of machine in Figure 4.1 . . 86
(a) The observation table after adding the suffix b. 90
(b) The observation table after adding the suffix ab. 90

(c) Make the observation table closed by moving the row a to S. . 90
(d) Make the table closed by moving the row aa to S. 90
6.3 The HVAC controller's inference with ${L_M}^*$ 92
6.4 Initial Observation Table for Mealy inference with L_1 of the HVAC controller 93
(a) The observation table after adding the suffix $T5$. 94
(b) Make the observation table closed by moving the row ON to S. 94
(c) Make the observation table closed by moving the row $ON \cdot T25$ to S. 94
(a) The observation table after adding the suffix $T25$ to E. 96
(b) Make the observation table closed by moving the row $ON \cdot T5$ to S. 96

7.1 Initial observation table for Mealy machine in Figure 2.2 108
7.2 Observation table after calculating the output queries $a \cdot a$ and $b \cdot b$. 113
7.3 The observation table after adding the answer of the output query $a \cdot b$ 113
7.4 Adding the suffix aa to the observation table 113
7.5 Unanswered output queries 114
7.6 Calculating the output query $b \cdot aa$ 114
7.7 Adding the suffix ab to the observation table 114
7.8 Adding the output for the output query $a \cdot ab$ 114
7.9 Moving the row a to S 115
7.10 Moving the row aa to S 115
7.11 Adding the output for the output query $b \cdot ab$ 115
7.12 Adding the output for the output queries $ab \cdot a$ and $ab \cdot ab$ 116
7.13 Calculating the output query $aaa \cdot b$ and updating cells ($a \cdot aa$, $aa \cdot a$, $aa \cdot ab$,$aaa \cdot b$) 116
7.14 Adding the output for the output queries $aab \cdot ab$ and $aab \cdot b$ 117

8.1 The observation table for model inference with the complex structure algorithm 124
8.2 Initial Observation Table of the coffee machine for Mealy inference with the L_1 algorithm. 125
(a) The observation table after adding the suffix *button* and $pad \cdot button$. 126
(b) Make the observation table closed by moving the row *water* to S. 126
(c) Make the observation table closed by moving the row $water \cdot pad$ to S. 126
(d) Make the observation table closed by moving the row $water \cdot pad \cdot button$ to S. 126
(a) The observation table after adding the suffix $water \cdot button$ to E. 128
(b) Make the observation table closed by moving the row pad to S. 128

Chapter 1

Introduction

Contents

1.1 Software Testing **1**

1.2 Software Models from Implementations **2**

1.2.1 Poor Equivalence Oracle 4

1.2.2 Large Input Set 5

1.2.3 Useless Queries 5

1.3 Thesis Outline **5**

This chapter provides a general introduction to learning and testing. The first section provides a discussion on software testing and its purpose. It also contains a short description of black box testing and model based testing. The second section presents an overview for inferring models from implementations and their usage for testing. The final section contains an outline of the thesis.

1.1 Software Testing

Software systems are an important part of current mode of life and our dependency on their proper functioning is continuously increasing with the passage of time. They are getting used in safety critical applications (like medical devices and nuclear plants), where their failure can result in extensive damage. This makes it essential to develop methods that can ensure proper functioning of software systems. Testing [Myers 2004] is an approach that enhances the reliability of a software system. It is a process designed to ensure that a software application does what it is developed to do and it does not do anything surprising or unpredictable [Myers 2004]. Since testing helps to obtain reliable systems, it is used in all sectors that deal with software development.

Testing is a primary technique to show that errors are not present in a software. Its purpose is to show that a software system performs its intended functions correctly [Myers 2004]. It involves systematically checking the correctness of a system. Software testing usually involves interacting with a system under test (SUT) by providing it the inputs and observing the outputs to find errors. However, testing for all errors or estimating the trustworthiness of a software system is difficult [Parnas 1990]. The correctness criteria for a software is derived from the

software specifications. The specifications describe the functionality of a software system, consequently, they are taken as the basis for software testing.

Software testing can be based on formal methods [Hierons 2008] or formal models [Broy 2005]. Ideally it should be completely automatic, but practically it requires human involvement. At least one manual activity is to analyze the testing results. An example where a software fails, is provided in the case of negative results. This helps the engineers to find out the error prone part of a software system. Testing is an important phase in software development but it is difficult and expensive.

Software testing is often applied to the actual SUT to test against some specifications, without using the knowledge of its internal structure. Even if the internal structure of the SUT is accessible, usually it is not used as it may lead to a biased testing process. We may require to check few aspects of a whole very large system. Extracting a part of code that corresponds to a functionality may be impracticable in case of legacy systems. However, comparing the SUT with an abstract model to check for some functionality is a feasible option. Moore initiated the black box testing in his classical paper in 1956 [Moore 1956], where he pointed to several problems including the model identification (automata inference of unknown black box automaton).

Model based software testing requires software models which are used to represent the desired behavior of a SUT. The software models show that a certain set of inputs is applicable to the SUT and how it behaves when these inputs are applied under different circumstances. The models are presented as finite state machines that are usually an abstract, partial presentation of the desired behavior of the SUT. In model based testing the software models are used to derive test cases [Bertolino 2007]. Test cases can be generated from models using various graph traversal algorithms, which are at same level of abstraction as the models. All of the abstract test cases together can be named as a test suite. The test suite directly cannot be tested on the SUT because of the different level of abstraction. The test suite is mapped to concrete tests that can communicate with the SUT [Utting 2007, Li 2012].

We have many methods that can be applied to models, to derive tests [Apfelbaum 1997, Dalal 1999, Utting 2007, Meinke 2011b, Meinke 2011a]. Since a test suite is not constructed from source code but models of the SUT, model based testing is considered as a type of black box testing. The models can also be combined with source code to derive tests which is named as gray box testing [Kicillof 2007].

1.2 Software Models from Implementations

Software industry is focusing on to develop software systems using Components-Off-The-Shelf (COTS). The target is to develop new systems by assembling existing components. A new functional system is obtained with little effort which makes possible to develop systems with reduced cost. The focus is continuously shifting on evaluation, testing and integration of third party components. The source code, complete specifications

and models for COTS are not available and in literature such systems are referred as black box software systems. The COTS can be evaluated and tested with black box testing techniques like, comparison testing, fuzz testing, Model based testing. For Model based software testing, software models are required, which represent the desired behavior of a SUT.

Generally, testing is done on a subset of all possible system behaviors, so it can find the presence of errors but not their absence. The field of combining model learning and testing is getting momentum with the passage of time. Automatically learning and verifying software models has become a prominent approach for testing software systems. When software models are not available for software black box components, models can be learned from the interactions with components, available specifications, knowledge of the experts and other such sources. The software models help to steer the testing and evaluate the test results.

As a result, software model inference techniques have become popular in the area of software testing. Inferring a software model is an incremental approach which requires to take into consideration many aspects of the system under inference (SUI). A lot of behavioral information can be reused for future inference. The very first thing to infer a model of a software system, is to capture its state. At a particular instant of time, a state is an instantaneous description of values of system variables. It is also required to know that in a response to some action of the system, how the state of the system will change. The change is described by providing the state before the occurrence of an action and the state afterwards. The change in the software system from one state to another state is called a transition of the system. The model of a software system can be presented in terms of states and transitions. An initial state of a software model is a snapshot of the system that captures the initial values of the system variables. A software system can be modeled as a finite set of states where each state is obtained by transition from the initial state or a subsequent state.

Model learning from interactions with software components is also known as black box model inference. Many contributions on the subject are available in the literature [Gold 1967, Gold 1972, Biermann 1972, Trakhtenbrot 1973, Kearns 1994, Shu 2007, Angluin 1987, de la Higuera 2010, Steffen 2011]. For model inference, we use the L^* algorithm by Angluin [Angluin 1987]. We can find many variants of L^* in the literature [Rivest 1993, Maler 1995, Balcázar 1997, Hungar 2003b].

The algorithm L^* permits to infer models (finite state machines) by continuously interacting with black box systems or from their behavioral traces obtained by interacting with them. The interactions with software implementations are in fact the tests that are conducted to uncover the models of software implementations. The L^* algorithm assumes that there exists a *minimally adequate teacher* (MAT). MAT is assumed to answer two types of questions named *membership queries* and *equivalence queries*. The membership queries are strings on Σ and MAT replies "yes" if a membership query belongs to a target model and "no", otherwise. The equivalence queries are conjectured models. The algorithm assumes that in case of incorrect conjecture, MAT always comes up with a counterexample. In testing context, the

algorithm can be seen as a learner, teacher and oracle as depicted in Figure 1.1.

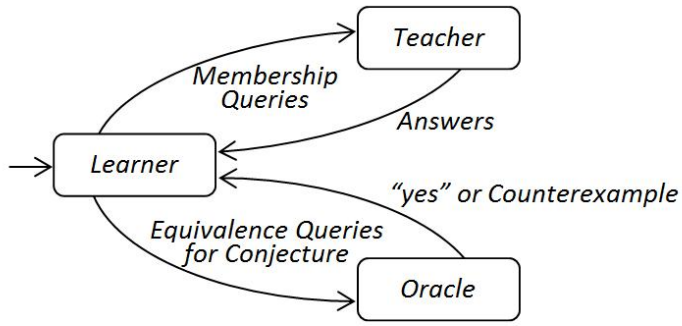

Figure 1.1: Learning Framework

The learner's job is to learn the observations and come up with a software model. The teacher answers the behavior of a particular string in the software implementation and the oracle replies whether the model conjectured by the learner is correct or not. It is worth mentioning that the goal of applying model inference algorithm is not only to discover the unknown behavior of a software system but is also to find out the errors by executing the tests. This can also be seen as learning and testing. For the learning phase membership queries are used to conjecture models of a software system, whereas equivalence queries are used for testing, by comparing respective conjectured model and the actual system. This process is repeated until a valid software model is conjectured.

The model inference technique extracts structural and design information of a software system and presents it as a formal model. The learned abstract software model is consistent with the behavior of the particular software system. However, the learned models are rarely complete and it is difficult to calculate the number of tests required to learn precise and complete model of a software black box system. The challenge is to find the best trade-off between precision and cost [Bertolino 2007].

The main problems that we address in this thesis are the following.

1.2.1 Poor Equivalence Oracle

To test a software system, after learning its model, the L^* algorithm requires to get it validated from an oracle (equivalence oracle). For software implementations such an oracle does not exist and this deficiency is alleviated by *random sampling oracle*. The random sampling oracle is an implementation which explores a learned model and a target system to search for discrepancies. This may result in increased number of attempts to find a discrepancy and the discrepancy trace may contain useless sequences which increases the number of queries required to be executed to learn a software model.

1.2.2 Large Input Set

Real world software systems operate on large input test sets. Most of the adaptations of L^* use the input set to initialize the observation table rows and columns. The initialization with large input set results in increased number of queries.

1.2.3 Useless Queries

The answer to query of every cell of an observation table and in some cases to queries for the rows of the observation table are not required. The learning job can be accomplished without asking all queries.

1.3 Thesis Outline

The outline of the following chapters is provided as follows.

- Chapter 2 provides the definitions and notations that are used through out the thesis manuscript. It also presents the models that we have used for software model inference.

State of the Art

- Chapter 3 provides state-of-the-art for related work and summary of contributions.
- Chapter 4 provides the existing model inference algorithm L^*, possible optimizations to learn Mealy models with L^*, and adaptation of L^* to infer Mealy models.
- Section 5.1 and Section 5.2 of Chapter 5 provide the existing methods to search and process counterexamples, respectively.

Contributions

- Section 5.3 of Chapter 5 provide the improved method to process counterexamples. The last section 5.4 provide the experimental results conducted to analyze the practical complexity.
- Chapter 6 provides the optimized Mealy inference algorithm L_1. Complexity discussion for the L_1 algorithm and experiments to evaluate its practical complexity.
- Chapter 7 provides the technique for organization of output queries and the Mealy inference without using counterexamples.
- Chapter 8 provides the tool and case studies that we have implemented to validate the algorithms.

Conclusion

- Chapter 9 provides the conclusion and future directions.

Chapter 2

Definitions and Notations

Contents

2.1 General Notations **7**

2.2 Finite State Machines **8**

2.2.1 Deterministic Finite Automaton 8

2.2.2 Mealy Machine 9

This chapter provides the definitions and notations that we use through out this manuscript. The first section provides the general notations that we use to present definitions and algorithms formally. The second section provides the models that we use to model the behavior of software applications.

2.1 General Notations

Let Σ be a finite set or alphabet for a Deterministic Finite Automaton DFA. The empty word is denoted by ϵ and the concatenation of two words s and e is expressed as $s \cdot e$ or se. The length of ω is the number of letters that a word ω contains and is denoted by $|\omega|$. If $\omega = u \cdot v$, then u and v are *prefix* and *suffix* of ω, respectively. Let Σ^+ is a sequence of letters constructed by taking one or more elements from Σ and Σ^* is a sequence of letters constructed by taking zero or more elements from Σ. Formally a word ω is a sequence of letters $a_1 a_2 \ldots a_n \in \Sigma^*$ and the empty word is denoted by ϵ. The finite set of words over Σ of length exactly n is denoted as Σ^n, where n is a positive integer and $\Sigma^0 = \{\epsilon\}$. The finite set of words over Σ of length at most n and less than n are denoted as $\Sigma^{\leq n}$ and $\Sigma^{<n}$, respectively. Let $\mathit{suffix}^j(\omega)$ denote the *suffix* of a word ω of length j and $\mathit{prefix}^j(\omega)$ denote the *prefix* of the word ω of length j, where $j \in \{1, 2, \ldots |\omega|\}$. Let $\mathit{prefixes}(\omega)$ denote the set of all prefixes of ω and $\mathit{suffixes}(\omega)$ denotes the set of all suffixes of ω. Let $\mathit{output}^j(\omega)$ denote the output for jth input symbol in ω. For example, if we have an output 0010 for a word $aaba$ then $\mathit{output}^3(aaba) = 1$. A set is *prefix closed iff* all the prefixes of every element of the set are also elements of the set. A set is *suffix closed iff* all the suffixes of every element of the set are also elements of the set. The cardinality of a set D is denoted by $|D|$.

2.2 Finite State Machines

We use finite state machines (FSM) to design models of software systems. It is conceived as an abstract machine that can be in one of a finite number of states. A state is an instantaneous description of a system that captures the values of the variables at a particular instant of time. The state in which it is in at any given time is called the current state. Typically a state is introduced when the system does not react the same way to the same trigger. It can change from one state to another when initiated by a triggering event or receiving an input, this is called a transition. A transition is a set of actions to be executed on receiving some input. A particular FSM is defined by a list of the possible transition states from each current state.

2.2.1 Deterministic Finite Automaton

The notion of DFA is formally defined as follows.

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

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

A DFA on input set $I = \{a, b\}$ is presented in Figure 2.1.

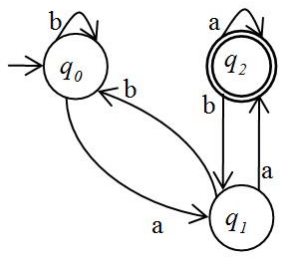

Figure 2.1: Deterministic Finite Automaton

Initially the deterministic finite automaton is in the initial state q_0. From a current state the automaton uses the transition function δ to determine the next state. It reads a letter or a word of letters and using δ, it identifies a state in *DFA*, which can be accepting or non-accepting. The transition function for a word ω is extended as $\delta(q_0, \omega) = \delta(\ldots\delta(\delta(q_0, a_1), a_2)\ldots, a_n)$. A word ω is accepted by *DFA iff* $\delta(q_0, \omega) \in F$. We define an output function $\Lambda : Q \times \Sigma^* \to \{0, 1\}$, where $\Lambda(q_0, \omega) = 1$, if $\delta(q_0, \omega) \in F$, and $\Lambda(q_0, \omega) = 0$, otherwise.

2.2.2 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 non-empty finite set of input symbols,*
- O *is the non-empty finite set of output symbols,*
- $\delta : Q \times I \rightarrow Q$ *is the transition function, which maps pairs of states and input symbols to the corresponding next states,*
- $\lambda : Q \times I \rightarrow O$ *is the output function, which maps pairs of states and input symbols to the corresponding output symbols.*

Moreover, we assume $dom(\delta) = dom(\lambda) = Q \times I$ *i.e. the Mealy machine is input enabled.*

For software systems all inputs may not be valid for every state. To make software systems input enabled, we introduce the output Ω for invalid inputs with transitions from the current state to itself. A Mealy machine on input set $I = \{a, b\}$ is presented in Figure 2.2.

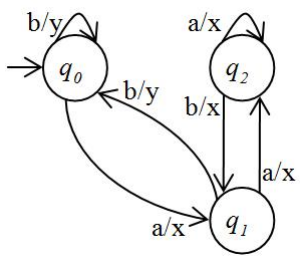

Figure 2.2: Mealy Machine

Initially the Mealy machine is in the initial state q_0. For a transition from a state $q \in Q$ an input $i \in I$ is read, output is produced by using the output function $\lambda(q, i)$ and the state reached is identified by using the transition function $\delta(q, i)$. When an input string ω composed of inputs $i_1 i_2 \ldots i_n \in I^+$ is provided to the Mealy machine, to calculate the state reached and output produced by the Mealy machine, both the transition and output functions are extended as follows. The transition function for ω is extended as $\delta(q_0, \omega) = \delta(\ldots \delta(\delta(q_0, i_1), i_2) \ldots, i_n)$. To extend output function for ω i.e. $\lambda(q_0, \omega)$, the transition function is used to identify the state reached and the output function is used to calculate the output. The process is started by calculating $\lambda(q_0, i_1)$ and $\delta(q_0, i_1)$ from the initial state q_0, this takes to the next

state $q_1 \in Q$. From q_1 output and next state are calculated by $\lambda(q_1, i_2)$ and $\delta(q_1, i_2)$, respectively. This process is continued until the state q_n is reached by $\lambda(q_{n-1}, i_n)$ and $\delta(q_{n-1}, i_n)$, and $i_1/o_1, i_2/o_2, \ldots i_n/o_n$ results in an output string $o_1 o_2 \ldots o_n$ of length n, where $n = |\omega|$. If the input string ω is provided to the Mealy machine and the state s_ω is reached then ω is the *access string* for s_ω. Let O^+ is a sequence of outputs constructed by taking one or more elements from O and O^* is a sequence of outputs constructed by taking zero or more elements from O.

Chapter 3

State of the Art

Contents

3.1 Model Inference **12**
3.2 Passive Learning **13**
3.2.1 Passive Learning Approaches in General 13
3.2.2 Inferring Models of Software Processes 13
3.2.3 Generating Software Behavioral Models 14
3.2.4 Exact *DFA* Identification 15
3.2.5 Framework for Evaluating Passive Learning Techniques . . . 16
3.3 Active Learning **17**
3.3.1 Overview of the Learning Algorithm L^* 18
3.3.2 Searching for Counterexamples 19
3.3.3 Processing Counterexamples 19
3.3.4 Executing new Experiments in Canonic Order 20
3.3.5 Tables with Holes 21
3.3.6 Optimized Learning with the Algorithm L^* 23
3.3.7 Learning from Membership Queries Alone 25
3.3.8 Learning from Counterexamples Alone 25
3.3.9 Mealy Adaptation of the Algorithm L^* 26
3.3.10 Optimized Mealy Inference 26
3.3.11 Learning NFA Models 27
3.3.12 Framework for Evaluating Active Learning Techniques 28
3.4 Applications of Model Inference **29**
3.4.1 Specification Mining 30
3.4.2 Integration Testing 31
3.4.3 Dynamic Testing 33
3.4.4 Communication Protocol Entities Models 34
3.4.5 Security Testing 35
3.5 Problem Statement **36**
3.6 Contributions **38**
3.6.1 Optimized Counterexample Processing Method 38
3.6.2 Optimized Mealy Inference Algorithm 39
3.6.3 Organizing Output Queries Calculation 39
3.6.4 Mealy Inference Without Using Counterexamples 40

This chapter provides a state-of-the-art review for model inference of black box implementations and possible optimizations for such techniques. It begins with a brief introduction to model inference and testing. Then, it briefly presents the passive model learning techniques and goes on to present active learning model inference algorithm L^* [Angluin 1987].

The algorithm L^* learns the software models by continuously posing membership queries to a teacher (software black box implementation). Once a model is learned, the algorithm requires to search for *counterexamples*. A counterexample *CE* is a string on the inputs set which is accepted by the black box and refused by the conjectured model or vice-versa. If *CE* is such a smallest sequence then the counterexample is *optimal* and *non-optimal*, otherwise. For software applications, the default method to search for counterexamples is random sampling, however, counterexample search can be improved by increasing the probability to reach unexplored states in black box models [Howar 2010, Balle 2010]. An appropriate method selected to process counterexamples can significantly reduce the number of membership queries required to learn the black box models [Rivest 1993, Shahbaz 2009]. The order to calculate membership queries is also important, calculating them in a different order exhibits a different behavior [Garcia 2010]. When a limited number of membership queries is allowed, then one may require to conjecture models from sparse observations [de la Higuera 2010]. The models can be conjectured from such observations by predicting some answers [Eisenstat 2010].

The algorithm L^* uses membership and equivalence queries to learn the models but models can be learned from membership or equivalence queries alone [Ibarra 1991, Birkendorf 1998, Eisenstat 2010]. The Mealy adaptation of L^* requires to ask *output queries* instead of membership queries [Niese 2003]. The answers to output queries are output strings. The Mealy adaptation of the algorithm L^* requires less output queries and the number can be further reduced by introducing optimization filters [Niese 2003]. Model inference has plenty of applications like program analysis [Walkinshaw 2008], software testing [Aarts 2010b], security testing [Cho 2010], dynamic testing [Raffelt 2009], and integration testing [Shahbaz 2008].

3.1 Model Inference

Software model inference from implementations has become an important field in software testing [Raffelt 2009, Meinke 2010]. It has been used for the verification and testing of black box components [Niese 2003, Shahbaz 2008, Shu 2007]. It can be used to find the behavior of software systems where specifications are missing [Muccini 2007, Peled 1999]. These techniques ease the use of third party components by providing unit and integration testing [Shahbaz 2008]. Model inference approaches can mainly be divided into active and passive learning. Passive learning algorithms construct models from a fixed set of positive and negative examples, whereas active learning algorithms iteratively ask new queries during the model con-

struction process. In the following sections, we discuss passive and active automata learning techniques in turn.

3.2 Passive Learning

The passive learning algorithms learn the models from the provided set of positive traces, i.e. the traces which belong to the language of the unknown model $L(DFA)$ and negative traces, i.e. the set of traces which do not belong to the target model language or belong to $L(DFA)^C$ (complement of the target language). The technique restricts learning from the given data. These algorithms cannot perform tests by creating new tests and it is difficult to analyze how far one may be from the solution.

3.2.1 Passive Learning Approaches in General

For model inference, the passive model inference techniques [Biermann 1972, Cook 1998, Lorenzoli 2008] assume that sufficient positive and negative traces about the behavior of target system are available. From a provided set of positive and negative traces, it is not guaranteed that exact models will be inferred. The notion of "complete" sample varies for different algorithms and domains. The de facto notion of a complete sample was established by Oncina *et al.* [Oncina 1992] in their work on the Regular Positive Negative Inference algorithm. To ensure learning accurate models Dupont *et al.* [Dupont 1994] showed that the sets of positive and negative traces would have to be structurally complete and characteristic of the target model. In other words, one would need to contain a succinctly diverse set of samples to cover every state transition and contain enough information in positive traces to distinguish every pair of non-equivalent states. The problem is, characteristic samples tend to be vast, whereas in a practical software engineering setting, it tends to be the case that we have only a small subset of program execution traces or scenarios. The challenge is to generate a model that is reasonably accurate, even if the given set of samples is sparse.

3.2.2 Inferring Models of Software Processes

A *software process* is a set of activities applied to artifacts, leading to the design, development, or maintenance of a software system. Software process examples are designing methods, change or control procedures, and testing techniques. The motive behind software process is to coordinate different activities so that they achieve the same common goal. Managing and improving a software process is a challenge. That is the reason methods and tools have been devised support various aspects of the software process.

Cook *et al.* [Cook 1998] developed a technique termed as process discovery for the analysis of the data describing the process events. They present three methods named as *RNet*, *Ktail* and *Markov*, which range from purely algorithmic to purely statistical. They developed Markov method specifically for process discovery and

adopted the other two methods from different domains. They automatically derive the formal model of a process from basic event data collected on the process [Cook 1995]. Their approach views the process discovery as one of the grammatical inference problem. The data describing the behavior of the process are the sentences of some language and the grammar of that language is the formal model of the process. Cook *et al.* [Cook 1996] implemented the proposed methods in a tool operating on process data sets. Although the proposed methods are automated, they still require the guidance from the process engineers who are familiar with the considered process. The process engineers guide the methods for tuning the parameters built into them and for the selection and application of event data. The process models produced by the developed methods are initial models that can be refined by the process engineers. The methods are developed for the model inference of software processes, however, they can be applied to other processes and behaviors. The methods were successful to discover the interprocess communication protocols at operating system level.

3.2.3 Generating Software Behavioral Models

One can learn the model for constraints on the data for the functional behavior as boolean expressions [Ernst 2001] and the component interactions as the state machines [Biermann 1972, Cook 1998]. Both types of models are important for testing and verifying different aspects of the software behavior. However, this type of models does not capture the interplay between data values and component interactions. To address this problem Lorenzoli *et al.* [Lorenzoli 2008] presented the *GK-tail* algorithm, which automatically generates an extended finite state machines *(EFSMs)* from the software interaction strings. The *GK-tail* algorithm does not rely on the additional source of information like teachers and operates mainly as described in the following steps:

- merge the traces, which are input equivalent to create a unique trace annotated with multiple data values,
- generate the transition associated predicates with the help of *DAIKON* [Ernst 2001] from multiple data values,
- construct a tree like structure *EFSM* from the interaction traces annotated with predicates,
- iteratively merge the states, which are equivalent to each other in the *EFSM* to construct the final *EFSM*.

The input equivalent traces are different input traces, which invoke the same sequence of methods in the target system and the equivalent transitions are the transitions, which have the same "tail". If the states cannot be distinguished by looking at the outgoing sequences, then they are said to have the same "tail". The method of merging the equivalent states is an extension to the *Ktail* algorithm by

Biermann *et al.* [Biermann 1972], who consider a set of states to be equivalent, if all the member elements of the set of states are followed by the same paths. The sets of paths exiting a state can be infinite, thus comparing the states for all the possible exiting paths may go very expensive, so the length of the paths sequences to be compared is limited to k, exactly like Biermann *et al.* To evaluate *GK-tail*, Lorenzoli *et al.* implemented a prototype for Java programs based on Aspectwerkz [Aspectwerkz 2007], theorem prover Simplify by Detlefs *et al.* [Detlefs 2005] and *DAIKON*. To monitor systems and record the traces, this prototype is based on Aspectwerkz, to check for equivalence and implication between annotations on Simplify theorem prover and *DAIKON* for detecting the invariants to annotate the edges of the *EFSM*. For the evaluation of *GK-tail* algorithm, the *EFSM* models were generated for a set of open source software applications, having different sizes and natures. The results by Lorenzoli *et al.* indicate that the size of the generated *EFSM* models does not depend on the size of the software implementations but on the size of interaction patterns within software components. They constructed *EFSMs* for a set of sample open source applications of different size including Jabref [JabRef] and Jedit [jEdit], the calculated results show that the *EFSM* models were better and often more accurate than generating the *FSMs* [Biermann 1972, Cook 1998] and learning the constraints [Ernst 2001] independently, especially for models of a non trivial size.

3.2.4 Exact *DFA* Identification

Heule and Verwer [Heule 2010] present an exact deterministic finite automata (*DFA*) learning algorithm, which is based on satisfiability (*SAT*) solvers. They propose a compact translation of *DFA* identification into *SAT*, then they reduce the *SAT* search space by adding the lower bound information using a fast max-clique symmetry breaking algorithm [Sakallah 2009]. They add many redundant clauses to improve the performance of the *SAT* solver and show how flexibility for their translation can be used to apply over very hard problems. The evidence driven state-merging (*EDSM*) algorithms are normally based on successful techniques such as satisfiability and graph coloring [Lang 1998]. To increase the quality of the solution, the *EDSM* is a greedy procedure that uses simple heuristic to identify which merges to perform. The examples for such heuristics are dependency directed backtracking [Oliveira 1998], mutually incompatible merges [Abela 2004], and searching most constrained nodes first [Lang 1999]. Bugalho and Oliveria compared different search techniques for *EDSM* [Bugalho 2005]. To reduce the size of the problem Heule and Verwer [Heule 2010] applied few steps of the *EDSM* algorithm and then applied their translation to *SAT*. Their algorithm for exact *DFA* learning named as *DFASAT* was the winner for the *STAMINA* competition [Walkinshaw 2010a].

3.2.5 Framework for Evaluating Passive Learning Techniques

It is difficult to compare different software inference algorithms as their evaluation available in the literature revolves around a limited number of benchmarks. The selected models are rarely diverse enough to make a systematic comparison. They might have an arbitrary number of states or specific model type, which is not appropriate for all the techniques. All this increases the need of a framework which can evaluate the model inference approaches. The *STAMINA* competition [Walkinshaw 2010a] provides an online deployed framework for the comparison of state machine inference approaches [Challenge a]. The purpose of this competition is to compare passive learning techniques. The advantage of such a framework is that the participants get their techniques evaluated, which provides an opportunity to have a snapshot of best performing inference techniques. However, if such a competition terminates with no dominant technique, it will help to find the weaknesses and strengths of the participating techniques. For the *STAMINA* framework, the learning algorithm are supposed to start with a sample of behavior of the target software system, and that behavior can be presented by a *DFA*. The sample is in the form of a set of sequences in Σ^*. For model inference, the sample can be program traces or scenarios supplied by the developer. The sample is supplied in two sets of traces (sequences), the positive traces and the negative traces. A random walk algorithm has been implemented to generate these traces. A positive trace is generated by starting a random walk from the initial state and then selecting a transition randomly with the uniform distribution on the outgoing transitions of the states. The random walk terminates with the probability

$$\left(\frac{1.0}{1 + 2 * outdegree(s)}\right),$$

after reaching a non terminal state s. Negative traces are obtained by editing the positive traces by substitution, insertion or deletion of a symbol. The traces are discarded if the edited versions are still accepted.

The *STAMINA* framework offers the inference of a set of random *DFAs*. This set includes 100 *DFAs* and the elements on the average have state size 50. For every problem, participants are provided with the training set, the positive and negative traces. To conjecture the models from the provided samples, the competitors can download the sample tests of traces from the online available framework[1].

[1] http://stamina.chefbe.net/home

		Sparsity of the training sample			
		100%	50%	25%	12.5%
Alphabet size	2	1 – 5	6 – 10	11 – 15	16 – 20
	5	21 – 25	26 – 30	31 – 35	36 – 40
	10	41 – 45	46 – 50	51 – 55	56 – 60
	20	61 – 65	66 – 70	71 – 75	76 – 80
	50	81 – 85	86 – 90	91 – 95	96 – 100

Table 3.1: 5 problems to solve in each cell

The participants were supposed to submit the solutions as binary strings and one was supposed to use "1" to represent the accepted tests by the learned conjecture and "0" to represent the rejected tests. The provided examples vary in the difficulty level depending on the size of the target model's alphabet and the sparsity of the provided traces. To infer the state machines, implementation of a well documented version of the baseline technique, the *EDSM* Blue-Fringe algorithm [Lang 1998] was made openly available on the competitions website. The participants were supposed to add their improvements to online available implementation and evaluate their variant.

		Sparsity of the training sample	
		100%	50%
Alphabet size	2	*MVdPyA/Equipo*	*MVdPyA/Menor*
	5	*DFASAT/DFASAT*	*DFASAT/DFASAT*
	10	*DFASAT/DFASAT*	*DFASAT/DFASAT*
	20	*DFASAT/DFASAT*	*DFASAT/DFASAT*
	50	*DFASAT/DFASAT*	*DFASAT/DFASAT*

Table 3.2: Cell winners and best challengers

The grids in Table 3.2 provide data about the competition results. The column labeled with 100% presents cell winners and underlined are the best challengers for the competition. The *DFASAT* algorithm discussed in Section 3.2.4 by Heule and Verwer was the official winner for the competition.

3.3 Active Learning

For passive learning techniques, one is bound to learn a model from the provided set of traces. Such traces may not contain all the necessary information about the behavior of the system. It is not possible for the model inference techniques to learn a correct model from an arbitrary set of traces. If the provided set of traces includes sufficient information about an implementation, i.e. what it can do and what it cannot, then inference techniques will be able to identify every state transition and distinguish all the non equivalent states from each other. However, considering

that a provided set of traces is unlikely to contain all the necessary information about the target implementation, the learned model is a poor approximation of the real implementation. In effort to solve this problem a number of active learning techniques have been proposed. Active learning algorithms actively interact with the system to learn the models. Instead of relying on given traces, these algorithms can create and perform new tests on target systems and find out how far they might be from the solution. The mathematical setting where the queries are asked to an oracle is called active learning. Active learning is used to construct the exact models of unknown black box systems in different communities. These techniques continuously interact with the target systems to produce the models.

3.3.1 Overview of the Learning Algorithm L^*

Angluin introduced an active learning model named as the algorithm L^* [Angluin 1987]. The algorithm queries sequences of inputs known as *membership queries* to the target system and organizes the replies in a table named as the observation table. The observation table row labels are *prefix closed* access strings for states and columns are *suffix closed* distinguishing strings for states.

Table 3.3: An observation table for a machine with $\Sigma = \{a, b\}$

		E
		ϵ
S	ϵ	0
	a	1
$S \cdot \Sigma$	b	1
	aa	1
	ab	0

To conjecture a model from the observation table the algorithm requires the table to be *closed* and *compatible* (in the original version of the algorithm L^*, Angluin named the compatibility concept as consistency). The observation table is closed if every element of $S \cdot \Sigma$ is equal to at least one element of S and it is compatible if two elements of S are equal, then their one letter extensions are also equal. The conjectured model is iteratively improved with the help of a teacher (*oracle*) by asking equivalence queries. If the learned model is correct, the oracle replies "yes" or a *counterexample*, otherwise. By taking the counterexample into account, the algorithm iterates by asking new membership queries and constructing an improved conjecture until we get the automaton that is equivalent to the black box. The worst case complexity of the algorithm in terms of membership queries is $O(|\Sigma|mn^2)$, where $|\Sigma|$ is the size of alphabet, m is the length of the longest counterexample provided by oracle and n is the number of states in the inferred model.

3.3.2 Searching for Counterexamples

After conjecturing the initial model, the algorithm L^* requires the oracle to provide the counterexamples. The methods for state space exploration [Keidar 2003] and functional test coverage [Walkinshaw 2010b] can be used to search for the counterexamples. Angluin [Angluin 1987] proposed a random sampling oracle that constructs a string by selecting elements from alphabet Σ^+ and returns string with "yes" or "no", yes if the string belongs to the language of the unknown model and no, otherwise. This naive method to search for the counterexamples may return very long counterexamples and may consume a lot of iterations to find such counterexamples.

Howar *et al.* [Howar 2010] proposed the Evolving Hypothesis Blocking (*E.H. Blocking*) and Evolving Hypothesis Weighted (*E.H. Weighted*) counterexample searching algorithms. The *E.H. Blocking* algorithm constructs the strings by randomly selecting the elements from the set $S \cdot \Sigma$, then the continuation of the strings is randomly generated by increasing length on Σ^+ and the constructed strings are candidates for counterexamples. The *E.H. Weighted* algorithm uses weights along the transitions of the conjecture. Every time a transition is traversed the weight gets incremented and its selection for traversal is inversely proportional to its weight.

Balle [Balle 2010] used the algorithms $Balle_{L_1}$ and $Balle_{L_2}$ to search for counterexamples. To search for counterexamples the algorithm $Balle_{L_1}$ builds the strings by uniform distribution over alphabet Σ, whereas $Balle_{L_2}$ constructs the strings by exploring the transitions of the conjecture, the probability that a transition is selected for exploration depends on its destination height.

3.3.3 Processing Counterexamples

Processing the counterexamples is an important part of learning with the learning algorithm L^*. To process a counterexample CE either the prefixes of CE can be added to rows S or the suffixes of CE can be added to the columns of the observation table. The counterexamples processing method by Angluin [Angluin 1987] requires to add all the prefixes of CE to the rows S of the observation table, which results in increased number of membership queries. Every prefix of CE is not necessarily a distinct row and some of the prefixes in S may point to equivalent states thus requiring to check the compatibility for such states and increased number of membership queries.

Rivest and Schapire [Rivest 1993] provided a counterexamples processing method, which uses binary search to find a distinguishing sequences from CE and adds the distinguishing sequence to the columns E of the observation table. This method always keeps the elements of S distinct and increments the size of S only to make the table closed, thus compatibility condition is trivially satisfied. The learning algorithm L^* keeps the row and column labels of the observation table prefix and suffix closed, respectively. But the method by Rivest and Schapire does not add the suffixes of distinguishing sequence thus requiring a compromise on suffix closure property.

To alleviate this problem, Maler and Pnueli [Maler 1995] proposed a counterexample processing method, which adds a counterexample and all of its suffixes to the columns E of the observation table. The resulting observation table is always prefix and suffix closed and trivially compatible. Shahbaz and Groz [Shahbaz 2009] identify that to process the counterexample CE, it is not inevitable to add all the suffixes to E. Their method searches for the longest member of the set $S \cup S \cdot \Sigma$ that is prefix to CE, and after dropping such a prefix from CE adds the suffixes of remaining part of CE to the columns E of the observation table.

Luque [Luque 2010] proposed a counterexample processing method which operates like the counterexample processing method by Angluin [Angluin 1987]. The difference lies at the point of adding the prefixes of CE to rows S of the observation table. The counterexample processing method by Luque adds the prefixes of CE one by one to the rows S in the hope that it is not necessary to add all the prefixes of CE to identify new states.

3.3.4 Executing new Experiments in Canonic Order

Garcia *et al.* [Garcia 2010] proposed two variants of the algorithm L^* known as *Depth-M** and *Breadth-M** for the ZULU challenge [Combe 2010]. To calculate the membership queries, the algorithms require to execute them in canonic order. The *Depth-M** algorithm fills the cells of the observation table with a top down (depth-first) technique by selecting the columns from left to right. After recording results for every test in the observation table, the algorithm checks for the closure property. If the table is not closed it moves the row with the newly added cell from equivalent states part $S \cdot \Sigma$ to states part S of the observation table and extends the table. To make the table closed new rows are added and empty table cells are filled with top down and left to right approach. This technique is explained with the help of Figure 3.1 where learning from a sample automaton is presented. The sequence a is added to columns of tables to process a counterexample (an arbitrary sequence).

	ϵ
ϵ	0
a	1
b	

	ϵ
ϵ	0
a	1
b	
aa	
ab	

	ϵ
ϵ	0
a	1
b	0
aa	0
ab	0

	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	
ab	0	

	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	
ab	0	
ba	0	
bb		

Figure 3.1: Learning with the *Depth-M** algorithm

The *Breadth-M** algorithm fills the cells of the observation table with a left to right (breadth-first) technique by selecting the rows from top to bottom. The algorithm checks for the closure property after recording results for every test in the

observation table. After making the table closed newly added rows are filled with the same left to right and top down pattern. This technique is explained with the help of Figure 3.2 where learning from a sample automaton is presented.

	ϵ
ϵ	0
a	1
b	0
aa	0
ab	0

	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	
ab	0	

	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	0
ab	0	1
ba		
bb		

	ϵ	a
ϵ	0	1
a	1	0
b	0	0
aa	0	0
ab	0	1
ba	0	0
bb		

Figure 3.2: Learning with the *Breadth-M** algorithm

Both of the algorithms *Depth-M** and *Breadth-M** operate similarly, if they are required to ask membership queries for only one column. They operate differently when they have to calculate queries for multiple columns. The algorithms calculate queries on the same pattern up to fourth table (from left to right) in Figure 3.1 and second table (from left to right) in Figure 3.2. After this, the *Depth-M** algorithm will calculate the membership query $ba \cdot \epsilon$, whereas the *Breadth-M** algorithm has to calculate $aa \cdot a$ and $ab \cdot a$ before calculating the query $ba \cdot \epsilon$.

Garcia *et al.* claim that the behavior of both *Depth-M** and *Breath-M** is quite different and therefore, both are considered as different algorithms.

3.3.5 Tables with Holes

At times while learning with L^*, we may come across situations when we do not have all the information to conjecture a model and we are required to construct models from incomplete or sparse observations. We can quite often have such problems, specifically in learning natural languages. When limited queries are allowed, the learner may run out of the queries limit with a sparse observation table (we might consume the entire available query budget before achieving the target model). In such cases and many others, we need to construct models from sparse observation tables. A sparse/incomplete observation table is a table, which has holes. A hole in the observation table is a pair (s, e) where s is an element of the set of rows indices and e is an element of the set of columns indices of the table such that $s \cdot e = *$, here "$*$" denotes the empty cells. A table is complete if it does not have any holes and is incomplete or sparse, if it has holes. To conjecture a model, we need the observation table to be complete and satisfy the closure and compatibility properties. But for incomplete tables, it is difficult to check such properties because of sparse cells.

For instance, if we have the observation table presented in Table 3.4, de la Higuera [de la Higuera 2010] (Grammatical Inference, page 271) proposes a method to conjecture the models from such tables. The method figures that the entries in the column labeled with ϵ decide whether a state in the conjectured model is final

Table 3.4: An incomplete table

		E		
		ϵ	a	b
S	ϵ		1	1
	a	1	1	0
	ab	0	0	
$S \cdot \Sigma$	b	1		1
	aa	1	1	
	aba	0	1	0
	abb			1

or non final. Thus, the states q_b, q_a and q_{aa} are final, and the states q_{ab} and q_{aba} are non final. The corresponding entries for q_ϵ and q_{abb} are missing so they are neither final nor non final. Figure 3.3 shows the conjectured model.

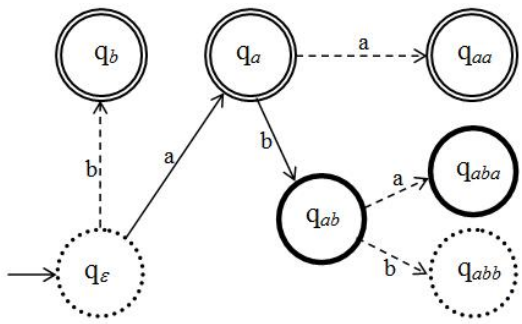

Figure 3.3: The conjecture from Table 3.4

Eisenstat and Angluin [Eisenstat 2010] proposed an algorithm for ZULU competition [Combe 2010] to learn the challenge problems where participants were allowed to make a limited number of membership queries. To conjecture a model from a sparse observation table, the algorithm requires to have all distinct rows in S, which are the states. If the membership query for the ϵ column of any of the states has not been queried, then the algorithm selects it as final or non final from a uniform distribution over "final" and "non final". For transitions, if the one letter extension of a state is equivalent to more than one state in S, then the algorithm selects one state from a uniform distribution over the set of all such states.

To infer a model from Table 3.4, first of all the algorithm will move all the distinct states to S. The row *aba* is not equivalent to any of the distinct states, it is moved to S and its one letter extensions are added to $S \cdot \Sigma$. The queries *abaa* and *abab* for the new added rows can be inferred from known queries (with the help of dictionaries). The updated observation table is shown in Table 3.5.

To conjecture a model, the algorithm needs to know that the distinct states are "final" or "non final". From the observation table one cannot decide about the state

Table 3.5: By moving the row *aba* to S

		E		
		ϵ	a	b
S	ϵ		1	1
	a	1	1	0
	ab	0	0	
	aba	0	1	0
$S \cdot \Sigma$	b	1		1
	aa	1	1	
	abb			1
	abaa	1		
	abab	0		

ϵ. Let us say the algorithm selects it as "final" by uniform distribution over "final" and "non final". The cell $\epsilon \cdot \epsilon$ is filled with 1. For the transition a from the state a, it can either go to the state ϵ or a, one from these is selected. The algorithm uses this technique wherever required and conjectures the model shown in Figure 3.4.

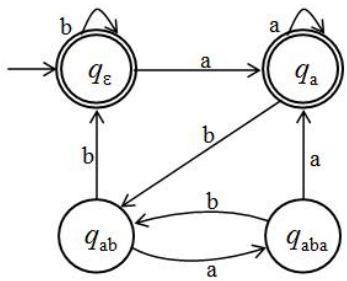

Figure 3.4: The conjecture from Table 3.5

3.3.6 Optimized Learning with the Algorithm L^*

Different type of optimizations can be introduced to the algorithm L^* to reduce the number of membership queries. Hungar *et al.* [Hungar 2003b] proposed optimizations to learn models of reactive systems. The states of such systems are accepting states with all the non accepting strings leading to a sink state, thus the language defined by such a state machine is prefix closed. Contrary to general languages, there is no switching from non accepting to accepting states. They introduced filters which use properties like input determinism, prefix closure, independence and symmetry of events. They reduce the number of membership queries by getting new membership queries answered from filters and previously observed answers.

Balcázar *et al.* introduce observation packs [Balcázar 1997] as a unified view of the algorithm L^*, discrimination trees of Kearns and Vazirani [Kearns 1994], and

the counterexample processing method by Rivest and Schapire [Rivest 1993]. They point out that the conjectured model after processing a counterexample with Rivest and Schapire [Rivest 1993] method may still classify the counterexample incorrectly and the same counterexample can be reused to answer several equivalence queries. Howar *et al.* [Howar 2010] used observation packs along their improved algorithms to search for the counterexamples for learning the problems provided by ZULU challenge [Combe 2010]. Their algorithm was the most efficient one, and asked less membership queries to learn better models as compared to the other participants [Eisenstat 2010, Balle 2010, Garcia 2010, Vilar 2010, Luque 2010].

Chaki and Strichman propose three optimizations for the algorithm L^* based on assume guarantee reasoning [Chaki 2007, Chaki 2008]. Balcázar *et al.* [Balcázar 1997] find that while using the Rivest and Schapire [Rivest 1993] counterexample processing method, a counterexample can be reused to refine conjectured models. The first improvement by Chaki and Strichman is based on the same observation that sometimes a counterexample can be reused to improve the conjectured models.

They consider a system composed of two components M_1 and M_2, and the components are required to synchronize on a set of shared actions. The system is supposed to be verified against a property φ. The automated assume guarantee procedure uses a constant assumption alphabet $\Sigma = (\Sigma_1 \cup \Sigma_\varphi) \cap \Sigma_2$, Where Σ_1, Σ_2 and Σ_φ are the alphabets for M_1, M_2 and φ, respectively. The unknown language to be learned is $U = \mathcal{L}\overline{((M_1 \times \bar{\varphi}) \downharpoonleft \Sigma)}$ over the alphabet Σ.

Their second improvement is based on the observation that most of the membership queries can be avoided, if the algorithm L^* exploits the internal knowledge of M_1, M_2 and φ. This internal knowledge helps the algorithm L^* to ask the membership queries in a more intelligent manner. On finding the observation table not closed, the algorithm L^* requires to move the row causing unclosure from $S \cdot \Sigma$ to S and add all of its one letter extensions to $S \cdot \Sigma$. To complete the cells of these rows, the algorithm requires to ask $|\Sigma| \times |E|$ membership queries. For membership queries, any $\omega \in \Sigma^*$, let $Sim(\omega)$ be the set of states of $M_1 \times \bar{\varphi}$ reached by simulating ω on $M_1 \times \bar{\varphi}$ from initial state. Then, ω is in the unknown language iff any of the accepting states of $M_1 \times \bar{\varphi}$ does not belong to $Sim(\omega)$. Let $En(s) = \{a \in \Sigma | output(Sim(s), a) \neq \emptyset\}$ be the set of enabled actions from $Sim(s)$ in $M_1 \times \bar{\varphi}$. Now, for any $e \in E$ and $a \notin En(s)$, $Sim(s \cdot a \cdot e = \emptyset)$ guarantees that $s \cdot a \cdot e$ belongs to the unknown language. Thus, for all the cells of rows $s \cdot a$ where $a \notin En(s)$ they directly put "1" and ask membership queries only for the rows $s \cdot a$ where $a \in En(s)$. The motivation behind this optimization is that the enabled actions size $|En(s)|$ is usually much smaller than the alphabet size $|\Sigma|$ for any s.

The third improvement is similar to the one proposed by Gheorghiu *et al.* [Gheorghiu 2007], this improvement is based on the assumption that it is always possibly to complete the verification with the much smaller size of alphabet than Σ. They start with the empty alphabet and keep refining on the basis of counterexamples, this procedure terminates with a minimal alphabet that is sufficient for the overall verification. Since the overall complexity depends on the size of alphabet, a smaller alphabet can improve the overall performance. Their improvement reduces

the size of alphabet and states in the learned *DFA*.

3.3.7 Learning from Membership Queries Alone

Most of the variants of L^* require asking membership and equivalence queries to learn the unknown models [Bshouty 1994, Shahbaz 2007, Pasareanu 2008, Aarts 2010a]. However, in the literature we can find some techniques that use only membership queries to learn the models, like the method proposed by Eisenstat and Angluin [Eisenstat 2010]. They proposed an algorithm for ZULU competition [Combe 2010] where the participants were required to learn from the limited budget of membership queries. Their work is inspired from Korshunov's work [Korshunov 1967], which implies that in almost all DFAs of n states with input alphabet of size k, a test set consisting of all strings of length at most about $log(k)log_2(n)$ is sufficient to distinguish all in-equivalent pairs of states. They alter the algorithm L^* to use such strings, but for black box models n number of states is not known, so $log(k)log_2(n)$ cannot be calculated. As a solution, their method adds suffixes of length $0, 1, 2, \ldots$ until the budget to ask the queries is exhausted. The method starts by adding empty string to columns of the observation table and carries on adding incrementing suffixes. When most of the suffixes of a certain length are added, then the length of the suffixes getting added to the observation table is incremented.

3.3.8 Learning from Counterexamples Alone

The complexity of the target model is measured by two parameters: the number of inputs Σ and the number of states n. Angluin claims that equivalence queries alone are not sufficient to learn the target models [Angluin 1990] though her L^* learning algorithm learns the models with a minimally adequate teacher [Angluin 1987]. Ibarra and Jiang [Ibarra 1991] show that membership queries are not needed for *DFA* inference if one puts additional information to the answers of the equivalence queries. They show that *DFA* models can be learned from $|\Sigma|n^3$ smallest counterexamples. Birkendorf *et al.* [Birkendorf 1998] improve these results by presenting a new algorithm named as *REDIRECT*, which requires only $|\Sigma|n^2$ smallest counterexamples. To get started *REDIRECT* requires a first partially correct *DFA* M_1. They suppose the *DFAs* M_1^1 and M_1^0 as shown in Figure 3.5, where ϵ denotes the empty string.

Figure 3.5: Initial *DFAs* M_1^1 and M_1^0 for $\Sigma = \{0, 1\}$. State q_ϵ is accepting in M_1^1 and non-accepting in M_1^0.

The state diagrams consist of a single state q_ϵ with the edges $q_\epsilon \xrightarrow{a} q_\epsilon$ for all $a \in \Sigma$. Thus, M_1^1 and M_1^0 are acceptors of $\emptyset$ and Σ^*. Let M_* and M_k denote the target unknown *DFA* and learned *DFA*, respectively. Now M_1^0 is partially correct if $M_*(\epsilon) = 0$ or M_1^0 is partially correct, otherwise. Thus, by using one counterexample to determine $M_*(\epsilon)$, M_1 can be set to $M_0^{M_*(\epsilon)}$. If $\text{mincex}(M_1, M_*)$ denotes the smallest counterexample for M_1, then the algorithm proceeds by finding a smallest counterexample $c_1 = \text{mincex}(M_1, M_*)$. They prove that their algorithm requires at most $|\Sigma|n^2$ counterexamples.

3.3.9 Mealy Adaptation of the Algorithm L^*

Mealy machine models are more appropriate to model the systems that reply with outputs, e.g. online commerce, communication protocols, web services, etc. The transitions of Mealy machine models are labeled with *input/output* (i/o) instead of *DFA* that are labeled with only input symbols. The (i/o) models (Mealy machines) are more appropriate to model reactive systems [Aarts 2010b, Bohlin 2009]. Bohlin, and Aarts *et al.* used Mealy inference to infer the models of communication protocols [Aarts 2010b, Bohlin 2009]. Mealy machines need less states and queries than DFA to model the same problem [Niese 2003, Hungar 2003b]. For Mealy machine models, the distinction between accepting and non accepting states is not like *DFA* models, rather Mealy machine states are distinguished by real outputs that they produce for a given input for a specific state. The algorithm L^* was adapted formally [Niese 2003, Berg 2005a, Shu 2007, Shahbaz 2009, Bohlin 2009] and informally [Pena 1998, Margaria 2004] to infer the Mealy machine models. To annotate transition edges of (i/o) models, the Mealy adaptation of the algorithm L^* requires to initialize the columns of the observation table with the input set I. The concepts to make the observation table closed and compatible before conjecturing a model remains the same.

3.3.10 Optimized Mealy Inference

Cho *et al.* [Cho 2010] noticed that the Mealy adaptation of the algorithm L^* asks a number of output queries that are independent from each other and can be executed in parallel on their case study (a network protocol). They implemented a file named as query cache to collect the parallel query responses, this file stores pairs of input strings and corresponding output sequences. Their implementation issues output queries in parallel and processes the responses sequentially. They divided the queries among 8 machines, which emit them independently in parallel to run them on the network. The deterministic assumption enables them to avoid asking duplicate queries that can be calculated from cached responses.

They identified that there is a significant amount of redundancy in the learning process due to the states that have self loops. These loops increase the number of queries for model inference without identifying new states. They believe that there are two factors that result in a number of self loops for network protocol model

inference:

- One needs to calculate the outputs for all inputs at every state as the target is to learn the complete models. Most of the protocols use every message for a dedicated task, which means that asking an unexpected input from a state leads to the error state or is ignored while remaining on the same state.

- The protocol interaction messages are supposed to be abstracted to input and output alphabet before the protocol model is learned. The abstraction overestimation may increase the redundancy in the model (e.g. increase the number of self loops that can not be eliminated before the model is learned). Reducing the input alphabet size may result in over simplified models.

They propose that a distinct state d (where $d \in S$, S the upper part of the observation table) with self-loops will have many equivalent rows in $S \cdot I$ (the lower part of the observation table). The responses for these equivalent rows can be predicted from the responses for d. To explain the intuition for response prediction, they use an example with states having a number of self loops. They propose a two level heuristic for predicting the query responses. The first level named as *restriction-based* prediction exploits the fact that protocols can have states with number of self-loops, for predicting the output query answers. The second level called *probability-based* prediction exploits input determinism (same input sequences result in same output sequences). The prediction errors are detected by random sampling. If an error is detected, the algorithm backtracks to the step with the first output query with an erroneous response and continues the learning process after fixing that error. They use same sampling queries to detect new states and to fix the wrong predictions.

3.3.11 Learning NFA Models

In many testing and verification applications the learned non-deterministic finite state automata (NFA) models are exponentially smaller than the corresponding DFA models. So there is a significant gain, when we use learning algorithms that learn the models as succinct NFA instead of DFA. But the issue is that the class of NFA lacks the significant properties that most of the learning algorithms available in the literature use. For a given regular language there is no minimally unique NFA, so it is not clear which of the automaton is required to be learned. Denis *et al.* [Denis 2002] introduced a subclass of NFA, the residual finite state automata (RFSA), which shares important properties with the DFA class. For example, there is unique minimal canonical RFSA for all the regular languages accepting them, which is more succinct than the corresponding DFA. Bollig *et al.* [Bollig 2008, Bollig 2009] proposed an active learning algorithm NL^* which learns the RFSA models. This algorithm alters the Angluin algorithm L^* to learn the RFSA models using the membership and equivalence queries. The NL^* algorithm requires $O(n^2)$ equivalence queries and mn^3 membership queries where n is the minimal number states in the target

model and m is the length of the longest counterexample provided by the oracle. They introduced the concepts of *Composed and Prime Rows* to redefine closure and compatibility as *RFSA-Closure* and *RFSA-Compatibility*, respectively. They implemented the algorithm and used it to learn the regular languages described by the regular expressions. For the considered set of experiments, in most of the cases, they observed that they required very few membership and equivalence queries as compared to L^*, and the RFSA learned by the NL^* algorithm were much smaller than the corresponding DFA learned by the algorithm L^*. Yokomori [Yokomori 1994] has also shown that a specific class of NFA, called polynomially deterministic, is learnable in polynomial time using membership and equivalence queries.

3.3.12 Framework for Evaluating Active Learning Techniques

In the context of active learning, numerous algorithms have been developed to infer the unknown models as state machines. It is complicated to decide which one is better amongst them, as different algorithms may perform better in different settings. If we want to work on a given problem, then finding all the possible solutions available in the literature is a difficult task. Furthermore, the research community is growing and working fast and always we have the possibility of better recently proposed solutions. One of the possible solutions to all such problems can be development of a framework, which can evaluate the solutions and bring the community closer to provide an opportunity to have a look at other solutions. Similarly, there could be a common source where all contributions on a specific subject are recorded and evaluated, e.g. the bibliographical study of grammatical inference [de la Higuera 2005] and grammatical inference for learning automata and grammar [de la Higuera 2010].

The finite state machine or grammar learning competitions like the Omphalos Context-Free Grammar Learning Competition [Starkie 2004], the Tenjinno Machine Translation Competition [Starkie 2006] and ZULU: An interactive learning competition [Combe 2010] are the efforts in this regard. The ZULU challenge provides an online deployed framework for evaluation and comparison of active learning state machine inference approaches. It is a challenge to test the algorithms more specifically the variants of Angluin's algorithm L^* [Angluin 1987]. The baseline of the framework implemented in Java, is based on the algorithm L^* and is openly available on the ZULU challenge website [Challenge b]. Randomly generated machines have often been used to assess testing techniques, e.g. work by Sidhu and Leung [Sidhu 1989], Yevtushenko and El Fakih [Dorofeeva 2005], Berg *et al.* [Berg 2005b], more recently by Bollig *et al.* [Bollig 2009]. For the ZULU challenge the minimal *DFAs* were generated randomly with variation on number of states and alphabet size. The problems were divided into 12 different categories, by different combinations of "small, medium and large" states and alphabet sizes combined with the number of allowed queries. The ZULU framework operates with the concept of two oracles; the first oracle can be accessed by a participant after getting registered. The participant can then request for a target *DFA*. The oracle computes the number of membership

queries k required to learn a reasonable machine and offers the participant to learn that machine with k membership queries. Here reasonable means a machine with less than 30% classification errors. The second oracle interacts with a participant and answers to k queries. Once, the participant has learned a state machine for the provided task. Now in order to judge the correctness of the participant's technique, the oracle generates 1800 strings and sends them to the participant. The participant runs these 1800 strings on his learned conjecture and records the results as "yes" or "No", ("Yes" for accepted strings and "No" for the rejected strings) and sends the result to the oracle. The oracle receives the 1800 labels along with answers and computes the score.

Task	queries	alphabet	states	Best%
1	304	3	8	100,00
2	199	3	16	100,00
3	1197	3	81	96,50
4	1384	3	100	93,22
5	1971	3	151	85,89
6	3625	3	176	100,00
7	429	5	15	100,00
8	375	5	18	100,00
9	2524	5	84	96,44
10	3021	5	90	100,00
11	5428	5	153	99,94
12	4616	5	123	100,00

Task	queries	alphabet	states	Best %
13	725	15	10	100,00
14	1365	15	17	100,00
15	5266	15	60	100,00
16	7570	15	71	100,00
17	17034	15	147	100,00
18	16914	15	143	87,94
19	1970	5	93	81,67
20	1329	5	61	70,00
21	571	5	40	69,22
22	735	5	57	65,11
23	483	5	73	86,61
24	632	5	78	100,00

Figure 3.6: ZULU challenge results.

There were 23 competing algorithms registered by 11 players, Figure 3.6 shows the results for all the 24 tasks which were grouped in 12 categories. The figure presents only the best results by any of the algorithms. The winners were calculated by averaging at the performance of the algorithms in all of the 12 categories. The algorithm by Howar *et al.* [Howar 2010] was ranked first, the algorithm from Balle [Balle 2010] was the second, and the algorithm by Eisenstat and Angluin [Eisenstat 2010] was rated third.

3.4 Applications of Model Inference

Grammatical inference has a number of applications like machine learning [Mitchell 1997], speech recognition [Jelinek 1997] and formal language theory [Harrison 1978]. Software model inference from implementations has been used for software testing [Niese 2003, Shu 2007, Raffelt 2009, Bué 2010], specification mining [Xie 2004] and model checking [Peled 1999, Berg 2005c, Elkind 2006]. It can be used to find behavior of software systems where specifications are missing [Peled 1999].

3.4.1 Specification Mining

The high level specifications can be extracted from the existing code of components and can be used for understanding the components. The specifications can also be used for formal verification and regression testing of programs or software components. A model of well functioning system captures properties of the system which can be considered as specification of the system. Mining specifications is a well known technique and in the literature, we can find significant number of contributions on the subject, which can be divided into dynamic and static approaches. From event traces, the finite state machine based models can be extracted. There are techniques which consider the general problem of extracting finite state machine based models from the event traces like Cook and Wolf [Cook 1998]. They take the model inference problem as well-known grammar inference problem [Gold 1967] and discuss algorithmic, statistical and hybrid approaches. Mining the specifications can be used for automatic verification of the programs; in this regard there is a contribution from Ammons *et al.* [Ammons 2002]. The proposed approach addresses *C* programs and learns the models as probabilistic finite state automata. For static mining specification techniques, there is a rich body of literature. There are techniques which mine object usage models that describe the usage of an object in a program [Wasylkowski 2007]. Wasylkowski *et al.* [Wasylkowski 2007] apply concept analysis to find code locations where rules derived from usage models are violated. Ramanathan *et al.* [Ramanathan 2007] use an inter-procedural path sensitive analysis to infer preconditions for method invocations. Shoham *et al.* [Shoham 2007] discover that static mining of automata based specifications requires precise aliasing information to produce reliable results.

To have specifications reflect normal rather than potential usage, dynamic specification mining observes executions to infer common properties. Typical examples of dynamic approaches include DAIKON [Ernst 2001] for invariants or GK-tail [Lorenzoli 2008] for object states. The issue with these approaches is that they are limited to the possibly small set of observed executions. If a piece of code is not executed, it will not be considered in the specifications, if it is executed only once, we do not know about alternative behavior. To address this problem, test case generation to systematically enrich dynamically mined specifications is used. Combined this way, both techniques benefit from each other: Dynamic specification mining profits from test case generation, additional executions can be observed to enrich the mined specifications. Test case generation, on the other hand, can profit from mined specifications, as their complement points to yet unobserved behavior.

The idea of combining test case generation with specification mining was conceived by Xie and Notkin [Xie 2004]. They presented a generic feedback loop framework where specifications are fed into a test case generator, the generated tests are used to refine the specifications, and the refined specifications are again given as input to the test case generator. Dallmeier *et al.* [Dallmeier 2010] extended this work by providing an implementation of the framework for type state mining, as well as an evaluation of how useful enriched specifications are for a real-world application.

The initially mined specifications by their technique contain only observed transitions. Then, to enrich the specifications, the TAUTOKO tool generates the test cases to cover all possible transitions between all observed states and thus extracts the additional transitions from their executions.

Michael Pradel *et al.* [Pradel 2010] present a framework for the systematic evaluation of different specification mining approaches. They tailor the metrics for accounting incompleteness and imprecision in mined specifications. Their framework consists of two parts: i) a mechanism that helps to formalize the knowledge of API usage constraints. On the basis of these formalizations the framework generates *FSMs* that serve as a reference for evaluation of mined specifications. ii) the metrics for computing precision and incompleteness. The precision indicates about the correctness of mined specification. Bogdanov and Walkinshaw proposed an algorithm [Bogdanov 2009] to structurally compare two *FSMs*. Their algorithm selects pairs of states, one from each of the machines under consideration with similar incoming and outgoing transitions and then expresses the difference between the *FSM*s in terms of added and removed transitions, whereas the framework from Pradel *et al.* compares the languages accepted by two *FSMs*.

3.4.2 Integration Testing

The fact that the integration of good quality components may not give rise to the good quality software systems yields the importance of integration testing. It may be complicated to cover all interactions between components to discover all possible errors and it can be more complicated when the provided components lack the source code and formal models. In such circumstances the learning and testing techniques to test the integrated systems [Li 2006, Groz 2008, Shahbaz 2008] can be very helpful. For integration testing the test cases are constructed following some test generation strategy. The selected strategy is built on some coverage criteria. These techniques explore how the components interact with each other in a system. Learning and testing individual components is not reliable. Indeed, the learned models are partial and they cannot depict integrated systems completely. With the integration of components the system may experience the issues like dead-locks and live-locks, which can be detected and tested in the integrated systems.

Groz *et al.* [Groz 2008] address the problem of testing intermittent errors for communicating components when models of components are not available. They propose a technique named as *k-quotient* to infer models of software components where k is based on state distinguishability. These models are used to detect compositional problems and intermittent errors. They perform reachability analysis of the learned models to identify traces of compositional errors like unspecified receptions, live-locks, and races.

Shahbaz [Shahbaz 2008, Shahbaz 2011] presents the integration framework for software systems developed from black box components using their partially learned models. The testing activity gives the results in two directions. The component models are iteratively refined after finding the discrepancies between the learned

model behavior and the original system. Secondly during testing compositional errors can be discovered. The proposed framework assumes that architecture of the system under test SUT is known, it is known that how the components are bound together through their interfaces and the SUT can be assembled and disassembled at the testers will. It considers a system built from black-box components which communicate asynchronously through their input output interfaces. The interfaces of the SUT are distinguished as internal and external interfaces. The internal interfaces are the interfaces with which the components of the SUT interact with each other, whereas the external interfaces are the interfaces through which the SUT interacts with the environment. The internal and external inputs and outputs are distinguished with the help of these internal and external interfaces. These interfaces are observable, external interfaces are controllable, whereas the internals are not. This means that the interaction between the components of the SUT can be monitored but cannot be interrupted in the integrated system. This framework assumes that the SUT is input deterministic and input enabled. On receiving an input on any of the states, only one transition labeled with that input is enabled and produces an observable output. The system is in a stable state when none of the SUT component's transitions is enabled. The SUT can accept the external inputs only when it is in a stable state. The components are black boxes but their input set is always known and the models of these components can be learned as finite state machines. The five steps of this approach are explained in Figure 3.7.

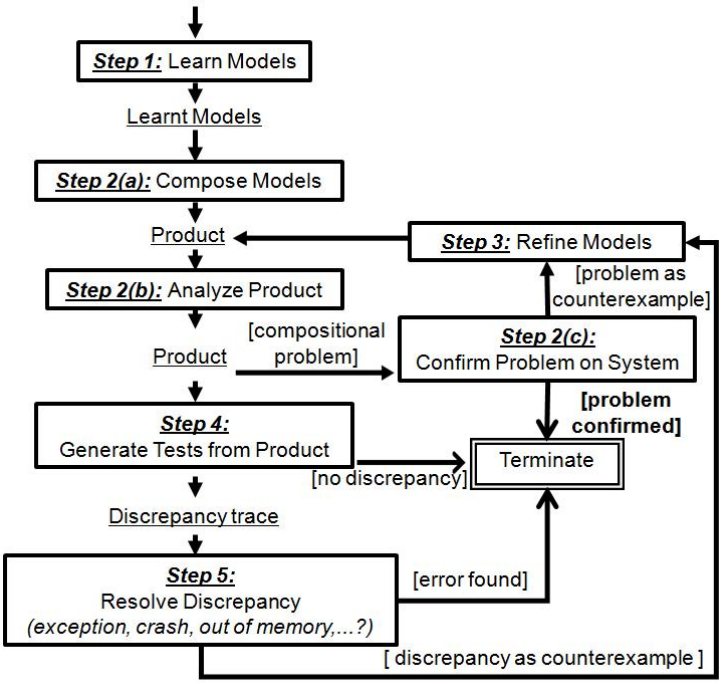

Figure 3.7: Learning and testing approach for integrated systems.

At first a complex integrated system is disassembled into components to learn and test each unit component in isolation. Then, at the second step, from the learned formal models of the components, the product of the unit models is constructed. This product may partially present the SUT, therefore it is analyzed for the compositional problems. If a problem is reported it is confirmed on the SUT, if it is confirmed, then procedure terminates after reporting it. If it is not confirmed, then it is an artifact and the problem is considered as counterexample for refining the conjectured model. At step three the product of the learned models is refined by relearning some components after which the second step is again repeated. At step four, tests are generated from the product of the formal models of the components of the SUT, which contains no compositional problems as it was verified at the initial stages. The generated tests are confirmed on the SUT, if any discrepancy is reported, then proceed to step five. At step five, the discrepancy is resolved either by reporting it as an error or a counterexample. If it is a counterexample, then conjectured model is refined by moving to step three. This procedure is repeated until any compositional problem, real error or no discrepancy between the product of the learned formal models of the components and SUT is reported.

3.4.3 Dynamic Testing

Grammatical inference can be used for dynamic testing [Raffelt 2009]. Raffelt *et al.* [Raffelt 2009] used Integrated Test Environment (*ITE*) [Niese 2001] along the *Learnlib* framework [Raffelt 2005, Raffelt 2006, Raffelt 2008] to propose a dynamic testing method. They improve the practicality of their technique by integrating the common record and replay testing. They use *ITE* for recording and replaying the test cases and the *Learnlib* framework for model inference. The *jABC* framework [Margaria 2005b, Steffen 2006] is used to graphically model the entire learning process that includes modeling conditional and interactive behavior of the system under test.

They used the web based bug tracking case study [Tracker] to illustrate the features of their technique. Their technique successively explores the system under test *SUT* to infer a behavioral model and the model is used to steer the further exploration. The technique traverses the *SUT* like real human users. It retrieves and analyses the web pages by using an application like a web robot along *SUT*'s functionality. It considers the dynamic behavior of *SUT* along the static elements like URL. For example, the operations like change password and update file are visible in the next test run and the technique adapts according to the changed scenario.

The *ITE* environment provides the *Webtest* testing solution which facilitates to record and execute tests for web applications. The *Webtest* is a support tool for dynamic testing of web applications using the machine inference. The capabilities of *Webtest* can be summarized as the identification and execution of test actions (the inputs), recording test cases, replaying the test cases, and integration of all the collected information into dynamic testing process. The inputs are captured and

defined by guiding the user through the process of configuring the desired actions interactively with a browser. The test cases can be recorded in modeling or harvesting mode. In modeling mode the available actions are displayed as trees. The test engineer can select one action and append it to executable graph after configuring it. However, in harvesting mode actions are recorded while the test engineer browses the *SUT*. The recorded tests are executed with the help of a browser component that exposes an interface to trigger the desired actions on a web page.

The dynamic testing framework uses the *Learnlib* framework along the application and structure specific optimizations to reduce the number of test cases [Hungar 2003a, Margaria 2005a]. The *Learnlib* framework starts learning by initializing the algorithm L^*. The algorithm asks the tests, that are executed via the *SUT* interface. The test cases are executed by repeatedly using the *ITE* and replies are recorded. While learning the web applications, every page may lead to new actions (inputs) by offering new hyperlinks and forms. This dynamic alphabet handling is done as follows: after recording the replies for tests, if the alphabet size has been changed, the previous step is repeated, otherwise, the algorithm L^* conjectures a model after making the observations closed and compatible (as discussed in Section 3.3.1). The equivalence between the conjectured model and *SUT* is done by checking the test suite generated by using the *WpMethod* [Fujiwara 1991]. If the conjectured model is not correct the learning process continues, otherwise, the test engineer can decide to introduce further actions and refine the abstraction level.

3.4.4 Communication Protocol Entities Models

Communication protocol entities models can be learned from the external behaviors of protocols. Aarts *et al.* [Aarts 2010b] proposed a framework which uses model inference to infer models of communication protocols. They use the *Learnlib* model inference framework [Raffelt 2005, Raffelt 2006, Raffelt 2008] along ideas from predicate abstraction [Loiseaux 1995, Clarke 2003] to infer the models as Symbolic Mealy machines. They implemented their approach to infer models of the Session Initiation Protocol (*SIP*) and the Transmission Control Protocol (*TCP*). To infer the protocol models they use the Network Simulator ns-2 [The Network Simulator - ns-2], which provides implementation of many network protocols. The *SIP* protocol is an application layer protocol that is responsible for creating and managing multimedia communication sessions. They infer the behavior of the *SIP* server entity at the time of establishing connections with a *SIP* client. The input messages from the client to the server are represented as *Method*(*From*, *To*, *Contact*, *CallId*, *CSeq*,*Via*), where

- *Method* defines the type of request that can be INVITE, PRACK, or ACK,
- *From* and *To* are addresses of the originator and receiver of the request,
- *CallId* is an unique session identifier,
- *CSeq* is a sequence number that orders transactions is a session,

– *Contact* is the address where the Client wants to receive input messages, and

– *Via* indicates the transport path that is used for transaction.

The output messages from server to client are represented as *StatusCode*(*From*, *To*, *Contact*, *CallId*, *CSeq*, *Via*), where, *StatusCode* is a three digit status code that indicates the outcome of the previous request.

They constructed an abstraction mapping, which maps each parameter to an abstract value for server. Before starting a session with The Network Simulator ns-2 the parameters *From*, *To* and *Contact* are configured and they remain constant for the whole session duration. The parameter *Via* consists of default address and variable branch. The Network Simulator ns-2 implementation retains the values of *CallId*, *CSeq* and *Via* parameters that it received in the first invite message and it also retains these parameters for the most recent input message but after producing an output these parameters are omitted. This is the reason for the abstraction they use 6 state variables. The variables *firstId* and *lastId*, *firstCSeq* and *lastCSeq*, and, *firstVia* and *lastVia* store the values for *CallId*, *CSeq*, and *Via*, respectively. For their experiment, the *Learnlib* framework asked about one thousand output queries and one counterexample to conjecture a model of 10 states and 70 transitions.

Bohlin *et al.* [Bohlin 2010] developed a technique that infers symbolic Mealy machines using user supplied criteria for forming state variables and control locations to override the default ones. They generated a model of the Mobile Location Center (*A-MLC*) protocol by using their technique. The *A-MLC* protocol is a commercially available middleware software product that enables the mobile network operators to identify the current information about the current status, geographical location and mobile telephone device details. The protocol is implemented in Erlang [Armstrong 2007]. They selected *A-MLC* because they had access to the executable specification of *A-MLC* which is created to improve the understanding of developers and testers. They were able to conjecture the models by executing the output queries and comparing the conjectured models to the provided models, this made *A-MLC* case study more appropriate for their research.

3.4.5 Security Testing

Cho *et al.* [Cho 2010] propose a technique to infer Mealy models of botnet Command and Control (C&C) protocols. A botnet is a network which consists of hosts remotely controlled by botmasters to carry out important activities like protecting personal information and spamming by denying the unauthorized access. They show that inferred machines enable to analyze formally the botnet defense. Mainly, they show three things:

– How models can be used to identify the weakest links in a protocol when multiple pools of bots partially share the same source. These weakest links are critical for normal functioning of one or more participating agents in communication.

– The inferred Mealy models can be used to discover the protocol design flaws.

– How models can be used to prove the existence of unobservable communication back-channels among botnet servers.

They analyzed the MegaD mass spamming botnet model and discovered a design flaw that was not known previously. They identified the important components of the botnet that are shared among multiple pools of bots by analyzing the critical links. By analyzing the back channels they prove that the MegaD servers communicate with each other.

The implementation of their technique consists of a bot emulator (a script written by them) and the algorithm L^* along the optimizations. The bot emulator receives the output queries from the learning algorithm and concertizes the output queries to valid protocol messages. After receiving a response from the botnet server, the bot emulator abstracts the responses to strings of output alphabet and provides such strings to the learning algorithm. They build the bot emulator from scratch in order to ensure that it does not perform any malicious activity. They designed experiments carefully so that they do not cause any harm to any party involved (e.g. the bot emulator carefully avoids constructing corrupted messages).

For their experiments the response messages were deterministic except for one exception: sometimes the master servers reply with an arbitrarily long sequence of *INFO* messages that always terminate with a non *INFO* message. Their inference infrastructure discards all the *INFO* messages and treats the first non *INFO* message as a response. This was the only source of non-determinism. They used sampling based approach for counterexamples search by generating uniformly distributed strings of input messages. Their implementation makes the table closed and conjectures a Mealy machine model, which is tested iteratively through sampling until the complete model is attained.

3.5 Problem Statement

The algorithm L^* can be used to infer the models of software black box implementations. It uses the minimally adequate teacher to answer two types of questions: membership queries and correctness of a conjecture. The algorithm L^* can be adapted to learn Mealy models directly and the Mealy adaptation asks output queries instead of membership queries [Niese 2003, Shahbaz 2009]. We search for answers to the following questions.

1. Can we reduce the impact of non optimal counterexamples?

2. Can we reduce the number of output queries required to infer a software black box model?

3. Can we learn models without using counterexamples?

For black box model inference, to reply about the correctness of a conjecture, the minimally adequate teacher is substituted by a random sampling oracle. The counterexamples provided by the random sampling oracle may include some futile prefixes to the actual distinguishing sequences. Can we get rid from these useless sequences? This gives rise to the Suffix1by1 counterexample processing method [Irfan 2010c].

– The Suffix1by1 counterexample processing method adds the suffixes of a counterexample to the columns of the observation table to find all distinguishing sequences from the counterexample.

The Mealy adaptation of L^* requires less queries as compared to L^* for inferring the model of the same problem [Niese 2003, Hungar 2003b, Shahbaz 2009]. The Mealy adaptations of L^* initialize the columns of the observation table with the set of inputs. The columns of the observation table contain distinguishing sequences. Are all of the inputs set elements distinguishing sequences for states of target models? Like L^*, on finding a distinct state the Mealy adaptations of L^* add its one letter extensions to the rows of the observation table. All of the inputs are not valid for every state of software applications. Should we add the rows to the observation table for an invalid one letter extension of a state (invalid state successors)? All these factors are the motivation for the algorithm L_1.

– Initially, the algorithm L_1 keeps the columns of the observation table empty and the size of the columns of the observation table augments only to process counterexamples. It uses the Suffix1by1 counterexample processing method to process the counterexamples. The algorithm keeps only valid one letter extensions of the distinct states (valid state successors) in the rows of the observation table.

If an output query is asked again to a deterministic system, it will reply the same answer as it replied previously. The construction of output queries for some cells of the observation results in similar output queries or prefix to the output queries for other cells. Recording the answers for output queries in dictionaries helps to avoid recalculating the output queries. The answers for output queries that are prefix to already calculated output queries can be inferred from the dictionaries. This motivated us for the following heuristic to calculate output queries.

– Instead of calculating the output queries according to their order of cells in the observation table, calculate an output query that can answer a greater number of other unanswered output queries.

The random sampling oracle replies with non optimal counterexamples. One solution as we have seen is to process the counterexamples efficiently, another solution is to avoid using these counterexamples. The GoodSplit algorithm learns the DFA model by taking a bound on the number of membership queries added

to the observation table. This algorithm initializes the rows and columns of the observation table with inputs set and then following a greedy choice fills the cells of the observation table to find distinct rows (states). Mealy variants of L^* require a smaller number of output queries as compared to the algorithm L^*. This motivated us for the Mealy adaption of the GoodSplit algorithm.

– We present the Mealy adaptation of the GoodSplit algorithm named as the learning algorithm $L_{M\text{-}GS}$. The greedy choice proposed for the GoodSplit algorithm is applicable to the boolean queries. We propose a greedy choice for $L_{M\text{-}GS}$ that helps the algorithm to find distinct rows fast. The algorithm $L_{M\text{-}GS}$ wherever possible uses the improved heuristic to decide about the calculation order of output queries.

3.6 Contributions

The summary for each of the main contributions is provided in the following.

3.6.1 Optimized Counterexample Processing Method

The Suffix1by1 algorithm operates with the spirit of Rivest and Schapire [Rivest 1993], Maler and Pnueli [Maler 1995], and Shahbaz and Groz [Shahbaz 2009] algorithms. Instead of adding prefixes of a counterexample to the rows of the observation table like Angluin's method [Angluin 1987], it adds the counterexample suffixes to the columns by increasing length (one by one). After adding every suffix, the observation table is completed and the closure property is checked. The algorithm continues adding suffixes until a suffix is found, which makes the table not closed and forces refinement. On finding such a suffix, this method stops adding the suffixes to the columns of the observation table. Now, the algorithm makes the table closed by finding a row, which is not equivalent to any of the distinct rows. This row is moved to distinct rows part of the observation table and its one letter extensions are added to the observation table. The process is iterated to make the table closed. Like Rivest and Schapire, Maler and Pnueli, and Shahbaz and Groz, in the upper part of the observation table this method keeps the non equivalent distinct rows. Therefore the observation table is always compatible. On finding the table closed, a conjecture is constructed from the observation table.

At this stage, the algorithm has processed only one distinguishing sequence from the counterexample. There is a possibility that the counterexample string includes other longer distinguishing sequences. Thus, the counterexample is tested on the conjecture. If required, the process of adding sequences from the counterexample is resumed and continued until all of the distinguishing sequences are added.

The Suffix1by1 method adheres to all the qualities of the existing counterexample processing methods. It adds the distinguishing sequences from a counterexample like Rivest and Schapire, it keeps the observation table suffix closed like Maler and Pnueli, and it removes the useless counterexample prefixes like Shahbaz and Groz.

3.6.2 Optimized Mealy Inference Algorithm

The Mealy variants of the algorithm L^* [Niese 2003, Li 2006, Shahbaz 2009] on pattern of the algorithm L^*, use an observation table to record answers of output queries asked to the teacher. These learning algorithms initialize the columns with the elements of inputs set to construct the transition tables for conjectured models. The columns of the observation table are distinguishing sequences for the states of the target models. If the size of inputs set is large, then all of the elements of the inputs set may not be distinguishing sequences. If we initialize the columns of the observation table with a large number of inputs, then it will result in requiring a greater number of output queries. The optimized Mealy inference algorithm L_1 keeps the columns empty and adds only distinguishing sequences (of target model states) and their suffixes to the columns of the observation table. The suffixes are added to keep the observation table suffix closed. Angluin used the prefix closure and suffix closure properties of the observation table to prove the correctness of a conjecture [Angluin 1987].

The Mealy adaptations of L^* initialize the columns with inputs for transition table of conjectured models. For transition tables, the algorithm L_1 records the output for the last input sequence of the access strings labeling the rows of the observation table. The Mealy inference algorithms assume that target models are input enabled as on identifying a distinct state it has to find its successor states for every input (all the one letter extensions). But for software applications, valid inputs for every state are smaller than the set of all inputs. For L_1 the output recorded along access strings helps to identify that an access string is valid or invalid, and only valid access strings are kept in the observation table. The algorithm L_1 augments the columns size of the observation table only to process the counterexamples, i.e. columns of the observation table contain only the distinguishing sequences, and sequences to keep set of column labels suffix closed. To process counterexamples L_1 uses the Suffix1by1 counterexample processing algorithm.

This algorithm does not initialize the columns with I and adds only those elements from I to the columns that are required to distinguish the target model states. For the rows of the observation table this algorithm adds only state successor access strings for valid inputs.

3.6.3 Organizing Output Queries Calculation

For model inference, we are supposed to calculate a set of output queries to complete the observation table. The improved method calculates the *number of prefix output queries* of every output query that belong to the set. Then, an output query is chosen to maximize the number of prefix output queries for every output query. The output query is calculated and its answer is recorded in the dictionaries. The output query and its prefixes are removed from the set and their answers are updated in the observation table. This process is iterated until the set is empty.

3.6.4 Mealy Inference Without Using Counterexamples

The DFA inference algorithms can learn the i/o systems (the software black box implementations) by model transformation techniques by taking the union or cross product of inputs set and outputs set as alphabet. The alphabet size plays a key role for the complexity of the algorithm. The increased size of alphabet results in an increased time complexity of the algorithm. We propose the Mealy inference algorithm $L_{M\text{-}GS}$ which learns i/o systems directly. To fill the observation table, the algorithm asks output queries to the teacher instead of membership queries, using the main settings from the GoodSplit algorithm. The length of sequences added to the columns of the observation table by $L_{M\text{-}GS}$ is $\leq l$ and initially l is 1. The algorithm initializes the columns and rows with the inputs set. Then, it calculates some of the output queries for the observation table, while keeping the other cells empty. All of the output queries are not calculated as the algorithm targets at learning models by consuming less output queries. After calculating 80% of the table cells the algorithm increments l. The examples that we have learned, filling 80% of the cells was sufficient to learn correct models. However, the decision about the number of observation table cells filled and empty is based on target models and it can be changed to adapt in accordance with a target model. This algorithm uses the heuristic explained in Section 3.6.3 to get answers for more output queries by calculating fewer queries.

Like L_1, the algorithm $L_{M\text{-}GS}$ on identifying a distinct state, keeps only valid access strings (for the state successors) in the observation table. Thus, it avoids to calculate the output queries for the rows labeled with invalid access strings. While adding new rows and columns, $L_{M\text{-}GS}$ updates the output query answers that can be inferred from the dictionaries.

In order to construct the transition table, every one letter extension of a distinct row (state) should be a distinct row or at most consistent with one distinct row. Two rows of the observation table are consistent: if the cells of the rows for all of the columns are calculated, then they are equal. Since all of the table cells are not filled, the observation table can have a row that is consistent with more than one distinct state. If we calculate output queries for all of the cells of the row, then immediately we would be able to figure out the corresponding distinct state. But we do not want to consume too much output queries. The algorithm greedily calculates the output queries for the row to find out the corresponding distinct row. The process is iterated until every row becomes distinct or consistent with at most one distinct row.

The algorithm $L_{M\text{-}GS}$ assumes that it knows a bound on the number of states of a target model and when distinct rows size becomes equal to this bound, the algorithm terminates by conjecturing a model. This algorithm is based on the result from Trakhtenbrot and Barzdin [Trakhtenbrot 1973] which indicates that a test set consisting of all strings of length at most about $log_g log_h(n)$ is sufficient to distinguish all distinct states in almost all complete finite state machines, where n is the number of states, g is the number of inputs and h is the number of outputs.

However, if the number of states for a target system are known, we can also calculate the number of membership queries required for inferring the model of the system [Domaratzki 2002].

Chapter 4

Model Inference with the Algorithm L^*

Contents

4.1 The Learning Algorithm L^* 43

4.1.1 Observation Table for Learning Algorithm L^* 44

4.1.2 The Algorithm L^* 46

4.1.3 Complexity of L^* 47

4.2 *DFA* Inference of Mealy Machines and Possible Optimizations 47

4.2.1 Prefix Closure 48

4.2.2 Input Determinism 48

4.2.3 Independence of Events 49

4.3 Mealy Inference 50

4.3.1 The Mealy Inference Algorithm ${L_M}^*$ 51

4.4 Conclusion 55

Angluin introduced the active learning model [Angluin 1981] that infers the models from the answers of queries asked to a teacher. She proved that regular languages can be learned with a polynomial amount of queries [Angluin 1987], the name of the proposed algorithm is L^*. Model inference with the algorithm L^* is a form of machine learning that focuses on induction of formal models. Such induction is particularly useful for learning abstractions of software systems through queries.

In the first section of this chapter, we provide a detailed description of the algorithm L^* along the data structure used for recording the observations and complexity analysis. The second section shows how the algorithm L^* infers the *input*/*output* (i/o) models and how the optimizations adapted for the algorithm L^* enable it to learn models with fewer queries. The third section presents the Mealy adaptation of the algorithm L^*, and the altered data structure used to record the query answers and complexity analysis. The final section provides a conclusion for the chapter.

4.1 The Learning Algorithm L^*

The learning algorithm L^* by Angluin [Angluin 1987] can learn the models of black box implementations. This algorithm assumes the formal model of black box implementations as unknown regular language and this language is learned with the following assumptions:

– the alphabet set Σ is known,

– and the target black box implementation can be reset before executing a query.

The algorithm L^* infers the model of a target black box implementation using the input alphabet Σ of the black box. The algorithm queries sequences of inputs to the black box and organizes the replies in a table known as the observation table. The rows and columns of the observation table are labeled with prefix closed and suffix closed sets of strings, respectively. This algorithm is based on two kinds of interactions with the black box: *membership* and *equivalence queries*. Making a membership query consists in giving to the black box a string that is constructed from the inputs set Σ and observing the answer, which can be recorded as 0 or "reject" and 1 or "accept". The membership queries are iteratively asked until conditions of compatibility and closure on the recorded answers are satisfied (in the original version of the algorithm L^*, the compatibility concept is named as consistency). The answers recorded into an observation table enable the algorithm to construct a conjecture, which is always consistent with the answers. The algorithm then relies on the existence of a *minimally adequate teacher* or *oracle* that knows the unknown model. It asks an equivalence query to the oracle in order to determine whether the conjecture is equivalent to the black box or not. If the black box is not equivalent to the conjecture, the oracle replies with a counterexample or "yes", otherwise. By taking the counterexample into account, the algorithm iterates by asking new membership queries and constructing an improved conjecture, until we get an automaton that is equivalent to the black box. If the inferred *DFA* is not partial model of the black box implementation, then it is the exact behavior of implementation.

4.1.1 Observation Table for Learning Algorithm L^*

The information collected by the algorithm as answers to the membership queries is organized in the observation table. Let $S \subseteq \Sigma^*$ be a *prefix closed* non empty finite set, $E \subseteq \Sigma^*$ a *suffix closed* non empty finite set, and T a finite function defined as $T : ((S \cup S \cdot \Sigma) \times E) \to \{0, 1\}$. The observation table is a triple over the given alphabet Σ and is denoted as (S, E, T). The rows of the observation table are labeled with $S \cup S \cdot \Sigma$ and columns are labeled with E. For a row $s \in S \cup S \cdot \Sigma$ and column $e \in E$, the corresponding cell in the observation table is equal to $T(s, e)$. Now $T(s, e)$ is "1", if $s \cdot e$ is accepted by the target model and "0", otherwise, that is $T(s, e) = \Lambda(q_0, s \cdot e)$. The observation table rows S and columns E are non empty and initially they contain ϵ, that is $S = E = \{\epsilon\}$. The algorithm runs by asking the membership and equivalence queries iteratively. Two rows $s_1, s_2 \in S \cup S \cdot \Sigma$ are said to be equivalent, *iff* $\forall e \in E$, $T(s_1, e) = T(s_2, e)$, and it is denoted as $s_1 \cong s_2$. For every row $s \in S \cup S \cdot \Sigma$, the equivalence class of row s is denoted by $[s]$. The observation table is finally used to construct a *DFA* conjecture. The rows labeled with strings from the prefix closed set S are candidate states for the *DFA* conjecture and the columns labeled with strings from the suffix closed set E are the sequences

to distinguish these states. The $S \cdot \Sigma$ elements are used to build the transitions. An example of the observation table (S, E, T) for *DFA* learning is given in Table 4.1.

		E
		ϵ
S	ϵ	
$S \cdot \Sigma$	a	
	b	

Table 4.1: Example for the observation table (S, E, T), where $\Sigma = \{a, b\}$

To construct a *DFA* conjecture from an observation table, the table must satisfy two properties, closure and compatibility (in the original work the compatibility concept is called consistency). An observation table is closed if for each $s_1 \in S \cdot \Sigma$, there exists $s_2 \in S$ and $s_1 \cong s_2$. The observation table is compatible, if any two rows $s_1, s_2 \in S$, such that $s_1 \cong s_2$, then $s_1 \cdot a \cong s_2 \cdot a$, for $\forall a \in \Sigma$. To construct a conjecture that is consistent with the answers in (S, E, T), the table must be closed and compatible. If the observation table is not closed, then a possible state, which is present in the observation table may not appear in the conjecture. If the observation table is not compatible, then two states marked as equivalent in the observation table might be leading to two different states with same letter $a \in \Sigma$. In other words, if (S, E, T) is not compatible, then there exists $s_1, s_2 \in S$ and $s_1 \cong s_2$, and for some $a \in \Sigma$, $s_1 \cdot a \ncong s_2 \cdot a$. When the observation table (S, E, T) satisfies the closure and compatibility properties, a *DFA* conjecture is build over the alphabet Σ as follows.

Definition 3 *Let the observation table (S, E, T) be closed and compatible, then DFA conjecture $Conj = (Q, \Sigma, \delta, F, q_0)$ is defined, where*

- $Q = \{[s] | s \in S\}$
- $q_0 = [\epsilon]$
- $\delta([s], i) = [s \cdot i], \forall s \in S, \forall i \in \Sigma$
- $F = \{[s] | s \in S \wedge T(s, \epsilon) = 1\}$

In order to verify that this conjecture is well defined with respect to the observations recorded in the table (S, E, T), one can note that as S is a prefix closed non empty set and it always contains ϵ, so q_0 is defined. Similarly as E is non empty suffix closed set, it also always contains ϵ. Thus, if $s_1, s_2 \in S$ and $[s_1] = [s_2]$, then $T(s_1) = T(s_1 \cdot \epsilon)$ and $T(s_2) = T(s_2 \cdot \epsilon)$, are defined and equal, which implies F is well defined. To see that δ is well defined, suppose two elements $s_1, s_2 \in S$ such that $[s_1] = [s_2]$. Since the observation table is compatible, $\forall a \in \Sigma$, $[s_1 \cdot a] = [s_2 \cdot a]$ and the observation table is also closed, so the classes of rows $[s_1 \cdot a]$ and $[s_2 \cdot a]$ are equal to a common row $s \in S$. Hence, the conjecture is well defined.

4.1.2 The Algorithm L^*

The learning algorithm L^* organizes the gathered information into (S, E, T). It starts by initializing the rows S and columns E to $\{\epsilon\}$.

Algorithm 1 The Learning Algorithm L^*

Input: The alphabet Σ
Output: *DFA* conjecture *Conj*

```
begin
    initialize the rows S and columns E of (S,E,T) to {ϵ} and S·Σ = {ϵ·a}, ∀a ∈ Σ
    complete (S,E,T) by asking membership queries s·e such that s ∈ (S ∪ S·Σ)
    ∧ e ∈ E
    repeat
        while (S,E,T) is not closed or not compatible do
            if (S,E,T) is not compatible then
                find s1, s2 ∈ S, a ∈ Σ and e ∈ E, such that
                s1 ≅ s2, but T(s1·a, e) ≠ T(s2·a, e)
                add a·e to E
                complete the table by asking membership queries for the column a·e
            end
            if (S,E,T) is not closed then
                find s1 ∈ S·Σ such that s1 ≇ s2, ∀s2 ∈ S
                move the row s1 to S
                add s1·a to S·Σ, ∀a ∈ Σ
                complete the table by asking membership queries for new added rows
            end
        end
        construct the conjecture Conj from (S,E,T)
        ask the equivalence query to oracle for Conj
        if oracle/teacher replies with a counterexample CE then
            add CE and all the prefixes of CE to S
            complete the table by asking membership queries for new added rows
        end
    until oracle replies "yes" to the conjecture Conj;
    return the conjecture Conj from (S,E,T)
end
```

For all $s \in S$ and $e \in E$, the algorithm performs membership queries $s \cdot e$ and fills the cells of the observation table. Now algorithm ensures that (S, E, T) is compatible and closed. If (S, E, T) is not compatible, incompatibility is resolved by finding two rows $s_1, s_2 \in S$, an alphabet letter $a \in \Sigma$ and and a column $e \in E$ such that $[s_1] = [s_2]$ but $T(s_1 \cdot a, e) \neq T(s_2 \cdot a, e)$, and adding the new suffix $a \cdot e$ to E. If (S, E, T) is not closed, the algorithm searches for $s_1 \in S \cdot \Sigma$ such that $s_1 \ncong s_2$, for all $s_2 \in S$ and makes it closed by adding s_1 to S. This process is iterated until (S, E, T) is closed and compatible. The algorithm L^* eventually constructs

the conjecture from (S, E, T) in accordance with the Definition 3.

4.1.3 Complexity of L^*

Angluin [Angluin 1987] showed that the algorithm L^* can conjecture a minimum *DFA* in polynomial time on the alphabet size $|\Sigma|$, the number of states in minimal conjecture n, and the length of the longest counterexample m. Initially, S and E contain ϵ, each time (S, E, T) is discovered to be not compatible, one string is added to E. Since the observation table can be incompatible for at most $n-1$ times, the total number of strings in E cannot exceed n. Each time (S, E, T) is discovered to be not closed one element is moved from $S \cdot \Sigma$ to S. This can happen for at most $n-1$ times, and there can be at most $n-1$ counterexamples. If the length of the longest counterexample *CE* provided by the oracle is m, for each counterexample at most m strings are added to S. Thus the total number of strings in S cannot exceed $n + m(n-1)$. The worst case complexity of the algorithm in terms of membership queries is: $(n + m(n-1) + (n + m(n-1))|\Sigma|)\ (n) = O(|\Sigma| m n^2)$.

4.2 *DFA* Inference of Mealy Machines and Possible Optimizations

One bottle neck for the learning algorithm L^* is that it may require executing a lot of membership queries, although equivalence query is another tricky issue. Hungar *et al.* [Hungar 2003b] proposed domain specific optimizations for reducing the number of membership queries for regular inference with the Angluin algorithm L^*. To reduce the number of membership queries, they introduced filters, which use properties like input determinism, prefix closure, independence and symmetry of events. Most of the filters are linked to the specific structure of *DFA*, which enables the learning algorithm L^* to learn Mealy models. The reduction is done by having membership queries answered by the filters, in cases where the answer can be deduced from the membership queries of the so-far accumulated knowledge of the system. They proposed the filters for testing reactive systems. They transform Mealy machines to *DFA* models with the help of Definition 4.

Definition 4 *The transformation of a Mealy machine* $\mathcal{M} = (Q_{\mathcal{M}}, I, O, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}}, q_{0\mathcal{M}})$ *to a DFA model* $(Q, \Sigma, \delta, F, q_0)$ *is defined as:*

- $Q \supseteq Q_{\mathcal{M}} \cup \{q_{err}\}$, *where* q_{err} *is a sink state*
- $q_0 = q_{0\mathcal{M}}$,
- $\Sigma = I \cup O$,
- *for each transition* $\delta_{\mathcal{M}}(q, i) = q'$ *where* $i \in I$, $q, q' \in Q_{\mathcal{M}}$ *and* $\lambda_{\mathcal{M}}(q, i) = o$ *where* $o \in O$, *the set* Q *contains the transient states* $q_1 \cdot \ldots \cdot q_n$, *with transitions*
  - $\delta(q, i) = q_1$ *and* $\delta(q_1, o) = q'$

- *the transitions* $\delta(q, o) = q_{err}$ *for all* $q \in Q_{\mathcal{M}}$, *and* $o \in O$,
- *the transitions* $\delta(q_{err}, a) = q_{err}$ *for all* $a \in \Sigma$,
- $F = Q \backslash \{q_{err}\}$.

Mealy models are learned as *DFA*, which have only one sink state, i.e. all the rejecting strings lead to the sink state as described in Definition 4. Contrary to general languages, there is no switching from non-accepting to accepting states. For such languages, the optimization filters can be introduced that are described in the following. For qualitative evaluation of these filters, Hungar *et al.* [Hungar 2003b] carried out experiments on four specific implementations of call center solutions. Each call center solution consists of a telephone switch connected to a fixed number of telephones.

4.2.1 Prefix Closure

The unknown regular language of systems discussed above is prefix closed. In this case, once a string has been evaluated as accepting then in the whole learning process, all the prefixes of the string will be evaluated to accepting without asking to the teacher. Similarly, once a string has been evaluated as rejected, then all continuations of that string will be evaluated to rejected, i.e. all strings which have rejected string prefix to them will be evaluated to rejected without asking to the teacher. Thus the language to be learned is prefix closed. Formally these filters can be expressed as,

- for $\omega, \omega', \omega'' \in \Sigma^*$ and $\omega = \omega' \cdot \omega''$ and $T(\omega) = 1 \implies \Lambda(\omega') = accept$
- for $\omega, \omega' \in \Sigma^*$ and $\omega' \in prefix(\omega)$ and $T(\omega') = 0 \implies \Lambda(\omega) = reject$

For learning prefix closed languages of *DFA* models having single sink state, usage of such filters significantly reduces the number of membership queries. These two optimizations for a set of experiments conducted by Hungar *et al.* on scenarios for call center solution [Hungar 2003b] gave a reduction of 72.22% to 77.85% membership queries.

4.2.2 Input Determinism

The second type of filters use the input determinism property, input determinism means that for a sequence of inputs, from the same state, we always have the same sequence of outputs. These filters are proposed for the scenarios when learning reactive systems as *DFA*, and alphabet for learning such systems is union of input set I and output set O of the target system. The first filter of this type proposes that replacing just one output symbol in a word of an input deterministic language cannot be the word of the same language, for decomposition of ω, a word from the language $\mathcal{L}$, as $\omega = \omega' \cdot o \cdot \omega''$, where o belongs to O, then all the other words made

by replacing o with other elements from alphabet $\Sigma\backslash\{o\}$ does not belong to the language $\mathcal{L}$ or in simple words all such words will be evaluated to rejected. The second filter of this type proposes that replacing one input symbol from I with input symbol from O in a word of an input deterministic language cannot be the word of the same language. For decomposition of ω a word from the language $\mathcal{L}$ as $\omega = \omega' \cdot i \cdot \omega''$, where i belongs to I, then all the other words made by replacing i with elements from O do not belong to the language $\mathcal{L}$ or simply all such words will be rejected. Formally these filters can be expressed as,

- $\exists o \in O, a \in \Sigma\backslash\{o\}$, $\omega, \omega', \omega'' \in \Sigma^*$ such that $\omega = \omega' \cdot o \cdot \omega'' \wedge T(\omega' \cdot a \cdot \omega'') = 1 \wedge o \neq a \implies \Lambda(\omega) = reject$

- $\exists i \in I, o \in O, \omega, \omega' \in \Sigma^*$ such that $\omega' \cdot o \in \mathit{prefix}(\omega) \wedge T(\omega' \cdot i) = 1 \implies \Lambda(\omega) = reject$

The learning results for a set of experiments conducted on scenarios for call center solution [Hungar 2003b] presented by Hungar *et al.* show that after introducing such filters the number of membership queries was reduced further from 50% up to 94.48%. This reduction is the additional reduction for the membership queries after the reduction by first type of filters.

4.2.3 Independence of Events

The intuition of independency in reactive systems can help to avoid asking unnecessary membership queries. Two devices in a system may require to perform actions that are independent of each other, i.e. the order of actions does not affect the results. If an input trace constructed from input traces executing two events independent of each other in two devices of the same system is accepted, then equivalence class of all traces to this trace is accepted. At first independent subparts of a trace with respect to the independence relation are identified, then shuffling these independent sub-traces in any order makes the equivalence class of a trace. The application of this filter along with first two types of filters further reduces the number of membership queries. The experiments on scenarios for call center solution [Hungar 2003b] by Hungar *et al.* show that the reduction with such filters varies from 3.23% to 44.64%.

The experimental results [Hungar 2003b] show that for the considered set of experiments on call center solutions, using all types of filters, there was an overall reduction of 87.03% to 99.78%. For one of the examples initially 132,340 membership queries were asked, whereas after the introduction of all the proposed filters it required only 289 membership queries, hence a reduction of 99.78%. Thus, modifying the learning algorithm by filtering out unnecessary queries enabled to perform quick and efficient learning. The approach of filters is quite flexible and is practical for the fast adaptations to different application domains.

4.3 Mealy Inference

We target at providing the optimized algorithms that can be applied to software testing. Software systems characterize their behaviors in terms of input/output. These systems on getting an input perform computations to take decision on internal transitions and produce output. More natural modeling of such models is Mealy machines. In Section 4.2, we have presented how Mealy machines can be learned with DFA algorithms. In this section we focus on to learn the Mealy models directly. The algorithm L^* can be adapted to learn reactive systems as Mealy machines [Niese 2003, Li 2006, Shahbaz 2009] instead of *DFA* and this considerably improves the efficiency of the learning algorithm. *DFA* models require an intermediate state to model the i/o behavior of reactive systems, whereas Mealy models do not. This is the reason why learning as *DFA* models require far more states to represent the same system as compared to Mealy models. *DFA* models lack the structure of i/o based behavior; Mealy models are more succinct to represent reactive systems. Although, the algorithm L^* can learn Mealy models by model transformation techniques by taking the inputs I and outputs O of target unknown Mealy machine as alphabet for *DFA* as presented in section 4.2. Alphabet set can be collection or product of the inputs and outputs, i.e. $\Sigma = I \cup O$ [Hungar 2003b, Groce 2006] or $\Sigma = I \times O$ [Mäkinen 2001]. But this significantly increases size of alphabet and number of states in learned model, which increases the time complexity of the learning algorithm. The simpler way of handling this problem is to use the modified algorithm L^*, which learns Mealy machines directly.

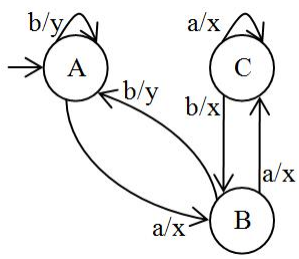

Figure 4.1: Mealy machine.

O. Niese [Niese 2003] proposed a Mealy adaptation of the algorithm L^* to learn reactive systems. He implemented the Mealy inference algorithm and conducted the experiments on four specific implementations of call center solutions. He observed that with the Mealy adaptation of the learning algorithm there was a noteworthy gain in terms of the membership queries and number of states. For instance, one of the examples for call center solutions discussed in O. Niese thesis and also presented in [Hungar 2003b] required 132,340 membership queries without any filter and with filters it required 289 membership queries, whereas O. Niese showed inferring the same example with Mealy inference technique required only 42 membership queries, which is an enormous reduction. For the number of states, there was also a reduction

of 60% to 90.12%.

The Mealy inference algorithm learns the target model by asking *output queries* [Shu 2007]. For Mealy inference, the observation table is filled with output strings instead of *accept* or "1" and *reject* or "0". For the algorithm L^*, the columns of the observation table are initialized with ϵ, whereas the Mealy inference adaptation initializes the columns with the input set I. This change enables the algorithm to detect i/o to annotate the edges in the Mealy machines. The concept of closure and compatibility remains the same as for the algorithm L^*. However, to make the observation table closed and compatible, instead of comparing the boolean values, the outputs recorded in the table cells are compared. To process a counterexample *CE*, the prefixes of *CE* are added to the rows of observation table. To learn the models of software black box components, the Mealy inference algorithm can be used to infer the models with the following assumptions:

- input set I for the target machine is known,
- before each query the learner can always reset the target system to the initial state,
- the i/o interfaces of the machine are accessible, the interface from where an input can be sent is a input interface and the interface from where an output can be observed is an output interface.

4.3.1 The Mealy Inference Algorithm ${L_M}^*$

The Mealy machine inference algorithm ${L_M}^*$ [Niese 2003, Li 2006, Shahbaz 2009] learns the models as Mealy machines using the general settings of the algorithm L^*. The algorithm explores the target model by asking *output queries* [Shu 2007] and organizes the outputs in the observation table. The output queries are iteratively asked until the observation table is closed and compatible. When the observation table is closed and compatible, a Mealy machine conjecture is constructed. The ${L_M}^*$ algorithm then asks equivalence query to the oracle, if the conjectured model is not correct, the oracle replies with a counterexample. The algorithm processes the counterexample to improve the conjecture. If the oracle replies "yes", then the conjecture is correct and the algorithm terminates. We denote the Mealy machine to be learned by the ${L_M}^*$ algorithm in Figure 4.1 as $\mathcal{M} = (Q_{\mathcal{M}}, I, O, \delta_{\mathcal{M}}, \lambda_{\mathcal{M}}, q_{0\mathcal{M}})$. The observation table used by the ${L_M}^*$ algorithm is described as follows.

4.3.1.1 Observation Table for the Mealy Inference Algorithm ${L_M}^*$

The observation table contains the outputs from O^+ by the interactions with the target black box. The ${L_M}^*$ algorithm sends the input strings from I^+ and records the outputs in the observation table. The observation table is defined as a triple (S_M, E_M, T_M), where $S_M \subseteq I^*$ is a prefix closed non empty finite set, which labels the rows of the observation table, $E_M \subseteq I^+$ is a suffix closed non empty finite set, which labels the columns of the observation table and T_M is a finite function,

which maps $(S_M \cup S_M \cdot I) \times E_M$ to outputs O^+. The observation table rows S_M and columns E_M are non empty and initially $S_M = \{\epsilon\}$, $S_M \cdot I = \{i \cdot \epsilon\}$ for all $i \in I$ and $E_M = I$. In the observation table, $\forall s \in S_M \cup S_M \cdot I$, $\forall e \in E_M$, $T_M(s, e) = \mathit{suffix}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e))$. Since S_M is a prefix closed set, which includes ϵ and the columns of the observation table always contain the input set elements, this means that $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s)$ can be calculated from the outputs already recorded in the observation table. Thus, recording the suffixes of answers to the output queries $\mathit{suffix}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, s \cdot e))$ to the observation table is sufficient. A word $\omega = s \cdot e$ is an input string or output query and on executing this output query the black box machine replies with $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, \omega)$. However, in the observation table only $\mathit{suffix}^{|e|}(\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, \omega))$ is recorded. The initial observation table (S_M, E_M, T_M) for Mealy inference of the machine in Figure 4.1 is presented in the Table 4.2, where the input set I has two elements $\{a, b\}$.

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

Table 4.2: Initial observation table for the Mealy machine in Figure 4.1

The equivalence of rows in the observation table is defined with the help of function T_M. Two rows $s_1, s_2 \in S_M \cup S_M \cdot I$ are said to be equivalent, *iff* $\forall e \in E_M$, $T_M(s_1, e) = T(s_2, e)$, and it is denoted as $s_1 \cong s_2$. For every row $s \in S_M \cup S_M \cdot I$, the equivalence class of row s is denoted by $[s]$. Like the algorithm L^*, to construct the conjecture, the ${L_M}^*$ algorithm requires the observation table to satisfy the closure and compatibility properties. The observation table is closed, if $\forall s_1 \in S_M \cdot I$, there exists $s_2 \in S_M$ such that $s_1 \cong s_2$. The observation table is compatible whenever two rows $s_1 \cong s_2$ for $s_1, s_2 \in S_M$ then $s_1 \cdot i \cong s_2 \cdot i$ for $\forall i \in I$. On finding the observation table closed and compatible, the ${L_M}^*$ algorithm eventually conjectures a Mealy machine. The rows labeled with strings from the prefix closed set S_M are the candidate states for the conjecture and the columns labeled with strings from suffix closed set E_M are the sequences to distinguish these states. The Mealy machine conjectured by the ${L_M}^*$ algorithm is always a minimal machine.

Definition 5 *Let (S_M, E_M, T_M) be a closed and compatible observation table, then the Mealy machine conjecture $M_M = (Q_M, I, O, \delta_M, \lambda_M, q_{0M})$ is defined, where*

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

One must show that M_M is a well defined conjecture. Since S_M is a non empty prefix closed set and always contains ϵ, q_{0M} is well defined. Suppose we have two elements $s_1, s_2 \in S_M$ such that $[s_1] = [s_2]$. Since the observation table is compatible, $\forall i \in I$, $[s_1 \cdot i] = [s_2 \cdot i]$ and since the observation table is also closed, the rows $[s_1 \cdot i]$ and $[s_2 \cdot i]$ are equal to a common row $s \in S_M$. Hence, δ_M is well defined. Since E_M is non empty and always contains inputs I, if there exists $s_1, s_2 \in S_M$ such that $s_1 \cong s_2$, then for all $i \in I$, we have $T_M(s_1, i) = T_M(s_2, i)$. Hence λ_M is also well defined.

4.3.1.2 The $L_M{}^*$ Algorithm

The $L_M{}^*$ learning algorithm maintains the observation table (S_M, E_M, T_M).

Algorithm 2 The $L_M{}^*$ Algorithm

Input: Black box and input set I
Output: Mealy machine conjecture M_M
begin
  initialize the rows $S_M = \{\epsilon\}$, columns $E_M = I$ and $S_M \cdot I = \{\epsilon \cdot i\}$, $\forall i \in I$
  complete (S_M, E_M, T_M) by asking output queries $s \cdot e$ such that $s \in (S_M \cup S_M \cdot I) \wedge e \in E_M$
  repeat
    while (S_M, E_M, T_M) *is not closed or not compatible* **do**
      if (S_M, E_M, T_M) *is not compatible* **then**
        find $s_1, s_2 \in S_M, e \in E_M, i \in I$ such that $s_1 \cong s_2$, but $T_M(s_1 \cdot i, e) \neq T_M(s_2 \cdot i, e)$
        add $i \cdot e$ to E_M
        complete the table by asking output queries for the column $i \cdot e$
      end
      if (S_M, E_M, T_M) *is not closed* **then**
        find $s_1 \in S_M \cdot I$ such that $s_1 \ncong s_2$, for all $s_2 \in S_M$
        move s_1 to S_M
        add $s_1 \cdot i$ to $S_M \cdot I$, for all $i \in I$
        complete the table by asking output queries for new added rows
      end
    end
    construct the conjecture M_M from (S_M, E_M, T_M)
    ask the equivalence query to oracle for M_M
    if *if oracle replies with a counterexample CE for* M_M **then**
      add all the prefixes of *CE* to S_M
      complete the table by asking output queries for new added rows
    end
  until *oracle replies "yes" to the conjecture* M_M;
  return the conjecture M_M from (S_M, E_M, T_M)
end

The set of rows S_M is initialized to $\{\epsilon\}$. The Mealy machine associates an output value with each transition edge, this output value is determined both by its current state and the input of the transition edge. The edges of Mealy machines are annotated with i/o, where $i \in I$ and $o \in O$. Thus the set of columns E_M is initialized to I, it enables the algorithm to calculate the corresponding output label o for every $i \in I$ for all the transition edges in the conjectured model.

The output queries are constructed as $s \cdot e$, for all $s \in S_M \cup S_M \cdot I$ and $e \in E_M$, the observation table is completed by asking the output queries. The main loop of the ${L_M}^*$ algorithm tests if (S_M, E_M, T_M) is closed and compatible. If (S_M, E_M, T_M) is not closed, then the ${L_M}^*$ algorithm finds a row $s_1 \in S_M \cdot I$, such that $s_1 \ncong s_2$, for all $s_2 \in S_M$. Then the ${L_M}^*$ algorithm moves s_1 to S_M and completes the table. If (S_M, E_M, T_M) is not compatible, then the ${L_M}^*$ algorithm finds $s_1, s_2 \in S_M$, $e \in E_M$, and $i \in I$ such that $s_1 \cong s_2$ but $T_M(s_1 \cdot i, e)$ is not equal to $T_M(s_2 \cdot i, e)$. Then the string $i \cdot e$ is added to E_M and T_M is extended to $(S_M \cup S_M \cdot I) \cdot (i \cdot e)$ by asking the output queries for missing elements. On finding the observation table (S_M, E_M, T_M) closed and compatible, the ${L_M}^*$ algorithm builds the Mealy machine conjecture in accordance with Definition 5. The algorithm is explained with the help of the Mealy inference of the machine $\mathcal{M}$ in Figure 4.1.

4.3.1.3 Example for Learning with ${L_M}^*$

The ${L_M}^*$ learning algorithm begins by initializing (S_M, E_M, T_M). The rows S_M are initialized to $\{\epsilon\}$ and the columns E_M to $\{a, b\}$. The output queries are asked to complete the table. The initial observation table is shown in Table 4.2. This observation table is closed and compatible, thus the ${L_M}^*$ algorithm conjectures the Mealy machine ${M_M}' = ({Q_M}', I, O, {\delta_M}', {\lambda_M}', {q_0}_M{}')$ shown in Figure 4.2.

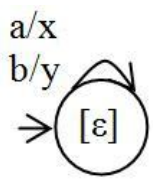

Figure 4.2: The Mealy machine conjecture ${M_M}'$ from Table 4.2

Then, the ${L_M}^*$ algorithm asks the equivalence query to the oracle. Since the conjectured model ${M_M}'$ is not correct, the oracle replies with a counterexample *CE*. There can be more than one counterexamples and oracle selects one from them. Let the counterexample *CE* selected by the oracle be *ababaab*, since

- $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, ababaab) = xyxyxxx$, but
- ${\lambda_M}'({q_0}_M{}', ababaab) = xyxyxxy$.

We have different methods to process the counterexamples, the processing of the counterexample *CE* with different methods is illustrated in Chapter 5.

4.3.1.4 Complexity of $L_M{}^*$

The learning algorithm $L_M{}^*$ can conjecture a minimal Mealy machine in polynomial time on the input size $|I|$, the number of states in minimum conjecture n, and the length of the longest counterexample m. Initially, E_M contains elements from input set I and its size is $|I|$, each time (S_M, E_M, T_M) is discovered to be not compatible, one string is added to E_M. Since the observation table can be incompatible for at most $n-1$ times, the total number of strings in E_M cannot exceed $|I|+n-1$. Initially, S_M contains ϵ, i.e. one element. Each time the observation table (S_M, E_M, T_M) is discovered to be not closed, one element is moved from $S_M \cdot I$ to S_M. This can happen for at most $n-1$ times, and there can be at most $n-1$ counterexamples. If length of longest counterexample CE provided by the oracle is m, for each counterexample at most m strings are added to S. Thus the total number of strings in S cannot exceed $n+m(n-1)$. The worst case complexity of the algorithm for the number of output queries is described as follows:

$$(n+m(n-1) + (n+m(n-1))|I|) \times (|I|+n-1) = O(|I|^2 mn + |I|mn^2).$$

4.4 Conclusion

This chapter presents the DFA model inference algorithm L^*, which can infer the DFA models of black box implementations with membership queries and counterexamples. The learner L^* asks membership queries to the minimally adequate teacher in order to conjecture a model. The correctness of the model is verified from the oracle. If the learned model is not correct, the oracle replies with a counterexample. The algorithm processes the counterexample and this process is iterated until the oracle replies "yes" the learned model is correct.

The algorithm L^* can be used to infer Mealy models by model transformation techniques, but it increases the size of alphabet. Since the size of alphabet is one of the key parameters to the time complexity of the learning algorithm, increased alphabet size results in requiring greater number of membership queries. The filters presented in Section 4.2 can be used to reduce the membership queries.

The Mealy inference algorithm $L_M{}^*$ (the Mealy adaptation of L^*) requires fewer output queries to infer i/o models as compared to L^*. This adaptation requires to change the observation table slightly. For the transition table of a conjectured model, it initializes the columns of the observation table with I. It asks output queries instead of membership queries. On finding the table closed and compatible it conjectures a Mealy machine from the observation table. A detailed study on searching and processing counterexamples for $L_M{}^*$ is provided in Chapter 5.

Chapter 5

Searching and Processing Counterexamples

Contents

5.1 Searching for Counterexamples 58

5.1.1 Counterexamples Search by Random Sampling 59

5.1.2 Howar Algorithm for Counterexample Search 59

5.1.3 Balle Algorithm for Counterexamples Search 60

5.2 Processing Counterexamples 60

5.2.1 Counterexample Processing Algorithm by Angluin 61

5.2.2 Counterexample Processing Algorithm by Rivest and Schapire 63

5.2.3 Counterexample Processing Algorithm by Maler and Pnueli . 65

5.2.4 Counterexample Processing Algorithm by Shahbaz and Groz 68

5.2.5 Issue with Rivest and Schapire Algorithm 69

5.3 The Improved Counterexample Processing Algorithm . . . 72

5.3.1 Motivation for Improved Counterexample Processing Algorithm 72

5.3.2 Counterexample Processing Algorithm Suffix1by1 73

5.3.3 Example for Processing Counterexamples with Suffix1by1 Algorithm 73

5.3.4 Complexity 74

5.3.5 Complexity Comparison 75

5.4 Experiments to Analyze Practical Complexity of Suffix1by1 76

5.4.1 CWB Examples 76

5.4.2 Random Machines 78

5.5 Conclusion 80

Once the learning algorithm $L_M{}^*$ has conjectured a model from the observation table then it relies on the existence of an oracle (teacher), which replies "yes" if the learned model is correct or provides a counterexample, otherwise.

For software black box systems the existence of such an oracle is a strong assumption, which is not met generally. Random sampling as suggested by Angluin [Angluin 1987] and its variants [Cho 2010, Irfan 2010a, Irfan 2010c, Howar 2010, Balle 2010] are simple heuristics to find counterexamples. A counterexample is a string on the inputs I^+, whose output for a black box system under inference and a conjectured model are different. Most of the oracle implementations involve a compromise on the precision and often counterexamples identified by them are not

optimal. For non optimal counterexamples, it becomes important to process them efficiently to avoid asking numerous queries. To address this issue, a number of counterexample processing methods have been proposed [Angluin 1987, Rivest 1993, Maler 1995, Shahbaz 2009]. Angluin proposed a method to process the counterexamples for L^*, this method adds the prefixes of a counterexample to the columns of the observation table [Angluin 1987].

Rivest and Schapire identified that there is always a distinguishing sequence in a counterexample that along an access string from the observation table makes a shorter counterexample. This method adds such a distinguishing sequence to the columns of the observation table requiring a relaxation on the suffix closure property of the observation table. Since this method adds only a single distinguishing sequence from a counterexample, the number of queries for model inference is significantly reduced. Maler and Pnueli [Maler 1995] proposed to add all the suffixes of a counterexample to the columns of the observation table. Their method keeps the observation table suffix closed, however, again the number of queries increases. In an attempt to reduce the number of queries required to learn models, Shahbaz and Groz [Shahbaz 2009] propose to drop the longest prefix of a counterexample which matches any of the access strings in the observation table and add the suffixes of remaining counterexample to the columns of the observation table. The newly proposed counterexample processing method adds the suffixes of a counterexample and stops adding them when a distinguishing suffix is added.

The first section of this chapter discusses the random sampling technique to search for the counterexamples. The second section provides the counterexample processing methods available from the literature. The third section presents the improved counterexample processing method suffix1by1 along the complexity discussion. In order to analyze the practical complexity of the algorithms, the fourth section shows some experimental results on simulated machines and we conclude this chapter in the final section.

5.1 Searching for Counterexamples

For inferring and testing black box software systems, a common procedure to search for counterexamples is a random walk on the inputs. It consists in providing the resulting input sequence in parallel to a black box and a conjectured model to find the differences. On finding such a difference, the trace of all the inputs from the first input on the initial state of the black box to the last input resulting in output difference, is considered as a counterexample. The counterexamples found by this method very often are non optimal and there is a possibility that before reaching a state, where a black box and a conjectured model differ in behavior, many states are compared where in fact the comparison was not required (as they are already there in the conjectured model) [Irfan 2010c]. It is not evident that oracle finds a counterexample at the very first attempt, it may require a number of iterations before finding a counterexample.

To find the counterexamples one can also use conformance testing methods as was done for instance in [Peled 1999] and [Margaria 2004]. But such methods are expensive, especially the Vasilevskii and Chow method [Vasilevskii 1973, Chow 1978] comes at a high exponential cost, its time complexity is $O(l^2 n|\Sigma|^{n-l+1})$, where n is the assumed size of states of the black box automaton, $l \leq n$ is the actual size states (l is the size of states of the conjectured model), and $|\Sigma|$ is the alphabet size.

5.1.1 Counterexamples Search by Random Sampling

In Angluin's black box learning framework, the counterexamples help to iteratively refine the conjectured models. Angluin [Angluin 1987] proposed a random sampling oracle that selects a string x from input set I^+ according to some distribution and returns x with "yes" or "no", yes if x belongs to the language of the unknown model and no, otherwise. This method adapted for Mealy inference constructs the input strings from uniform distribution on inputs and calculates the outputs from the target system and the conjecture to find the discrepancies. All the calls to this oracle are independent from each other. This method may use a lot of strings x before finding a counterexample string. This technique can be improved by generating the counterexample search strings with the objective that every new string covers a different set of states in order to increase the probability of finding undiscovered states.

5.1.2 Howar Algorithm for Counterexample Search

Howar *et al.* [Howar 2010] proposed the Evolving Hypothesis Weighted (*E.H. Weighted*) and Evolving Hypothesis Blocking (*E.H. Blocking*) algorithms that steer the search to find the counterexamples quickly. The *E.H. Weighted* algorithm requires annotating the conjecture transitions with a variable (counter) used to record the number of traversals for a transition. This counter is reset whenever the transition changes. For counterexample search the algorithm uses the conjectured model transition weights (counters) to select a transition for traversal. The probability of selecting a transition is inversely proportional to the increasing weight associated with the transition.

The *E.H. Blocking* counterexample searching method by Howar *et al.* [Howar 2010] exploits the fact that processing a counterexample CE results in moving a row $si \in S \cdot I$ to S. Where, $CE = si \cdot d$ and $d \in I^+$ is a distinguishing sequence, i.e. a counterexample sequence always has an element of set $S.I$ prefix to it followed by some distinguishing sequence (they use Rivest and Schapire [Rivest 1993] counterexample processing method, described later in Section 5.2.2). The *E.H. Blocking* algorithm randomly selects an element from the set $S \cdot I$ and then by randomly selecting inputs from the set I, constructs a string by increasing length, which is tested for being a distinguishing sequence. The length of the string is initialized with ratio to the number of states in a conjecture and is increased after a certain number of unsuccessful attempts. In order to avoid the states that have already been tested, once an $S \cdot I$ element is used, it is excluded from the subsequent counterexample

search tests. This process continues until all the elements of $S \cdot I$ are disabled. If all the elements of $S \cdot I$ are disabled, without learning the target model are enabled again.

5.1.3 Balle Algorithm for Counterexamples Search

Balle *et al.* [Balle 2010] used the algorithms $Balle_{L_1}$ and $Balle_{L_2}$ for learning unknown *DFA* models in the ZULU competition [Combe 2010]. The difference between the two algorithms lies in the method to search for counterexamples. The algorithm $Balle_{L_1}$ uses the uniform distribution over alphabet for the counterexample search, whereas $Balle_{L_2}$ searches the counterexamples from a *random walk* over the states of a conjecture with the probability of selecting each transition between states depending on height of the state successor. The search is based on the assumption that strings generated by traversing more transitions towards shorter leaves in the current conjecture are more likely to be counterexamples. In their implementation they assigned a weight to each transition using the expression

$$w\left(s,\sigma\right) = \left(\frac{1}{h_{\tau(s,\sigma)} - h_{min} + 1}\right)^2,$$

where $h_{\tau(s,\sigma)}$ is the height of the leaf corresponding to the state $\tau(s,\sigma)$ and h_{min} is the height of the shortest leaf in the discrimination tree. Transition probabilities are obtained by normalizing these weights for each state: $p(s,\sigma) = w(s,\sigma)/W_s$ where $W_s = \Sigma_\sigma w(s,\sigma)$. They computed the transition probabilities for the hypothesis according to this rule. Even though they tried to make the $Balle_{L_2}$ algorithm efficient by introducing improvements described above and were expecting gain as compared to $Balle_{L_1}$ for learning the tasks offered by the ZULU challenge, however, they observed that both algorithms performed similarly and statistically there was no significant difference.

5.2 Processing Counterexamples

The length of counterexamples is an important parameter to the complexity of the algorithm ${L_M}^*$, which is evident from the Section 4.3.1.4. The counterexample processing methods play a vital role and directly affect the complexity. That is the reason why a significant number of different counterexample processing methods [Rivest 1993, Maler 1995, Shahbaz 2009] have been proposed.

Angluin's method [Angluin 1987] to process counterexamples adds all prefixes of a counterexample to S_M, and two rows of S_M become equivalent only after processing the counterexamples. Rivest and Schapire [Rivest 1993] identified that incompatibilities in the observation table (S_M, E_M, T_M) can be avoided by keeping the rows S_M distinct. The compatibility condition requires that whenever two rows of S_M are equal, $s_1, s_2 \in S_M$, $s_1 \cong s_2$ then for $\forall i \in I$, $s_1 \cdot i \cong s_2 \cdot i$. But if the S_M rows are always distinct, that is for all $s_1, s_2 \in S_M$, always $s_1 \ncong s_2$, then the compatibility condition is trivially satisfied. The counterexample processing

method by Rivest and Schapire adds only a single distinguishing string from CE to E_M. However, it may make up to $log(m)$ output queries to find such a string, where $m = |CE|$. This method maintains the condition that all rows S_M of observation table are distinct, that is for all $s_1, s_2 \in S_M$, $s_1 \ncong s_2$.

The counterexample processing methods proposed by Maler and Pnueli [Maler 1995], and Shahbaz and Groz [Shahbaz 2009] also add sequences only to columns E_M. Since for these methods S_M augments only when the observation table is not closed, this keeps the S_M elements distinct. Thus, always $|S_M| \leq n$, where n is the number of states in a conjectured model. We present the counterexample processing methods adapted for the algorithm ${L_M}^*$ in the following.

5.2.1 Counterexample Processing Algorithm by Angluin

The counterexample processing method by Angluin [Angluin 1987] requires to add all the prefixes of a counterexample to the rows of the observation table. This method can be adapted for Mealy inference. If we have a counterexample CE, this method adds CE and all the prefixes of CE to S_M. Then the observation table is completed by extending T_M to $(S_M \cup S_M \cdot I) \cdot (E_M)$ and by asking output queries for new added rows. This method adapted for Mealy inference can be presented as Algorithm 3.

Algorithm 3 Counterexample Processing by Angluin for ${L_M}^*$

Input: Pre-refined observation table (S_M, E_M, T_M), CE
Output: Refined observation table (S_M, E_M, T_M)

```
begin
    for j = 1 to |CE| do
        if prefix^j(CE) ∉ S_M then
            if prefix^j(CE) ∈ S_M · I then
                move the row prefix^j(CE) to S_M
            end
            else
                add prefix^j(CE) to S_M
            end
        end
    end
    construct the output queries for the new rows
    complete (S_M, E_M, T_M) by executing output queries
    make (S_M, E_M, T_M) closed and compatible
    return refined observation table (S_M, E_M, T_M)
end
```

After adding the prefixes of CE to S_M and completing the observation table, the table can be not closed or incompatible. The algorithm ${L_M}^*$ makes the observation

table closed and compatible, when both of these properties are satisfied, ${L_M}^*$ conjectures the Mealy machine from the table. This method is explained with the help of following example.

5.2.1.1 Example for Processing Counterexamples with Angluin Algorithm

While learning the Mealy machine $\mathcal{M}$ in Figure 4.1 with ${L_M}^*$, for the initial conjecture ${M_M}'$ in Figure 4.2, the oracle replies with a counterexample $CE = ababaab$. The counterexample processing method by Angluin requires to add CE and all of its prefixes that are not in S_M to to the distinct rows of the observation table S_M, i.e. $a, ab, aba, abab, ababa, ababaa, ababaab$ to the set S_M and the one letter extensions of these prefixes to $S_M \cdot I$ (one letter extensions that are not already member of $S_M \cup S_M \cdot I$), i.e. $aa, abb, abaa, ababb, ababab, ababaaa, ababaaba, ababaabb$ to the set $S_M \cdot I$. The function T_M is extended to $(S_M \cup S_M \cdot I) \cdot E_M$ by means of output queries for the missing entries. After adding the prefixes of counterexample the observation table is presented in Table 5.1a.

	a	b
$\underline{\epsilon}$	x	y
a	x	y
ab	x	y
aba	x	y
$abab$	x	y
$\underline{ababa}$	x	y
$ababaa$	x	x
$ababaab$	x	y
b	x	y
aa	x	x
abb	x	y
$abaa$	x	x
$ababb$	x	y
$ababab$	x	y
$ababaaa$	x	x
$ababaaba$	x	x
$ababaabb$	x	y

(a) The observation table after adding prefixes of CE.

	a	b	ab
ϵ	x	y	xy
a	x	y	xx
ab	x	y	xy
aba	x	y	xx
$abab$	x	y	xy
$ababa$	x	y	xx
$ababaa$	x	x	xx
$ababaab$	x	y	xx
b	x	y	xy
aa	x	x	xx
abb	x	y	xy
$abaa$	x	x	xx
$ababb$	x	y	xy
$ababab$	x	y	xy
$ababaaa$	x	x	xx
$ababaaba$	x	x	xx
$ababaabb$	x	y	xy

(b) To make the observation table compatible ab is added to E_M.

Table 5.1: The observation table in Table 5.1a after adding prefixes of CE is incompatible for rows with underlined labels. The observation table in Table 5.1b is made compatible by adding ab to E_M.

Since $\epsilon, ababa \in S_M$, $a \in I$ and $b \in E_M$ such that the underlined row labels $\epsilon \cong ababa$, but $T_M(\epsilon \cdot a, b)$ is not equal to $T_M(ababa \cdot a, b)$, the observation table in Table 5.1a is closed but not compatible. Thus, this method adds the string ab to E_M. The observation table is completed by executing the output queries for the newly added column ab. The observation table in Table 5.1b is closed and compatible, thus, $L_M{}^*$ conjectures the Mealy machine shown in Figure 7.4.3.

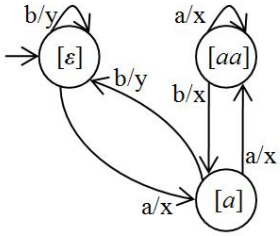

Figure 5.1: Conjectured Mealy machine.

The conjectured model is correct, so the oracle replies "yes" and $L_M{}^*$ terminates. Thus, a total of 51 output queries were asked to learn this example. The worst case complexity of the algorithm for number of output queries is $O(|I|^2mn + |I|mn^2)$, where $|I|$ is the size of input set, m is the length of the longest counterexample provided by the oracle and n is the number of states in the learned model.

5.2.2 Counterexample Processing Algorithm by Rivest and Schapire

The counterexample processing method by Rivest and Schapire [Rivest 1993] adapted for $L_M{}^*$ significantly improves the worst case number of output queries required to learn the Mealy machines. In the observation table (S_M, E_M, T_M), S_M is a prefix closed set representing the states of the conjecture. The counterexample processing method by Rivest and Schapire maintains the condition that all rows S_M of observation table are distinct, i.e. for all $s_1, s_2 \in S_M$, $s_1 \ncong s_2$. Instead of adding prefixes of a counterexample CE to S_M, it adds only a single distinguishing string from CE to E_M. However, it may make up to $log(m)$ output queries to find such a string, where m is the length of CE, i.e. $m = |CE|$. The counterexample processing method by Rivest and Schapire can be described as follows.

Let u_j be a sequence made up of first j actions in CE and v_j be a sequence made up of actions after first j actions, thus, $CE = u_j \cdot v_j$, where $0 \leq j \leq m$ such that $u_0 = v_m = \epsilon$ and $u_m = v_0 = CE$. Now, if we run u_j on a conjecture, the conjecture moves to some state q, where $q \in Q_M$. By construction this state q corresponds to a row $s \in S_M$. For the output query $s \cdot v_j$, let α_j be the output from a target Mealy machine $\mathcal{M}$ and β_j be the output from a conjecture. For a counterexample CE, we have $\alpha_0 \neq \beta_0$ and $\alpha_m = \beta_m$, and the point where $\alpha_j \neq \beta_j$ and $\alpha_{j+1} = \beta_{j+1}$ can be found using the binary search in $log(m)$ output queries. Binary search can be performed by initializing j to $m/2$, if $\alpha_j \neq \beta_j$ then $j = 3(m/4)$ and $j = m/4$,

otherwise. On finding such j, this method adds v_{j+1} to E_M. For $j < m$ and $i_j \in I$, the counterexample is $u_j \cdot i_j \cdot v_{j+1} = u_j \cdot v_j = CE$ and v_{j+1} distinguishes $s \cdot i_j$ from distinct rows of S_m, where $s \cdot i_j \in S_M \cdot I$. It is presented as Algorithm 4.

Algorithm 4 Counterexample Processing by Rivest and Schapire for ${L_M}^*$

Input: Pre-refined observation table (S_M, E_M, T_M), CE, Conjecture
Output: Refined observation table (S_M, E_M, T_M)
1: $CE = u_j \cdot v_j$, where u_j are first i actions and v_j are subsequent actions in CE
2: $q = \delta(q_{0M}, u_j)$ by running u_j on Conjecture, where $q \in Q_M$
3: Find the row $s \in S$, which corresponds to q
4: Calculate outputs α_j, β_j for output query $s \cdot v_j$ from target Mealy machine and learned Conjecture, respectively
5: By binary search find the point where $\alpha_j \neq \beta_i$ and $\alpha_{j+1} = \beta_{j+1}$
6: $u_j \cdot i_j \cdot v_{j+1} = u_j \cdot v_j = CE$, $j < m$
7: Add v_{j+1} to E_M
8: Construct the output queries for the new column
9: Complete (S_M, E_M, T_M) by executing output queries
10:Make (S_M, E_M, T_M) closed
11:**return** refined observation table (S_M, E_M, T_M)

If ${L_M}^*$ processes counterexamples with this method then size of S_M augments only when the observation table is not closed. Thus, we always have $|S_M| \leq n$, where n is the number of states in the conjecture. Since the compatibility condition requires that whenever two rows of S_M are equal, $s_1, s_2 \in S_M$, $s_1 \cong s_2$ then for $\forall i \in I$, we have $s_1 \cdot i \cong s_2 \cdot i$. To process counterexamples with Angluin's algorithm, ${L_M}^*$ requires to add all prefixes of a counterexample to S_M and two rows of S_M become equivalent only after processing counterexamples. Since this counterexample processing method maintains the condition for all $s_1, s_2 \in S_M$, $s_1 \ncong s_2$, compatibility condition is always trivially satisfied.

Balcázar *et al.* pointed out that after processing a counterexample with this method the conjectured model may still classify the counterexample incorrectly, as another longer distinguish sequence from the same counterexample can improve the conjecture [Balcázar 1997]. To address this, they propose to process distinguishing sequences from a counterexample until no further distinguishing sequence can be identified. This method adds only one suffix of a counterexample to E_M and requires a compromise on the suffix closure property for the observation table, consequently, the new conjecture may not be minimal and consistent with the observation table. The worst case complexity of ${L_M}^*$ adapting this counterexample processing method in terms of output queries is $O(|I|^2 n + |I| n^2 + n log(m))$.

5.2.2.1 Example for Processing Counterexamples with Rivest and Schapire Algorithm

While learning the Mealy machine $\mathcal{M}$ in Figure 4.1 with ${L_M}^*$, for the initial conjecture ${M_M}'$ in Figure 4.2, the oracle replies with a counterexample $CE = ababaab$.

The counterexample processing method by Rivest and Schapire searches for the distinguishing sequence by means of binary search. The binary search divides $ababaab$ as $u_j \cdot v_j$, here $u_j = abab$ and $v_j = aab$. Now, by running $abab$ on the conjecture $M_M{}'$, the corresponding string from the observation table for the state reached is ϵ. The potential shorter counterexample sequence is $aab = \epsilon \cdot aab$. By running the output query aab on $\mathcal{M}$, we get

- $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, aab) = xxx$, but
- $\lambda_M{}'(q_{0M}{}', aab) = xxy$.

The search for shorter distinguishing sequence continues by selecting $u_j = ababaa$ and $v_j = b$, the string from the observation table for state reached is ϵ. New candidate sequence is $b = \epsilon \cdot b$, but

- $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, b) = x$, and
- $\lambda_M{}'(q_{0M}{}', b) = x$.

The values for u_j and v_j are changed to $ababa$ and ab, respectively. New candidate sequence is $ab = \epsilon \cdot ab$, we get

- $\lambda_{\mathcal{M}}(q_{0\mathcal{M}}, ab) = xx$, and
- $\lambda_M{}'(q_{0M}{}', ab) = xx$.

The binary search ends by finalizing $u_j = abab$ and $v_j = aab$. By running $abab$ on the conjecture $M_M{}'$, we get the access string ϵ. The row $\epsilon \cdot a$ along the column ab will make the table not closed. After adding the suffix $v_{j+1} = ab$, the observation table is presented in Table 5.2a.

From the observation table in Table 5.2a, it can be observed that the row $a \in S_M \cdot I$ is not equal to the only member ϵ of S_M, thus, the observation table is not closed. To make the table closed, the row a is moved to S_M and its one letter extensions are added to $S_M \cdot I$. Now, again the row aa of the observation table in Table 5.2b makes the table not closed. It is made closed by adding aa to S_M. The observation table in Table 5.2c is closed. Since with this counterexample processing method the observation table is always compatible, the Mealy machine shown in Figure 7.4.3 is conjectured. The conjectured model is correct, so the oracle replies "yes" and the algorithm terminates. Thus, in total $24 = (21 + 3)$ output queries are asked.

5.2.3 Counterexample Processing Algorithm by Maler and Pnueli

The counterexample processing method by Maler and Pnueli [Maler 1995] does not require any compromise on the suffix closure property of the observation table. This method adds a counterexample CE and all the suffixes of CE to E_M that are not

	a	b	ab
ϵ	x	y	xy
$\underline{a}$	x	y	xx
b	x	y	xy

(a) Observation table after adding the distinguishing suffix $v_{j+1} = ab$.

	a	b	ab
ϵ	x	y	xy
a	x	y	xx
b	x	y	xy
$\underline{aa}$	x	x	xx
ab	x	y	xy

(b) Make the observation table closed by moving the row a to S_M.

	a	b	ab
ϵ	x	y	xy
a	x	y	xx
aa	x	x	xx
b	x	y	xy
ab	x	y	xy
aaa	x	x	xx
aab	x	y	xx

(c) Make the observation table closed by moving aa to S_M.

Table 5.2: After adding the distinguishing suffix $v_{j+1} = ab$ to E_M, we get the observation table in Table 5.2a. The underlined rows are the rows, which make the table not closed. The observation table in Table 5.2c is closed.

already member of the set E_M. Since E_M always contains the set of inputs I, the algorithm adds only the suffixes of size ≥ 2. Thus, the observation table is always suffix closed and the improved conjecture is always consistent to the observation table. Like Rivest and Schapire the size of distinct rows S_M augment only when a row from $S_M \cdot I$ is identified as distinct and is moved to S_M, thus, avoiding the incompatibilities trivially. The algorithm is presented as Algorithm 5.

Algorithm 5 Counterexample Processing by Maler and Pnueli for ${L_M}^*$

Input: Pre-refined observation table (S_M, E_M, T_M), CE
Output: Refined observation table (S_M, E_M, T_M)
begin
  for $j = 2$ *to* $|CE|$ **do**
    if $\mathit{suffix}^j(CE) \notin E_M$ **then**
      add $\mathit{suffix}^j(CE)$ to E_M
    end
  end
  construct the output queries for the new columns
  complete (S_M, E_M, T_M) by executing output queries
  make (S_M, E_M, T_M) closed
  return refined observation table (S_M, E_M, T_M)
end

The observation table is completed for missing entries by executing the output queries. As S_M elements are always distinct, the observation table is compatible. If the observation table is not closed, it is made closed by finding $s_1 \in S_M \cdot I$ such that $s_1 \ncong s_2$ for all $s_2 \in S_M$ and then adding s_1 to S_M. On finding the table closed

the Mealy machine is conjectured.

5.2.3.1 Example for Processing Counterexamples with Maler and Pnueli Algorithm

While learning the Mealy machine $\mathcal{M}$ in Figure 4.1 with $L_M{}^*$, for the initial conjecture $M_M{}'$ in Figure 4.2, the oracle replies with a counterexample $CE = ababaab$. The counterexample processing method by Maler and Pnueli adds the counterexample $ababaab$ and elements of its prefixes set to the set E_M (the elements that are not already member of E_M), i.e. $ababaab$ and $babaab, abaab, baab, aab, ab$ to columns of the observation table. The function T_M is extended to $(S_M \cup S_M \cdot I) \cdot E_M$ by means of output queries for the missing entries. After adding the suffixes of v, the observation table is presented in Table 5.3a.

	a	b	ab	aab	$baab$	$abaab$	$babaab$	$ababaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
$\underline{a}$	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$	$xyxyxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$

(a) Observation table after adding CE and its suffixes.

	a	b	ab	aab	$baab$	$abaab$	$babaab$	$ababaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
a	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$	$xyxyxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
$\underline{aa}$	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$	$xxxxxxx$
ab	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$

(b) Make the observation table closed by moving the row a to S_M.

	a	b	ab	aab	$baab$	$abaab$	$babaab$	$ababaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
a	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$	$xyxyxxx$
aa	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$	$xxxxxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
ab	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$	$xyxyxxx$
aaa	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$	$xxxxxxx$
aab	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$	$xyxyxxx$

(c) Make the observation table closed by moving aa to S_M.

Table 5.3: After adding $ababaab$ and its suffixes $babaab, abaab, baab, aab, ab$ to E_M, we get the observation table in Table 5.3a. The underlined rows are the rows, which make the table not closed. The observation table in Table 5.3c is closed.

From the observation table in Table 5.3a, it can be observed that the row $a \in S_M \cdot I$ is not equal to the only member ϵ of S_M, thus, the observation table is not

closed. To make the table closed, the row a is moved to S_M and its one letter extensions are added to $S_M \cdot I$. Now, again the row aa of the observation table in Table 5.3b makes the table not closed. It is made closed by moving aa to S_M. The observation table in Table 5.3c is closed. Since with this counterexample processing method the observation table is always compatible, the Mealy machine shown in Figure 7.4.3 is conjectured. The conjectured model is correct, so the oracle replies "yes" and the algorithm terminates. Thus, in total 56 output queries are asked.

5.2.4 Counterexample Processing Algorithm by Shahbaz and Groz

The counterexample processing method by Shahbaz and Groz [Shahbaz 2009] operates in the same manner as the counterexample processing method by Maler and Pnueli [Maler 1995]. The only difference is that before adding the suffixes of a counterexample CE to E_M, this method drops the longest prefix of CE that matches any element of $S_M \cup S_M \cdot I$, whereas the method by Maler and Pnueli adds all the suffixes of counterexample to E_M. Thus, for this method suffix closure property is also trivially satisfied. As shown by Rivest and Schapire [Rivest 1993], incompatibilities can arise only when we have equivalent states in S_M, which can happen on adding counterexample prefixes to S_M. The counterexample processing algorithm by Shahbaz and Groz adds counterexample suffixes to E_M, and S_M augments only when the observation table is not closed, thus, all the rows labeled by S_M elements are always distinct. After adding the suffixes to the observation table, it is required to make the table closed as it is always compatible. The algorithm is presented as Algorithm 6.

Algorithm 6 Counterexample Processing by Shahbaz and Groz for ${L_M}^*$

Input: Pre-refined observation table (S_M, E_M, T_M), CE
Output: Refined observation table (S_M, E_M, T_M)
begin
  divide CE as $u \cdot v$, where u is the longest prefix of CE such that $u \in (S_M \cup S_M \cdot I)$
  for $j = 2$ *to* $|v|$ **do**
    if $\mathit{suffix}^j(v) \notin E_M$ **then**
      add $\mathit{suffix}^j(v)$ to E_M
    end
  end
  construct the output queries for the new columns
  complete (S_M, E_M, T_M) by executing output queries
  make (S_M, E_M, T_M) closed
  return refined observation table (S_M, E_M, T_M)
end

The counterexample processing method by Shahbaz and Groz divides CE as $u \cdot v$ where u is the longest prefix in $S_M \cup S_M \cdot I$. It adds v and all the suffixes of v to E_M that are not already member of the set E_M. Since E_M always contains

the set of inputs I, the algorithm begins by adding the suffixes of size ≥ 2. Then observation table is completed for missing entries. As S_M elements are always distinct so observation table is compatible. If the observation table is not closed, it is made closed by finding $s_1 \in S_M \cdot I$ such that $s_1 \not\cong s_2$ for all $s_2 \in S_M$ and then moving s_1 to S_M. On finding the table closed the Mealy machine is conjectured.

5.2.4.1 Example for Processing Counterexamples with Shahbaz and Groz Algorithm

While learning the Mealy machine $\mathcal{M}$ in Figure 4.1 with ${L_M}^*$, for the initial conjecture ${M_M}'$ in Figure 4.2, the oracle replies with a counterexample $CE = ababaab$. The counterexample processing method by Shahbaz and Groz divides $ababaab$ as $u \cdot v$, where $u = a$ and $v = babaab$ ($a \in S_M \cup S_M \cdot I$ is the longest prefix of CE in the Table 4.2). Then it adds v and all the suffixes of v to E_M that are not already in E_M, i.e. $babaab, abaab, baab, aab, ab$ to the set E_M. The function T_M is extended to $(S_M \cup S_M \cdot I) \cdot E_M$ by means of output queries for the missing entries. After adding the suffixes of v, the observation table is presented in Table 5.4a.

From the observation table in Table 5.4a, it can be observed that the row $a \in S_M \cdot I$ is not equal to the only member ϵ of S_M, thus, the observation table is not closed. To make the table closed, the row a is moved to S_M and its one letter extensions are added to $S_M \cdot I$. Now, again the row aa of the observation table in Table 5.4b makes the table not closed. It is made closed by adding aa to S_M. The observation table in Table 5.4c is closed. Since with this counterexample processing method the observation table is always compatible, the Mealy machine shown in Figure 7.4.3 is conjectured. The conjectured model is correct, so the oracle replies "yes" and the algorithm terminates. Thus, in total 49 output queries were asked.

5.2.5 Issue with Rivest and Schapire Algorithm

The Rivest and Schapire counterexample processing algorithm searches for the smallest distinguishing sequence from a counterexample and adds the suffix to the columns E_M of the observation table. The learning algorithm ${L_M}^*$ requires the observation table to be prefix closed for the row labels S_M and suffix closed for column labels E_M. The Rivest and Schapire counterexample processing method requires a compromise on the suffix closure property of the observation table. Thus, the algorithm is not sound for the example in Figure 5.2. We explain this fact by learning the Mealy machine provided in Figure 5.2.

5.2.5.1 Example that creates problem with Rivest and Schapire Algorithm

To learn the Mealy machine $\mathcal{M}_1$ provided in Figure 5.2, the algorithm ${L_M}^*$ begins by initializing (S_M, E_M, T_M). The rows S_M are initialized to $\{\epsilon\}$, and columns E_M to I. The initial observation table after executing the output queries for $\{S_M \cup S_M \cdot I\} \times E_M$ is shown in Table 5.5.

	a	b	ab	aab	$baab$	$abaab$	$babaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
$\underline{a}$	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$

(a) Observation table after adding suffixes of v.

	a	b	ab	aab	$baab$	$abaab$	$babaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
a	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
$\underline{aa}$	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$
ab	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$

(b) Make the observation table closed by moving the row a to S_M.

	a	b	ab	aab	$baab$	$abaab$	$babaab$
ϵ	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
a	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$
aa	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$
b	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
ab	x	y	xy	xxx	$yxxx$	$xyxxx$	$yxyxxx$
aaa	x	x	xx	xxx	$xxxx$	$xxxxx$	$xxxxxx$
aab	x	y	xx	xxx	$yxxx$	$xxxxx$	$yxyxxx$

(c) Make the observation table closed by moving aa to S_M.

Table 5.4: After adding v and all suffixes of v to E_M, we get the observation table in Table 5.4a. The underlined rows are the rows, which make the table not closed. The observation table in Table 5.4c is closed.

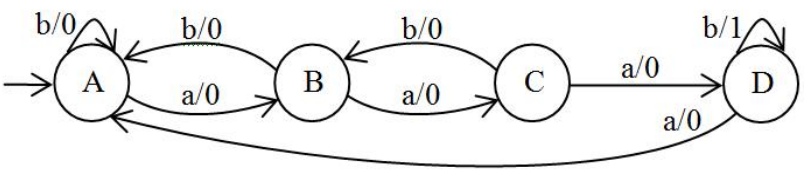

Figure 5.2: Mealy machine with $I = \{a, b\}$.

Table 5.5: Initial Observation Table for the Mealy machine in Figure 5.2

		E_M	
		a	b
S_M	ϵ	0	0
$S_M \cdot I$	a	0	0
	b	0	0

The observation table of Table 5.5 is closed and compatible, thus, the algorithm $L_M{}^*$ conjectures the Mealy machine $Conj' = (Q_C{}', I, O, \delta_C{}', \lambda_C{}', q_{0C}{}')$ shown in Figure 5.3.

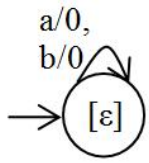

Figure 5.3: The Mealy machine conjecture $Conj'$ from Table 5.5

Now, in order to verify the correctness of the conjectured model, the algorithm $L_M{}^*$ asks an equivalence query to the oracle. Since the conjectured model $Conj'$ is not correct, the oracle replies with a counterexample. There can be more than one counterexamples and the oracle selects one from them. Since

- $\lambda(q_0, baaab) = 00001$, but
- $\lambda_C{}'(q_{0C}{}', baaab) = 00000$.

Thus, the counterexample CE returned by the oracle is *baaab/00001* and by using the binary search the Rivest and Schapire counterexample processing method finds $u_j = b$ and $v_j = aaab$. The corresponding access string from the table is ϵ. This method adds $v_{j+1} = aab$ to the columns E_M of the observation table. After adding this distinguishing string, the observation table is presented in Table 5.6a.

The observation table in Table 5.6a is not closed and it is made closed by moving the row a from $S_M \cdot I$ to the distinct rows S_M. The observation table in Table 5.6b is closed. Since with this counterexample processing method the observation table always remains compatible, the Mealy machine shown in Figure 5.4 is conjectured.

The conjectured Mealy machine shown in Figure 5.4 is neither a minimal machine nor consistent with the observations of Table 5.6b.

Again the oracle replies with the counterexample *baaab*, the algorithm again finds $u_j = b$ and $v_j = aaab$. But $v_{j+1} = aab$ is not a distinguishing sequence for the row $\epsilon \cdot a$, as the suffix *aab* is already there in the observation table and the row $\epsilon \cdot a \in S_M$. Thus, the algorithm fails to find a distinguishing sequence.

	a	b	aab
ϵ	0	0	000
$\underline{a}$	0	0	001
b	0	0	000

(a) Observation Table after adding distinguishing string $aaab$

	a	b	aab
ϵ	0	0	000
a	0	0	001
b	0	0	000
aa	0	0	000
ab	0	0	000

(b) To make the table closed moving the row a to S_M.

Table 5.6: The underlined rows are the rows which make the table not closed. The observation table in Table 5.6b is closed.

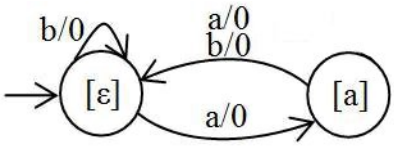

Figure 5.4: The Mealy machine conjecture $Conj''$ from Table 5.6b

5.3 The Improved Counterexample Processing Algorithm

In this section, we present an improved method to process the counterexamples, which is named as Suffix1by1 [Irfan 2010c]. This method adapted for the algorithm $L_M{}^*$ can be presented as the Algorithm 7. We illustrate our algorithm with the help of Example 5.3.3. The discussion about the complexity of the algorithm is provided in Section 5.3.4.

5.3.1 Motivation for Improved Counterexample Processing Algorithm

The Mealy machine conjecture $M_M{}'$ in Figure 4.2 of machine $\mathcal{M}$ in Figure 4.1 is not correct and in the previous section we have used the sequence of inputs *ababaab* as a counterexample. Now, if we look carefully at both the conjectured model $M_M{}'$ and the target machine $\mathcal{M}$, we may recognize the fact that the minimal string of inputs required to distinguish these machines is *aab*. The set of inputs I is $\{a, b\}$ and with random sampling the probability to select the string *aab* as a counterexample is 1/8. From this, it can be deduced that there is a strong possibility that the counterexample provided by the oracle will be preceded by some useless sequence. Even if we consider the guided counterexample searching techniques by Howar *et al.* [Howar 2010] and Balle *et al.* [Balle 2010], these techniques do not provide the minimal counterexamples. In the case when *ababaab* is a counterexample, it can be noticed that the minimal counterexample is a suffix to the provided counterexample. However, this is not the standard case, we can have useless sequences at the

beginning or at any later stage. Considering suffixes of a counterexample results in a smaller counterexample, which is closer to minimal counterexample as compared to originally found counterexample.

5.3.2 Counterexample Processing Algorithm Suffix1by1

Like Rivest and Schapire [Rivest 1993], Maler and Pnueli [Maler 1995], and Shahbaz and Groz [Shahbaz 2009], the Suffix1by1 counterexample processing method adds the suffixes of a counterexample to the columns E_M of the observation table and the size of S_M augments only to make the observation table closed, (when a row from $S_M \cdot I$ is moved to S_M). If we have a counterexample CE, this method adds the suffixes one by one by increasing length from CE to E_M. Each time a suffix is added, the observation table is completed and the closure property is checked. The algorithm carries on this process until a suffix is found, which makes the table not closed and forces refinement. On finding such a suffix, this method stops adding the suffixes of CE to E_M and conjectures a Mealy machine.

Since a part of the counterexample CE having a distinguishing sequence is added to the observation table, the conjectured model may still classify the sequence CE as counterexample. After adding a distinguishing suffix from CE and conjecturing a model, the algorithm checks if CE is again a counterexample for the conjecture. If CE is again a counterexample, this means a longer suffix from the same sequence CE can still improve the conjecture. The algorithm continues adding suffixes from CE until a suffix is found, which makes the table not closed. The algorithm makes the table closed and conjectures a model. This process is continued until CE can no longer help to improve the conjectured Mealy machine. This algorithm is described as Algorithm 7.

This method of processing the counterexamples also keeps the S_M members distinct, thus, while processing the counterexamples with this method, the observation table is always compatible. As the experiments have shown, in the case of non-optimal counterexamples, this method can have a big impact on complexity. The rationale behind the proposed improvement in this method comes from the observation that random walks on uniform distribution of inputs to search for the counterexamples can cycle through states of a black box that are already present in a conjectured model, before reaching undiscovered states. Therefore, only the tail parts of such counterexamples actually correspond to discriminating sequences. Thus, considering suffixes of such counterexamples makes it possible to reduce the negative impact of unproductive cycles.

5.3.3 Example for Processing Counterexamples with Suffix1by1 Algorithm

While learning the Mealy machine $\mathcal{M}$ in Figure 4.1 with ${L_M}^*$, for the initial conjecture ${M_M}'$ in Figure 4.2, the oracle replies with a counterexample $CE = ababaab$. The Suffix1by1 counterexample processing method adds the smallest suffix of CE to

Algorithm 7 Counterexample Processing Suffix1by1

Input: Pre-refined observation table (S_M, E_M, T_M), CE
Output: Refined observation table (S_M, E_M, T_M)

```
begin
    while CE is a counterexample do
        for j = 2 to |CE| do
            if suffix^j(CE) ∉ E_M then
                add suffix^j(CE) to E_M
                construct the output queries for the new columns
                complete (S_M, E_M, T_M) by executing output queries
                if (S_M, E_M, T_M) is not closed then
                    break for loop
                end
            end
        end
        make (S_M, E_M, T_M) closed
        construct the conjecture M_M
    end
    return refined observation table (S_M, E_M, T_M)
end
```

E_M that is not already member of E_M. Since E_M is initialized with I, it starts by adding the suffix of length 2 to the observation table, it adds ab to the set E_M. The function T_M is extended to $(S_M \cup S_M \cdot I) \cdot E_M$ by means of output queries for the newly added column. After adding the suffix ab, the observation table is presented in Table 5.7a.

From the observation table in Table 5.7a, it can be observed that row $a \in S_M \cdot I$ is not equal to the only member ϵ of S_M, thus, the observation table is not closed. To make the table closed, the row a is moved to S_M and its one letter extensions are added to $S_M \cdot I$. Now, again the row aa of the observation table in Table 5.7b makes the table not closed. It is made closed by moving aa to S_M. The observation table in Table 5.7c is closed. This counterexample processing method always keeps the observation table compatible. So the Mealy machine shown in Figure 7.4.3 is conjectured. The conjectured model is correct, so the oracle replies "yes" and the algorithm terminates. Thus, in total 21 output queries are asked.

5.3.4 Complexity

The learning algorithm $L_M{}^*$ adapting the Suffix1by1 counterexample processing method can conjecture a minimal Mealy machine in polynomial time on the following

	a	b	ab
ϵ	x	y	xy
$\underline{a}$	x	y	xx
b	x	y	xy

(a) The observation table after adding the suffix ab.

	a	b	ab
ϵ	x	y	xy
a	x	y	xx
b	x	y	xy
$\underline{aa}$	x	x	xx
ab	x	y	xy

(b) Make the observation table closed by moving the row a to S_M.

	a	b	ab
ϵ	x	y	xy
a	x	y	xx
aa	x	x	xx
b	x	y	xy
ab	x	y	xy
aaa	x	x	xx
aab	x	y	xx

(c) Make the table closed by moving the row aa to S_M.

Table 5.7: We get the observation table in Table 5.7a after adding suffix ab to E_M. The underlined rows are the rows, which make the table not closed. The observation table in Table 5.7c is closed.

factors.

- the size of inputs $|I|$,
- the number of states in minimal conjecture n,
- and the length of the longest distinguishing sequence from a counterexample p.

Using Suffix1by1 to process the counterexamples, the learning algorithm ${L_M}^*$ initializes the S_M with ϵ and E_M with I. The size of S_M augments only when one element from $S_M \cdot I$ is moved to S_M to make the table closed. This can happen at most $n-1$ times and hence, we always have $|S_M| \leq n$. Thus, the number of observation table rows $S_M \cup S_M \cdot I$ cannot exceed $n + n|I|$.

The columns E_M are initialized to I and to process a counterexample at most $p-1$ suffixes are added to E_M, where p is the length of the longest distinguishing sequence added to E_M. This can happen at most $n-1$ times. Thus, the size of E_M cannot exceed $|I|+(p-1)(n-1)$. Putting all this together the maximum cardinality of $\{S_M \cup S_M \cdot I\} \times E_M$ is at most

$$(n + n|I|) \times (|I| + (p-1)(n-1)).$$

The worst case time complexity for the learning algorithm ${L_M}^*$ adapting the improved counterexample processing method Suffix1by1 in terms of output queries is $O(|I|^2 n + |I|pn^2)$.

5.3.5 Complexity Comparison

The theoretical worst case time complexity for ${L_M}^*$ in terms of output queries is $O(|I|^2 mn + |I|mn^2)$, which is reduced to $O(|I|^2 n + |I|mn^2)$ by adapting the counterexample processing method from Maler and Pnueli [Maler 1995].

If Rivest and Schapire counterexample processing method is adapted for ${L_M}^*$ then the complexity is reduced to $O(|I|^2n+|I|n^2+nlog(m))$ output queries, but this method requires a compromise on the suffix closure property of the observation table.

By adapting the counterexample processing method from Shahbaz and Groz, the worst case time complexity remains $O(|I|^2n + |I|mn^2)$, but the gain is for the factor m, which is reduced from length of the longest counterexample to length of the longest suffix of a counterexample added to the observation table.

The worst case complexity for the improved method is $O(|I|^2n+|I|pn^2)$, where p is the length of the longest suffix added by the method. This suffix is a distinguishing sequence from a counterexample and non optimal counterexamples include useless prefixes to distinguishing sequences. Thus, the suffix added by the improved method is always much smaller than the suffix added by the Shahbaz and Groz method.

5.4 Experiments to Analyze Practical Complexity of Suffix1by1

We have performed an experimental evaluation of the algorithm ${L_M}^*$ with the counterexample processing techniques described in previous sections. Our experiments aim at finding out how all of these techniques perform in practice. The algorithms are implemented by closely following their high level description. In order to analyze these techniques, we performed two sets of experiments. We first executed the various algorithms on the *Edinburgh Concurrency Workbench (CWB)* examples [1] [Moller]. Then, we used random machines, which allowed us to study the influence of the various parameters (number of inputs, outputs, states) on the algorithms.

Randomly generated state machines have been used to investigate the application of state machine inference algorithms [Berg 2005b, Bollig 2008, Shahbaz 2009]. This testbed provides independence to generate state machines with required parameters. We record the number of output queries and counterexamples to analyze the practical complexity of the learning algorithms. Since the CPU time depends on the execution of output queries in the black box and we experiment on the simulated machines, it is useless to record the time. Execution of output queries varies a lot from one black box to another. For instance in the case of a web service of a slow interface (e.g. smart card) or a system with physical delays such as mechanical motions, the execution time of a query can be much greater than rest of the algorithm.

5.4.1 CWB Examples

Berg *et al.* [Berg 2005b], and Shahbaz and Groz [Shahbaz 2009] use the *CWB* examples to examine the practical applicability and analysis of their techniques. The *CWB* examples cater for the manipulation and analysis of concurrent systems. These examples allow evaluation of testing and model checking techniques. We

[1] Examples available at `http://homepages.inf.ed.ac.uk/perdita/cwb/Examples/ccs/`

have experimented with examples like buffers, vending machines, mutual exclusion protocols and schedulers shipped with the *CWB* examples. To all of the examples, we have added a sink state having transitions with invalid inputs from all states.

The Mealy machines of *CWB* examples are learned with the algorithm ${L_M}^*$ considering each of the counterexample techniques discussed in previous sections of this chapter.

For *CWB* examples we have implemented an oracle that finds the counterexamples by calculating the symmetric difference between a conjecture and a target example. Let C be a model conjectured from an unknown *DFA* $\mathcal{A}$. The language $\mathcal{L}(\bar{\mathcal{A}} \cap C)$ accepts the strings which are accepted by C but rejected by $\mathcal{A}$. The language $\mathcal{L}(\mathcal{A} \cap \bar{C})$ is defined analogously. We construct a *DFA* Z such that $\mathcal{L}(Z) = \mathcal{L}(\bar{\mathcal{A}} \cap C) \cup \mathcal{L}(\mathcal{A} \cap \bar{C})$. Thus, the language of Z contains the strings that are either accepted by C or $\mathcal{A}$, but not by both. Any string that belongs to $\mathcal{L}(Z)$ is a counterexample. If $\mathcal{L}(Z)$ is $\emptyset$, then this implementation returns, "yes" the conjecture is correct. Here we are inferring Mealy models and every Mealy machine can be expressed as a DFA.

Table 5.8: Number of output queries required

	$\|Q\|$	$\|I\|$	*RS*	*1by1*	*Ang*	*Sh*	*MP*
ABP-Lossy	11	3	306	340	754	340	578
Peterson2	11	3	306	340	880	374	850
Small	11	5	392	392	462	392	672
VM	11	5	392	392	891	448	672
Sched2	13	6	553	553	824	790	1027
ABP-Safe	19	3	638	754	2336	754	870
TMR1	19	5	1152	1632	1392	1728	1920
Vmnew	29	4	2106	1638	2941	2106	3510
CSPROT	44	5	3536	3094	4864	3757	6188
Jobshop	39	7	5754	5206	3960	5206	8768

Table 5.8 shows the number of output queries required by all of the considered counterexample processing methods. The columns of the table are labeled with the number of states $|Q|$, the size of input set $|I|$, *RS* the Rivest and Schapire counterexample processing, *1by1* the Suffix1by1 counterexample processing, *Ang* the Angluin counterexample processing, *Sh* the Shahbaz and Groz counterexample processing, and *MP* the Maler and Pnueli counterexample processing. The recorded results show that generally the Rivest and Schapire counterexample processing method is better than the others, however, in some cases it requires more queries than the other methods. For the examples *Small* and *VM*, the Rivest and Schapire and Suffix1by1 both require only two counterexamples, and both of the methods add only one sequence to distinguish new states. Since both of the methods add the same number of sequences to the columns E_M and every sequence distinguishes the similar number of states, they result in requiring the equal number of output queries. A

distinguishing sequence is a sequence which can distinguish a state from other states of a model. A distinguishing sequence can identify one or more than one state. For the examples *Vmnew*, *CSPROT* and *Jobshop*, the distinguishing sequences added by the Rivest and Schapire method distinguished relatively less states as compared to other methods, which results in requiring more output queries 5.8.

Table 5.9: Number of counterexamples required

	$\|Q\|$	$\|I\|$	*RS*	*1by1*	*Ang*	*Sh*	*MP*
ABP-Lossy	11	3	5	5	4	5	3
Peterson2	11	3	5	5	5	5	4
Small	11	5	2	2	2	2	2
VM	11	5	2	2	2	2	2
Sched2	13	6	1	1	1	1	1
ABP-Safe	19	3	8	4	4	4	3
TMR1	19	5	7	2	2	2	2
Vmnew	29	4	14	8	6	8	5
CSPROT	44	5	11	7	5	6	5
Jobshop	39	7	14	6	4	6	6

The Table 5.9 shows the number of counterexamples required by the considered counterexamples processing methods. We can observe that the Maler and Pnueli method requires least number of counterexamples. However, it asked most number of output queries. This method adds all the suffixes of counterexamples to the observation table, some of them are distinguishing sequences while other are not. The non distinguishing sequences may eventually become distinguishing sequences after adding new rows to the observation table. Thus, this method suits most where finding the counterexamples is hard but output queries can be tolerated to some extent.

5.4.2 Random Machines

We generate the input deterministic machines by variating the inputs $|I|$, outputs $|O|$ and state sizes $|Q|$. We learn Mealy machines of these random machines with the counterexample processing methods provided in this chapter. We have simulated an oracle so that the algorithm can ask the equivalence queries for the correctness of conjectured models. The oracle selects an input from uniform distribution over I and provides this input in parallel to conjecture and target system and observes the outputs. This process is continued until it finds an input whose output for conjecture and target system are different. The sequence of inputs from the initial input to the one which causes the difference in outputs is returned as a counterexample. Since the counterexamples are searched with the random search, the counterexamples found are often not optimal. In order to increase our confidence in results, we repeat the learning for every required machine for 30 times and average on the calculated data.

We generate Mealy machines by fixing the output and state sizes but varying

the number of inputs. Here machines are generated with inputs $|I|\in\{2,3,\ldots 8\}$, outputs $|O|$=7 and states $|Q|$=40. In Figure 5.5 the number of output queries asked by all of the algorithms are presented. The vertical axis shows the number of output queries and horizontal axis shows the number of inputs. The curves for the Rivest and Schapire, and the Suffix1by1 methods are very close as the maximum difference for the number of output queries asked is around 60 on the average. Both of these methods perform better than the others. However, on the average these methods require around 1.7 counterexamples, whereas the others on the average require around 1.2 counterexamples. For the smallest machine with input size $|I| = 2$, the Rivest and Schapire, Suffix1by1, Angluin, Shahbaz and Groz, and Maler and Pnueli methods on the average require 274, 331, 586, 1069 and 1102 output queries, respectively, and for the largest machine with $|I| = 8$ require 2895, 2910, 37778, 14166, and 14873 output queries, respectively.

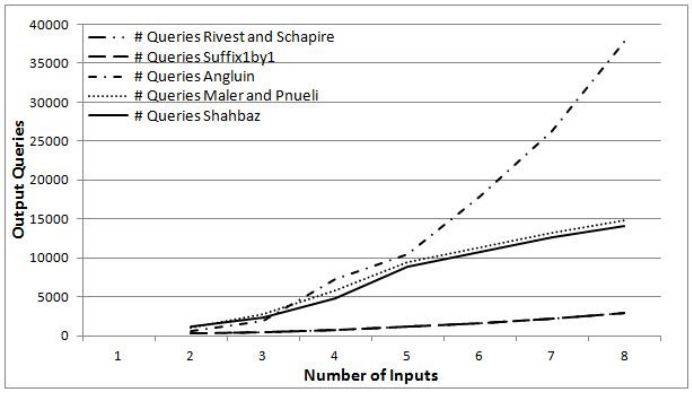

Figure 5.5: $|I|\in\{2,3\ldots 8\}$,$|O|$=7 and $|Q| = 40$

The second set of random machines is generated with inputs $|I|$=2, the outputs $|O|$=2 and states $|Q|\in\{3,4\ldots 40\}$. The algorithm $L_M{}^*$ is executed to learn these machines repeatedly with the counterexample processing methods. The Figure 5.6 presents the number of output queries asked by all of the methods. The vertical axis shows the number of output queries and horizontal axis show the number of states.

We can observe that by increasing the number of states, the gain with Rivest and Schapire method becomes more significant. For the smallest machine with State size $|Q| = 3$, the Rivest and Schapire, Suffix1by1, Angluin, Shahbaz and Groz, and Maler and Pnueli methods on the average require 21, 21, 63, 42 and 56 output queries, respectively. For the largest machine with $|Q| = 40$ require 610, 775, 1287, 1215, and 972 output queries, respectively. For the smallest machine all of the methods required only one counterexample, whereas for the largest machine the Rivest and Schapire, Suffix1by1, Angluin, Shahbaz and Groz, and Maler and Pnueli methods require 9, 5, 2.22, 2.75, and 1.72 counterexamples respectively.

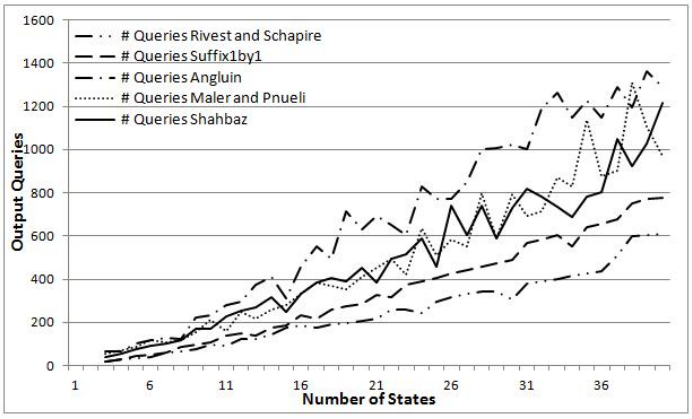

Figure 5.6: $|I|$=2,$|O|$=2 and $|Q|\in\{3, 4 \ldots 40\}$

5.5 Conclusion

This chapter presents the techniques for searching and processing the counterexamples. The counterexamples help to improve models conjectured with ${L_M}^*$. However, for software black box system inference searching for counterexamples is a hard task. Random sampling is a simple method to search for the counterexamples. However, this method finds non optimal counterexamples that have useless sequences prefix to them. We can use the algorithms for conformance testing of a conjectured machine with a black box machine but the complexity of such algorithms is exponential [Vasilevskii 1973, Chow 1978].

Howar *et al.* propose to search for the counterexamples by introducing some weights along the transition edges of the conjectured models. The selection of an edge for counterexample search is inversely proportional to the weight of the edge. This algorithm worked well for randomly generated machines for ZULU competition, however, this method may not be efficient for all models. Balle found that the transitions ending in shorter leaves of a discrimination tree close to root are less informed. Thus, the traversal of transitions towards the shorter leaves increases the probability to find counterexamples. Their experiments could not witness gain with this method.

For software black box model inference, almost all of the counterexample searching methods provide non optimal counterexamples. To circumvent this deficiency a number of counterexample processing methods are proposed. The method by Angluin adds all the prefixes of a counterexample to S rows of the observation table. Before conjecturing a model it ensures that the observation table is closed and compatible. Rivest and Schapire found that compatibility check can be avoided if we keep S elements distinct. Their method finds a smallest distinguishing sequence from a counterexample by querying $log(m)$ output queries. Balcázar *et al.* identified that after processing the smallest distinguishing sequence, a larger may still be there in the counterexample, which can still improve the conjecture. Rivest and

Schapire method adds only a distinguishing sequence to the columns of the observation table by requiring a compromise on the suffix closure property. The conjecture constructed from such tables can go inconsistent with the observations in tables.

The method from Maler and Pnueli adds all the suffixes to columns of the observation table where fewer might be required. However, it keeps the observation table suffix and prefix closed. In an effort to reduce the elements added to the columns of an observation table, Shahbaz and Groz proposed a method to process the counterexamples. This method drops the longest prefix from *CE* that is equivalent to any element of access strings in the observation table and then adds all the suffixes of remaining *CE* to the columns of the observation table. For inferring black box systems, it is not easy to find "smart" counterexamples. The counterexamples are often non optimal and a counterexample may contain some useless sequence as prefix to it. The Shahbaz and Groz method removes only a small prefix of such a useless counterexample prefix. The Suffix1by1 algorithm adds the suffixes from *CE* one by one to E, as soon as it finds a distinguishing suffix from *CE* which can improve the conjecture it stops adding suffixes to E. Thus, such useless prefixes are avoided.

The counterexample processing method by [Rivest 1993] is the best competitor of the improved counterexample processing method. It adds only a distinguishing sequence from a counterexample *CE* and finds it in $log(m)$ output queries. The compromise on suffix closure property by this method can result in inconsistent conjecture. *Merten et al.* provide a solution in LearnLib (*RivestAllSuffixesSplitterCreator*) [Merten 2011], which adds the distinguishing sequence along its suffixes to E_M. But they ask $log(m)$ output queries just to search for a distinguishing sequence, which is not necessarily the smallest one. A suffix of such a distinguishing sequence can be a distinguishing sequence along other access strings which were not tested during the binary search to identify the smallest distinguishing sequence from *CE*. The Suffix1by1 counterexample processing method always finds the smallest distinguishing sequence as it adds the suffixes of *CE* by increasing length to the columns of the observation table by beginning from the smallest suffix. This method finds the smallest distinguishing suffix without asking $log(m)$ output queries.

The counterexample processing methods are analyzed with the systematic and the random generation of counterexamples for *CWB* examples and randomly generated machines, respectively. The systematic counterexample generation results in relatively smaller counterexamples and consists of calculating the symmetric differences, which is not possible for black box model inference. However, we have considered both counterexample searching techniques to observe the behavior of the improved method. We have compared the number of output queries and counterexamples required to learn the considered models. We have observed that the results for Suffix1by1 counterexample processing method are encouraging.

Chapter 6

Improved Model Inference

Contents

6.1 Motivation for the L_1 Algorithm 83

6.2 Improved Mealy Inference Algorithm 84

6.2.1 Observation Table 85

6.2.2 The L_1 Algorithm 87

6.2.3 Example for learning with L_1 88

6.2.4 Complexity of the L_1 Algorithm 90

6.3 Inferring the HVAC controller 90

6.3.1 Description of the HVAC controller 91

6.3.2 Inference of the HVAC controller with the ${L_M}^*$ Algorithm . . 91

6.3.3 Inference of the HVAC controller with the L_1 Algorithm . . . 92

6.4 Experiments to Analyze Practical Complexity of L_1 97

6.5 Conclusion 99

This chapter presents the improved Mealy inference algorithm named as the algorithm L_1. In the first section of this chapter, we provide discussion on the motivation that leads us to the optimized algorithm. The second section presents: the L_1 algorithm, the improved observation table (data structure) that it uses for recording the observations, the method to conjecture a model from the observation table, an example to illustrate the algorithm, and complexity discussion. The third section provides model learning of the HVAC controller with the ${L_M}^*$ algorithm and L_1 algorithm. To show the gain with the improved algorithm, we conduct experiments in the fourth section. The final section provides a conclusion for the chapter.

6.1 Motivation for the L_1 Algorithm

In Chapter 5, We have exploited the fact: for black box model inference the counterexamples reported by the oracle are often non optimal as they have useless sequences as prefixes. If we process the counterexamples with the improved method (the suffix1by1 method presented in Section 5.3), it results in getting rid of these useless sequences. In this chapter, we explore the other points where the black box model inference with ${L_M}^*$ can be optimized. This gives rise to the L_1 algorithm.

Mealy models associate an output with each transition i.e. Mealy machine transitions are annotated with i/o, where $i \in I$ and $o \in O$. The output o of a transition

in a model is determined both by the current state and the input of the transition. For the Mealy inference variant $L_M{}^*$ of the L^* algorithm, the columns of the observation table are initialized to input set I and this enables the algorithms to calculate the corresponding i/o for every transition in a conjectured model.

The rows of the observation table are labeled with sequences, which are access strings for the states and the columns of the observation table are labeled with the sequences, which distinguish the distinct states of a target system's model. If input set I has a large number of elements and we initialize the columns of the observation table with I, then there is a possibility that all of the input sequences are not distinguishing strings and we have initialized the columns with too many sequences, which results in an increased number of output queries. A heuristic, which can restrict the columns of the observation table only to the distinguishing sequences, can reduce the number of output queries required by the learning algorithm.

The $L_M{}^*$ algorithm assumes that the target system is input enabled. To attain this, it uses the collection of all possible inputs I for each state (either there exists a behavior for that input or not). When the $L_M{}^*$ algorithm identifies an access string for a distinct state, then its one letter extensions are added to the observation table to identify its successor states. But all inputs may not be valid for every state. For invalid inputs a transition from current state with output Ω to itself is introduced. Thus, the successor state reached with invalid input is again the current state. We need not to record the behavior of a state twice. A row in the observation table corresponds to behavior of a state and it can be marked as *unnecessary row* if it has a row in the observation table that will always be equal to it (the rows corresponding to the states reached with transitions having output Ω are such cases). A technique, which avoids asking output queries for unnecessary rows of the observation table reduces the number of output queries required to learn a model of the target system.

6.2 Improved Mealy Inference Algorithm

The Mealy inference algorithm L_1 is an improved version of the learning algorithm $L_M{}^*$. The L_1 algorithm initially keeps the columns of the observation table empty with the intent to add only those elements from the set I, which are really required. But, Mealy adaptations [Niese 2003, Li 2006, Shahbaz 2009] of the L^* algorithm initialize the columns with the input set I to calculate the annotations (labeling) for the transitions of conjectured models. To enable the L_1 algorithm to calculate the output labels for the transitions of the conjectured Mealy models, we record the output for the last input symbol of the access strings (input sequences) that label rows of the observation table. This output also helps to identify unnecessary rows of the observation table. The columns of the observation table are populated only to process the counterexamples, i.e. columns of the observation table contain only the distinguishing sequences, and sequences to keep the set of column labels suffix closed. To process a counterexample L_1 adds sequences from the counterexample to the columns of the observation table like Rivest and Schapire [Rivest 1993], Maler

and Pnueli [Maler 1995], and Shahbaz and Groz [Shahbaz 2009]. To process a counterexample, the L_1 algorithm adds the suffixes of the counterexample by increasing length to the columns of the observation table [Irfan 2010c]. The observation table is populated with the outputs from O^+ that are calculated from the target system by sending input strings from I^+. The observation table used by the L_1 algorithm is described as follows.

6.2.1 Observation Table

The observation table is defined as a quadruple (S, E, L, T), where

- $S \subseteq I^*$ is a prefix closed non empty finite set of access strings, which labels the rows of the observation table,
- $E \subseteq I^+$ is a suffix closed finite set, which labels the columns of the observation table,
- for $S' = S \cup S \cdot I$, the finite function T maps $S' \times E$ to the set of non empty output sequences $O^* \backslash \{\epsilon\}$ or O^+,
- and the finite function L maps $S' \backslash \{\epsilon\}$ to outputs O which are used to label the transitions.

The observation table rows S' are non empty and initially, $S = \{\epsilon\}$ and $S \cdot I = I$. The output for the last input element for all the members of $S' \backslash \{\epsilon\}$ are recorded with the access strings S' by L. The columns E are initially $\emptyset$ and E augments only after processing the counterexamples. To process a counterexample, the suffixes of the counterexample are added to E by increasing length. The observation table is completed by extending T to $S' \cdot E$ by asking the output queries. The access strings are concatenated with the distinguishing sequences to construct the output queries as $s \cdot e$[1], for all $s \in S'$ and $e \in E$. In the observation table, $\forall s \in S'$, $\forall e \in E$, $T(s, e) = \mathit{suffix}^{|e|}(\lambda(q_0, s \cdot e))$, and $L(s) = \mathit{output}^{|s|}(\lambda(q_0, s \cdot e))$. By means of function L, all the access strings $s \in S' \backslash \{\epsilon\}$ labeling the rows of the observation table contain the output o for the last input symbol of the access string s, and S' is prefix closed. This implies that $\lambda(q_0, s)$ can be calculated from the row labels. Thus, recording the suffix of the output query answer $\mathit{suffix}^{|e|}(\lambda(q_0, s \cdot e))$ to the cell labeled by row s and column e in the observation table is sufficient. Initial observation table (S, E, L, T) for Mealy inference of the machine in Figure 4.1 is presented in Table 6.1, where the input set I is $\{a, b\}$.

The equivalence of rows in the observation table is defined with the help of function T. Two rows $s_1, s_2 \in S'$ are said to be equivalent, *iff* $\forall e \in E$, $T(s_1, e) =$

[1] The sequences $s \cdot e$ for $s \in S' \wedge e \in E$ and $s \cdot i$ for $s \in S' \wedge i \in I$ are constructed by considering the access strings only. The output for last input symbol of access strings recorded along rows $S' \backslash \{\epsilon\}$ of the observation table, is used for mapping pair of a state and an input symbol to the corresponding output symbol (for annotating the transitions during conjecture construction) and identifying the valid access strings.

Table 6.1: Initial table for mealy inference with L_1 of machine in Figure 4.1

		E
		$\emptyset$
S	ϵ	
$S \cdot I \backslash S$	a/x	
	b/y	

$T(s_2, e)$, and it is denoted as $s_1 \cong s_2$. For every row $s \in S'$, the equivalence class of a row s is denoted by $[s]$. To construct the conjecture, the L_1 algorithm requires the observation table to satisfy the closure and compatibility properties. The observation table is closed, if $\forall s_1 \in S \cdot I$, there exists $s_2 \in S$ such that $s_1 \cong s_2$. The observation table is compatible, if whenever two rows $s_1 \cong s_2$ for $s_1, s_2 \in S$, then $s_1 \cdot i \cong s_2 \cdot i$ for $\forall i \in I$. Since size of the rows S of observation table (S, E, L, T) increases only to make the table closed, the L_1 algorithm always maintains the condition that all rows S are distinct, i.e. for all $s_1, s_2 \in S$, $s_1 \ncong s_2$. Thus, the compatibility condition is always trivially satisfied. On finding the observation table closed and compatible, the L_1 algorithm eventually conjectures a Mealy machine. The access strings S are the states for the conjecture and columns labeled with strings from suffix closed set E are the sequences that distinguish these states. The conjecture $Conj_1$ is defined as:

Definition 6 *Let (S, E, L, T) be a closed and compatible observation table, then the Mealy machine conjecture $Conj_1 = (Q_C, I, O, \delta_C, \lambda_C, q_{0C})$ is defined, where*

- $Q_C = \{[s] | s \in S\}$, *(since $\forall s_1, s_2 \in S$, always $s_1 \ncong s_2$ thus $|S| = |Q_C|$)*
- $q_{0C} = [\epsilon]$, *$\epsilon \in S$ is the initial state of the conjecture*
- $\delta_C([s], i) = [s \cdot i], \forall s \in S, i \in I$
- $\lambda_C([s], i) = L(s \cdot i), \forall s \in S, \forall i \in I \ \exists! s \cdot i \in S'$

To verify that $Conj_1$ is a well defined conjecture: since S is a non empty prefix closed set and always contains ϵ, q_{0C} and Q_C are defined. For all $s \in S$ and $i \in I$, the string $s \cdot i$ is added to $S \cdot I$ exactly once. Thus, for every s and every i, there exists uniquely one $s \cdot i$ in S. Since the observation table is closed, the row $s \cdot i \cong s_1$ for some row $s_1 \in S$. Hence, δ_C is well defined. Since every $s \cdot i$ is also associated with some output o by function L, λ_C is well defined.

Theorem 1 *If (S, E, L, T) is a closed and compatible observation table, then the Mealy conjecture $Conj_1$ from (S, E, L, T) is consistent with the finite function T. Any other conjecture consistent with T but inequivalent to $Conj_1$ must have more states.*

Theorem 1 claims the correctness of the conjecture $Conj_1$. The formal proof by Niese [Niese 2003] for the correctness of a model conjectured by Mealy adaptation

of L^*, uses the prefix closure and suffix closure properties of the observation table. The L_1 algorithm always keeps the observation table prefix and suffix closed and the conjecture is a minimal machine by construction.

6.2.2 The L_1 Algorithm

The L_1 learning algorithm maintains an observation table (S, E, L, T) to record the answers O^+ of the output queries I^+. The set of rows S is initialized to $\{\epsilon\}$ and the columns E at the beginning are $\emptyset$.

Algorithm 8 The Algorithm L_1

```
Input: Black box and input set I
Output: Mealy Machine Conjecture
begin
    initialize the rows S = {ε}, and columns E = ∅;
    execute output queries for S · I strings;
    since columns are empty, table is closed;
    construct a conjecture C;
    repeat
        search for counterexamples;
        if the oracle replies with a counterexample CE then
            while CE is a counterexample do
                for j = 1 to |CE| do
                    if suffix^j(CE) ∉ E then
                        add suffix^j(CE) to E;
                        complete (S, E, T, L) by asking output queries s · e such that
                        s ∈ S' ∧ e ∈ E;
                        if (S, E, T, L) is not closed then
                            break for loop;
                        end
                    end
                end
                while (S, E, L, T) is not closed do
                    find s1 ∈ S · I\S such that s1 ≇ s2, for all s2 ∈ S;
                    move s1 to S;
                    add s1 · i to S · I, for all i ∈ I;
                    complete table by asking output queries for new added rows and
                    find unnecessary rows;
                end
                construct a conjecture C;
            end
        end
    until oracle replies yes to the conjecture C;
end
```

For the set of inputs I labeling rows $S \cdot I$, the outputs are calculated. All of the columns are empty, which implies that all the rows of the observation table are equivalent, i.e. $\forall s_1 \in S \cdot I$, there exists $s_2 \in S$, such that $s_1 \cong s_2$, thus, the observation table is closed. Since S has only one element ϵ, the compatibility condition is trivially satisfied. On finding the observation table closed and compatible, the L_1 algorithm conjectures a model (the initial conjecture is a single state "daisy" machine). For every access string $s \in S$ and for every input $i \in I$, there exists exactly one $s \cdot i \in S'$ and the output for the last input i of the S' elements enables the L_1 algorithm to calculate the corresponding output label o for all the transitions of the conjectured model. Now the L_1 algorithm asks the equivalence query to the oracle. If the oracle replies with a counterexample CE, then L_1 adds one by one the suffixes by increasing length from CE to E. The observation table is completed by asking the output queries. After adding a suffix and completing the observation table, the closure property is checked. The algorithm continues adding suffixes until a suffix of CE is found, which makes the table not closed and forces refinement. On finding such a suffix, this method stops adding suffixes to E. Now the algorithm makes (S, E, L, T) closed by finding a row $s_1 \in S \cdot I$, such that $s_1 \ncong s_2$, for all $s_2 \in S$ and moving s_1 to S. The one letter extensions of s_1 are added to $S \cdot I$. Since a row in the observation table corresponds to behavior of a state and Ω is output for a transition having same current and target state, on asking an output query for any of new added rows if the output found to be stored with access string of that row is Ω then it is marked as unnecessary row. On completing the table, again the closure property is checked. Since the observation table (S, E, L, T) is always compatible, on finding it closed, L_1 builds the Mealy machine conjecture in accordance with the Definition 6.

Now the main while loop checks, if processed CE is again a counterexample for the learned conjecture. If "yes" (CE is still a counterexample), this means a longer suffix from the same counterexample CE can improve the conjecture. The algorithm L_1 relearns with CE and this process is continued until the counterexample CE can no longer help to improve the conjectured Mealy machine. After processing a counterexample and making the table closed, the algorithm again asks equivalence queries to the oracle. This process is continued until oracle is unable to find a counterexample and replies "yes" to the conjectured model. The algorithm is explained with the help of the Mealy inference of the machine $\mathcal{M}$ in Figure 4.1.

6.2.3 Example for learning with L_1

To illustrate the L_1 algorithm, we learn the Mealy machine $\mathcal{M}$ given in Figure 4.1. The L_1 algorithm begins by initializing (S, E, L, T), the rows S are initialized to $\{\epsilon\}$ and columns E to $\emptyset$. The output queries a and b are asked to find the outputs for input symbols $\{a, b\}$ labeling the rows $S \cdot I$. The initial observation table is shown in Table 6.1. This observation table has empty columns for all of the rows, thus, for every $s_2 \in S \cdot I$, there exists $s_1 \in S$ such that $s_2 \cong s_1$. Hence, the table is closed. Since for the L_1 algorithm the set of rows S always

has distinct members, trivially the observation table is compatible. On finding the observation table closed and compatible, the L_1 algorithm conjectures the model $Conj_1 = (Q_{C1}, I, O, \delta_{C1}, \lambda_{C1}, q_{0C1})$ shown in Figure 6.1.

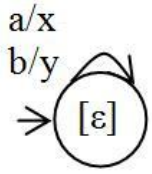

Figure 6.1: The Mealy machine conjecture $Conj_1$ from Table 6.1

To verify correctness of the one state machine conjecture, the L_1 algorithm asks an equivalence query to the oracle. The conjectured model $Conj_1$ is not correct, and there can be more than one counterexample and oracle replies with one from them.

- $\lambda(q_0, ababaab) = xyxyxxx$, but
- $\lambda_{C1}(q_{0C1}, ababaab) = xyxyxxy$.

Let us assume the oracle replies with the counterexample *ababaab/xyxyxxx*. The L_1 algorithm adds the smallest suffix b of the counterexample *ababaab* to the columns E and completes the table, which remains closed as presented in Table 6.2a. Then suffix ab is added, which makes the observation table not closed as presented in Table 6.2b.

The row a as shown in Table 6.2b makes the observation table not closed. It is moved to S and its one letter extensions aa and ab are added to $S \cdot I$. To calculate the output for last input of access strings aa and ab, the algorithm does not require to execute the output queries separately. The output for the last input of access strings can be calculated from any of the output queries executed for that row. For instance if the algorithm asks the output query $ab \cdot ab$, the answer from the target machine is $xyxy$. The output for the last input string in ab is calculated as y and remaining xy is recorded in column ab of the observation table. After completing the table, it can be observed from Table 6.2c that the row aa makes the observation table not closed. Again the table is made closed by moving the row aa to S. The observation table in Table 6.2d is closed. Since the algorithm L_1 maintains the condition $\forall s_1, s_2 \in S$ always $s_1 \not\cong s_2$, the observation table is always compatible. Hence, L_1 conjectures the Mealy machine shown in Figure 7.4.3.

The conjectured model is correct, the oracle replies "yes" and L_1 terminates. Initially 2 output queries were executed to calculate the outputs for $\{a, b\}$ labeling $S \cdot I$, thus, L_1 asks a total of 16 output queries to learn this example. Since all the inputs are valid for all of the states, none of the observation table row is marked as unnecessary row. Here, we have considered an example with a small set of inputs. If we learn a system with a large input set, the L_1 algorithm can restrict a bigger number of columns.

	b
ϵ	y
a/x	y
b/y	y

(a) The observation table after adding the suffix b.

	b	ab
ϵ	y	xy
$\underline{a}/x$	y	xx
b/y	y	xy

(b) The observation table after adding the suffix ab.

	b	ab
ϵ	y	xy
a/x	y	xx
b/y	y	xy
$\underline{aa}/x$	x	xx
ab/y	y	xy

(c) Make the observation table closed by moving the row a to S.

	b	ab
ϵ	y	xy
a/x	y	xx
aa/x	x	xx
b/y	y	xy
ab/y	y	xy
aaa/x	x	xx
aab/x	y	xx

(d) Make the table closed by moving the row aa to S.

Table 6.2: Model inference of Mealy machine in Figure 7.4.3 with L_1

6.2.4 Complexity of the L_1 Algorithm

The learning algorithm L_1 can conjecture a minimal Mealy machine in polynomial time on, the inputs size $|I|$, the number of states in the minimal conjecture n, and m the length of the longest distinguishing sequence added to the observation table (S, E, L, T) from a counterexample. Initially S contains ϵ, i.e. one element. Each time the observation table is discovered to be not closed, one element is moved from $S \cdot I \backslash S$ to S. This can happen for at most $n-1$ times, hence, we always have $|S| \leq n$. The algorithm begins with columns $E = \emptyset$ and $|E| = 0$, each time to process a counterexample at most m suffixes can be added to E and at most there can be $n-1$ counterexamples. Thus, the size of E cannot exceed $m(n-1)$. Putting all this together the maximum cardinality of $S' \times E$ is $(n + n|I|) \times m(n-1)$. The worst case complexity of L_1 in terms of output queries is $O(|I|mn^2)$. Since the algorithm avoids asking output queries for unnecessary rows (avoids asking output queries for observation table rows that have output Ω recorded with access strings), for every state the size of inputs $|I|$ is reduced to the size of the valid inputs.

6.3 Inferring the HVAC controller

In this section, we infer a simplified version of an HVAC (Heating-Ventilation-Air-Conditioning) controller with the $L_M{}^*$ and the L_1 algorithms and show differences.

6.3.1 Description of the HVAC controller

The HVAC controller regulates heating, ventilation and air conditioning components in a building. Usually, a sensing device is used to compare the actual temperature with a target temperature. Then, the control system determines an action (e.g. start heater). We have taken the specifications of the HVAC controller from the UPNP standardization[2].

The HVAC Controller operations on different temperature values that are:
– for low temperature, i.e. $[-20, 11]$*, turn on the heater,*
– for high temperature, i.e. $[16, 50]$*, turn on the fan,*
– for average temperature, i.e. $[11, 16]$*, stop both heater and fan.*

Figure 6.2 presents the HVAC controller. The controller has various modes depending on the specifications. We infer its very generic behavior, that is, controlling the heating and cooling components on the change of temperature. The controller accepts inputs from its environment to control the connected components. It uses ON and OFF for starting and shutting down the components, respectively. The temperature T ranges from -20℃ to +50℃. The control modes are: it turns on the heater H when temperature is between -20℃ and 11℃, it turns on the fan F when the temperature is between 16℃ and 50℃. The controller shuts down a component, when the temperature goes out of the provided range for a component or the controller receives the instruction OFF, and shut down is denoted by S. The outputs for invalid inputs are recorded as Ω.

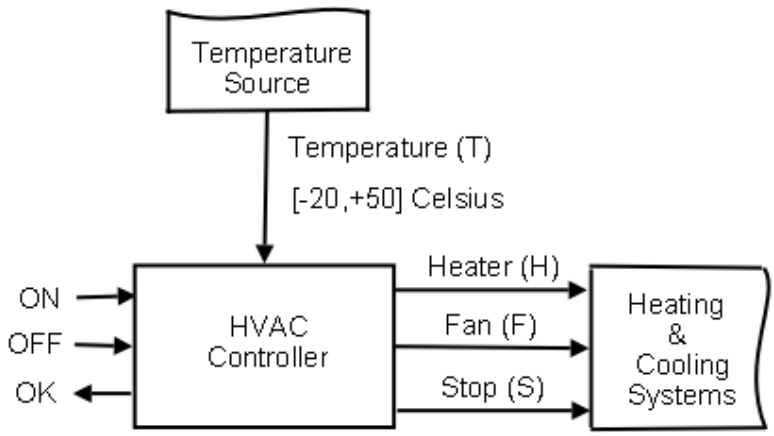

Figure 6.2: HVAC controller

6.3.2 Inference of the HVAC controller with the $L_M{}^*$ Algorithm

The behavior of the HVAC controller can be modeled as a Mealy machine. We use the $L_M{}^*$ algorithm to infer controller's interactions with the heating and cooling

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

components. The elements of the input set I are ON, OFF, and variating temperature. We are interested to infer the behavior of controller when temperature is -5, 5, 15, 25 and 35 denoted by T-5, $T5$, $T15$, $T25$ and $T35$, respectively. Thus, the input set is $I = \{ON, OFF, T\text{-}5, T5, T15, T25, T35\}$. The observation table (S_M, E_M, T_M) is completed by asking the output queries and the algorithm ${L_M}^*$ iterates till the table is closed. The observation table (S_M, E_M, T_M) in Table 6.3 is closed. Since all of the S_M rows are distinct, the table is compatible. To save the space, we have excluded the rows $\{OFF, T\text{-}5, T5, T15, T25, T35, ON \cdot ON, ON \cdot T5 \cdot ON, ON \cdot T25 \cdot ON\}$ from the table 6.3.

		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
	$ON \cdot T\text{-}5$	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
	$ON \cdot T5 \cdot T\text{-}5$	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
	$ON \cdot T25 \cdot T\text{-}5$	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 6.3: The HVAC controller's inference with ${L_M}^*$

A total of 203 output queries (29 rows $\times$ 7 columns) are asked to conjecture Mealy machine of the HVAC controller. Figure 6.3 shows the model conjectured from Table 6.3.

6.3.3 Inference of the HVAC controller with the L_1 Algorithm

We illustrate the L_1 algorithm by learning the Mealy model $\mathcal{M}_{hvac}$ of the HVAC controller, and show how unnecessary rows can be identified using the output for the last input of the access strings. The L_1 algorithm begins by initializing the

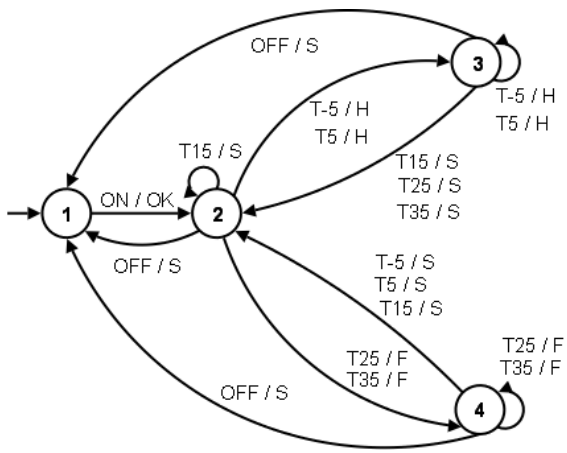

Figure 6.3: The HVAC controller's model conjectured from Table 6.3

observation table (S, E, L, T), the rows S are initialized to $\{\epsilon\}$, and columns E are $\emptyset$. The output queries $ON, OFF, T\text{-}5, T5, T15, T25$ and $T35$ are asked to find the outputs for input symbols $\{ON, OFF, T\text{-}5, T5, T15, T25, T35\}$ labeling the rows $S \cdot I$.

		E
		$\emptyset$
S	ϵ	
$S \cdot I \backslash S$	ON/OK ~~OFF/Ω~~ ~~$T\text{-}5/\Omega$~~ ~~$T5/\Omega$~~ ~~$T15/\Omega$~~ ~~$T25/\Omega$~~ ~~$T35/\Omega$~~	

Table 6.4: Initial Observation Table for Mealy inference with L_1 of the HVAC controller

The rows with access strings whose output for the last input symbol is Ω are marked as unnecessary rows and are excluded from the subsequent tests. The access strings of unnecessary rows are struck out in initial observation table as shown in Table 6.4. This observation table has empty columns for all of the rows, thus, for every $s_2 \in S \cdot I$, there exists $s_1 \in S$ such that $s_2 \cong s_1$. Hence, the table is closed. Since for the L_1 algorithm the set of rows S always has distinct members, trivially the observation table is compatible. On finding the observation table closed and compatible, the L_1 algorithm conjectures the model $Conj_{hvac_1} = (Q_{Chvac_1}, I, O, \delta_{Chvac_1}, \lambda_{Chvac_1}, q_{0Chvac_1})$ shown in Figure 6.4.

To verify correctness of the one state machine conjecture, the L_1 algorithm asks

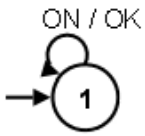

Figure 6.4: The Mealy machine conjecture $Conj_{hvac_1}$ from Table 6.4

an equivalence query to the oracle.

	$T5$
ϵ	Ω
$\underline{ON}/OK$	H

(a) The observation table after adding the suffix $T5$.

	$T5$
ϵ	Ω
ON/OK	H
$\cancel{ON \cdot ON/\Omega}$	$\cancel{\Omega}$
$ON \cdot OFF/S$	Ω
$ON \cdot T\text{-}5/H$	H
$ON \cdot T5/H$	H
$ON \cdot T15/S$	H
$\underline{ON \cdot T25}/F$	S
$ON \cdot T35/F$	S

(b) Make the observation table closed by moving the row ON to S.

	$T5$
ϵ	Ω
ON/OK	H
$ON \cdot T25/F$	S
$ON \cdot OFF/S$	Ω
$ON \cdot T\text{-}5/H$	H
$ON \cdot T5/H$	H
$ON \cdot T15/S$	H
$ON \cdot T35/F$	S
$\cancel{ON \cdot T25 \cdot ON/\Omega}$	$\cancel{\Omega}$
$ON \cdot T25 \cdot OFF/S$	Ω
$ON \cdot T25 \cdot T\text{-}5/S$	H
$ON \cdot T25 \cdot T5/S$	H
$ON \cdot T25 \cdot T15/S$	H
$ON \cdot T25 \cdot T25/F$	S
$ON \cdot T25 \cdot T35/F$	S

(c) Make the observation table closed by moving the row $ON \cdot T25$ to S.

Table 6.5: The HVAC controller inference observation table after adding the suffix $T5$ to E.

The conjectured model $Conj_{hvac_1}$ is not correct, and there can be more than one counterexample and oracle replies with one from them.

- $\lambda_{\mathcal{M}_{hvac}}(q_{0\mathcal{M}_{hvac}}, ON \cdot T5) = OK \cdot H$, but
- $\lambda_{Chvac_1}(q_{0hvac_1}, ON \cdot T5) = OK \cdot \Omega$.

The oracle replies with a counterexample $ON \cdot T5$. The L_1 algorithm adds the smallest suffix $T5$ of the counterexample $ON \cdot T5$ to the columns E and completes the table. This suffix makes the observation table not closed as presented in Table 6.5a.

The row ON makes the observation table not closed. It is moved to S and its one letter extensions $\{ON \cdot ON, ON \cdot OFF, ON \cdot T\text{-}5, ON \cdot T5, ON \cdot T15, ON \cdot T25, ON \cdot T35\}$ are added to $S \cdot I$. To calculate the output for last input of access strings in $S \cdot I$, the algorithm does not require to execute these output queries separately. The output for the last input of access strings can be calculated from the output query executed for any of the cells of that row. For instance, for row $ON \cdot T15$ and column $T5$, if the algorithm asks the output query $ON \cdot T15 \cdot T5$, the answer from the HVAC controller is $OK \cdot S \cdot H$. The output for the last input string of $ON \cdot T15$ is calculated as S, and H is recorded in the column $T5$ of the observation table. After completing the table, it can be observed from Table 6.5b that the row $ON \cdot T25$ makes the observation table not closed. Again the table is made closed by moving the row $ON \cdot T25$ to S. The observation table in Table 6.5c is closed. Since the algorithm L_1 maintains the condition $\forall s_1, s_2 \in S$ always $s_1 \not\cong s_2$, the observation table is always compatible. Hence, L_1 conjectures the Mealy machine $Conj_{hvac_2} = (Q_{Chvac_2}, I, O, \delta_{Chvac_2}, \lambda_{Chvac_2}, q_{0Chvac_2})$ shown in Figure 6.5.

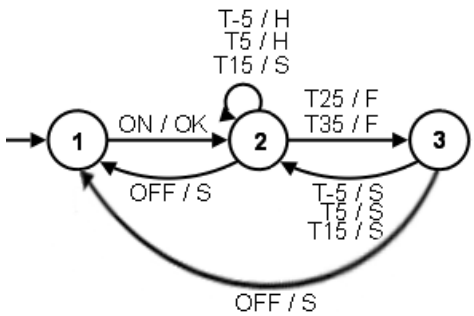

Figure 6.5: The Mealy machine conjecture $Conj_{hvac_2}$ from Table 6.5c

The L_1 algorithm asks an equivalence query to the oracle.

- $\lambda_{\mathcal{M}_{hvac}}(q_{0\mathcal{M}_{hvac}}, ON \cdot T\text{-}5 \cdot T25) = OK \cdot H \cdot S$, but
- $\lambda_{Chvac_2}(q_{0hvac_2}, ON \cdot T\text{-}5 \cdot T25) = OK \cdot H \cdot F$.

The oracle replies with a counterexample $ON \cdot T\text{-}5 \cdot T25$. The L_1 algorithm adds the smallest suffix $T25$ of the counterexample $ON \cdot T\text{-}5 \cdot T25$ to the columns E and completes the table. This suffix makes the observation table not closed as presented in Table 6.6a.

The row $ON \cdot T5$ as shown in Table 6.6a makes the observation table not closed. It is moved to S and its one letter extensions are added to $S \cdot I$. After completing the observation table, we get Table 6.6b. It is closed and compatible observation table.

	$T5$	$T25$
ϵ	Ω	Ω
ON/OK	H	F
$ON \cdot T25/F$	S	F
$ON \cdot OFF/S$	Ω	Ω
$ON \cdot T\text{-}5/H$	H	S
$\underline{ON \cdot T5}/H$	H	S
$ON \cdot T15/S$	H	F
$ON \cdot T35/F$	S	F
$ON \cdot T25 \cdot OFF/S$	Ω	Ω
$ON \cdot T25 \cdot T\text{-}5/S$	H	F
$ON \cdot T25 \cdot T5/S$	H	F
$ON \cdot T25 \cdot T15/S$	H	F
$ON \cdot T25 \cdot T25/F$	S	F
$ON \cdot T25 \cdot T35/F$	S	F

(a) The observation table after adding the suffix $T25$ to E.

	$T5$	$T25$
ϵ	Ω	Ω
ON/OK	H	F
$ON \cdot T25/F$	S	F
$ON \cdot T5/H$	H	S
$ON \cdot OFF/S$	Ω	Ω
$ON \cdot T\text{-}5/H$	H	S
$ON \cdot T15/S$	H	F
$ON \cdot T35/F$	S	F
$ON \cdot T25 \cdot OFF/S$	Ω	Ω
$ON \cdot T25 \cdot T\text{-}5/S$	H	F
$ON \cdot T25 \cdot T5/S$	H	F
$ON \cdot T25 \cdot T15/S$	H	F
$ON \cdot T25 \cdot T25/F$	S	F
$ON \cdot T25 \cdot T35/F$	S	F
$\cancel{ON \cdot T5 \cdot ON/\Omega}$	$\cancel{\Omega}$	
$ON \cdot T5 \cdot OFF/S$	Ω	Ω
$ON \cdot T5 \cdot T\text{-}5/H$	H	S
$ON \cdot T5 \cdot T5/H$	H	S
$ON \cdot T5 \cdot T15/S$	H	F
$ON \cdot T5 \cdot T25/S$	H	F
$ON \cdot T5 \cdot T35/S$	H	F

(b) Make the observation table closed by moving the row $ON \cdot T5$ to S.

Table 6.6: The HVAC controller observation table after adding the suffix $T25$ to E.

The L_1 algorithm conjectures the Mealy machine shown in Figure 6.3. The conjecture is a correct behavior of the HVAC controller and the algorithm terminates. The L_1 algorithm asks 7 output queries to construct Table 6.4, 7 output queries for Table 6.5b, 7 output queries for Table 6.5c, 14 output queries for Table 6.6a, and 13 output queries for Table 6.6b. Thus, in total it requires 38 output queries to learn the Mealy model of the HVAC controller instead of 203 output queries for the ${L_M}^*$ algorithm.

6.4 Experiments to Analyze Practical Complexity of L_1

In Section 5.2, we have presented 4 counterexample processing methods [Angluin 1987, Rivest 1993, Maler 1995, Shahbaz 2009] for ${L_M}^*$. The Rivest and Schapire method adds only a distinguishing sequence to the columns of the observation table by requiring a compromise on the suffix closure property of the observation table. Section 5.2.5 shows that a conjecture built from such an observation table may not be minimal and consistent with the observation table.

Shahbaz and Groz [Shahbaz 2009] show that their Mealy adaptation algorithm ${L_M}^+$ (their counterexample processing method adapted for ${L_M}^*$) performs better than Mealy inference with rest of the counterexample processing methods. We have performed an experimental evaluation to compare L_1 and ${L_M}^+$. Our experiments aim at finding out how both algorithms perform in practice. The algorithms are implemented by closely following their high level description. In the previous section, the worst case theoretical complexity analysis shows that L_1 performs better than ${L_M}^+$ in terms of required output queries. This is important to find the average case practical complexity of these algorithms.

Randomly generated state machines have been used to investigate the application of state machine inference algorithms ([Berg 2005b, Bollig 2009]). This testbed provides independence to generate state machines with given parameters. We use random machines, which allow us to study the influence of the various parameters (number of inputs, states, etc) on the learning algorithms. For both of the following sets of experiments in order to increase our confidence, we repeat the learning for every target machine for 30 times and average on the calculated data.

A set of Mealy machines is generated by fixing outputs and states sizes, but varying the number of inputs. We record the number of output queries to analyze the practical complexity of the learning algorithms. Since the CPU time depends on the execution of output queries in a black box implementation and we experiment on the simulated machines, it is useless to record the time. Execution of output queries varies a lot from one black box to another. For instance in the case of a web service of a slow interface (e.g. smart card) or a system with physical delays such as mechanical motions, the execution time of a query can be much greater than rest of the algorithm. We generate machines for inputs $|I|\in\{2, 3, \ldots 10\}$, outputs $|O|$=5 and states n=40. We learn the Mealy machines of these machines with both algorithms. We have simulated an oracle so that the algorithms can ask the equivalence queries

for the correctness of conjectured models. The oracle systematically compares the conjecture and target system's model to obtain the counterexamples.

In Figure 6.6 the number of output queries asked by both algorithms are presented. The vertical axis shows the number of output queries and horizontal axis shows the number of inputs. The results for output queries clearly show that L_1 outperforms ${L_M}^+$. We can observe that for the machines generated with smaller input set the gain is small and it increases with increase in the size of the input set. For instance, for the smallest machine with inputs size $|I| = 2$, on the average L_1 asks 385 output queries and ${L_M}^+$ asks 405 output queries. There is a 4.94% gain of output queries. Now, if we consider the machine generated with inputs size $|I| = 10$, on the average L_1 asks 1604 output queries and ${L_M}^+$ asks 4010 output queries. There is a gain of 60% for output queries.

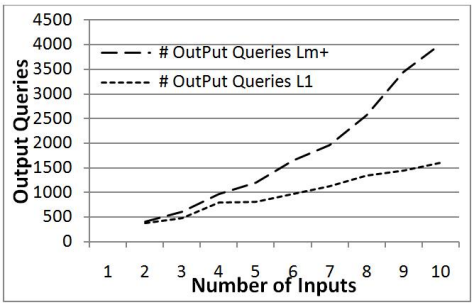

Figure 6.6: $|I|\in\{2, 3, \ldots 10\}$ and $|O|$=5,n=40

The second set of experiments is generated by fixing the size of input set and size of outputs set, but varying the number of states. The machines are generated with input set size $|I|$=5, the outputs set size $|O|$=7 and number of states $n\in\{3, 4 \ldots 40\}$. The size of outputs set is changed to get slightly different machines as compared to the first set of experiments. The L_1 and ${L_M}^+$ algorithms are executed on these machines one by one. Figure 6.7 presents the number of output queries asked by both algorithms. The vertical axis shows the number of output queries and horizontal axis shows the number of states. Again for this set of experiments L_1 outperforms ${L_M}^+$.

On the average L_1 requires 16 output queries to learn the machine with states size $n = 3$ and ${L_M}^+$ requires 80 output queries. So there is a gain of 80% for output queries. For the largest machine having states size $n = 40$, L_1 requires 784 output queries and ${L_M}^+$ requires 1202 output queries, there is a gain of 34.78% output queries. Thus, clearly L_1 outperforms ${L_M}^+$ for both of the considered sets of experiments. Here, we have inferred examples with relatively small input set, the gain with L_1 increases when we are required to learn black box systems with large input set. The counterexample processing method adapted for L_1 is also a contributing factor towards gain.

However, L_1 may require more counterexamples than ${L_M}^+$. To start learning L_1 asks equivalence query to the oracle. As the columns E of the observation table

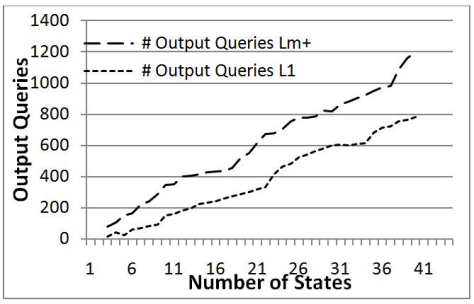

Figure 6.7: $|I|$=5,$|O|$=7 and $n\in\{3, 4 \ldots 40\}$

(S, E, L, T) are initially empty and they are populated only to process the counterexamples. Initially for one state conjecture and conjectures with a small number of states, very small sequences are counterexamples and such counterexample search is not costlier than an output query.

6.5 Conclusion

The Mealy adaptations of L^* available in the literature initialize the columns E_M with I. The L_1 algorithm does not initialize the columns with I and adds only those elements of I to the columns E which are distinguishing sequences or suffixes of distinguishing sequences. Adding fewer elements from I to E results in a reduced number of output queries and the gain with L_1 increases for systems with large input set. Secondly, all the members of I are not valid inputs for every state, the existing Mealy inference algorithms on identifying an access string for a state, add its one letter extensions for all inputs (valid and invalid) to the observation table. The one letter extensions of access strings with invalid inputs are unnecessary rows. The output recorded along access strings in the observation table helps L_1 to identify the unnecessary rows. Keeping only valid access strings in the observation table results in fewer rows S' and thus, fewer output queries.

The theoretical worst case time complexity for ${L_M}^*$ in terms of output queries is $O(|I|^2mn + |I|mn^2)$, which is reduced to $O(|I|^2n + |I|mn^2)$ by adapting the counterexample processing method from Maler and Pnueli [Maler 1995]. If Rivest and Schapire [Rivest 1993] counterexample processing method is adapted for ${L_M}^*$, then the complexity is reduced to $O(|I|^2n+|I|n^2+nlog(m))$ output queries, but this method requires a compromise on the suffix closure property of the observation table. The worst case time complexity for L_1 in terms of output queries is $O(|I|mn^2)$. Thus, there is a gain of $|I|^2n$ output queries for L_1. Here, m is the length of the longest distinguishing sequence added from a counterexample, which is always smaller than length of the longest counterexample processed by L_1. The size of inputs $|I|$ is also reduced to only the valid inputs for every state. Real world systems work on huge data sets as their possible inputs, thus, gain with L_1 over ${L_M}^*$ and its variants becomes more visible on inferring models of such systems.

Chapter 7

Mealy Inference without Using Counterexamples

Contents

7.1 Organization of Output Queries 102
- 7.1.1 Output Queries and Dictionaries 102
- 7.1.2 Motivation 103
- 7.1.3 Improved Heuristic 103

7.2 Motivation 103

7.3 Learning without Counterexamples 104
- 7.3.1 The GoodSplit Algorithm 104
- 7.3.2 Termination Criteria for the GoodSplit Algorithm 106
- 7.3.3 Greedy Choice for the GoodSplit Algorithm 107

7.4 Mealy Adaptation of the GoodSplit Algorithm 107
- 7.4.1 Observation Table 108
- 7.4.2 The $L_{M\text{-}GS}$ Algorithm 110
- 7.4.3 Example for learning with $L_{M\text{-}GS}$ Algorithm 112
- 7.4.4 Complexity 117

7.5 Discussion 117

7.6 Conclusion 118

We propose a technique which infers models of software components as Mealy machines without using counterexamples. The technique is based on the GoodSplit algorithm, which was proposed by Eisenstat and Angluin [Eisenstat 2010] as an improvement over the L^* algorithm. For black box model inference, the L^* algorithm requires an oracle that can answer equivalence queries. The GoodSplit algorithm learns black box models without using equivalence queries and by taking into account an estimate on the length of distinguishing sequences. The GoodSplit algorithm uses a greedy choice to ask membership queries which is only applicable queries with answers "accept/reject". In this chapter, we propose the Mealy adaptation of the GoodSplit algorithm. The learning algorithm is reorganized along some improvements that enable it to learn Mealy models. The algorithm uses a technique to ask the output queries. The technique is based on the following observation: some output queries required to learn a black box model that are prefixes to other tests

can be deduced from the answers executeded for the larger tests. The improvements for the Mealy inference algorithm enable it to learn the software black box models with a reduced number of output queries.

In the first section of this chapter, we provide a discussion on the concept of dictionaries, the motivation that leads us to devise the technique for calculating output queries and the new technique. The second section presents the generic concept of oracle. The third section discusses the need to learn without counterexamples and entails the details about the issues with oracle. The fourth section introduces the GoodSplit algorithm. In the fifth section, the Mealy GoodSplit algorithm $L_{M\text{-}GS}$ is formally presented and explained with the help of an example. The chapter provides a discussion before concluding the chapter.

7.1 Organization of Output Queries

The DFA inference algorithms ask membership queries that are answered as accept or reject [Angluin 1987, Eisenstat 2010], whereas the Mealy inference algorithms ask output queries that are replied with an output string [Niese 2003, Shu 2007].

7.1.1 Output Queries and Dictionaries

It is observed that in the process of learning unknown models with the variants of Angluin algorithm L^* [Howar 2010, Niese 2003, Shahbaz 2009], the algorithms may repeatedly need to ask the same output queries. For deterministic systems, asking again the same output query from a target system will result in the same answer. If an output query for a cell of the observation table is calculated, then its answer can be reused to answer the same or its prefix output query for a different cell. It consumes resources and takes time to calculate the output queries from a black box. Moreover, recalculating the output queries increases the learning cost. If we use dictionaries to record answers for output queries for software model inference, it will accelerate the learning process by consuming fewer resources [Niese 2003]. To calculate output queries, in the literature, we can find techniques that consult answers of already calculated output queries before executing them into a target system [Niese 2003, Eisenstat 2010, Howar 2010]. The dictionaries and filters can be used to avoid asking the output queries repeatedly. To answer the output queries the filters are constructed from the assumption of input determinism, knowledge of the target system (domain specific constraints) and the known answers to the output queries (optimization filters are presented in Section 4.2). With the implementation of dictionaries, an output query will be asked to a target black box system only when its output cannot be inferred from the already known queries. Whenever an output query is calculated, it is recorded in the dictionaries before adding its answer to an observation table.

7.1.2 Motivation

In the process of learning unknown models with the algorithm ${L_M}^*$ (the Mealy adaptation of L^*) and even with the L_1 algorithm, we may very often repeatedly require to ask the output queries that have already been asked (but for a different cell of the observation table). This is because the concatenation of different access strings and distinguishing strings of an observation table may result in a similar output query. It will reduce the learning cost, if we infer the answer for an output query from the answers of output queries whose outputs/answers are already calculated (without consulting the target black box system). The concept of query caching has been used for learning Mealy models [Niese 2003, Cho 2010, Howar 2010]. The idea of dictionaries implementation is general and helps to avoid asking queries repeatedly. An output query is executed in a target black box system only when its output cannot be inferred from the dictionaries. The answer of an output query from the dictionaries can be used to answer the output query and its prefix output queries.

Generally, the learning algorithms ask the output queries by concatenating the access strings and distinguishing strings in the order they are found in the observation table. However, if from unanswered output queries, we first calculate an output query which can answer relatively larger number of unanswered output queries, then the number of output queries required for model inference of a black box system is reduced. Thus, the order in which output queries are asked is important.

7.1.3 Improved Heuristic

We have observed that while learning with ${L_M}^*$, at times we may require to ask a number of output queries at once, especially when ${L_M}^*$ adapts the counterexample processing method from Maler and Pnueli [Maler 1995]. To process a counterexample CE with this method, the algorithm requires to ask $|CE| \times |S_M \cup S_M \cdot I|$ output queries. Asking longer queries prior to smaller queries helps to calculate the answers for the smaller queries (that are prefix to these longer queries). The intuition behind this technique is to ask those output queries first that have greater number of unanswered output queries as prefix to them. This technique proposes to choose an output query with a maximum number of prefixes in a current set of unanswered output queries. This output query is executed in the target black box system and its output is recorded in the dictionaries. This output also helps to infer outputs for its prefix output queries.

7.2 Motivation

For software black box implementations in reality an oracle does not exist. A number of heuristics have been proposed to circumvent this deficiency. Almost all of them involve a compromise on the precision. For inferring and testing black box software systems, the most common procedure is the construction of an input se-

quence by a random walk on a uniform distribution of inputs. The resulting input sequence is provided in parallel to the conjectured model and black box to find the differences [Angluin 1987, Howar 2010, Irfan 2010c]. The counterexamples found by this method are often very long. Since the counterexample sequences are built randomly, there is a strong possibility that such sequence are constructed by traversing iteratively the states which are already present in a learned model. This type of long counterexamples include most of the information that is already known to the learned model. To circumvent the negative impact of long counterexamples, in the literature we can find the counterexample processing methods [Angluin 1987, Maler 1995, Rivest 1993, Shahbaz 2009, Irfan 2010c]. However, if we can avoid the usage of an oracle or equivalence oracle, it may ease the job of test engineers.

7.3 Learning without Counterexamples

Since searching for counterexamples for black box model inference is a complicated task, construction of a technique which can learn the models of black box implementations without requiring counterexamples is important. The GoodSplit learning algorithm is a worthy contribution in this regard. To find new states, this algorithm does not require the counterexamples from an oracle. Instead, it adds all the sequences of a given length to the columns of the observation table.

The oracle also provides a termination criteria by saying "yes" to a conjectured model. In the absence of an oracle the GoodSplit algorithm requires an alternative criterion to stop the learning process. There are many possibilities in this regard; the original version of the GoogSplit algorithm which stood third in the Zulu competition [Combe 2010] uses a limit on the number of membership queries.

7.3.1 The GoodSplit Algorithm

The GoodSplit algorithm [Eisenstat 2010] uses an observation table to record the answers from target model interactions. This algorithm adds the distinguishing sequences to the columns of the observation table like [Rivest 1993, Maler 1995, Shahbaz 2009, Irfan 2010c]. The models are learned as DFA accepting an unknown regular language over a given alphabet Σ. The outputs of the queries are recorded in the cells of the observation table in the form of *accept* or "1" and with *reject* or "0", otherwise. This algorithm does not fill the observation table completely, some cells contain either *accept* or *reject* depending on the output of the corresponding membership query, while other cells do not contain anything at all until queries for those cells are calculated. The rows of the observation table are states and the columns are the distinguishing sequences. The set of columns is denoted by $\Sigma^{\leq l}$, where l is the length of the longest suffix added to the observation table. The set of distinct rows is denoted by D and it is initialized with ϵ, where ϵ is an empty string. The set of all distinct and equivalent to distinct rows is denoted by P, i.e. $P = D \cup D \cdot \Sigma$.

Two rows r_1, $r_2 \in P$ are said to be *consistent* iff $\forall e \in \Sigma^{\leq l}$, if both $r_1 \cdot e$ and $r_2 \cdot e$ are non-empty, then they must be equal, and it is denoted by $r_1 \cong_c r_2$.

The inconsistency of rows r_1 and r_2 is denoted as $r_1 \ncong_c r_2$. For any row r the set D_r denotes the rows from D consistent with r. A row is identified as distinct, if it is not consistent with any of the distinct rows. For all $r \in P$, $[r]$ denotes the equivalence class of a row r, the equivalence class of rows includes all the rows that are consistent with r. For a row r we always have $D_r \sqsubseteq [r]$. Initially, the observation table has one row and one column, both consisting of the empty string ϵ. The answer for the only cell of the observation table is calculated and recorded. The set of executed queries is cached and consulted before asking new queries and is maintained as $A = A_0 \cup A_1$, where A_0 are the queries answered "0" and A_1 are the queries answered "1". The algorithm was proposed for the ZULU competition [Combe 2010], where a limited number of queries were allowed. This algorithm uses the query limit as the termination criteria. The algorithm operates in the following steps.

1. For the transitions of the states, $\forall r \in D$ and $a \in \Sigma$, if $r \cdot a \notin P$, then add $r \cdot a$ to P and complete the table by asking membership queries.

2. If $\exists r_1 \in (P \backslash D)$ and $\forall r_2 \in D$, $r_1 \ncong_c r_2$, then move r_1 to D. Repeat this process until P does not change anymore.

3. $\forall r \in (P \backslash D)$, as long as $|D_r| > 1$, the algorithm selects greedily a suffix $e \in \Sigma^{\leq l}$ and queries $r \cdot e$.

   The greedy choice is made as follows: For $b \in \{0, 1\}$, let
   $$v_b(r, e) = |\{r' \in D_r : r' \cdot e \in A_b\}|$$
   that is the number of $r' \in D_r$ such that $r' \cdot e$ has been queried and answered b, then $e \in \Sigma^{\leq l}$ is chosen to maximize
   $$v(r, e) = min\{v_0(r, e), v_1(r, e)\}.$$
   Since elements of D_r are inconsistent with each other, there will be at least one e with $v(s, e) \geq 1$. The greedy choice maximizes the minimum number of possible identifications that could be eliminated by a membership query.

4. The algorithm uses the following heuristic to decide whether or not to increment the current suffix length l. If more than 90% of table cells are filled, then l is incremented by 1, i.e. $(r, e) \in (P \backslash D) \times \Sigma^{\leq l}$ and $r \cdot e \in A$ is greater than 90%, then increment l by 1.

5. For $\lceil |D|/2 \rceil$ random choices $(r, e) \in (P \backslash D) \times \Sigma^{\leq l}$ such that answer for $r \cdot e$ is not known, query both $r \cdot e$ and $r' \cdot e$ such that $r' \in D$ is consistent with r. Return to step 1.

After reaching the query limit, the algorithm executes step 1 and step 2, as if at this stage there are distinct states in $(P \backslash D)$, they can be moved to D. The conjecture is constructed from the observation table as follows:

Definition 7 *The GoodSplit algorithm constructs a DFA conjecture* $Conj = (Q, \Sigma, \delta, F, q_0)$ *from the observation table defined as:*

- $Q = \{[r] | r \in D\}$;
- $q_0 = [\epsilon]$;
- $F = \{[r] | r \in D \wedge r \in A_1\}$,
  $\exists r \in D$, *such that* $r \notin A$, *then it is randomly selected as accepting or rejecting;*
- $\delta([r], a) = [r \cdot a], \forall r \in D, a \in \Sigma$,
  if $r \cdot a$ *is consistent with more than one* $r' \in D$, *then one from them is selected randomly.*

The strings in D are the distinct states for the inferred model and the initial state is ϵ. If the corresponding string of the state in the observation table belongs to A_1, then it is an accepting state, and if it belongs to A_0, then it is a rejecting state. If the string does not belong to A, or the string has not been queried, then it is selected from a uniform distribution over *accept* or *reject*. For all $r \in D$, the transition function $\delta(r, a)$ maps the transition for state r with letter a to $r \cdot a$, if r is in D. Otherwise, $r' \in D$ is selected such that $r' \cong_c r \cdot a$, if $r \cdot a$ is consistent with more than one r', then from such r', one distinct state is selected randomly.

7.3.2 Termination Criteria for the GoodSplit Algorithm

The GoodSplit learning algorithm does not use an oracle which can provide an answer "yes", when the learned model is correct and learning process can be stopped. However, it needs a "Heuristic" that may answer "yes" when the correct model is learned and learning process can be stopped. The "Heuristic" can use the information about the number of states to terminate the learning process. For black box model inference such a guess could be difficult. So we look for the other options that can be a limit on the number of output queries or limit on the length of suffixes l added to the observation table. The monograph from Trakhtenbrot and Barzdin [Trakhtenbrot 1973] indicates that for complete finite state machines with n states, g inputs, and h outputs, the length of input sequences reaching all n states is asymptotically equal to $log_g(n)$ and distinguishing states just $log_g log_h(n)$.

This means that we can use this result for the length of l to *terminate* the learning process if we know about the sizes of inputs, outputs and number of states. For $d = \xi + log_g log_h(n)$, querying all suffixes in $\Sigma^{\leq d}$ for each state and its one letter extension (state successor) entails about $k^{1+\epsilon} n log_h(n)$ membership queries, where ξ is a small constant. But again, we need to know the number of states to determine the number of membership queries required for a model.

For the GoodSplit learning algorithm, for any two rows $r_1, r_2 \in D$, we always have $r_1 \ncong_c r_2$, which means that the number of rows in D corresponds to the number of states in a conjectured model. For black box checking Peled *et al.* assume that a bound n on the number of states of the checked system is known [Peled 1999]. If

we know about a bound n on the number of states, then as soon as the size of D becomes equal to n, the learning algorithm stops the learning process and terminates by constructing a conjecture.

7.3.3 Greedy Choice for the GoodSplit Algorithm

Each distinct row $r \in D$ of the observation table is an access string to a state. We want to be able to figure out the transitions with every $i \in \Sigma$ for these states. For every $r \in D$, its one letter extension is $r \cdot a \in P$ for some $a \in \Sigma$. Since to save membership queries the GoodSplit algorithm keeps some cells of the observation table empty, it is possible to have some $r \cdot a \in P \backslash D$ such that $r \cdot a$ is consistent with more than one $r' \in D$. The algorithm wants to bring it down to one. This can be done immediately if all of the table cells for the row $s \cdot a$ are filled.

This is where the greedy part of the algorithm comes in. To optimize the query consumption the algorithm wants to query some of the entries in row $r \cdot a$. The algorithm queries to eliminate as many consistent distinct states as possible (with $s \cdot a$). It examines each column one by one to count the number of entries filled with "accept" and "reject". If the algorithm queries $(r \cdot a, e)$ for some $e \in \Sigma^{\leq l}$ and the answer is "accept", then all of the rows with rejecting entries will be eliminated. On the other hand, if the answer is "reject", then all of the rows with accepting entries will be eliminated. The greedy choice maximizes the worst case number of rows eliminated as follows: take the minimum of the number of accepting and rejecting entries for each column, then maximize this value.

For DFA model inference the membership queries have boolean answers, whereas for Mealy inference, the output queries are answered with strings of outputs. Thus, this greedy choice will not work for Mealy inference.

7.4 Mealy Adaptation of the GoodSplit Algorithm

Like the L^* algorithm, the GoodSplit algorithm can learn Mealy models of input/output (i/o) systems using model transformation techniques by taking the union or the cross product of sets of inputs I and outputs O of the target unknown Mealy machine as alphabet for DFA [Hungar 2003b, Groce 2006, Mäkinen 2001]. However, this increases the size of alphabet which results in increasing the complexity of the learning algorithm. Learning directly Mealy models results in requiring less number of output queries [Niese 2003, Shahbaz 2009].

We propose the Mealy version $L_{M\text{-}GS}$ of the GoodSplit algorithm which learns the models of target systems as Mealy machines. The $L_{M\text{-}GS}$ learning algorithm operates with an additional assumption along the assumptions described in Section 4.3.

– A bound on the number of states in the target black box software is known.

This algorithm learns the models as Mealy machines by using the general settings of the GoodSplit algorithm. The algorithm explores the target model by asking

output queries [Niese 2003, Shu 2007] and organizes the outputs in an observation table. This algorithm introduces some improvements which are as follows:

- The observation table keeps only valid access strings.
- On adding new rows and columns for the observation table, the answers for their output queries which can be inferred from known answers (dictionaries) are added at the time of adding new rows or columns. This helps to find distinct states fast.
- The greedy choice of the GoodSplit algorithm for asking boolean membership queries does not work for the Mealy version of the GoodSplit algorithm. A greedy choice is proposed which is applicable to the answers of the output queries.

The observation table for $L_{M\text{-}GS}$ is defined as follows.

7.4.1 Observation Table

We define an observation table (S, E, T) for the Mealy learning algorithm.

- $S \subseteq I^*$ is a prefix closed non empty finite set of access strings, which labels the rows of the observation table,
- $E \subseteq I^{\leq l}$ is a suffix closed finite set, which labels the columns of the observation table, $\epsilon \notin E$ and l is the length of the longest suffix added to the columns.
- for $S' = S \cup S \cdot I$, the finite function T maps $S' \times E$ to outputs O^+,

The observation table rows S' are non empty and initially, $S = \{\epsilon\}$ and $S \cdot I = I$. At the beginning, the value for l is 1 as the columns E are initialized to I and E augments only when l in incremented. The observation table is completed by extending T to $S' \cdot E$ by asking the output queries greedily. The access strings are concatenated with the distinguishing strings to construct the output queries as $s \cdot e$, for all $s \in S'$ and $e \in E$. In the observation table, $\forall s \in S'$, $\forall e \in E$, $T(s, e) = \mathit{suffix}^{|e|}(\lambda(q_0, s \cdot e))$. The initial observation table (S, E, T) for Mealy inference of the machine in Figure 2.2 is presented in Table 7.1, where the inputs set I is $\{a, b\}$.

Table 7.1: Initial observation table for Mealy machine in Figure 2.2

		E	
		a	b
S	ϵ		
$S \cdot I \backslash S$	a		
	b		

The consistency of two rows of the observation table is defined with the help of the function T.

Two rows $s_1, s_2 \in S'$ are *consistent*, if
$\forall e \in E, T(s_1, e) \wedge T(s_2, e)$ are known $\Rightarrow T(s_1, e) = T(s_2, e)$. It is denoted as $s_1 \cong_c s_2$.

Two rows $s_1, s_2 \in S'$ are *inconsistent*, if
$\exists e \in E$ *such that* $T(s_1, e) \wedge T(s_2, e)$ are known and $T(s_1, e) \neq T(s_2, e)$ and it is denoted as $s_1 \ncong_c s_2$.

Two rows of the observation table are said to be equivalent if they are consistent, the equivalence class of a row $r \in S'$ is denoted by $[r]$. The distinct rows $s \in S$ of the observation table are always inconsistent. The observation table is *not closed*, if $\exists s_1 \in S \cdot I \backslash S$ such that $\forall s_2 \in S$, $s_1 \ncong_c s_2$. If s_1 makes the observation table not closed, then s_1 is moved to distinct rows S.

For all $s \in S$, the set D_s denotes the set of rows $s_1 \in S$ which are consistent with s. From the observation table *all transitions cannot be determined*, if $\exists s \in S \cdot I \backslash S$ and $|D_s| > 1$.

The observation table is eventually used to construct a Mealy conjecture. The Mealy machine conjecture is a minimal machine which is consistent with the observations in the table. To construct a conjecture, we use the following definition.

Definition 8 *If an observation table (S, E, T) is such that all the one letter extensions of distinct rows are distinct or consistent with one distinct row and all the columns for the inputs of distinct rows are filled, then a Mealy Conjecture $Conj_1 = (I, O, Q_C, q_{0C}, \delta_C, \lambda_C)$ is defined as:*

- $Q_C = [s] | s \in S$;
- $q_{0C} = [\epsilon]$, $\epsilon \in S$ *is the initial state of the conjecture;*
- $\delta_C([s], i) = [s \cdot i], \forall s \in S, \forall i \in I$;
- $\lambda_C([s], i) = T(s, i), \forall s \in S, \forall i \in I$.

Now to verify that the Mealy conjecture $Conj_1$ is well defined, we know that S is a non-empty prefix closed set and it always contains at least one row ϵ, hence Q_C and q_{0C} are well defined. Now $\forall s_1, s_2 \in S'$ if $s_1 \cong_c s_2$, we have $[s_1] = [s_2]$ and the distinct rows of the observation table are always inconsistent, then $(s_1 \cong_c s_2) \Rightarrow \forall i \in I (s_1 \cdot i \cong_c s_2 \cdot i)$ [Rivest 1993, Shahbaz 2009], which implies that $\forall i \in I, [s_1 \cdot i] = [s_2 \cdot i]$. Since every non distinct row is consistent with a distinct row, $\exists s \in S$ such that $[s] = [s_1 \cdot i] = [s_2 \cdot i]$. Hence the transition function for the conjecture δ_C is well defined. The set of columns E is a non-empty set and $E \supseteq I$ always holds, if there exists $s_1, s_2 \in S'$ such that $s_1 \cong_c s_2$, then for all inputs $i \in I$, $T([s_1], i) = T([s_2], i)$, which implies that, the output function λ_C is also well defined. Since for every distinct row (distinct access string) its one letter extensions are added to $S \cdot I$ and at least output query for one cell of every one letter extension is queried to identify the corresponding distinct row from S (to which it is consistent), all the

output queries for the columns for the inputs of distinct rows are known (thanks to dictionaries).

Theorem 2 *If (S, E, T) is a closed and compatible observation table, then the Mealy machine conjecture from (S, E, T) is consistent with the finite function T. That is for every $s \in S'$ and $e \in E$, $\lambda(\delta(q_{0C}, s), e) = T(s, e)$. Any other conjecture consistent with T but inequivalent to $Conj_1$ must have more states.*

The correctness of the conjectured Mealy machine is claimed by the Theorem 2. Niese [Niese 2003] provides the formal proof for Mealy inference by adapting the formal proofs for L^*. He proves that the inferred Mealy machine is consistent with the observation table by using the prefix closure and suffix closure properties for the rows and columns of the observation table. The $L_{M\text{-}GS}$ learning algorithm respects all the properties used by Niese for the formal proofs. The learning algorithm by Niese makes the observation table closed and compatible before constructing the conjecture. The closure property means, if $\exists s' \in S \cdot I \backslash S$ such that $\forall \in S, s' \not\approx_c s$, then such a row is moved to distinct rows S and observation table for the $L_{M\text{-}GS}$ algorithm always preserves this property. The compatibility property is only required to be checked, when we have equivalent rows in S [Rivest 1993, Shahbaz 2009], but $L_{M\text{-}GS}$ learning algorithm always keeps the distinct rows in S (which are inconsistent with each other), thus the observation table is always compatible. ∎

We define N_e and $P_{s \cdot e}$ notations as follows:

- $\forall s \in S \wedge \forall e \in E$, N_e is the set of queries where $T(s, e)$ has been calculated (the output queries that have been calculated for distinct rows of a column e),
- $P_{s \cdot e}$ is the set of not calculated *prefixes*$(s \cdot e)$ output queries where $s \in S' \wedge e \in E \wedge T(s, e)$ is not calculated (the set of all prefix output queries of an output query that have not been calculated yet).

7.4.2 The $L_{M\text{-}GS}$ Algorithm

The $L_{M\text{-}GS}$ learning algorithm maintains an observation table (S, E, T) to record the answers O^+ of the output queries I^+. The set of distinct rows S is initialized to $\{\epsilon\}$ and the columns E are initialized to I, i.e. the length of the suffixes which are added to the columns of the observation table is one ($l = 1$). The learning algorithm targets at learning a model without using counterexamples and by using minimum possible output queries. The output queries are constructed by concatenating the row labels (access strings) with column labels (distinguishing strings) as $s \cdot e$ where $s \in S'$ and $e \in E$. To maximize the utilization of answers calculated for output queries with the help of dictionaries, whenever there are no constraints to execute a specific output query, the algorithm uses the heuristic described in Section 7.1.3 to ask output queries. Initially, the output queries for the cells having similar columns

and rows labels are such possibilities. Executing such an output query results in answering output queries for two cells in a column. After calculating these output queries the algorithm searches for distinct states. If (S, E, T) is not closed, then the $L_{M\text{-}GS}$ algorithm finds a row $s_1 \in S \cdot I \backslash S$, such that $s_1 \not\cong_c s_2$, for all $s_2 \in S$, then the $L_{M\text{-}GS}$ algorithm moves s_1 to S and adds its valid one letter extensions to the observation table. The algorithm fills the cells of newly added rows whose output queries can be inferred from the dictionaries.

Algorithm 9 The $L_{M\text{-}GS}$ Learning Algorithm

Input: Black-box, Number of States, k and set of inputs I
Output: Mealy Machine Conjecture

```
begin
    initialize (S, E, T) with S = ϵ, E = I and l = 1;
    calculate output queries s · e where s = e for s ∈ S′ ∧ e ∈ E;
    repeat
        do
            while (S, E, T) is not closed do
                find s1 ∈ S · I\S such that s1 ≇c s2, for all s2 ∈ S;
                move s1 to S;
                add valid s1 · i to S · I, for all i ∈ I;
                update added row cells whose queries are prefixes(calculated queries);
            end
            while all transitions cannot be determined from (S, E, T) do
                find s ∈ S · I\S such that |Ds| > 1;
                ∀e2 ∈ E, select e1 ∈ E where T(s, e1) is not calculated and (|Ne1| ≥
                |Ne2|);
                calculate s · e1 and record outputs to prefixes(s · e1) output queries;
            end
        while (S, E, T) is not closed;
        while the table cells filled < k% do
            calculate s1 · e1 for s1 ∈ S′, e1 ∈ E where ∀s2 ∈ S′, ∀e2 ∈ E(s1 ≠ s2 ∨ e1 ≠
            e2) ∧ (|Ps1·e1| ≥ |Ps2·e2|);
            record outputs to prefixes(s1 · e1) output queries;
            if (S, E, T) is not closed then break while loop;
        end
        if the table cells filled > k% then
            if "most" of the suffixes of length l are added then increment l;
            else add a new suffix of length l;
        end
    until S is equal to Number of States;
    return the conjecture C from (S, E, T)
end
```

For the transitions, the algorithm ensures that all the one letter extensions (state successors) of the distinct rows are distinct or consistent with at most one distinct

row. If all transitions cannot be determined from (S, E, T), then the algorithm finds $s \in S \cdot I \backslash S$ such that $D_s > 1$. The algorithm greedily choses a suffix $e \in E$ and queries $s.e$. The greedy choice is made as follows. For all the empty cells of the row s, a column label $e \in E$ is chosen to maximize $|N_e|$.

If the size of S is not equal to the number of states and $k\%$ of the table cells are not filled, the algorithm asks extra output queries by selecting with the improved technique described in Section 7.1.3 until $k\%$ of the table cells are filled or the observation table becomes not closed. If the improved technique is not applicable to output queries, then the algorithm selects an output query randomly. Default value for k is 80, as for machines that we have inferred querying 80% of total output queries was sufficient to infer a correct model. However, this may change depending on the target system.

When output queries for 80% of the table cells have been queried, the algorithm increments l. It can be useless consumption of output queries to query every suffix of length l when fewer are required. So the algorithm adds suffixes one by one. Since last few suffixes may not distinguish any new state, after adding "most" of the suffixes of length l, it is incremented. Here we use a guess on number of states to terminate the learning process. However, this can be changed with a limit on number of output queries or maximum possible length for l. On finding the size of S equal to the provided number of states, the algorithm conjectures a Mealy machine model and terminates.

7.4.3 Example for learning with $L_{M\text{-}GS}$ Algorithm

As an application example of the $L_{M\text{-}GS}$ learning algorithm, we consider the "unknown" Mealy model shown in Figure 2.2 with three states that are labeled as A (the initial state), B, and C. This machine is defined over inputs $\{a, b\}$ and outputs $\{x, y\}$. For this example, we use the number of states (which is 3 in this case) to stop the learning process of the $L_{M\text{-}GS}$ algorithm, i.e. when the size of S reaches 3 the algorithm terminates by conjecturing a Mealy machine model.

The learning algorithm starts learning the unknown model by initializing the observation table (S, E, T). The column indices E are initialized with I and the row indices S are initialized with ϵ and one letter extensions of ϵ are updated in $S \cdot I$ as shown in Table 7.1. Now, in order to fill in the observation table, the algorithm selects all the cells of the observation table that have the same input symbol as their row and column label. Executing these output queries results in filling two cells for every column of the observation table. From the output of the query $a \cdot a$, output for $\epsilon \cdot a$ can also be derived and this fills two cells of the column a. After filling the table by executing the queries for such cells, the table is presented in Table 7.2.

Now the $L_{M\text{-}GS}$ algorithm searches for any row in $S \cdot I \backslash S$ that is inconsistent with S rows. In Table 7.2, none of the rows from row a and row b is inconsistent with ϵ. The algorithm finds the rows in $(S \cdot I) \backslash S$ that are consistent with more than one distinct row. Since there is only one row in S, this condition is satisfied. Again the observation table is checked for the closure property but here nothing has been

Table 7.2: Observation table after calculating the output queries $a \cdot a$ and $b \cdot b$.

		E	
		a	b
S	ϵ	x	y
$S \cdot I \backslash S$	a	x	
	b		y

changed for making every one letter extension of distinct rows equal to a unique distinct row, the closure property is satisfied. Since the observation table cells filled are less than 80%, the algorithm continues filling them. From the output queries that have not been calculated yet, none of them is prefix of others. So the improved technique to execute output queries described in Section 7.1.3 is not aplicable. The output query $a \cdot b$ is selected randomly and its output is calculated.

Table 7.3: The observation table after adding the answer of the output query $a \cdot b$

		E	
		a	b
S	ϵ	x	y
$S \cdot I \backslash S$	a	x	y
	b		y

Now more than 80% of the table cells are filled and the number of distinct rows S in the observation table is less than the number of states in the target model. The algorithm increments l (the length of suffixes added to the columns of the observation table). Since it can be wasteful to query every suffix of length l, the algorithm adds a suffix aa of length l to the observation table and updates the cell $\epsilon \cdot aa$, as the output query aa can be inferred from the dictionaries.

Table 7.4: Adding the suffix aa to the observation table

	a	**b**	**aa**
ϵ	x	y	xx
a	x	y	
b		y	

The algorithm checks if the table is not closed or any of the one letter extensions is consistent with more than one distinct states, which is not the case. Now, the algorithm considers all the unanswered output queries and selects the one which can answer the maximum number of output queries using the heuristic described in Section 7.1.3. In Table 7.5, the output queries are provided which are not calculated yet.

From Table 7.5, it can be observed that $b \cdot a$ is prefix to another output query

Table 7.5: Unanswered output queries

$a \cdot aa$	$b \cdot a$	$b \cdot aa$

$b \cdot aa$. The improved heuristic to calculate output queries selects the output query $b \cdot aa$. The algorithm calculates $b \cdot aa$ and updates the cells $b \cdot a$ and $b \cdot aa$.

Table 7.6: Calculating the output query $b \cdot aa$

	a	**b**	**aa**
ϵ	x	y	xx
a	x	y	
b	x	y	xx

None of the rows has made the observation table not closed. Now, more than 80% of the table cells are filled and the number of distinct rows in S of the observation table is less than number of states in the target model. The algorithm adds another suffix ab of the length l to the observation table and updates the cell $\epsilon \cdot ab$, as the output query ab can be inferred from the dictionaries.

Table 7.7: Adding the suffix ab to the observation table

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		
b	x	y	xx	

The algorithms checks if the table is not closed or any of the one letter extensions is consistent with more than one distinct states, which is not the case. Since the improved heuristic to calculate output queries is not applicable here, the algorithm selects an output query randomly.

Table 7.8: Adding the output for the output query $a \cdot ab$

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		xx
b	x	y	xx	

Now, the observation table is not closed so the algorithm breaks the process of calculating the output queries and makes it closed as shown in observation table presented in Table 7.9.

The row a is inconsistent with the distinct row ϵ, so the algorithm moves it to the distinct states and adds its one letter extensions to $S \cdot I$ as shown in Table 7.9. Now the algorithm checks if the observation table is not closed, which is the case.

Table 7.9: Moving the row a to S

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		xx
b	x	y	xx	
aa		x		
ab				

As the row aa is inconsistent with the distinct rows ϵ and a, the algorithm moves it to the distinct states and adds its one letter extensions to $S \cdot I$ as shown in Table 7.10.

Table 7.10: Moving the row aa to S

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		xx
aa		x		
b	x	y	xx	
ab				
aaa				
aab				

The algorithm checks again if the observation table is not closed, which is not the case. To figure out the transitions table, the algorithm checks if all the transition can be determined from the observation table. None of the $S \cdot I \backslash S$ rows is consistent with only one distinct row. The algorithm fills the cells for these rows to bring it down to one. For the row b, it has only one cell which has not been calculated yet, the output query for this cell is calculated and it becomes consistent to only one distinct row ϵ as presented in Table 7.11.

Table 7.11: Adding the output for the output query $b \cdot ab$

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		xx
aa		x		
b	x	y	xx	xy
ab				
aaa				
aab				

For the row ab all of the cells are empty, so the output query for the column

cell having maximum number of cells filled in distinct rows will be calculated. The output query $ab \cdot b$ is calculated, but even after updating the answer, the row ab is consistent with more than one distinct rows.

Now, the columns a and ab have equivalent number of cells filled in the distinct rows. But a is prefix to ab, so the output query $ab \cdot ab$ is calculated. After updating the answers for output queries $ab \cdot ab$ and $ab \cdot a$, the row ab becomes consistent with only one row ϵ. The observation table is presented in Table 7.12.

Table 7.12: Adding the output for the output queries $ab \cdot a$ and $ab \cdot ab$

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y		xx
aa		x		
b	x	y	xx	xy
ab	x	y		xy
aaa				
aab				

For the row aaa all of the cells are empty so the output query for the column cell having maximum number of cells filled in distinct rows will be calculated. The output query $aaa \cdot b$ is calculated and recorded in the observation table. The row becomes consistent to only one distinct row aa as presented in Table 7.13.

Table 7.13: Calculating the output query $aaa \cdot b$ and updating cells ($a \cdot aa$, $aa \cdot a$, $aa \cdot ab$,$aaa \cdot b$)

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y	xx	xx
aa	x	x		xx
b	x	y	xx	xy
ab	x	y		xy
aaa		x		
aab				

For the row aab all of the cells are empty so the output query for the column cell having maximum number of cells filled in distinct rows will be calculated. The columns a and ab have equivalent number of cells filled in the distinct rows.

But a is prefix to ab, so the output query $aab \cdot ab$ is calculated. After updating the answers for output queries $aab \cdot ab$ and $aab \cdot a$, still the row aab is consistent with more than one distinct rows. The algorithm calculates $aab \cdot b$, the row aab becomes consistent with only one distinct row a as presented in Table 7.14. The row aab is consistent with only one distinct row a. The distinct rows are equal to states of

Table 7.14: Adding the output for the output queries $aab \cdot ab$ and $aab \cdot b$

	a	**b**	**aa**	**ab**
ϵ	x	y	xx	xy
a	x	y	xx	xx
aa	x	x		xx
b	x	y	xx	xy
ab	x	y		xy
aaa		x		
aab	x	y		xx

the target model, the algorithm terminates the learning process and conjectures a Mealy machine model. The conjectured model is presented in Figure .

The algorithm calculates 11 output queries, whereas in the table the outputs for 22 output queries can be observed, where 11 output queries were inferred from the dictionaries.

7.4.4 Complexity

For the $L_{M\text{-}GS}$ algorithm, initially S contains ϵ, i.e. one element. Each time the observation table (S, E, T) is discovered to be not closed, one element is moved from $S \cdot I \backslash S$ to S. This can happen for at most $n-1$ times, hence, always $|S| \leq n$, where n is the number of states in the minimal conjecture. Thus, the total number of the observation table rows cannot exceed $(n + n \cdot |I|)$.

The algorithm begins with columns $E = I$ and $|E| = |I|$, each time after incrementing l, the total number of suffixes in E cannot exceed $|I|^l$, where l is the length of longest distinguishing sequences added to E and $|I|$ is the size of inputs. Putting all this together the maximum cardinality of $S' \times E$ is at most $((n + n \cdot |I|) \times |I|^l)$.

The worst case complexity of the $L_{M\text{-}GS}$ learning algorithm in terms of output queries is $O(n \cdot |I|^{l+1})$.

7.5 Discussion

The Mealy inference algorithm L_1 presented in Chapter 6 learns the unknown models in $O(|I|mn^2)$ output queries, where I is the set of inputs, n is the number of states and m is the length of longest suffix processed by the learning algorithm from a counterexample. However, this complexity formula does not take the equivalence oracle into account. The existence of an oracle is a strong assumption for software model inference (that oracle can always come up with a counterexample if the learned model is not correct). For software black box inference, the oracle is an implementation which constructs input strings from a uniform distribution over the set of inputs I and provides the resulting sequences in parallel to conjecture and target black box to search for the differences. The oracle might iterate

repeatedly with very long input sequences in the software black box implementation without finding a counterexample. Along the random exploration, the conformance testing methods can also be used to accommodate the deficiency of an oracle [Margaria 2004, Peled 1999], but such methods for instance the one from Vasilevskii and Chow [Vasilevskii 1973, Chow 1978] come at high exponential cost.

Domaratzki *et al.* [Domaratzki 2002] provided a lower bound on membership queries for the identification of exact language, they show that $(g-1)nlog_h(n)+O(n)$ bits are necessary and sufficient to specify a language accepted by automata, where n is the number of states, g is the number of inputs and h is the number of outputs.

7.6 Conclusion

The GoodSplit algorithm can learn i/o systems by taking the union or cross product of inputs set and outputs set as alphabet, but it significantly increases the size of alphabet. From the complexity discussion in Section 7.4.4, it can be observed that the size of alphabet is one of the key parameters to the time complexity of the algorithm. The objective is to learn the models with less number of output queries. The improved technique to calculate output queries makes the dictionaries more effective by enabling them to reply more output queries without consulting black box implementations.

We have proposed the Mealy inference algorithm $L_{M\text{-}GS}$ which can directly learn i/o systems using improved technique to calculate output queries. The algorithm keeps only valid one letter extensions of distinct states (valid access strings) in the observation table. On calculating an output query for any column of an access string it can be identified that it is valid or invalid. The algorithm avoids calculating output queries for invalid access strings by keeping only valid access strings in the observation table. At the time of adding new rows and columns, the algorithm updates their output queries which can be calculated from the dictionaries, this accelerates the process of identifying distinct states. The algorithm proposes a greedy choice which helps to identify the corresponding distinct row of a non distinct row in question by requiring less output queries. This algorithm does not require any counterexamples at all. Since searching for the counterexamples is not an easy task, the gain for $L_{M\text{-}GS}$ also includes the fact that it does not need counterexamples.

This algorithm is based on the result from Trakhtenbrot and Barzdin which indicate that a test set consisting of all strings of length at most about $log_g log_h(n)$ is sufficient to distinguish all distinct states in almost all complete finite state machines. However, this algorithm may not be appropriate for the models which require very long distinguishing sequences, combination locks are a well known example in this regard. For them, we require distinguishing sequences of much higher length, and for these models $L_{M\text{-}GS}$ would need to perform a lot of output queries.

Chapter 8

Tool and Case Studies

Contents

8.1 RALT **119**
8.1.1 Test Drivers 120
8.1.2 Learning Platform 120
8.1.3 Learner 120
8.2 Case Studies **122**
8.2.1 Random Machine Generator 122
8.2.2 Edinburgh Concurrency Workbench (CWB Examples) 122
8.2.3 HVAC controller 122
8.2.4 Coffee Machine 123
8.3 Conclusion **129**

In this chapter, we present the tool *Rich Automata Learning and Testing* (RALT) developed by Shabaz [Shahbaz 2008]. We have added the implementation of our software model inference algorithms in RALT. Current version is RALT 5.0. This chapter also includes the case studies that we have used for analysis and evaluation of our algorithms.

8.1 RALT

We have extended the RALT toolbox version 4.0 to evaluate our software model inference algorithms. This toolbox is developed with Java 2 Platform Standard Edition 5.0. It is developed for learning deterministic and non-deterministic observable black-box systems. These systems can be software components (COTS) or hardware devices such as UPnP, X10, DPWS or Bonjour devices. RALT can learn various models: DFA [Angluin 1987], Mealy [Shahbaz 2009], Parameterized finite state machines [Li 2006] and observable non-deterministic finite state machines [El-Fakih 2010].

The tool RALT interacts with a target implementation with the help of a test driver. After asking a certain number of tests, it conjectures a model from the answers calculated from the implementation. This model can be produced as ".jff" or ".dot" files. The general learning framework for RALT version 5.0 is presented in Figure 8.1.

Figure 8.1 depicts functioning of RALT in general and execution order of the various modules.

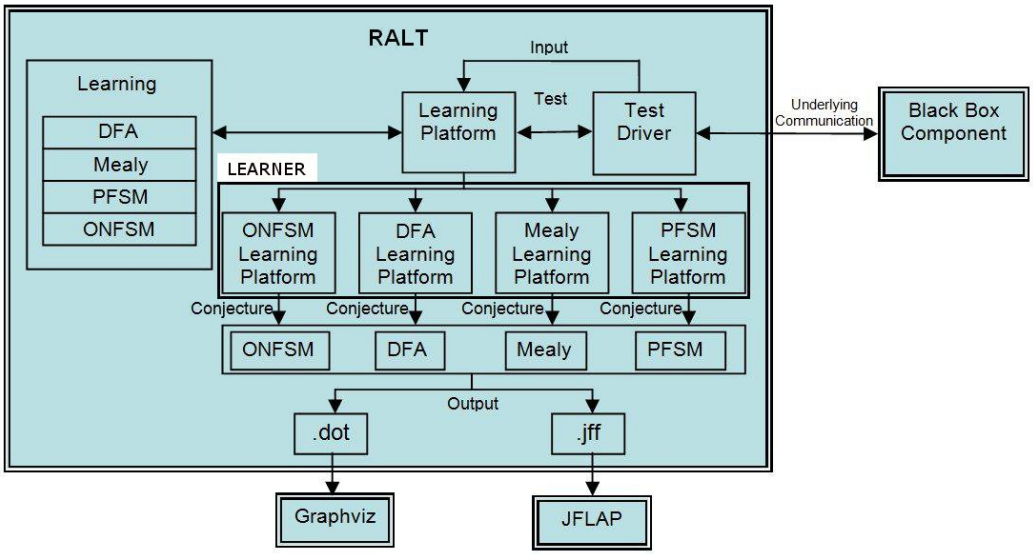

Figure 8.1: Rich Automata Learning and Testing Framework

8.1.1 Test Drivers

The tool RALT requires test drivers to enable a learning algorithm implementation to interact with a simulated or real life example. The test drivers are responsible for ensuring the communication between the learning algorithm and a target software implementation. Test drivers receive abstract output queries from the learning algorithm and translate them to concrete inputs for the target software implementation. They are also responsible for translating the concrete outputs to abstract outputs that are understandable for the learning algorithm. Most of the test drivers are system specific, however, generic test drivers can be written for applications developed respecting some standard ontologies.

8.1.2 Learning Platform

This is the main module through which all of the modules communicate with each other. This module contains the abstract test drivers and methods that help the learning algorithm implementations to access the internal data structures implemented in the tool.

8.1.3 Learner

All of the learning algorithms use this module to calculate output queries by interacting with the test drivers. Our tool can learn DFA, Mealy machines, parameterized finite state machines (PFSM) and observable non-deterministic finite state machines (ONFSM). We have added the implementation of the L_1 (the Algorithm 8) and L_N algorithms [El-Fakih 2010] (the L_N algorithm learns the ONFSM models). The im-

plementation of the $L_{M\text{-}GS}$ (the Algorithm 9) is in process. The detailed Learner module is presented in Figure 8.2.

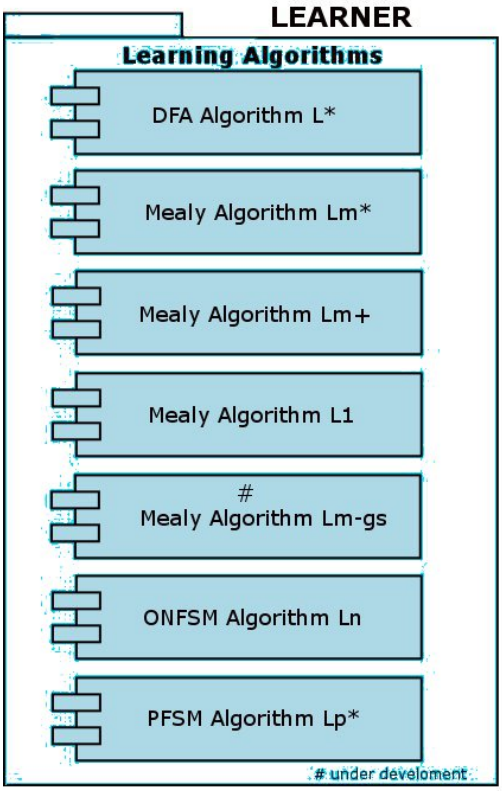

Figure 8.2: Detailed Learner Framework

For all of the learning algorithms presented in Figure 8.2, the following counterexample search and counterexample processing algorithms are implemented.

8.1.3.1 Counterexample search

The L^* algorithm and its variants require to validate a learned model from an oracle. To verify the learned models of software implementations we have implemented the random sampling oracle proposed by Angluin (provided in Section 5.1.1).

8.1.3.2 Counterexample Processing

RALT contains five counterexample processing methods. The counterexample processing method from Angluin and Shahbaz were already implemented in RALT version 4.0. We have implemented the counterexample processing method from Maler and Pnueli (provided in Section 5.2.3), Rivest and Schapire (provided in Section 5.2.2), and the Suffix1by1 counterexample processing method (provided in Section 5.3).

8.2 Case Studies

The efficiency of a learning algorithm can change with the nature of considered example to be learned. An algorithm can be efficient for one example, whereas it can be less efficient for the other. In order to analyze an algorithm, we need to study different aspects for a variety of examples. Selecting a set of examples which can exhibit the advantages and disadvantages of a learning algorithm is a complex task. This problem can be addressed by using a random machines generator which provides liberty to generate a variety of machines ranging from simple to complex.

8.2.1 Random Machine Generator

For finite state machines, we have implemented a generator in RALT, which constructs random finite state machines with a given number of inputs, outputs and states. The method for generating these machines is straightforward: we define the states of the machine and choose an initial state. Then from the initial state, for each input $i \in I$, we select a random output from the given set of outputs $o \in O$ and select a target state randomly from the given set of states $q \in Q$, we continue this process until we are done for all the states Q. The constructed FSM is finally minimized (thus it does not have equivalent states) and serves as a target black box example for a learning algorithm. We may want to learn a FSM with a given state size, so we iterate generation while constructing the minimized machines to fit the size, until we get a machine of the required size. With random counterexample searching method we can find the long and optimal counterexamples simultaneously, which can make the learning very erratic. To overcome the erratic behavior, we average on the output parameters to the learning algorithms for a given set of input parameters. This generator basically takes as input: the number of states n, the number of inputs $|I|$ the number of outputs $|O|$ and returns a FSM.

8.2.2 Edinburgh Concurrency Workbench (CWB Examples)

We have used this case study to calculate the practical complexity of proposed learning algorithms. We have analyzed our counterexample processing method by comparing it with rest of the counterexample processing methods. The experiments show the gain.

8.2.3 HVAC controller

We have used the HVAC controller to evaluate the L_1 algorithm. We have learned the controller with ${L_M}^*$ and L_1 algorithms. The experiments show that the L_1 algorithm requires much lesser output queries as compared to the L_1 algorithm.

8.2.4 Coffee Machine

Merten *et al.* [Merten 2011] provide the LearnLib tool[1] for inferring models of black box implementations. The LearnLib tool offers a tutorial for model inference of a coffee machine presented in Figure 8.3. In this section, we provide model inference of the coffee machine with the L_1 algorithm and compare with the best inference algorithm implemented by the LearnLib tool named as the complex setup algorithm. The complex setup algorithm comprises of the $L_M{}^*$ (direct Mealy adaptation of L^* presented in 4.3.1) algorithm and the counterexample processing method by Rivest and Schapire [Rivest 1993].

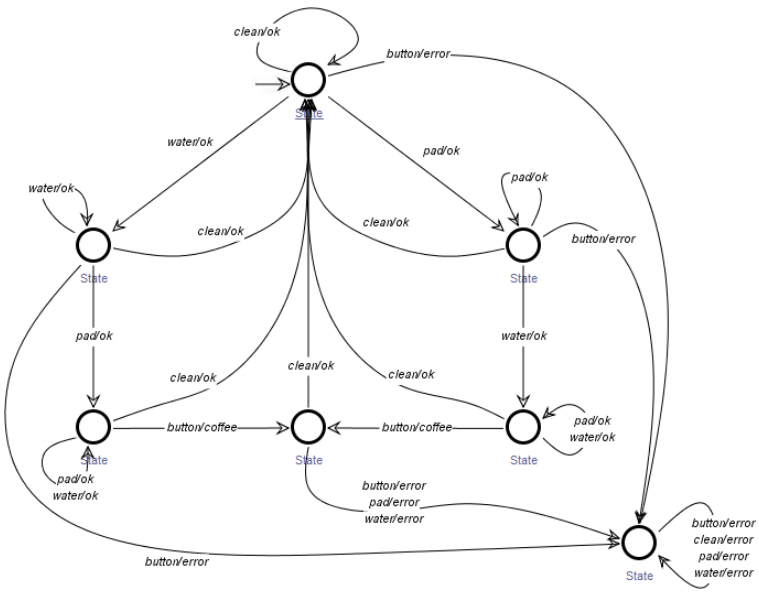

Figure 8.3: Coffee machine

8.2.4.1 Model Inference of the Coffee Machine with the Complex Setup Algorithm

The behavior of the coffee machine is modeled as a Mealy machine. We present the inference results for the complex setup algorithm of the Learnlib tool. The elements of the input set I for the coffee machine are *pad*, *water*, *button* and *clean*. The final observation table (S_M, E_M, T_M) after processing the counterexamples is presented in Table 8.1.

They asked 150 output queries (25 rows $\times$ 6 columns) to conjecture Mealy machine model of the coffee machine [Steffen 2011]. Additional $3 + 2 = 5$ output queries are used to process counterexamples. Thus, in total they asked 155 output queries. Figure 8.4 shows the model conjectured from Table 8.1.

[1] Online available to download from http://faelis.cs.uni-dortmund.de/learnlib.de/index.php

Table 8.1: The observation table for model inference with the complex structure algorithm

	water	pad	button	clean	pad· button	water· water· button
ϵ	ok	ok	error	ok	error	error
pad	ok	ok	error	ok	error	coffee
water	ok	ok	error	ok	coffee	error
button	error	error	error	error	error	error
water· pad	ok	ok	coffee	ok	coffee	coffee
water· pad· button	error	error	error	ok	error	error
clean	ok	ok	error	ok	error	error
water· clean	ok	ok	error	ok	error	error
pad· button	error	error	error	error	error	error
pad· clean	ok	ok	error	ok	error	error
button· water	error	error	error	error	error	error
button· clean	error	error	error	error	error	error
button· pad	error	error	error	error	error	error
button· button	error	error	error	error	error	error
water· water	ok	ok	error	ok	coffee	error
pad· water	ok	ok	coffee	ok	coffee	coffee
water· button	error	error	error	error	error	error
pad· pad	ok	ok	error	ok	error	coffee
water· pad· pad	ok	ok	coffee	ok	coffee	coffee
water· pad· water	ok	ok	coffee	ok	coffee	coffee
water· pad· clean	ok	ok	error	ok	error	error
water· pad· button· clean	ok	ok	error	ok	error	error
water· pad· button· button	error	error	error	error	error	error
water· pad· button· pad	error	error	error	error	error	error
water· pad· button· water	error	error	error	error	error	error

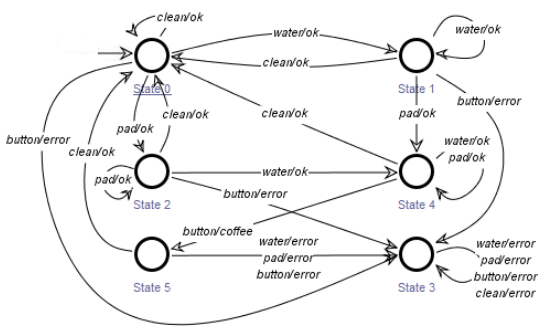

Figure 8.4: The Coffee Machine model conjectured from Table 8.1

8.2.4.2 Model Inference of the Coffee Machine with the L_1 Algorithm

We illustrate the L_1 algorithm by learning the Mealy model $\mathcal{M}_{coffee}$ of the coffee machine presented in Figure 8.3 and show how invalid access strings can be excluded from subsequent tests using the output for the last input of the access strings. The L_1 algorithm begins by initializing the observation table (S, E, L, T), the rows S are initialized to $\{\epsilon\}$, and columns are initially empty, i.e. $E = \emptyset$. The output queries *pad*, *water*, *button* and *clean* are asked to find the outputs for the set of input symbols $\{pad, water, button, clean\}$ labeling the rows $S \cdot I$.

Table 8.2: Initial Observation Table of the coffee machine for Mealy inference with the L_1 algorithm.

		E
		$\emptyset$
S	ϵ	
$S \cdot I \backslash S$	pad/ok $water/ok$ ~~$button/error$~~ $clean/ok$	

The coffee machine has an error state and all inputs with output error have transitions to this state. The access strings whose output for the last input symbol is *error* are marked unnecessary rows and are excluded from the subsequent tests. The initial observation table is shown in Table 8.2. This observation table has empty columns for all of the rows, thus, for every $s_2 \in S \cdot I$, there exists $s_1 \in S$ such that $s_2 \cong s_1$. Hence the observation table is closed. The table has only one row in S, trivially the observation table is compatible. The row whose access string is struck out is unnecessary row. On finding the observation table closed and compatible, the L_1 algorithm conjectures the model $Conj_{coffee_1} = (Q_{C\,coffee_1}, I, O, \delta_{C\,coffee_1}, \lambda_{C\,coffee_1}, q_{0C\,coffee_1})$ shown in Figure 8.5.

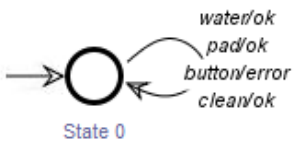

Figure 8.5: The Mealy machine conjecture $Conj_{coffee_1}$ from Table 8.2

To verify correctness of the one state machine conjecture, the L_1 algorithm asks an equivalence query to the oracle. The conjectured model $Conj_{coffee_1}$ is not correct, and there can be more than one counterexamples and oracle replies with one from them. Since

– $\lambda_{\mathcal{M}_{coffee}}(q_{0\mathcal{M}_{coffee}}, water \cdot pad \cdot button) = ok \cdot ok \cdot coffee$, but

– $\lambda_{C\,coffee_1}(q_{0\,coffee_1}, water \cdot pad \cdot button) = ok \cdot ok \cdot error.$

The oracle replies with a counterexample $water \cdot pad \cdot button$. The L_1 algorithm adds the smallest suffix *button* of the counterexample $water \cdot pad \cdot button$ to the columns E and completes the table. The observation table remains closed so the suffix $pad \cdot button$ is added, which makes the observation table unclosed as presented in Table 8.3a.

	$button$	$pad \cdot button$
ϵ	$error$	$ok \cdot error$
pad/ok	$error$	$ok \cdot error$
$\underline{water}/ok$	$error$	$ok \cdot coffee$
$clean/ok$	$error$	$ok \cdot error$

(a) The observation table after adding the suffix *button* and $pad \cdot button$.

	$button$	$pad \cdot button$
ϵ	$error$	$ok \cdot error$
$water/ok$	$error$	$ok \cdot coffee$
pad/ok	$error$	$ok \cdot error$
$clean/ok$	$error$	$ok \cdot error$
$water \cdot water/ok$	$error$	$ok \cdot coffee$
$\underline{water \cdot pad}/ok$	$coffee$	$ok \cdot coffee$
$\cancel{water \cdot button}/error$	$error$	
$water \cdot clean/ok$	$error$	$ok \cdot error$

(b) Make the observation table closed by moving the row *water* to S.

	$button$	$pad \cdot button$
ϵ	$error$	$ok \cdot error$
$water/ok$	$error$	$ok \cdot coffee$
$water \cdot pad/ok$	$coffee$	$ok \cdot coffee$
pad/ok	$error$	$ok \cdot error$
$clean/ok$	$error$	$ok \cdot error$
$water \cdot water/ok$	$error$	$ok \cdot coffee$
$water \cdot clean/ok$	$error$	$ok \cdot error$
$water \cdot pad \cdot water/ok$	$coffee$	$ok \cdot coffee$
$water \cdot pad \cdot pad/ok$	$coffee$	$ok \cdot coffee$
$\underline{water \cdot pad \cdot button}/coffee$	$error$	$error \cdot error$
$water \cdot pad \cdot clean/ok$	$error$	$ok \cdot error$

(c) Make the observation table closed by moving the row $water \cdot pad$ to S.

	$button$	$pad \cdot button$
ϵ	$error$	$ok \cdot error$
$water/ok$	$error$	$ok \cdot coffee$
$water \cdot pad/ok$	$coffee$	$ok \cdot coffee$
$water \cdot pad \cdot button/coffee$	$error$	$error \cdot error$
pad/ok	$error$	$ok \cdot error$
$clean/ok$	$error$	$ok \cdot error$
$water \cdot water/ok$	$error$	$ok \cdot coffee$
$water \cdot clean/ok$	$error$	$ok \cdot error$
$water \cdot pad \cdot water/ok$	$coffee$	$ok \cdot coffee$
$water \cdot pad \cdot pad/ok$	$coffee$	$ok \cdot coffee$
$water \cdot pad \cdot clean/ok$	$error$	$ok \cdot error$
$\cancel{water \cdot pad \cdot button \cdot water}/error$	$error$	
$\cancel{water \cdot pad \cdot button \cdot pad}/error$	$error$	
$\cancel{water \cdot pad \cdot button \cdot button}/error$	$error$	
$water \cdot pad \cdot button \cdot clean/ok$	$error$	$ok \cdot error$

(d) Make the observation table closed by moving the row $water \cdot pad \cdot button$ to S.

Table 8.3: We get the observation table in Table 8.3a after adding the suffixes *button* and $pad \cdot button$ to E. The underlined rows are the rows, which make the table not closed. The observation table in Table 8.3d is closed.

The row *water* makes the observation table not closed. It is moved to S and its one letter extensions $\{water \cdot water, water \cdot pad, water \cdot button, water \cdot clean\}$ are added to $S \cdot I$. To calculate the output for last input of access strings in $S \cdot I$, the algorithm does not require to execute the output queries separately. The output for

the last input of access strings can be calculated from the output query executed for any of the cells of that row. For instance, for row $water \cdot pad$ and column $button$, if the algorithm asks the output query $water \cdot pad \cdot button$, the answer calculated from the coffee machine is $ok \cdot ok \cdot coffee$. The output for the last input string of $water \cdot pad$ is calculated as ok and $coffee$ is recorded in the column $button$ of the observation table. After completing the table, it can be observed from Table 8.3b that the row $water \cdot pad$ makes the observation table not closed. Again the table is made closed by moving the row $water \cdot pad$ to S. This procedure is iterated until the observation table is closed as shown in Table 8.3d. Since the algorithm L_1 maintains the condition $\forall s_1, s_2 \in S$ always $s_1 \ncong s_2$, the observation table is always compatible. Hence L_1 conjectures the Mealy machine model $Conj_{coffee_2} = (Q_{C\,coffee_2}, I, O, \delta_{C\,coffee_2}, \lambda_{C\,coffee_2}, q_{0C\,coffee_2})$ shown in Figure 8.6.

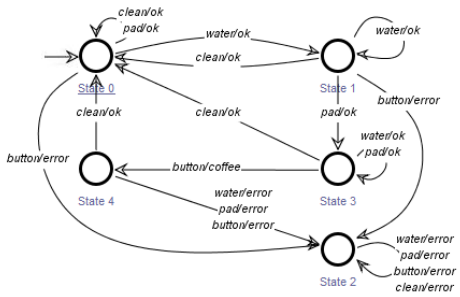

Figure 8.6: The Mealy machine conjecture $Conj_{coffee_2}$ from Table 8.3d

The L_1 algorithm asks an equivalence query to the oracle. Since

- $\lambda_{\mathcal{M}_{coffee}}(q_{0\mathcal{M}_{coffee}}, pad \cdot water \cdot button) = ok \cdot ok \cdot coffee$, but
- $\lambda_{C\,coffee_1}(q_{0\,coffee_1}, pad \cdot water \cdot button) = ok \cdot ok \cdot error$

The oracle replies with a counterexample $pad \cdot water \cdot button$. Since the smallest suffix $button$ is already member of E, the L_1 algorithm adds the suffix $water \cdot button$ to the columns E and completes the table. This suffix makes the observation table unclosed as presented in Table 8.4a.

The row pad as shown in Table 8.4a makes the observation table not closed. It is moved to S and its one letter extensions are added to $S \cdot I$. After completing the observation table, we get the Table 8.4b. It is closed and compatible observation table. The L_1 algorithm conjectures the Mealy machine model shown in Figure 8.4. The conjecture is a correct behavior of the coffee machine, thus, the algorithm terminates. The L_1 algorithm asks 4 output queries to construct Table 8.2, 8 output queries for Table 8.3a, 7 output queries for Table 8.3b, 8 output queries for Table 8.3c, 5 output queries for Table 8.3d , 12 output queries for Table 8.4a, and 10 output queries for Table 8.4b, Thus, in total it asks 54 output queries to learn the Mealy model of the coffee machine.

	$button$	$pad \cdot button$	$water \cdot button$
ϵ	$error$	$ok \cdot error$	$ok \cdot error$
$water/ok$	$error$	$ok \cdot coffee$	$ok \cdot error$
$water \cdot pad/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot button/coffee$	$error$	$error \cdot error$	$error \cdot error$
$\underline{pad}/ok$	$error$	$ok \cdot error$	$ok \cdot coffee$
$clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot water/ok$	$error$	$ok \cdot coffee$	$ok \cdot error$
$water \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot pad \cdot water/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot pad/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot pad \cdot button \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$

(a) The observation table after adding the suffix $water \cdot button$ to E.

	$button$	$pad \cdot button$	$water \cdot button$
ϵ	$error$	$ok \cdot error$	$ok \cdot error$
$water/ok$	$error$	$ok \cdot coffee$	$ok \cdot error$
$water \cdot pad/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot button/coffee$	$error$	$error \cdot error$	$error \cdot error$
pad/ok	$error$	$ok \cdot error$	$ok \cdot coffee$
$clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot water/ok$	$error$	$ok \cdot coffee$	$ok \cdot error$
$water \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot pad \cdot water/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot pad/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$water \cdot pad \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$water \cdot pad \cdot button \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$
$pad \cdot water/ok$	$coffee$	$ok \cdot coffee$	$ok \cdot coffee$
$pad \cdot pad/ok$	$error$	$ok \cdot error$	$ok \cdot coffee$
$\cancel{pad \cdot button}/error$	$error$		
$pad \cdot clean/ok$	$error$	$ok \cdot error$	$ok \cdot error$

(b) Make the observation table closed by moving the row pad to S.

Table 8.4: We get the observation table in Table 8.4a after adding the suffix $water \cdot button$ to E. The underlined rows are the rows, which make the table not closed. The observation table in Table 8.4b is closed.

8.3 Conclusion

We have presented the tool RALT 5.0 which is an extension to RALT 4.0. The extension contains the implementation of the algorithms proposed by us in the thesis. The main modules of the tool are presented to show the functioning of the tool. All of the learning algorithms are implemented in the module called learner which interacts via test drivers with software implementations to be learned.

We have used the case studies presented in this chapter for learning and testing. The detailed analysis of learning results is provided in the respective chapters. We have used the random machines mainly for evaluation of our algorithms. However, wherever other case studies are more appropriate, we have used them.

This tool can be applied for learning and testing of software systems. One needs to write a test driver which can cater with translating the RALT queries to concrete system inputs and system outputs to the messages that are understandable by the RALT algorithms.

Chapter 9

Conclusion and Perspectives

Contents

9.1 Summary **131**

9.2 Publications **132**

9.3 Perspectives **135**

9.3.1 Optimizing and extending the $L_{M\text{-}GS}$ Algorithm 135

9.3.2 Generic Test driver for RALT 136

9.3.3 Learning non-deterministic Machines 136

9.3.4 Addressing non-deterministic values 136

9.3.5 Improving Partial Models of Implementations 136

This chapter concludes the thesis. First section provides the summary for the contributions presented in the previous chapters. The second section presents all of the papers and a book chapter published during the thesis along the abstracts. Final section provides some future directions.

9.1 Summary

Inferring models of software systems is an active approach and it is gaining momentum for testing software systems. The main advantage for this technique is that it enables to test even a part of the whole functionality of a large SUT. The thesis addresses the problem of learning software models with reduced number of tests. The reduction has been done by proposing various techniques which can be summarized as below.

- We studied the various counterexample processing methods for the L^* algorithm. Most of them are unable to reduce the effect of long counterexamples provided by random sampling oracle. We propose a new counterexample processing algorithm named as the Suffix1by1 algorithm. The Suffix1by1 algorithm reduces the random sampling oracle effect and results in reduced number of output queries. This gain is confirmed by experimenting with random examples and CWB workbench.

- Mealy models are more appropriate to learn the i/o systems. We have proposed the L_1 Mealy inference algorithm. This algorithm has a significant gain

over the other Mealy adaptations of L^* when the size of the input set is large. The gain with this algorithm also comes from the fact that all of the inputs are not valid for every state and the algorithm avoids asking output queries for the states reached with the transitions of invalid inputs. This algorithm is tested on randomly generated machines and experiments confirm the gain.

– Random sampling is used to find the counterexamples while learning the software black box implementations with the L^* algorithm. One may require many resets before finding a counterexample. We propose the GoodSplit algorithm which learns Mealy models of software implementations without requiring the counterexamples. This algorithm greedily selects output queries to calculate them from a system under inference and keeps some of the entries in the observation table sparse which results in reduced number of output queries.

– To validate all of our proposed algorithms we have implemented them in the tool RALT. This enables us to test our algorithms on various case studies to calculate the practical complexity of the algorithms.

9.2 Publications

In the following, we present the list of papers and a book chapter published in the course of the thesis along their abstracts.

1. **Muhammad Naeem Irfan**, Roland Groz, Catherine Oriat. Improving Model Inference of Black Box Components having Large Input Test Set. ICGI 2012 [Irfan 2012a].

   The deterministic finite automata learning algorithm L^* has been extended to learn Mealy machine models which are more succinct for input/output based systems. We propose an optimized learning algorithm L_1 to infer Mealy models of software black box components. The L_1 algorithm uses a modified observation table and adds only valid access strings to the rows of the observation table. The columns of the observation table are initially kept empty and their size augments only to process the counterexamples. The proposed improvements reduce the worst case time complexity. The L_1 algorithm is compared with the existing Mealy inference algorithms and the experiments conducted on a comprehensive set of examples confirm the gain.

2. Roland Groz, **Muhammad Naeem Irfan**, Catherine Oriat. Algorithmic Improvements on Regular Inference of Software Models and Perspectives for Security Testing. ISoLA 2012 [Groz 2012].

   Among the various techniques for mining models from software, regular inference of black-box systems has been a central technique in the last decade. In

this paper, we present various directions we have investigated for improving the efficiency of algorithms based on L^* in a software testing context where interactions with systems entail large and complex input domains. In particular we consider algorithmic optimizations for large input sets, for parameterized inputs, for processing counterexamples. We also present our current directions motivated by application to security testing: focusing on specific sequences, identifying randomly generated values, combining with other adaptive techniques.

3. **Muhammad Naeem Irfan**, Catherine Oriat, Roland Groz. Model Inference and testing, Book chapter "Advances in Computers, Volume 89". Editor: Atif Memon [Irfan 2012b]

   For software systems, models can be learned from behavioral traces, available specifications, knowledge of experts and other such sources. Software models help to steer testing and model checking of software systems. The model inference techniques extract structural and design information of a software system and present it as a formal model. This chapter briefly discusses the passive model inference and goes on to present the active model inference of software systems using the algorithm L^*. This algorithm switches between model inference and testing phases. In model inference phase it asks membership queries and records answers in a table to conjecture a model of a software system under inference. In testing phase it compares a conjectured model with the system under inference. If a test for a conjectured model fails, a counterexample is provided which helps to improve the conjectured model. Different counterexample processing methods are presented and analyzed to identify an efficient counterexample processing method. A counterexample processing method is said to be efficient if it helps to infer a model with fewer membership queries. An improved version of L^* is provided which avoids asking queries for some rows and columns of the table which helps to learn models with fewer queries.

4. **Muhammad Naeem Irfan**. State Machine Inference in Testing Context with Long Counterexamples. In Third International Conference on Software Testing, Verification and Validation, ICST 2010, Pages 508-511, Paris, France, 2010 (PhD symposium) [Irfan 2010a].

   We are working on the techniques which iteratively learn the formal models from black box implementations by testing. The novelty of the approach addressed here is our processing of the long counterexamples. There is a possibility that the counterexamples generated by a counterexample generator include needless sub sequences. We address the techniques which are developed to avoid the impact of such unwanted sequences on the learning process. The gain of the proposed algorithm is confirmed by considering a comprehensive set of experiments on the finite sate machines.

5. Khaled El-Fakih, Roland Groz, **Muhammad Naeem Irfan**, Muzammil Shahbaz. Learning Finite State Models of Observable Nondeterministic Systems in a Testing Context. In 22nd IFIP International Conference on Testing Software and Systems, Pages 97-102, Natal, Brazil, 2010 (short paper) [El-Fakih 2010].

   Learning models from test observations can be adapted to the case when the system provides nondeterministic answers. In this paper we propose an algorithm for inferring observable nondeterministic finite state machines (ONFSMs). The algorithm is based on Angluin L^* algorithm for learning DFAs. We define rules for constructing and updating learning queries taking into account the properties of ONFSMs. Application examples, complexity analysis and an experimental evaluation of the proposed algorithm are provided.

6. **Muhammad Naeem Irfan**, Roland Groz, Catherine Oriat. Optimizing Angluin Algorithm L^* by Minimising the Number of Membership Queries to Process Counterexamples. In Zulu Workshop, Valencia, September 2010 [Irfan 2010b].

   Angluin algorithm L^* is a well known approach for learning unknown models as minimal deterministic finite automata (DFA) in polynomial time. It uses concept of oracle which presumably knows the target model and comes up with a counterexample, if the conjectured model is not correct. This algorithm can be used to infer the models of software artefacts and a cheap oracle for such components uses random strings (built from inputs) to verify the inferred models. In such cases and others the provided counterexamples are rarely minimal. The length of the counterexample is an important parameter to the complexity of the algorithm. The proposed technique tends to reduce the impact of non minimal counterexamples. The gain of the proposed algorithm is confirmed by considering a set of experiments on DFA learning.

7. **Muhammad Naeem Irfan**, Catherine Oriat, Roland Groz. Angluin Style Finite State Machine Inference with Non-optimal Counterexamples. In 1st International Workshop on Model Inference In Testing, MIIT 2010, July 2010 [Irfan 2010c].

   Angluin's algorithm is a well known approach for learning black boxes as minimal deterministic finite automata in polynomial time. In order to infer finite state machines instead of automata, different variants of this algorithm have been proposed. These algorithms rely on two types of queries which can be asked to an oracle: output and equivalence queries. If the black box is not equivalent to the learned model, the oracle replies with a counterexample. The complexity of these algorithms depends on the length of the counterexamples

provided by the oracle. The aim of this paper is to compare the average practical complexity of three of these algorithms in the case of poor oracles, in particular when the counterexamples are constructed with random walks.

8. **Muhammad Naeem Irfan**. Heuristics for Improving Model Learning Based Software Testing. In Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART 2009), Windsor, UK, September 2009 [Irfan 2009].

   In order to reduce the cost and provide rapid development, most of the modern and complex systems are built integrating prefabricated third party components COTS. We have been investigating techniques to build formal models for black box components. The integration testing framework developed by our team leaves several open strategies; we will be investigating variations of these open strategies to enhance applicability. We are investigating the heuristics to improve the existing methodologies for learning black boxes and integration testing. We are addressing the counter-example part of the learning algorithm for improvements and are examining different techniques to identify the counterexamples in a more efficient way.

9.3 Perspectives

We have been working on finding new model inference algorithms and improving the existing algorithms to make them more efficient and effective in testing context. Testing is an approximate and time consuming task. We have optimized the algorithms so that they can be used widely in the testing community. The work can be extended in many directions. However, we plan to consider some possible extensions of our contributions. We are already working on few of them and we plan to work on others in near future. In the following, we present these directions briefly.

9.3.1 Optimizing and extending the $L_{M\text{-}GS}$ Algorithm

Searching for the counterexamples to improve the inferred software models is not an easy task. An equivalence oracle may fail a number of times before finding a counterexample, which results in an increased cost for model inference. The $L_{M\text{-}GS}$ algorithm is an important contribution in this regard. We are working on optimizations for this algorithm. Currently there is a lot of room for the improvement of this algorithm. While asking the output queries, this algorithm can be adapted in different ways. The realization and adaption of a better method will result in a gain. We plan to extend this algorithm for non-deterministic and parameterized machines.

9.3.2 Generic Test driver for RALT

To learn model of a software implementation our tool RALT requires a test driver. This test driver is a target application specific. Writing test drivers is a time consuming task. We are working on a generic test driver which can at least learn a number of web applications with few customizations. We are implementing a generic test driver which has some variables that can be set according to a target application.

9.3.3 Learning non-deterministic Machines

In [El-Fakih 2010], we propose an algorithm L_N for inferring observable non deterministic finite state machines (ONFSMs). This algorithm is an extension to the L^* algorithm. The rules for constructing and updating learning queries are defined by considering the properties of the observable non-deterministic finite state machines (ONFSMs). We plan to extend this work for non-deterministic machines. The non-deterministic machines produce multiple outputs if an input is provided repeatedly. The algorithm L_N learns a sub class of these machines by using the so-called "all-weather conditions" [Milner 1982]. The algorithm records all of the outputs produced for an input into the corresponding cell of the observation table. This algorithm assumes that with a pair of an input and output the machine under inference has a transition to only one state, which is not the case for non-deterministic machines. Bollig *et al.* propose the NL^* algorithm for learning residual finite state machine automata (RFSA). RFSA is a sub-class of NFA and it shares important properties with DFA. We plan to devise a method that benefits from the qualities of both L_N and NL^*.

9.3.4 Addressing non-deterministic values

To infer the non-deterministic machines every time a transition is triggered from a state, it can provide new values. It is important to identify that all such outputs from the black box actually correspond to the same transition of the state. We are working on methods that can recognize such values and record them in the observation table with reference to the same transition. This will help to infer models that include this kind of data non-determinism.

9.3.5 Improving Partial Models of Implementations

Model inference can be used when models are not available at all. However, it can also be used to improve existing partial models. Improving existing models is akin to processing the counterexamples for the learned models. The difference is that by processing the counterexamples the software models (conjectured models) improve monotonically, whereas for improving the partial models one may also need to remove some states or transitions. This is because the initial software model may have some behavior that has been changed or removed in the revisions afterwards. An observation table can be constructed from a partial model provided along the

system under inference. But the problem is that the observations in the observation table might not be consistent with the system under inference. Although each observation can be verified from the system under inference but at the cost of losing the advantage of available partial model (if each observation is to be cross checked then the observation table can be directly filled by asking the output queries to the system under inference).

The inconsistency of the observation table and system under inference can be found and addressed on the counterexample search pattern. There are many way outs to optimize searching and addressing such inconsistencies [Chaki 2008, Cho 2010]. Cho *et al.* [Cho 2010] predict the answers to the output queries. They try to make accurate prediction of responses. However, the erroneous predictions are detected using sampling queries and fixed by backtracking to the first mistake made by the predictor.

Bibliography

[Aarts 2010a] F. Aarts and F. W. Vaandrager. *Learning I/O Automata*. In CONCUR 2010 - Concurrency Theory, 21th International Conference, volume 6269, pages 71–85, 2010. (Cited on page 25.)

[Aarts 2010b] Fides Aarts, Bengt Jonsson and Johan Uijen. *Generating Models of Infinite-State Communication Protocols Using Regular Inference with Abstraction*. In ICTSS, pages 188–204, 2010. (Cited on pages 12, 26 and 34.)

[Abela 2004] J. Abela, F. Coste and S. Spina. *Mutually Compatible and Incompatible Merges for the Search of the Smallest Consistent DFA*. In Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, volume 3264, pages 28–39, 2004. (Cited on page 15.)

[Ammons 2002] Glenn Ammons, Rastislav Bodík and James R. Larus. *Mining specifications*. In POPL, pages 4–16, 2002. (Cited on page 30.)

[Angluin 1981] Dana Angluin. *A Note on the Number of Queries Needed to Identify Regular Languages*. Information and Control, vol. 51, no. 1, pages 76–87, 1981. (Cited on page 43.)

[Angluin 1987] Dana Angluin. *Learning Regular Sets from Queries and Counterexamples*. Inf. Comput., vol. 75, no. 2, pages 87–106, 1987. (Cited on pages 3, 12, 18, 19, 20, 25, 28, 38, 39, 43, 47, 57, 58, 59, 60, 61, 97, 102, 104 and 119.)

[Angluin 1990] Dana Angluin. *Negative Results for Equivalence Queries*. Mach. Learn., vol. 5, pages 121–150, July 1990. (Cited on page 25.)

[Apfelbaum 1997] Larry Apfelbaum and John Doyle. *Model Based Testing*. In Software Quality Week Conference, pages 296–300, 1997. (Cited on page 2.)

[Armstrong 2007] Joe Armstrong. Programming erlang: Software for a concurrent world. Pragmatic Bookshelf, July 2007. (Cited on page 35.)

[Aspectwerkz 2007] Aspectwerkz, 2007. Available from http://aspectwerkz.codehaus.org/. (Cited on page 15.)

[Balcázar 1997] José L. Balcázar, Josep Díaz and Ricard Gavaldà. *Algorithms for Learning Finite Automata from Queries: A Unified View*. In Advances in Algorithms, Languages, and Complexity, pages 53–72, 1997. (Cited on pages 3, 23, 24 and 64.)

[Balle 2010] Borja Balle. *Implementing Kearns-Vazirani Algorithm for Learning DFA Only with Membership Queries*. ZULU workshop organised during ICGI, 2010. (Cited on pages 12, 19, 24, 29, 57, 60 and 72.)

[Berg 2005a] 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. (Cited on page 26.)

[Berg 2005b] Therese Berg, Bengt Jonsson, Martin Leucker and Mayank Saksena. *Insights to Angluin's Learning.* Electr. Notes Theor. Comput. Sci., vol. 118, pages 3–18, 2005. (Cited on pages 28, 76 and 97.)

[Berg 2005c] Therese Berg and Harald Raffelt. *Model Checking.* In Model-Based Testing of Reactive Systems, volume 3472 of *LNCS*, pages 557–603. Springer, 2005. (Cited on page 29.)

[Bertolino 2007] Antonia Bertolino. *Software Testing Research: Achievements, Challenges, Dreams.* In FOSE '07: 2007 Future of Software Engineering, pages 85–103. IEEE Computer Society, 2007. (Cited on pages 2 and 4.)

[Biermann 1972] A. W. Biermann and J. A. Feldman. *On the synthesis of finite-state machines from samples of their behaviour.* IEEE Trans Computers, vol. 21, pages 591–597, 1972. (Cited on pages 3, 13, 14 and 15.)

[Birkendorf 1998] Andreas Birkendorf, Andreas Böker and Hans Ulrich Simon. *Learning deterministic finite automata from smallest counterexamples.* In Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms, SODA '98, pages 599–608, Philadelphia, PA, USA, 1998. Society for Industrial and Applied Mathematics. (Cited on pages 12 and 25.)

[Bogdanov 2009] Kirill Bogdanov and Neil Walkinshaw. *Computing the Structural Difference between State-Based Models.* In WCRE, pages 177–186, 2009. (Cited on page 31.)

[Bohlin 2009] Therese Bohlin. *Regular Inference for Communication Protocol Entities.* PhD thesis, Uppsala UniversityUppsala University, Division of Computer Systems, Computer Systems, 2009. (Cited on page 26.)

[Bohlin 2010] Therese Bohlin, Bengt Jonsson and Siavash Soleimanifard. *Inferring Compact Models of Communication Protocol Entities.* In ISoLA (1), pages 658–672, 2010. (Cited on page 35.)

[Bollig 2008] Benedikt Bollig, Peter Habermehl, Carsten Kern and Martin Leucker. *Angluin-Style Learning of NFA.* Research Report LSV-08-28, Laboratoire Spécification et Vérification, ENS Cachan, France, October 2008. 30 pages. (Cited on pages 27 and 76.)

[Bollig 2009] Benedikt Bollig, Peter Habermehl, Carsten Kern and Martin Leucker. *Angluin-Style Learning of NFA.* In IJCAI, pages 1004–1009, 2009. (Cited on pages 27, 28 and 97.)

[Broy 2005] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker and Alexander Pretschner, editeurs. Model-based testing of reactive systems, advanced lectures [the volume is the outcome of a research seminar that was held in schloss dagstuhl in january 2004], volume 3472 of *Lecture Notes in Computer Science*. Springer, 2005. (Cited on page 2.)

[Bshouty 1994] Nader H. Bshouty, Richard Cleve, Sampath Kannan and Christino Tamon. *Oracles and queries that are sufficient for exact learning (extended abstract)*. In Proceedings of the seventh annual conference on Computational learning theory, COLT '94, pages 130–139, New York, NY, USA, 1994. ACM. (Cited on page 25.)

[Bué 2010] Pierre-Christophe Bué, Frédéric Dadeau and Pierre-Cyrille Héam. *Model-Based Testing Using Symbolic Animation and Machine Learning*. In ICST Workshops, pages 355–360, 2010. (Cited on page 29.)

[Bugalho 2005] M. M. F. Bugalho and A. L. Oliveira. *Inference of regular languages using state merging algorithms with search*. Pattern Recognition, vol. 38, pages 1457–1467, 2005. http://www.odysci.com/article/1010112991693583. (Cited on page 15.)

[Chaki 2007] Sagar Chaki and Ofer Strichman. *Optimized L*-Based Assume-Guarantee Reasoning*. In TACAS, pages 276–291, 2007. (Cited on page 24.)

[Chaki 2008] Sagar Chaki and Ofer Strichman. *Three optimizations for Assume-Guarantee reasoning with L^**. Formal Methods in System Design, vol. 32, no. 3, pages 267–284, 2008. (Cited on pages 24 and 137.)

[Challenge a] Stamina Challenge. Available from http://stamina.chefbe.net/home. (Cited on page 16.)

[Challenge b] ZULU Challenge. Available from http://labh-curien.univ-st-etienne.fr/zulu/index.php. (Cited on page 28.)

[Cho 2010] Chia Yuan Cho, Domagoj Babic, Eui Chul Richard Shin and Dawn Song. *Inference and analysis of formal models of botnet command and control protocols*. In ACM Conference on Computer and Communications Security, pages 426–439, 2010. (Cited on pages 12, 26, 35, 57, 103 and 137.)

[Chow 1978] T. S. Chow. *Testing Software Design Modeled by Finite-State Machines*. IEEE Trans. Softw. Eng., vol. 4, no. 3, pages 178–187, May 1978. (Cited on pages 59, 80 and 118.)

[Clarke 2003] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu and Helmut Veith. *Counterexample-guided abstraction refinement for symbolic model checking*. J. ACM, vol. 50, no. 5, pages 752–794, 2003. (Cited on page 34.)

[Combe 2010] David Combe, Myrtille Ponge, Colin de la Higuera and Jean-Christophe Janodet. *Zulu: An interactive learning competition.* ZULU workshop organised during ICGI, 2010. (Cited on pages 20, 22, 24, 25, 28, 60, 104 and 105.)

[Cook 1995] Jonathan E. Cook and Alexander L. Wolf. *Automating Process Discovery Through Event-Data Analysis.* In ICSE, pages 73–82, 1995. (Cited on page 14.)

[Cook 1996] Jonathan E. Cook, Artur Klauser, Alexander L. Wolf and Benjamin G. Zorn. *Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases.* In SIGMOD Conference, pages 377–388, 1996. (Cited on page 14.)

[Cook 1998] Jonathan E. Cook and Alexander L. Wolf. *Discovering Models of Software Processes from Event-Based Data.* ACM Trans. Softw. Eng. Methodol., vol. 7, no. 3, pages 215–249, 1998. (Cited on pages 13, 14, 15 and 30.)

[Dalal 1999] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton and B.M. Horowitz. *Model-based testing in practice.* In Software Engineering, 1999. Proceedings of the 1999 International Conference on, pages 285 –294, may 1999. (Cited on page 2.)

[Dallmeier 2010] V. Dallmeier, N. Knopp, C. Mallon, S. Hack and A. Zeller. *Generating test cases for specification mining.* In Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, pages 85–96, 2010. (Cited on page 30.)

[de la Higuera 2005] Colin de la Higuera. *A bibliographical study of grammatical inference.* Pattern Recognition, vol. 38, no. 9, pages 1332–1348, 2005. (Cited on page 28.)

[de la Higuera 2010] Colin de la Higuera. Grammatical inference: Learning automata and grammars. Cambridge University Press, New York, NY, USA, 2010. (Cited on pages 3, 12, 21 and 28.)

[Denis 2002] François Denis, Aurélien Lemay and Alain Terlutte. *Residual Finite State Automata.* Fundam. Inform., vol. 51, no. 4, pages 339–368, 2002. (Cited on page 27.)

[Detlefs 2005] David Detlefs, Greg Nelson and James B. Saxe. *Simplify: a theorem prover for program checking.* J. ACM, vol. 52, no. 3, pages 365–473, 2005. (Cited on page 15.)

[Domaratzki 2002] Michael Domaratzki, Derek Kisman and Jeffrey Shallit. *On the Number of Distinct Languages Accepted by Finite Automata with n States.* Journal of Automata, Languages and Combinatorics, vol. 7, no. 4, pages 469–486, 2002. (Cited on pages 41 and 118.)

[Dorofeeva 2005] Rita Dorofeeva, Nina Yevtushenko, Khaled El-Fakih and Ana R. Cavalli. *Experimental Evaluation of FSM-Based Testing Methods.* In SEFM, pages 23–32, 2005. (Cited on page 28.)

[Dupont 1994] P. Dupont, L. Miclet and E. Vidal. *What is the search space of the regular inference.* In Lecture N. in Artificial Intelligence, editeur, ICGI'94 - Lectures Notes in Computer Science, volume 862 - Grammatical Inference and Applications, pages 25–37, Heidelberg, 1994. (Cited on page 13.)

[Eisenstat 2010] Sarah Eisenstat and Dana Angluin. *Learning Random DFAs with Membership Queries: the GoodSplit Algorithm.* ZULU workshop organised during ICGI, 2010. (Cited on pages 12, 22, 24, 25, 29, 101, 102 and 104.)

[El-Fakih 2010] Khaled El-Fakih, Roland Groz, Muhammad Naeem Irfan and Muzammil Shahbaz. *Learning Finite State Models of Observable Nondeterministic Systems in a Testing Context.* In 22nd IFIP International Conference on Testing Software and Systems, pages 97–102, Natal, Brazil, 2010. (Cited on pages 119, 120, 134 and 136.)

[Elkind 2006] Edith Elkind, Blaise Genest, Doron Peled and Hongyang Qu. *Grey-Box Checking.* In FORTE, pages 420–435, 2006. (Cited on page 29.)

[Ernst 2001] M. D. Ernst, J. Cockrell, W. G. Griswold and D. Notkin. *Dynamically Discovering Likely Program Invariants to Support Program Evolution.* IEEE Transactions on Software Engineering, vol. 27, no. 2, pages 99–123, 2001. A previous version appeared in ICSE99, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, USA, 1999. (Cited on pages 14, 15 and 30.)

[Fujiwara 1991] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou and Abderrazak Ghedamsi. *Test Selection Based on Finite State Models.* IEEE Trans. Software Eng., vol. 17, no. 6, pages 591–603, 1991. (Cited on page 34.)

[Garcia 2010] Pedro Garcia, Manuel Vazquez de Parga and Damian Lopez. *Some ideas on active learning using membership queries.* ZULU workshop organised during ICGI'10, 2010. (Cited on pages 12, 20 and 24.)

[Gheorghiu 2007] Mihaela Gheorghiu, Dimitra Giannakopoulou and Corina S. Păsăreanu. *Refining interface alphabets for compositional verification.* In Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems, TACAS'07, pages 292–307, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 24.)

[Gold 1967] E. M. Gold. *Language Identification in the Limit.* Information and Computation, vol. 10, pages 447–474, 1967. (Cited on pages 3 and 30.)

[Gold 1972] E. Gold. *System identification via state characterization.* Automatica, vol. 8, pages 621–636, 1972. (Cited on page 3.)

[Groce 2006] A. Groce, D. Peled and M. Yannakakis. *Adaptive Model Checking.* Bulletin of the IGPL, vol. 14, no. 5, pages 729–744, 2006. (Cited on pages 50 and 107.)

[Groz 2008] 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. (Cited on page 31.)

[Groz 2012] Roland Groz, Muhammad-Naeem Irfan and Catherine Oriat. *Algorithmic Improvements on Regular Inference of Software Models and Perspectives for Security Testing.* In ISoLA (1), pages 444–457, 2012. (Cited on page 132.)

[Harrison 1978] M. A. Harrison. Introduction to formal language theory. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition, 1978. (Cited on page 29.)

[Heule 2010] Marijn Heule and Sicco Verwer. *Exact DFA Identification Using SAT Solvers.* In ICGI, pages 66–79, 2010. (Cited on page 15.)

[Hierons 2008] Robert M. Hierons, Jonathan P. Bowen and Mark Harman, editeurs. Formal methods and testing, an outcome of the fortest network, revised selected papers, volume 4949 of *Lecture Notes in Computer Science.* Springer, 2008. (Cited on page 2.)

[Howar 2010] Falk Howar, Bernhard Steffen and Maik Merten. *From ZULU to RERS - Lessons Learned in the ZULU Challenge.* In ISoLA (1), pages 687–704, 2010. (Cited on pages 12, 19, 24, 29, 57, 59, 72, 102, 103 and 104.)

[Hungar 2003a] Hardi Hungar, Tiziana Margaria and Bernhard Steffen. *Test-Based Model Generation For Legacy Systems.* In ITC, pages 971–980, 2003. (Cited on page 34.)

[Hungar 2003b] Hardi Hungar, Oliver Niese and Bernhard Steffen. *Domain-Specific Optimization in Automata Learning.* In CAV, pages 315–327, 2003. (Cited on pages 3, 23, 26, 37, 47, 48, 49, 50 and 107.)

[Ibarra 1991] Oscar H. Ibarra and Tao Jiang. *Learning regular languages from counterexamples.* J. Comput. Syst. Sci., vol. 43, pages 299–316, October 1991. (Cited on pages 12 and 25.)

[Irfan 2009] Muhammad Naeem Irfan. *Heuristics for Improving Model Learning Based Software Testing.* In Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART 2009), Windsor, UK, September 2009. (Cited on page 135.)

[Irfan 2010a] Muhammad Naeem Irfan. *State Machine Inference in Testing Context with Long Counterexamples.* In Third International Conference on Software Testing, Verification and Validation, ICST 2010, pages 508–511, Paris, France, 2010. (Cited on pages 57 and 133.)

[Irfan 2010b] Muhammad Naeem Irfan, Roland Groz and Catherine Oriat. *Optimising Angluin Algorithm L* by Minimising the Number of Membership Queries to Process Counterexamples.* In Zulu Workshop, Valencia, September 2010. (Cited on page 134.)

[Irfan 2010c] Muhammad Naeem Irfan, Catherine Oriat and Roland Groz. *Angluin style finite state machine inference with non-optimal counterexamples.* In Proceedings of the First International Workshop on Model Inference In Testing, MIIT '10, pages 11–19. ACM, 2010. (Cited on pages 37, 57, 58, 72, 85, 104 and 134.)

[Irfan 2012a] Muhammad Naeem Irfan, Roland Groz and Catherine Oriat. *Improving Model Inference of Black Box Components having Large Input Test Set.* In Proceedings of the 11th International Conference on Grammatical Inference, ICGI 2012, pages 133–138, September 2012. (Cited on page 132.)

[Irfan 2012b] Muhammad Naeem Irfan, Catherine Oriat and Roland Groz. *Model Inference and Testing.* volume 89, pages 121–182, 2012. (Cited on page 133.)

[JabRef] JabRef. reference manager available from http://jabref.sourceforge.net/. (Cited on page 15.)

[jEdit] jEdit. jEdit - Programmer's Text Editor availabe on http://www.jedit.org/. (Cited on page 15.)

[Jelinek 1997] Frederick Jelinek. Statistical methods for speech recognition. MIT Press, Cambridge, MA, USA, 1997. (Cited on page 29.)

[Kearns 1994] M.J. Kearns and U.V. Vazirani. An introduction to computational learning theory. MIT Press, 1994. (Cited on pages 3 and 23.)

[Keidar 2003] S. Keidar and Y. Rodeh. *Searching for Counter-Examples Adaptively.* In 6th International Workshop on Formal Methods, IWFM 2003, 2003. (Cited on page 19.)

[Kicillof 2007] Nicolas Kicillof, Wolfgang Grieskamp, Nikolai Tillmann and Víctor A. Braberman. *Achieving both model and code coverage with automated gray-box testing.* In A-MOST, pages 1–11, 2007. (Cited on page 2.)

[Korshunov 1967] A.D. Korshunov. *The degree of distinguishability of automata.* In Diskret. Analiz., pages 10 (36) 39–59, 1967. (Cited on page 25.)

[Lang 1998] K. J. Lang, B. A. Pearlmutter and R. A. Price. *Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm.* In Proc. 4th International Colloquium on Grammatical Inference - ICGI 98, volume 1433 of *Lecture Notes in Artificial Intelligence*, pages 1–12. Springer-Verlag, 1998. (Cited on pages 15 and 17.)

[Lang 1999] Kevin J. Lang. *Faster Algorithms for Finding Minimal Consistent DFAs.* Rapport technique, 1999. (Cited on page 15.)

[Li 2006] K. Li, R. Groz and M. Shahbaz. *Integration Testing of Components Guided by Incremental State Machine Learning.* In Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), pages 59–70, 2006. (Cited on pages 31, 39, 50, 51, 84 and 119.)

[Li 2012] Nan Li. *A Smart Structured Test Automation Language (SSTAL).* In ICST, pages 471–474, 2012. (Cited on page 2.)

[Loiseaux 1995] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani and Saddek Bensalem. *Property Preserving Abstractions for the Verification of Concurrent Systems.* Formal Methods in System Design, vol. 6, no. 1, pages 11–44, 1995. (Cited on page 34.)

[Lorenzoli 2008] D. Lorenzoli, L. Mariani and M. Pezzè. *Automatic generation of software behavioral models.* In 30th International Conference on Software Engineering (ICSE 2008), pages 501–510, 2008. (Cited on pages 13, 14 and 30.)

[Luque 2010] Franco M. Luque. *A Lazy L* Algorithm for Learning Regular Languages from Queries.* ZULU workshop organised during ICGI, 2010. (Cited on pages 20 and 24.)

[Mäkinen 2001] Erkki Mäkinen and Tarja Systä. *MAS - An Interactive Synthesizer to Support Behavioral Modeling in UML.* In ICSE, pages 15–24, 2001. (Cited on pages 50 and 107.)

[Maler 1995] Oded Maler and Amir Pnueli. *On the Learnability of Infinitary Regular Sets.* Inf. Comput., vol. 118, no. 2, pages 316–326, 1995. (Cited on pages 3, 20, 38, 58, 60, 61, 65, 68, 73, 75, 85, 97, 99, 103 and 104.)

[Margaria 2004] T. Margaria, O. Niese, H. Raffelt and B. Steffen. *Efficient test-based model generation for legacy reactive systems.* High-Level Design, Validation, and Test Workshop, IEEE International, vol. 0, pages 95–100, 2004. (Cited on pages 26, 59 and 118.)

[Margaria 2005a] Tiziana Margaria, Harald Raffelt and Bernhard Steffen. *Knowledge-based relevance filtering for efficient system-level test-based model generation.* ISSE, vol. 1, no. 2, pages 147–156, 2005. (Cited on page 34.)

[Margaria 2005b] Tiziana Margaria, Bernhard Steffen and Manfred Reitenspieß. *Service-Oriented Design: The jABC Approach.* In Service Oriented Computing, 2005. (Cited on page 33.)

[Meinke 2010] Karl Meinke and Fei Niu. *A Learning-Based Approach to Unit Testing of Numerical Software.* In ICTSS, pages 221–235, 2010. (Cited on page 12.)

[Meinke 2011a] Karl Meinke and Fei Niu. *Learning-Based Testing for Reactive Systems Using Term Rewriting Technology.* In ICTSS, pages 97–114, 2011. (Cited on page 2.)

[Meinke 2011b] Karl Meinke and Muddassar A. Sindhu. *Incremental Learning-Based Testing for Reactive Systems.* In TAP, pages 134–151, 2011. (Cited on page 2.)

[Merten 2011] Maik Merten, Bernhard Steffen, Falk Howar and Tiziana Margaria. *Next Generation LearnLib.* In TACAS, pages 220–223, 2011. (Cited on pages 81 and 123.)

[Milner 1982] R. Milner. A calculus of communicating systems. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. (Cited on page 136.)

[Mitchell 1997] Tom M. Mitchell. Machine learning. McGraw-Hill, New York, 1997. (Cited on page 29.)

[Moller] Faron Moller and Perdita Stevens. *Edinburgh Concurrency Workbench User Manual (Version 7.1).* Available from http://homepages.inf.ed.ac.uk/perdita/cwb/. (Cited on page 76.)

[Moore 1956] Edward F. Moore. *Gedanken Experiments on Sequential Machines.* In Automata Studies, pages 129–153. Princeton U., 1956. (Cited on page 2.)

[Muccini 2007] Henry Muccini, Andrea Polini, Fabiano Ricci and Antonia Bertolino. *Monitoring Architectural Properties in Dynamic Component-Based Systems.* In CBSE, pages 124–139, 2007. (Cited on page 12.)

[Myers 2004] Glenford J. Myers and Corey Sandler. The art of software testing. John Wiley & Sons, 2004. (Cited on page 1.)

[Niese 2001] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, W. Goerigk and H. Ide. *Automated Regression Testing of CTI-Systems.* In Sixth IEEE European Test Workshop (ETW'01), pages 51–51, 2001. (Cited on page 33.)

[Niese 2003] Oliver Niese. *An Integrated Approach to Testing Complex Systems.* PhD thesis, University of Dortmund, 2003. (Cited on pages 12, 26, 29, 36, 37, 39, 50, 51, 84, 86, 102, 103, 107, 108 and 110.)

[Oliveira 1998] A. Oliveira and J. Silva. *Efficient Search Techniques for the Inference of Minimum Size Finite Automata.* In String Processing and Information Retrieval: A South American Symposium, pages 81–89, 1998. http://www.odysci.com/article/1010112997165286. (Cited on page 15.)

[Oncina 1992] J. Oncina and P. Garcia. *Inferring Regular Languages in Polynomial Update Time.* Pattern Recognition and Image Analysis, pages 49–61, 1992. (Cited on page 13.)

[Parnas 1990] David L. Parnas, A. John van Schouwen and Shu Po Kwan. *Evaluation of safety-critical software.* Commun. ACM, vol. 33, no. 6, pages 636–648, June 1990. (Cited on page 1.)

[Pasareanu 2008] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh and H. Barringer. *Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning.* Formal Methods in System Design, vol. 32, pages 175–205, 2008. (Cited on page 25.)

[Peled 1999] Doron Peled, Moshe Y. Vardi and Mihalis Yannakakis. *Black Box Checking.* In FORTE, pages 225–240, 1999. (Cited on pages 12, 29, 59, 106 and 118.)

[Pena 1998] Jorge M. Pena and Arlindo L. Oliveira. *A new algorithm for the reduction of incompletely specified finite state machines.* In ICCAD, pages 482–489, 1998. (Cited on page 26.)

[Pradel 2010] M. Pradel, P. Bichsel and T. R. Gross. *A framework for the evaluation of specification miners based on finite state machines.* In 26th IEEE International Conference on Software Maintenance (ICSM 2010), pages 1–10, 2010. (Cited on page 31.)

[Raffelt 2005] H. Raffelt, B. Steffen and T. Berg. *LearnLib: a library for automata learning and experimentation.* In 10th international workshop on Formal methods for industrial critical systems, pages 62–71, 2005. (Cited on pages 33 and 34.)

[Raffelt 2006] H. Raffelt and B. Steffen. *LearnLib: A Library for Automata Learning and Experimentation.* In Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, volume 3922, pages 377–380, 2006. (Cited on pages 33 and 34.)

[Raffelt 2008] H. Raffelt, T. Margaria, B. Steffen and M. Merten. *Hybrid test of web applications with webtest.* In 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), pages 1–7, 2008. (Cited on pages 33 and 34.)

[Raffelt 2009] Harald Raffelt, Maik Merten, Bernhard Steffen and Tiziana Margaria. *Dynamic testing via automata learning.* STTT, vol. 11, no. 4, pages 307–324, 2009. (Cited on pages 12, 29 and 33.)

[Ramanathan 2007] M. K. Ramanathan, A. Grama and S. Jagannathan. *Static specification inference using predicate mining.* In ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pages 123–134, 2007. (Cited on page 30.)

[Rivest 1993] 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. (Cited on pages 3, 12, 19, 24, 38, 58, 59, 60, 63, 68, 73, 81, 84, 97, 99, 104, 109, 110 and 123.)

[Sakallah 2009] Karem A. Sakallah. *Symmetry and Satisfiability.* In Handbook of Satisfiability, pages 289–338. 2009. (Cited on page 15.)

[Shahbaz 2007] Muzammil Shahbaz, Keqin Li and Roland Groz. *Learning and Integration of Parameterized Components Through Testing.* In TestCom/FATES, pages 319–334, 2007. (Cited on page 25.)

[Shahbaz 2008] Muzammil Shahbaz. *Reverse Engineering Enhanced State Models of Black Box Components to Support Integration Testing.* PhD thesis, Grenoble Institute of Technology, 2008. (Cited on pages 12, 31 and 119.)

[Shahbaz 2009] Muzammil Shahbaz and Roland Groz. *Inferring Mealy Machines.* In FM, pages 207–222, 2009. (Cited on pages 12, 20, 26, 36, 37, 38, 39, 50, 51, 58, 60, 61, 68, 73, 76, 84, 85, 97, 102, 104, 107, 109, 110 and 119.)

[Shahbaz 2011] Muzammil Shahbaz, K. C. Shashidhar and Robert Eschbach. *Iterative refinement of specification for component based embedded systems.* In ISSTA, pages 276–286, 2011. (Cited on page 31.)

[Shoham 2007] S. Shoham, E. Yahav, S. J. Fink and M. Pistoia. *Static specification mining using automata-based abstractions.* In ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, pages 174–184, 2007. (Cited on page 30.)

[Shu 2007] 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. (Cited on pages 3, 12, 26, 29, 51, 102 and 108.)

[Sidhu 1989] D. P. Sidhu and T. Leung. *Formal Methods for Protocol Testing: A Detailed Study.* IEEE Transactions on Software Engineering, vol. 15, pages 413–426, 1989. (Cited on page 28.)

[Starkie 2004] Bradford Starkie, François Coste and Menno van Zaanen. *The Omphalos Context-Free Grammar Learning Competition.* In ICGI, pages 16–27, 2004. (Cited on page 28.)

[Starkie 2006] Bradford Starkie, Menno van Zaanen and Dominique Estival. *The Tenjinno Machine Translation Competition.* In ICGI, pages 214–226, 2006. (Cited on page 28.)

[Steffen 2006] Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges and Christian Kubczak. *Model-Driven Development with the jABC.* In Haifa Verification Conference, pages 92–108, 2006. (Cited on page 33.)

[Steffen 2011] Bernhard Steffen, Falk Howar and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective.* In SFM, pages 256–296, 2011. (Cited on pages 3 and 123.)

[The Network Simulator - ns-2] The Network Simulator - ns-2. Available from http://www.isi.edu/nsnam/ns/. (Cited on page 34.)

[Tracker] Mantis Bug Tracker. Available from http://www.mantisbt.org/, June 2007. (Cited on page 33.)

[Trakhtenbrot 1973] B. A. Trakhtenbrot and Ya. M. Barzdin. *Finite Automata, Behaviour and Synthesis.* North-Holland, 1973. (Cited on pages 3, 40 and 106.)

[Utting 2007] Mark Utting and Bruno Legeard. Practical model-based testing - a tools approach. Morgan Kaufmann, 2007. (Cited on page 2.)

[Vasilevskii 1973] M. P. Vasilevskii. *Failure diagnosis of automata.* Cybernetics and Systems Analysis, vol. 9, pages 653–665, 1973. 10.1007/BF01068590. (Cited on pages 59, 80 and 118.)

[Vilar 2010] Juan Miguel Vilar. *Next question, please. Two query learning algorithms participate in the Zulu challenge.* ZULU workshop organised during ICGI, 2010. (Cited on page 24.)

[Walkinshaw 2008] N. Walkinshaw, K. Bogdanov, S. Ali and M. Holcombe. *Automated discovery of state transitions and their functions in source code.* Softw. Test., Verif. Reliab, vol. 18, pages 99–121, 2008. (Cited on page 12.)

[Walkinshaw 2010a] N. Walkinshaw, K. Bogdanov, C. Damas, B. Lambeau and P. Dupont. *A framework for the competitive evaluation of model inference techniques.* In First International Workshop on Model Inference In Testing, pages 1–9, 2010. (Cited on pages 15 and 16.)

[Walkinshaw 2010b] Neil Walkinshaw, Kirill Bogdanov, John Derrick and Javier Paris. *Increasing Functional Coverage by Inductive Testing: A Case Study.* In ICTSS, pages 126–141, 2010. (Cited on page 19.)

[Wasylkowski 2007] A. Wasylkowski, A. Zeller and C. Lindig. *Detecting object usage anomalies*. In 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 35–44, 2007. (Cited on page 30.)

[Xie 2004] T. Xie and D. Notkin. *Mutually Enhancing Test Generation and Specification Inference*. In Formal Approaches to Software Testing, Third International Workshop on Formal Approaches to Testing of Software, FATES 2003, volume 2931, pages 60–69, 2004. (Cited on pages 29 and 30.)

[Yokomori 1994] T. Yokomori. *Learning non-deterministic finite automata from queries and counterexamples*. In Machine Intelligence 13, pages 169–189, 1994. (Cited on page 28.)