

Doctoral Thesis



**Faculty of Engineering
Department of Computer Science**

Testing and Active Learning of Resettable Finite-State Machines

Ing. Michal Soucha

**A thesis submitted towards the degree of
Doctor of Philosophy**

**January 2019
Supervisor: Dr Kirill Bogdanov**

Abstract

This thesis proposes novel active-learning algorithms and testing methods for deterministic finite-state machines that (i) have a specified transition from every state on each input of the (fixed) alphabet and (ii) can be reliably reset to the initial state on request. These algorithms rely on the novel methods of construction of separating sequences. Extensive evaluation demonstrates that the described testing and learning methods are the most efficient in terms of the amount of interaction by a tester with the system under test.

Keywords: finite-state machine, separating sequence, software testing, active learning, automata inference, deterministic finite acceptor, Mealy machine, Moore machine

Acknowledgement

Firstly, I would like to thank my supervisor, Dr Kirill Bogdanov, for his perfect guidance and detailed discussions of the topic. I would also like to thank Professor Georg Struth for his kindness and good advices, and Dr Mark Hepple for his valuable suggestions on my personal growth. I thank the Department of Computer Science at the University Of Sheffield that gave me the opportunity to make my research and contributed to the science.

My thanks also belong to my wife, my parents and my brother for their endless support and love.

Contents

Introduction	
1 Introduction	3
1.1 State Machines – Turnstile	
Example	4
1.2 Testing and Active Learning	5
1.3 Motivation of the Work in the	
Thesis	9
1.4 Contribution of the Work in the	
Thesis	9
1.5 Formal Languages and General	
Definitions	10
1.6 Finite-State Machine	11
1.6.1 Sequences for State	
Identification	14
1.6.2 State Pair Array	16
Part I	
Construction of State	
Identification Sequences	
2 Introduction	19
2.1 Research Questions	20
3 Existing Methods	21
3.1 Construction of Separating	
Sequences	21
3.1.1 Minimization Approach	21
3.1.2 Shortest Separating Sequences	
(SSS)	22
3.1.3 Splitting Tree with Minimal	
Separating Sequences (ST-MSS) .	24
3.2 Construction of Characterizing Set	
(CSet)	26
3.2.1 Minimization Approach	26
3.2.2 Incremental Method	26
3.2.3 From SSS	26
3.2.4 From ST-MSS	27
3.2.5 Reduction of CSet	27
3.3 Construction of Harmonized State	
Identifiers (HSI)	29
3.3.1 From CSet	29
3.3.2 From SSS	31
3.3.3 From ST-MSS	31
3.3.4 From Incomplete Adaptive	
Distinguishing Sequences (IADS)	31
4 Splitting Tree with Invalid Inputs	35
4.1 Motivation Example	35
4.2 Splitting Tree for Incomplete	
Adaptive Distinguishing Sequences	38
4.2.1 Input Validity Types	38
4.3 State Identification Sequences	
From ST-IADS	39
4.4 ST-IADS Construction Algorithm	44
4.5 Running Example	53
5 Experiments	57
5.1 Randomly Generated Machines .	57
5.2 Models of Real Systems	59
6 Conclusion	61
Part II	
Testing Methods	
7 Introduction	65
7.1 Research Questions	66
8 Test Suite Completeness	67
8.1 Convergence of Sequences	69
8.2 State Domains of Test Sequences	74
8.3 Sufficient Conditions	77
9 Existing Methods	83
9.1 ADS-method	84
9.2 SVS-method	84
9.3 W-method	84
9.4 Wp-method	84
9.5 HSI-method	85
9.6 H-method	85
9.6.1 Implementation	85
9.6.2 Running Example	89
9.7 SPY-method	91
9.7.1 Implementation	91
9.7.2 Running Example	93
9.8 Other Methods	95
10 SPYH-method	97
10.1 Implementation	98
10.2 Running Example	102
11 S-method	107
11.1 Implementation	108
11.2 Running Example	116
12 Experiments	119
12.1 Randomly Generated Machines	120
12.2 Models of Real Systems	124
13 Conclusion	127

Part III	
Active-Learning Algorithms	
14 Introduction	131
14.1 Active Learning	132
14.2 Research Questions	136
15 Observation Tree Approach	137
15.1 General Idea	141
15.2 Common Properties and Optimizations	142
15.2.1 Adaptive Separating Sequences	145
15.2.2 Types of Inconsistency	148
15.3 H-learner	153
15.3.1 Implementation	153
15.3.2 Running Example	159
15.4 SPY-learner	166
15.4.1 Implementation	166
15.4.2 Running Example	176
15.5 S-learner	183
15.5.1 Implementation	184
15.5.2 Running Example	188
15.6 General Learning Framework	193
16 Standard Learning Algorithms	195
16.1 L* algorithm	195
16.1.1 AllPrefixes	198
16.1.2 OneSuffix - binary search	198
16.1.3 AllSuffixesAfterLastState	198
16.1.4 Suffix1by1	199
16.1.5 SuffixAfterLastState	199
16.1.6 Other Counterexample Processing Approaches	199
16.1.7 Semantic Suffix Closedness	199
16.2 Discrimination Tree algorithm	202
16.3 Observation Pack algorithm	205
16.4 TTT algorithm	208
16.5 Quotient algorithm	212
16.6 GoodSplit algorithm	215
16.7 Other Learning Algorithms	220
16.8 Summary of the Algorithms	222
17 Experiments	225
17.1 Randomly Generated Machines	227
17.2 Models of Real Systems	231
17.3 GridWorld	232
18 Conclusion	237

Conclusions and Future Work	
19 Conclusions	241
20 Future Work	243
Appendices	
A FSMlib	247
B Generator of Random Finite-State Machines	249
C Description of Case Studies	251
C.1 Randomly Generated Machines	251
C.2 Models of Real Systems	252
C.3 GridWorld in the Brain Simulator	252
D Flaws in the Related Work	255
D.1 IADS construction algorithm	255
D.2 Convergence of Test Sequences	257
Bibliography	259

Figures

<p>1.1 Turnstile example 4</p> <p>1.2 Turnstile specification 5</p> <p>1.3 Turnstile implementation 5</p> <p>1.4 Testing and active learning 6</p> <p>1.5 Turnstile conjecture 7</p> <p>1.6 Turnstile implementation with an extra state 8</p> <p>1.7 Turnstile implementation with k extra states 9</p> <p>1.8 Types of finite-state machines . . 12</p> <p>3.1 Incomplete adaptive distinguishing sequences 33</p> <p>4.1 Mealy machine 36</p> <p>4.2 HSI construction approaches . . . 37</p> <p>4.3 IADSs from ST-IADS 43</p> <p>4.4 ST node splitting 45</p> <p>4.5 ST construction – root 53</p> <p>4.6 ST construction – invalid inputs 54</p> <p>4.7 ST construction halfway 55</p> <p>5.1 HSIs of randomly generated machines 58</p> <p>8.1 Convergent sequences in the implementation 72</p> <p>8.2 State domains 75</p> <p>8.3 Transition verification 79</p> <p>9.1 H-method 90</p> <p>9.2 SPY-method 94</p> <p>10.1 SPYH-method 103</p> <p>10.2 SPYH-method: convergent graph 105</p> <p>11.1 S-method 117</p> <p>11.2 S-method: convergent graph . 118</p> <p>12.1 Testing of randomly generated Moore machines: test suite sizes . 121</p> <p>12.2 Testing of randomly generated Moore machines: exploration efficiency and construction time . 122</p> <p>12.3 Testing of randomly generated Mealy machines: different number of extra states 123</p>	<p>12.4 Testing of real systems: different number of extra states 125</p> <p>14.1 Active learning of finite-state machines 135</p> <p>15.1 A sketch of observation tree with some state-relating parts coloured 139</p> <p>15.2 Adaptive separating sequence 147</p> <p>15.3 Resolving inconsistency II: inconsistent state domain 150</p> <p>15.4 Resolving inconsistency III: empty domain of convergent node 150</p> <p>15.5 DFA black box 159</p> <p>15.6 H-learner: 2nd state observed 160</p> <p>15.7 H-learner: 3rd state observed . 161</p> <p>15.8 H-learner: complete conjecture 162</p> <p>15.9 H-learner: DFA learned 163</p> <p>15.10 H-learner: extra state elimination 164</p> <p>15.11 H-learner: complete OTree . . 165</p> <p>15.12 Resolving inconsistency I . . . 171</p> <p>15.13 SPY-learner: 3 states revealed 177</p> <p>15.14 SPY-learner: complete 3-state conjecture 180</p> <p>15.15 SPY-learner: DFA learned . . 181</p> <p>15.16 SPY-learner: complete OTree 182</p> <p>15.17 S-learner: complete 4-state DFA 189</p> <p>15.18 S-learner: complete 4-state DFA 191</p> <p>15.19 S-learner: complete OTree . . 192</p> <p>16.1 An observation table (\bar{S}, E, T) 196</p> <p>16.2 L^* example: initial closed OT 200</p> <p>16.3 L^* example: 3rd state observed 201</p> <p>16.4 DT example: initialization . . . 203</p> <p>16.5 DT example: CE processing . . 204</p> <p>16.6 OP example: initialization . . . 205</p> <p>16.7 OP example: 2nd state observed 206</p> <p>16.8 OP example: 3rd state observed 207</p> <p>16.9 OP example: CE processing . . 208</p> <p>16.10 TTT example: CE processing 210</p> <p>16.11 TTT example: finalization . . 211</p> <p>16.12 Quotient example: initialization 213</p>
--	--

16.13 Quotient example: 3 states revealed	213
16.14 Quotient example: CE processing	214
16.15 GoodSplit example: 1st closed model	216
16.16 GoodSplit example: 3 states revealed	218
16.17 GoodSplit example: all states revealed	219
17.1 Learning randomly generated DFSMs: output and equivalence queries	228
17.2 Learning randomly generated DFSMs: queried symbols and resets	229
17.3 Learning randomly generated DFSMs: exploration efficiency and learning time	230
C.1 GridWorld map E	253
C.2 Model of GridWorld map E...	254
D.1 Sequence convergence	258

Tables

5.1 HSIs of real systems	60
12.1 Testing methods comparison on the example	120
16.1 Access and separating sequences used by the learning algorithms..	222
16.2 Interpretation of GLF by the learning algorithms	223
17.1 Learning the reference DFA ..	226
17.2 Learning peterson2	232
17.3 Learning sched4	233
17.4 Learning sched5	234
17.5 Learning GridWorld	235
C.1 DFA models of real systems ..	252

Algorithms

1	Construction of shortest separating sequences	23			34	154
2	Reduction LS-SL of CSet	28			35	155
3	Reduction EqualLen of CSet	30			36	156
4	ST – obtaining a separating sequence	40			37	157
5	ST – get separating node	41			38	157
6	ST – construction of HSIs	41			39	158
7	ST – obtaining an IADS	42			40	168
8	Construction of a splitting tree (ST-IADS)	47			41	169
9	ST – recording valid transitions of dependent nodes	48			42	170
10	ST – dependent resolving	50			43	171
11	ST – recording invalid transitions of dependent nodes	50			44	172
12	ST – separating sequence score	51			45	173
13	H-method	86			46	173
14	H-DISTINGUISH	86			47	174
15	H-DISTINGUISHFROMSC	87			48	176
16	H-DISTINGUISHFROMSET	87			49	185
17	H-GETBESTPREFIX	88			50	185
18	H-ESTIMATEGROWTHOFT	88			51	186
19	SPY-method	92			52	187
20	SPY-APPENDSEPARATING-SEQUENCE	93			53	188
21	SPYH-method	98			54	193
22	SPYH-DISTINGUISH	99			55	256
23	SPYH-DISTINGUISHFROMSET	100				
24	SPYH-GETBESTPREFIX	101				
25	S-method	109				
26	S-CREATEDIVERGENCE-PRESERVINGSC	111				
27	S-DISTINGUISH	112				
28	S-DISTINGUISHFROMSET	112				
29	S-APPENDSEPARATING-SEQUENCE	113				
30	S-AREDISTINGUISHED	114				
31	Learner based on the observation tree approach	143				
32	QUERY	145				
33	RESOLVEINCONSISTENCY	152				



Introduction



Chapter 1

Introduction

System quality and absence of faults are very important in the development of any software or hardware product. Quality of a system can be improved if one has or can construct an abstract model of the system that can be analysed and tests can be generated from such a model. One type of such abstract models is a finite-state machine (FSM) that can model a wide variety of systems, from communication protocols and software systems to hardware components. Use of finite-state machines is beneficial for the following three reasons. First, finite-state machines are easy to understand and analyse for both human and computer. Second, tests that check if the system works properly can be generated automatically from its model represented by a finite-state machine. Third, it is possible to learn a model of the system that can be described as a finite-state machine.

This thesis describes how to generate tests from FSM models and how to learn FSM models by interaction with the system much more efficiently than the other methods. There are plenty of methods for both testing and learning but most of them do not minimize the amount of interaction with the system. Each interaction requires some time and thus also money in practice. Therefore, it is important to test or learn the system using the least number of interactions.

Testing and learning of finite-state machines with terms used in this thesis are described informally on the example of a turnstile in the next two sections. The rest of this introduction chapter describes the motivation of research of finite-state machines, the contribution of this thesis, and definitions of finite-state machines and common terms. Then the thesis is divided into three parts. Part I deals with the construction of separating sequences that are important in both testing and learning. Part II describes testing of finite-state machines and proposes new testing methods. Part III is about active learning of finite-state machines and proposes novel learning algorithms. Each part introduces the topic and definitions of related terms at first. Then, particular approaches are discussed and proposed algorithms are experimentally evaluated on randomly-generated machines and another two case studies that are summarized in Appendix C. All parts end with a conclusion of the work done in the related part. The entire thesis is summed up in Chapter 19 followed by the future work in Chapter 20.

1.1 State Machines – Turnstile Example

The simple turnstile in Figure 1.1 is an example of system that can be described by a finite-state machine. It accepts two actions, or inputs, and responds with one of three possible outputs to each input. If one inserts a coin in the turnstile, then there is no observable response; the output for the action ‘insert a coin’, encoded as ‘c’, is ‘N’ as ‘No response’. If one pushes the bar of turnstile, the bar is either locked or free so that one can pass through. The input ‘p’ as ‘push the bar’ leads either to the output ‘L’ to mean ‘Locked’ or to the output ‘F’ to mean ‘Free’.

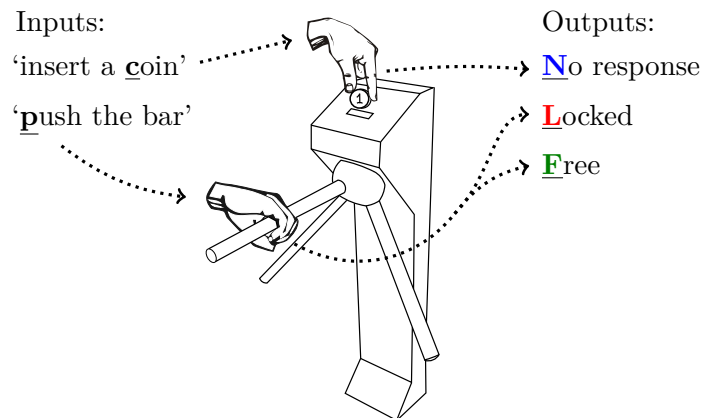


Figure 1.1: A simple turnstile for coins

The *specification*, or a model of how the system should work, of the simple turnstile is shown in Figure 1.2. It has two states called ‘Locked’ and ‘Unlocked’ that correspond to the states of the bar. The bar is initially locked and is locked until one inserts a coin; the initial state is indicated by a small arrow above the state ‘Locked’. Other arrows represent transitions, for example, the leftmost arrow labelled with ‘p/L’ corresponds to the action of the bar push with the response that it is locked. The state of the turnstile is changed to ‘Unlocked’ after a coin is inserted. The transition from ‘Unlocked’ corresponding to the bar push is labelled with ‘p/F’ so the bar is free. If more than one coin is inserted in a row, then the turnstile remains in the state ‘Unlocked’ which is captured by the transition ‘c/N’ that leads from ‘Unlocked’ back to that state. The specification of the simple turnstile thus can be summarized as ‘exactly one person can pass through after at least one coin is inserted’.

The specification in Figure 1.2 is also the first example of models that this thesis works with. It has at most one transition for each input from every state, therefore, it is *deterministic*. There is a transition for each input from every state, therefore, it is *completely-specified*. The specification is thus an example of a completely specified deterministic finite-state machine. In

addition, it is assumed that systems as well as models can be *reset* reliably to the initial state at any time; such machines are called *resettable*.

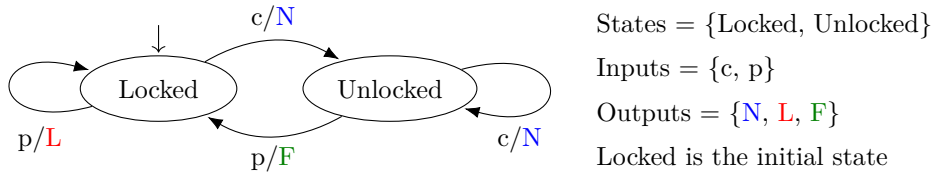


Figure 1.2: The specification of the simple turnstile (Figure 1.1)

An *implementation*, or a model of how the system works, can be different from the specification. The differences are then called faults. For example, the implementation of the simple turnstile in Figure 1.3 contains a transition fault. The transition ‘c/N’ from the state ‘Unlocked’ leads to the state ‘Locked’ instead of going back to ‘Unlocked’. It means that every second coin does not allow one to pass through the turnstile. Therefore, this implementation follows the description of ‘one can pass through after an odd number of coins is inserted’.

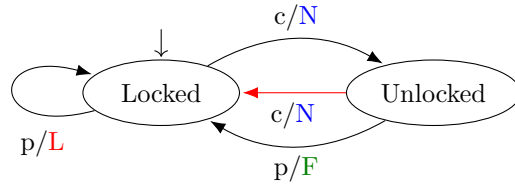


Figure 1.3: A faulty implementation of the simple turnstile (Figure 1.1)

1.2 Testing and Active Learning

Testing methods create a set of test sequences called a test suite such that test sequences as sequences of inputs asked from the initial state aim to reveal faults in the implementation. A fault is revealed by observing a response different from the expected one that is produced by the specification. If the implementation passes a test suite, that is, responses to all test sequences are equal to the expected ones, then a guarantee on the absence of particular faults is usually given. For example, the simplest testing method, the transition tour method, just checks the presence of all transitions so it can guarantee that no transition is missing if the implementation passes the test suite. The transition tour method cannot detect all transition faults. For instance, the test suite of sequences ‘p’, ‘c,c’ and ‘c,p’ covers all transition of the specification in Figure 1.2 but the implementation in Figure 1.3 with a transition fault passes it. The transition fault is not revealed because the target states of transitions are not verified. Such verification is done by using separating sequences. A *separating sequence* of two states is a sequence of inputs such that the

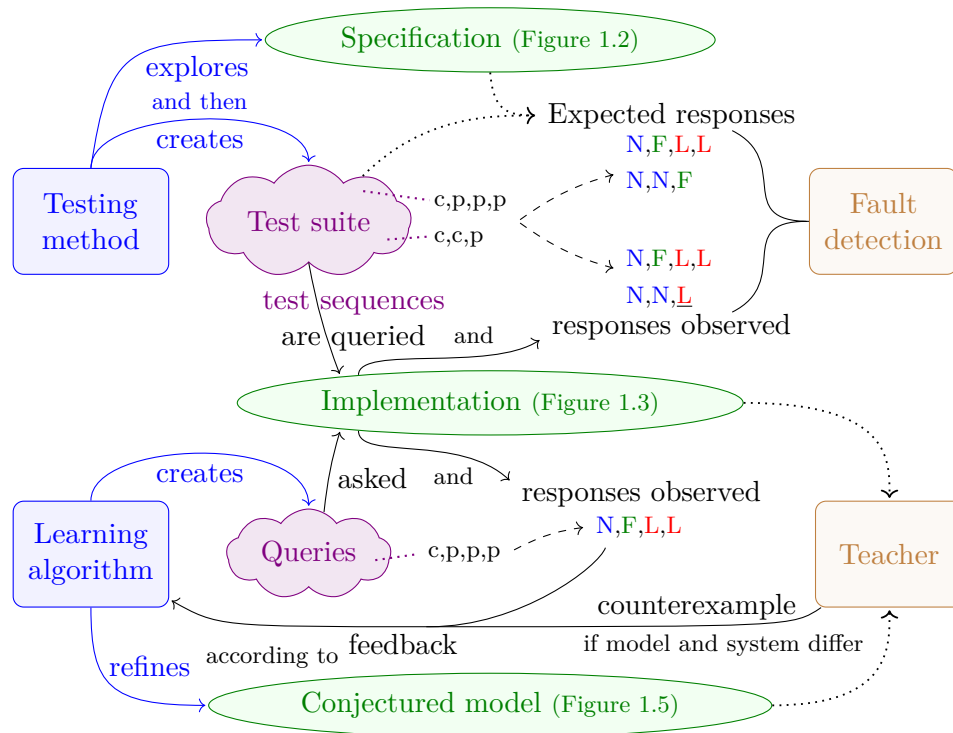


Figure 1.4: Standard setting of testing and active learning of finite-state machines

states respond with different outputs to the sequence and thus the sequence separates, or distinguishes, them. One separating sequence of states ‘Locked’ and ‘Unlocked’ of the simple turnstile’s specification is the input ‘p’. Therefore, it reveals the transition fault in the implementation in Figure 1.3 if it is asked after the sequence of ‘c,c’. The bar would be locked after insertion of two coins but it should be free according to the specification in Figure 1.2.

The upper half of Figure 1.4 captures a general flow of testing with an example of the test suite of the simple turnstile’s specification. At first, a testing method analyses the specification and builds a test suite based on the analysis. Then the responses to the test sequences are collected both from the specification and the implementation. If there is a difference between the responses of both models, then a fault is detected. Otherwise some guarantee on the correctness of the implementation is given. In the case of the simple turnstile, sequences ‘c,p,p,p’ and ‘c,c,p’ are sufficient to reveal any transition fault in the implementation.

The specification can often be unsuitable for the construction of test suites. It can be outdated, not described as a finite-state machine, partially specified, or completely missing. If one still wants to know how the implementation works, then its model can be learnt instead. There are two main approaches to learning, active and passive. *Passive learning* derives a model of a system just from provided traces, that is, input sequences with corresponding responses. In contrast, *active learning* does not limit itself to a set of a few traces but it

interacts with the system by asking for responses on particular input sequences. According to obtained outputs it refines the conjectured model and queries another sequence. The model derived by passive learning thus captures just properties covered in the provided traces while active learning can explore any behaviour of the system and so it is able to detect possible faults that are extremely rare. Therefore, this thesis deals with active learning.

The standard setting of active learning is captured in the lower half of Figure 1.4. A *learning algorithm*, or a *learner*, interacts with the implementation, or the system to learn. It asks for responses to particular sequences of actions based on the obtained outputs and refines the conjectured model. Then, it creates another query for the system. In addition, there is usually a teacher that can help the learner with the learning. The teacher provides a counterexample on request of the learner if the conjectured model differs from the implementation. Otherwise, the teacher confirms that both models are equivalent which means that the learner stops the learning.

In the case of the simple turnstile, the learner knows nothing about the system at first, so the conjectured model would be just the initial state s_0 . Let ‘c,p,p,p’ be the first query that is asked. According to the response ‘N,F,L,L’, the learner refines the conjectured model as shown in Figure 1.5. On the one hand, the conjectured model has two states as the implementation, because the separating sequence of single input ‘p’ was observed with two different outputs. On the other hand, the behaviours of the models differ because the bar is estimated to be free initially by the conjectured model. Note that there is no transition for the input ‘c’ from the state s_1 which means that the conjectured model is not completely specified.

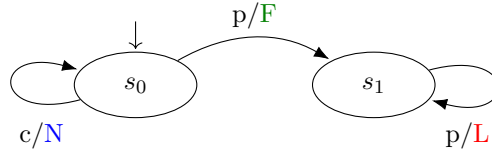


Figure 1.5: The conjectured model of the simple turnstile (Figure 1.1) after observing ‘N,F,L,L’ in response to ‘c,p,p,p’

Figure 1.4 highlights the significant similarity between active learning and testing. Both approaches work with the implementation by querying sequences of inputs and make further steps according to observed responses. These input sequences are formed mainly based on the knowledge of separating sequences acquired either from the specification in the case of testing or from observed responses and counterexamples in the case of learning. Both approaches also result in a guarantee of equivalence of either the implementation and the specification in testing or the implementation and the conjectured model in learning. Moreover, the abstract teacher is usually approximated by a testing method in practice. The conjectured model is then considered as the specification and thus the guarantee is the same in both approaches.

The difficulty of both approaches is hidden in providing a guarantee of equivalence and still querying as few input sequences as possible. There are many kinds of guarantees but this thesis works only with those based on the number of extra states. What are *extra states*? Consider a faulty implementation of the simple turnstile that allows exactly two people to pass through the turnstile after at least two coins are inserted. Such an implementation is captured in Figure 1.6. It has three states, the initial one where the bar is locked and two states, ‘U0’ and ‘U1’, where the bar is free. The implementation has thus one state more than the specification, that is, there is the fault of an extra state. A guarantee based on the number of extra states is usually in the form of ‘if the implementation passes the test suite but differs from the specification, then the implementation has more than k extra states’. Particular conditions that influence which sequences are queried need to be fulfilled to achieve such a guarantee.

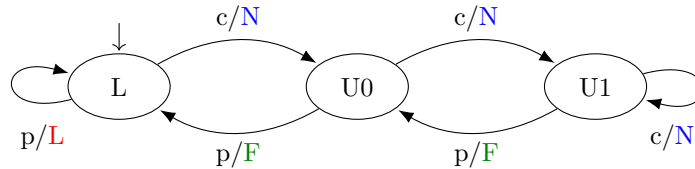


Figure 1.6: An implementation of the simple turnstile with an extra state

The complexity grows with the number of extra states. Consider a possible implementation of the simple turnstile in Figure 1.7 that has k extra states and thus $k + 2$ states in total. It describes the turnstile that counts the number of inserted coins but can count just up to $k + 1$. It also decreases the number of coins if one passes through. The shortest separating sequence of states ‘ U_i ’ and ‘ U_j ’, $i < j \leq k$, is $(i + 2)$ -times ‘p’; ‘U0’ responds to sequence ‘p,p’ with ‘F,L’ and ‘U1’ with ‘F,F’. If one wants to guarantee that there is no such extra state, then at least these separating sequences need to be queried from corresponding states. However, a testing method does not know that the implementation looks like the one in Figure 1.7 and does not know its separating sequences. Therefore, it needs to query many more sequences to eliminate all possible faulty implementations that have up to k extra states. Notice that the responses to all shortest separating sequences of two states differ only in the last output symbol. Therefore, each separating sequence is formed of a transfer sequence producing the same outputs, a *separating input*, and an optional suffix sequence; ‘p,p,p’ is a separating sequence of ‘U0’ and ‘U1’ but it is not the shortest one, the second ‘p’ is the separating input. Notice also that the sequence of $(k + 1)$ -times ‘p’ has an important property in the implementation in Figure 1.7 as it distinguishes all states, that is, all states respond to the sequence differently. Such a sequence is called a *distinguishing sequence* and it can rapidly reduce the number of sequences needed to be asked.

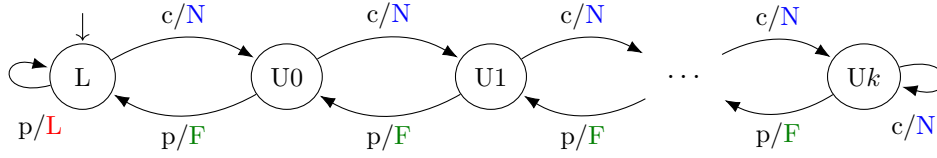


Figure 1.7: An implementation of the simple turnstile with k extra states

1.3 Motivation of the Work in the Thesis

Understanding the inner representation of a system is the main goal of reverse engineering and it is a crucial part for many other fields like testing and verification. Therefore, it has a great impact if there is a learning algorithm that can learn a model of the system efficiently and with as little external help as possible. If someone tries to use an active-learning algorithm, it is usually the oldest one, the L^* algorithm. It asks a lot of queries and it is also very dependent on the teacher knowing if the learnt automaton is the ‘correct’ one. Hence, the approach of active learning is usually given up as there are just a few learning algorithms performing a little better than the L^* algorithm but they are not well-known. How would the situation be changed if a learning algorithm learned with just half of what the L^* algorithm needs?

Testing is an important step in the development of any system as it can detect faults and thus helps to improve the quality. In practice, each test requires some resources such as money or time, therefore, testing is usually limited by the given resources. One should thus try to minimize the number of tests while retaining at least the same guarantee on the outcome. Similarly to the learning, if someone tries to use a testing method for resettable finite-state machines, it is usually the oldest one, the W-method. It is very simple but it generates a lot of test sequences. There are some advanced testing methods that performs much better than the W-method but are they at the edge of efficiency? Is there a testing method that creates test suites significantly smaller than the advanced testing methods?

1.4 Contribution of the Work in the Thesis

The thesis contains several contributions. The main five follow:

Part I — Construction of Separating Sequences

- Existing *splitting tree* (ST) construction procedure was amended to work with invalid inputs (Chapter 4). This makes it possible to reduce a test suite by constructing greatly reduced harmonized state identifiers (HSI) from the splitting tree.

Part II — Testing Methods

- New *sufficient condition on completeness of a test suite* (Section 8.3) that is weaker than the other sufficient conditions.

- *S-method* (Chapter 11) – a novel testing method using separating sequences formed from ST that makes it the most efficient testing method.

Part III — Active-Learning Algorithms

- *Observation tree approach* (Chapter 15) proposes a way how to learn efficiently with an assumption of extra states by including a testing method in the learning and using an appropriate learning structure. In addition, it benefits from asking for responses on single inputs instead of on entire queries that makes it more adaptive and flexible. It usually eliminates the dependence on the teacher and still needs less interaction with the system than the other learning algorithms.
- *S-learner* (Section 15.5) – a novel learning algorithm that follows the observation tree approach and employs the S-method to be the most efficient learning algorithm.

There are also a few small contributions. *FSMlib*, a C++ library described in Appendix A, contains implementation of a significant number of testing methods and learning algorithms that makes it the most comprehensive library for deterministic finite-state machines. Moreover, all algorithms are adjusted to work with all types of deterministic finite-state machines described in Section 1.6. These amendments are explained on an example after each algorithm is described in the particular section of the thesis. The original algorithms were usually proposed for one of the three most common types of finite-state machine. By introducing a special input symbol called *stOut*, a new more general definition of deterministic finite-state machine was developed such that it covers all three most common types and so it provides even a richer model. As a consequence, any algorithm that works with this new general model can implicitly work with any of the three most common types.

Two flaws were found in the related works. First, the proof of Lemma 3 in [SPY12] about convergent sequences does not cover all possible cases so it is just partially correct. Convergent sequences are the foundations of the SPY-method, a testing method. Second, semantic suffix closedness proposed in [SHM11] as an improvement of the L* algorithm is shown not to be a sufficient condition for the conjecture model to be minimal as it was claimed. Both flaws are explained in detail in the related sections. In the case of convergent sequences, an amendment with a proof is proposed.

1.5 Formal Languages and General Definitions

Basics of the theory of formal languages and some general structures such as a tree are introduced in this section.

An alphabet X is a nonempty finite set of symbols x_1, \dots, x_p , where p is the size, or cardinality, of X , that is, $p = |X|$. A *string*, or a *word*, over X is any finite sequence of symbols from X . The empty string is denoted ε and it

has length 0. Let u, v be strings over the alphabet X . Then $u \cdot v$, or simply uv , means concatenation of u and v . The length of u is denoted by $|u|$ so that $|\varepsilon| = 0$. X^* is the set of all strings over the alphabet X , that is, $u, v \in X^*$. X^k and $X^{\leq k}$ are sets of all strings over the alphabet X of (respectively up to) length k , that is, $u \in X^k \iff |u| = k$ and $u \in X^{\leq k} \iff |u| \leq k$. Note that the set of all strings X^* always contains ε and $\forall u \in X^* : \varepsilon \cdot u = u = u \cdot \varepsilon$. Concatenation of two sets of strings U, V is $U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$.

Let u, v, w be strings such that $w = u \cdot v$. Then u is a *prefix* of w , v is a *suffix* of w and v is an *extension* of u in w . If $v \neq \varepsilon$, u is a *proper prefix*. The set of prefixes of w is denoted $\text{pref}(w)$. A set of strings U is *prefix-closed* if for each string $u \in U$ there are all prefixes of u in U . Similarly, a set of strings U is *suffix-closed* if for each string $u \in U$, all suffixes of u are in U .

A *successor tree*, or a *rooted directed tree*, is a directed graph with a special node, called *root*, such that no edge leads to the root and there is exactly one directed path to each node from the root. Nodes in a successor tree have a special notation. Let (u, v) be an edge of a successor tree from node u to node v . Node u is called a *parent* of v and v is a *child*, or a *successor*, of u . If node u has no successor, then u is said to be a *leaf*. Otherwise, u is said to be an *internal node*. A *decision tree* and an *observation tree* are successor trees with nodes containing additional information.

1.6 Finite-State Machine

A finite-state machine (FSM) is a model consisting of states and transitions between states. According to the received input, an FSM changes its current state and responds with the corresponding output. Examples of FSMs represented as *state diagrams* were shown in Figure 1.2 and in Figure 1.3 as the specification and the implementation of simple turnstile. The behaviour of an FSM can also be listed in *transition* and *output tables*, called together a *behaviour table*.

There are many different definitions of finite-state machines in the literature. Automata active learning deals with *deterministic finite automata* (DFA) whereas active learning of finite-state machines works rather with *Mealy machines* as they describe reactive systems more precisely. A *Moore machine* is another type that is used as a model of finite-state machines. The difference is mainly the position of outputs in a model. Moore machines and deterministic finite automata have outputs tied to states. In contrast, outputs are only on transitions in the case of Mealy machines. This section proposes a general model called *deterministic finite-state machine* (DFSM) that have outputs both on states and on transitions. All four types are depicted in Figure 1.8. States are denoted ‘A’ and ‘B’, inputs ‘a’ and ‘b’, and outputs are 0, 1, 2. Note that DFA has just two outputs, one is represented by double line around the state.

There are two functions that describe the behaviour of a model, transition and output functions. Both functions take an input symbol and respectively produce a next state (a state where the transition on the input leads) and an

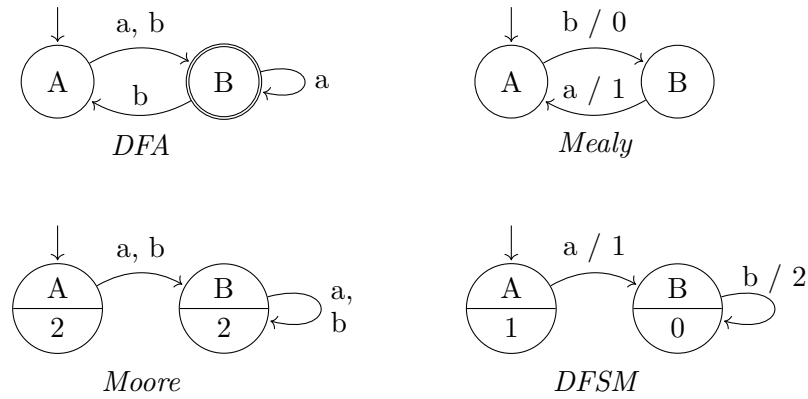


Figure 1.8: FSMs with $S = \{A, B\}$, $X = \{a, b\}$ and $Y = \{0, 1, 2\}$

output symbol that is observed if the transition is taken. Two special symbols are introduced to cover both Moore and Mealy machines in one definition. An input symbol \uparrow called *stOut* requests the state output and the current state of the machine is assumed to remain unchanged when it is used. An output symbol \downarrow called *noOut* represents ‘no response’.

Definition 1.1. A *deterministic finite-state machine* (DFSM) is a septuple $(S, X, Y, s_0, D, \delta, \lambda)$, where

- S is a finite nonempty set of states,
- X is an input alphabet (a finite nonempty set of symbols); $\uparrow \notin X$,
- Y is an output alphabet (a finite nonempty set of symbols),
- s_0 is an initial state, $s_0 \in S$,
- D is a domain of defined transitions; $D \subseteq S \times X$, $D_{\uparrow} = D \cup \{S \times \{\uparrow\}\}$,
- δ is a transition function $\delta : D_{\uparrow} \rightarrow S$ such that $\forall s \in S : \delta(s, \uparrow) = s$,
- λ is an output function $\lambda : D_{\uparrow} \rightarrow Y \cup \{\downarrow\}$.

Note that the *stOut* \uparrow is not in the input alphabet X so that it differs from all other input symbols. Similarly, the *noOut* \downarrow can be declared outside the output alphabet Y so as not to interfere with other output symbols but it is usually matched to the output of ‘timeout’ that is in Y . Therefore, it is not specified if \downarrow is or is not in Y . The timeout output represents that no response is observed during the predefined time limit.

The initial state s_0 is the current state of the machine before any input is asked. Moreover, s_0 is also the current state if the machine is *reset*. Machines that can be reset are called *resettable*.

Transitions are labelled with input and output symbols. The next state, or the target state, of a transition is defined by the transition function δ and the function λ assigns an output symbol to the transition. Not all transitions are defined as the domain D specifies. If an input $x \in X$ is applied, the current state of the machine is $s \in S$ and the transition (s, x) is not defined, that is, $(s, x) \notin D$, then the behaviour of the DFSM is undefined.

For simplicity, let $X_{\uparrow} = X \cup \{\uparrow\}$. Strings over X_{\uparrow} are called *input sequences* and strings over $Y \cup \{\downarrow\}$ are called *output sequences*. ‘Input’ and ‘output’ are usually omitted so only ‘sequence’ is used if it is clear from the context.

The transition function δ and the output function λ can be extended to work over input sequences. The extended transition function δ^* returns the target state reached by following the path labelled with the given input sequence. The output sequence formed of labels on this path is returned by the extended output function λ^* . If a transition on the path is not defined, the path and both functions are undefined. Otherwise, both functions are defined inductively as follows. If nothing is asked, then the machine stays in the same state and the empty string is obtained, that is, $\delta^*(s, \varepsilon) = s$ and $\lambda^*(s, \varepsilon) = \varepsilon$ for any state $s \in S$. If a sequence $x \cdot v$ is queried, then both functions follow the first symbol x and process the suffix v from the next state, that is, $\delta^*(s, x \cdot v) = \delta^*(\delta(s, x), v)$ and $\lambda^*(s, x \cdot v) = \lambda(s, x) \cdot \lambda^*(\delta(s, x), v)$ for all $(s, x) \in D_{\uparrow}$ and an input sequence v consisting of defined transitions. In addition, the transition and output function with their extended versions can be applied to a set of states and a set of sequences, that is, $\gamma(S', U) = \{\gamma(s, u) \mid s \in S' \wedge u \in U\}$ for all $\gamma \in \{\delta, \lambda, \delta^*, \lambda^*\}$, $S' \subseteq S$ and $U \subseteq X_{\uparrow}^*$ such that $|u| = 1$ for all $u \in U$ in the case of functions δ and λ .

Definitions of three standard models, Mealy and Moore machines and deterministic finite automaton, follow with their relations to DFSM.

Definition 1.2. A *Mealy machine* is a DFSM with a specified output function $\lambda_{Mealy} : D \rightarrow Y$.

The output function of DFSM is specified in the following manner:

$$\begin{aligned} \lambda(s, x) &= \lambda_{Mealy}(s, x) & \forall (s, x) \in D, \\ \lambda(s, \uparrow) &= \downarrow & \forall s \in S. \end{aligned}$$

The response to the stOut symbol \uparrow is always the noOut symbol \downarrow . Therefore, there is no point to apply \uparrow as the output does not provide any additional information about the model if the machine is modelled by a Mealy machine.

Definition 1.3. A *Moore machine* is a DFSM with a specified output function $\lambda_{Moore} : S \rightarrow Y$.

The specification of the output function for Moore machines assigns the output of a state to the output of the incoming transition:

$$\begin{aligned} \lambda(s, x) &= \lambda_{Moore}(\delta(s, x)) & \forall (s, x) \in D, \\ \lambda(s, \uparrow) &= \lambda_{Moore}(s) & \forall s \in S. \end{aligned}$$

This way, the stOut symbol is used only to obtain the output of the initial state. The outputs of other states are observed on transitions leading to the states. Therefore, \uparrow is applied only once in the case of a Moore machine.

Definition 1.4. A *deterministic finite automaton* is a quintuple (S, X, s_0, δ, F) , where S, X, s_0 are a set of states, an input alphabet and the initial state, respectively, and

$$\begin{aligned} \delta & \text{ is a transition function: } \delta : S \times X \rightarrow S, \\ F & \text{ is a set of } \textit{accepting}, \text{ or } \textit{final}, \text{ states; } F \subseteq S. \end{aligned}$$

A deterministic finite automaton (DFA) is a complete Moore machine $M = (S, X, Y, s_0, D, \delta, \lambda_{Moore})$ with binary output alphabet $Y = \{0, 1\}$, such that $\lambda_{Moore}(s) = 1$ if and only if $s \in F$. A complete machine has defined transitions on each input from all states. That is, a DFSM $M = (S, X, Y, s_0, D, \delta, \lambda)$ is *completely specified*, or *complete*, if and only if $D = S \times X$. Otherwise, the

- *Preset distinguishing sequence* (PDS) – a sequence u is a PDS if it separates (is a separating sequence of) all pairs of states, that is, each state responds with a unique output sequence to u .
- *State verifying sequence* (SVS), or *unique input output sequence*, of state s – a sequence u is SVS of s if it separates s from all other states, that is, s responds uniquely to u . *Verifying set* (VSet) contains an SVS for each state.
- *Adaptive distinguishing sequence* (ADS) is a VSet such that for each two states, the associated SVSs have a common prefix that separates the states.
- *State characterizing set* (SCSet) of state s is a set of separating sequences such that each other state is separated by a sequence of the SCSet. Each state of a reduced DFSM has always an SCSet. A set of separating sequences that separate all pairs of states is called *characterizing set* (CSet), sometimes denoted W .
- *Harmonized state identifiers* (HSI) are SCSets of all states such that for each pair of states s_i and s_j their separating sequence w is a prefix of a sequence in the HSIs of both s_i and s_j .

There are four main relations of these sequences with respect to a minimal DFSM M .

- i If M has a PDS, then M has an ADS.
 - Let d be a PDS. Then d is an SVS of each state and so a VSet complying with the definition of ADS can be formed as every two states have the same SVS.
- ii If M has an ADS, then M has a VSet, that is, every state of M has an SVS.
 - It follows from the definition.
- iii If a state s of M has an SVS, then s has an SCSet of one sequence.
 - Let w be an SVS of s and $W_s = \{w\}$ be an SCSet of s . Then W_s separates s from the other states because w does it by the definition of SVS.
- iv M has an ADS if and only if M has HSIs such that every HSI contains just one sequence.
 - ii and iii prove that if M has an ADS, then HSIs are singletons as HSIs are SCSets. The other direction follows from the fact that an HSI of one sequence is an SVS because it separates the related state from the others. Every two HSIs contain a common prefix that separates the related pair of states. Therefore, the condition on the SVSs in the definition of ADS is also met.

The first three relations and construction algorithms of these sequences are discussed in [Sou14]. The relation iv is proposed in [HT15].

1.6.2 State Pair Array


Algorithms dealing with finite-state machines frequently need to handle pairs of states, or *state pairs*, and store information related to them. A *state pair array* (SPA) is introduced for this purpose. The content of an SPA can be arbitrary but it relates to a particular machine M with n states. The size of SPA is always $\frac{n \cdot (n-1)}{2}$ that is about a half of the size of state pair table that has n rows and n columns. An example of an SPA is in Figure 4.2. The previous version of SPA with different indexing was proposed in [Sou15]. The cells of SPA are indexed according to the following formula that enables to increase the number of states n without re-indexing the stored content. Before n is increased by 1, n cells are appended to the end of SPA; the cells relate to the state pair $(s_0, s_n), \dots, (s_{n-1}, s_n)$ where s_n is the added state. Note that SPAs are different from the Pairs Table used in [Gil62] to minimize a machine.

$$\text{GETSTATEPAIRINDEX}(i, j) \rightarrow \frac{j \cdot (j - 1)}{2} + i \text{ if } i < j \text{ else swap } i \text{ and } j \quad (1.1)$$



Part I

Construction of State Identification Sequences



Chapter 2

Introduction

Separating sequences are essential elements in both testing and learning of finite-state machines. One would not be able to distinguish any two different states without them. Although most of the testing methods and active-learning algorithms use separating sequences, their construction did not get much attention. It is obvious that to be efficient in testing or learning, one needs to be efficient in the construction of separating sequences. If a testing method or a learner requires a separating sequence of a state pair, then one may want to know the shortest separating sequences of all state pairs. However, a special type of separating sequences introduced in Section 1.6.1 is usually requested. The most common are a characterizing set (CSet) and harmonized state identifiers (HSI) as all minimal deterministic finite-state machines have these sequences for state identification. In their cases, it is not just about the length of individual separating sequences but also about the number of sequences in these sets. It was shown in [Sou15] that the use of an adaptive distinguishing sequence (ADS), another special type of grouped separating sequences, results in much smaller test suites. The reason is that only one sequence is needed for state identification. However, not every DFSM has an ADS. Note that a machine has an ADS if and only if it also has HSIs such that every HSI contains just one sequence as is showed in Section 1.6.1.

This part of the thesis is structured as follows. The next chapter describes the published approaches to the construction of separating sequences, CSet and HSIs. Then, a new extension of the existing construction algorithm of a splitting tree is proposed in Chapter 4. The extension allows one to easily construct a sequence that separates a state from a subset of states based on the splitting tree. Moreover, HSIs constructed from the splitting tree contain a small number of sequences and if there is an ADS, then HSIs are singletons. Chapter 5 describes experiments comparing the proposed algorithm to the standard ones in the construction of HSIs. Chapter 6 concludes this part of thesis.

■ 2.1 Research Questions

- I.1 Is there a technique that can construct a sequence identifying a state in a subset of states?
- I.2 Is there a technique constructing harmonized state identifiers that are more suitable for testing, that is, the total length of test sequences is shorter than using a standard HSI-construction method?

Chapter 3

Existing Methods

This chapter describes the existing algorithms that construct separating sequences, characterizing set (CSet) and harmonized state identifiers (HSI) of a minimal completely-specified deterministic finite-state machine (DFSM) M with n states and p inputs.

3.1 Construction of Separating Sequences

There are three methods for the construction of separating sequences in the literature. The oldest one derives separating sequences from a minimization procedure that divides states of a possibly unreduced DFSM into equivalence classes. The other two methods are quite recent and both aim to construct separating sequences directly. All three methods build for each state pair a separating sequence of minimal length.

3.1.1 Minimization Approach

The oldest construction method of separating sequences described in [Gil62, Algorithm 4.1] is based on the minimization algorithm proposed in [Gil62, Section 3.6].

The minimization algorithm partitions states of M using so-called P_k tables that have the same structure as the transition table of M , that is, rows represent states, columns relate to inputs and each cell contains the next state $\delta(s, x)$ of the related row s and column x . Table P_1 groups the states that have the same responses to all inputs. Next states in the cells are labelled with a group in which the particular next state is. Table P_{k+1} refines the partition given by the groups of P_k such that groups are divided according to different contents of the related rows. Labels of next states correspond to the groups of states in P_{k+1} . If no group can be refined, the minimization algorithm stops and thus each group represents a state of the reduced machine.

A separating sequence of states (s_i, s_j) is constructed from the P_k tables in two steps. At first, a table P_l is found such that s_i and s_j are in different groups in P_l but they are in the same group in P_{l-1} . The second step traverses each of the P_i tables for $i < l$ and appends symbols gradually to the separating

sequence w . The first symbol of the separating sequence w is the input of the column in which rows of P_l related to s_i and s_j have different labels. Then, l is decreased by 1 and the input of the column in which rows of P_l related to $\delta^*(s_i, w)$ and $\delta^*(s_j, w)$ have different labels is appended to w . This is repeated until l reaches 1, then the input x that produces different outputs from the related states, that is, $\lambda(\delta^*(s_i, w), x) \neq \lambda(\delta^*(s_j, w), x)$, is appended to w as the last symbol of separating sequence of (s_i, s_j) .

The time complexity of the construction of separating sequences of all state pairs is as follows.

$$O\left((n-1) \cdot np + \frac{n \cdot (n-1)}{2} ((n-1) \cdot 2 + (n-1) \cdot 2p)\right) = O(n^3 p)$$

At first, at most $(n-1)$ P_k tables of size $n \times p$ are created; $n-1$ is a well-known upper bound on the length of the shortest separating sequence for any two states of a minimal completely specified DFSM [Moo56]. Then, the groups of a particular state pair in up to all P_k tables are compared to find P_l and finally two rows are compared in l P_k tables such that $l \leq (n-1)$. This is performed for all $\frac{n \cdot (n-1)}{2}$ state pairs. Note that there are other minimization algorithms. For instance, the algorithm by Hopcroft [Hop71] runs in $O(np \log n)$ but then it is not easy to reconstruct separating sequences from it.

3.1.2 Shortest Separating Sequences

The shortest separating sequences (SSS) of all state pairs can be constructed directly without the need of minimization procedure as it was proposed in [Sou14]. Algorithm 1 amends the original SSS algorithm such that it can work with all the types of DFSM.

The idea of the algorithm is that all state pairs having a separating input are distinguished at first. Then, the rest of state pairs are distinguished successively by prepending an input symbol x to the separating sequence of a distinguished state pair to which the particular undistinguished state pair leads on x .

The algorithm handles three structures: a state pair array V for separating sequences, a queue *distinguished* of state pairs with a separating sequence already constructed, and a state pair array *transitionsTo* for storing predecessor state pairs. There are two phases in Algorithm 1. The first phase (line 1–18) separates state pairs that have a separating input. It also stores important data in both *distinguished* and *transitionsTo* that are used in the second phase (line 19–26) to process the undistinguished state pairs efficiently. Lines 3–12 in the first phase check whether the state pair has a separating input. If so, the state pair is pushed into *distinguished* and the next state pair can be processed. Otherwise, lines 13–18 stores the transition between state pair (s_i, s_j) and its successor state pair $(\delta(s_i, x), \delta(s_j, x))$ for each $x \in X$ if the state pairs are different. Moreover, if the successor state pair is already separated (line 18) which means that the state pair (s_i, s_j) can be separated by a sequence of length 2, then the other inputs are not processed. Lines

Algorithm 1: Construction of shortest separating sequences

input : A minimal DFSM M with n states
output : A state pair array V of the shortest separating sequences

```

1 foreach pair of states  $(s_i, s_j)$  such that  $i < j$  do
2    $idx \leftarrow \text{GETSTATEPAIRINDEX}(s_i, s_j)$ 
3   if  $M$  has state outputs and  $\lambda(s_i, \uparrow) \neq \lambda(s_j, \uparrow)$  then
4      $V[idx] \leftarrow \uparrow$ 
5      $\text{distinguished.push}(idx)$ 
6     continue
7   foreach  $x \in X$  do
8     if  $\lambda(s_i, x) \neq \lambda(s_j, x)$  then
9        $V[idx] \leftarrow x$ 
10       $\text{distinguished.push}(idx)$ 
11      break
12  if  $V[idx]$  is set then continue
13  foreach  $x \in X$  do
14    if  $\delta(s_i, x) \neq \delta(s_j, x)$  then
15       $\text{succIdx} \leftarrow \text{GETSTATEPAIRINDEX}(\delta(s_i, x), \delta(s_j, x))$ 
16      if  $idx \neq \text{succIdx}$  then
17         $\text{transitionsTo}[\text{succIdx}].\text{push}((idx, x))$ 
18        if  $V[\text{succIdx}]$  is set then break
19  $n_u \leftarrow |V| - |\text{distinguished}|$  // number of undistinguished state pairs
20 while  $n_u > 0$  do
21    $\text{succIdx} \leftarrow \text{distinguished.pop}()$ 
22   foreach  $(idx, x) \in \text{transitionsTo}[\text{succIdx}]$  do
23     if  $V[idx]$  is not set then
24        $V[idx] \leftarrow x \cdot V[\text{succIdx}]$ 
25        $\text{distinguished.push}(idx)$ 
26        $n_u \leftarrow n_u - 1$ 
27 return  $V$ 

```

22–26 in the second phase go through all stored predecessor state pairs of distinguished state pairs and if the predecessor is not separated, it is separated with the input leading to the distinguished state pair and the related separating sequence. As *distinguished* is a queue, the shortest separating sequences are created.

An example of the SSS algorithm's outcome is in Figure 4.2 where the state pair array contains the shortest separating sequences of the Mealy machine defined in Figure 4.1. The first phase finds the separating input 'a' for all state pairs including state B and the separating input 'b' for all other state pairs including state E. All distinguished state pairs are present in the queue *distinguished* and *transitionsTo* is filled with ((C,D),b) for (A,C), ((A,C),c) for

(A,D), ((A,D),c) for (A,E), ((A,C),a) and ((C,D),a) for (B,E), and ((C,D),a) for (D,E). State pairs (A,C) and (D,C) are separated with ‘aa’ in the second phase when the distinguished state pair (E,B) to which they lead on ‘a’ is processed. Similarly, (A,D) is separated with ‘cb’ as it leads on ‘c’ to (A,E) with separating sequence of ‘b’.

The SSS algorithm runs in $O(\frac{n(n-1)}{2} \cdot 2p + \frac{n(n-1)}{2} \cdot p) = O(n^2p)$. It passes through all states pairs and in the worst case all inputs are checked in the first phase. The state pair array *transitionsTo* that represents the inverse transition function δ^{-1} can contain at most $\frac{n(n-1)}{2} \cdot p$ pairs (*idx*, *x*) in total. The queue *distinguished* includes each state pair once and so it would enable to process all elements of *transitionsTo*, however, all elements are not necessary. The second phase stops immediately after all state pairs are separated. This is controlled by the counter n_u . Both main cycles (lines 1 and 20) are thus bounded by $O(\frac{n(n-1)}{2}p)$. Notice that *transitionsTo* is never filled up completely as some state pairs are separated by an input or the input is unsuitable for separating. There are also inputs that are not processed in the first phase due to lines 6, 11, 12 and 18 in Algorithm 1. Hence, the average time complexity is well below $O(n^2p)$. The algorithm was parallelized in [Sou15] where two parallel algorithms running in $O(np)$ are proposed. Almost identical algorithm was proposed in [HT16]; all state pairs as possible predecessor are checked with every input instead of the use of queue *distinguished* and storing the inverse transition function (lines 20–22). The difference in the implementations increases the complexity to $O(n^4p)$ but it allowed the authors to easily develop a parallel version of the algorithm. This parallel algorithm proposed in [HT16] is different from both parallel algorithms in [Sou15] and runs in $O(\frac{n^4p}{\Gamma})$ where Γ is the number of threads.

3.1.3 Splitting Tree with Minimal Separating Sequences

The state-of-the-art construction algorithm of the shortest separating sequences of all state pairs was proposed in [SMJ16]. It is called the ST-MSS algorithm in this thesis as it constructs separating sequences of minimal length. Its idea of constructing separating sequences based on a distinguished successors is shared with the SSS algorithm. However, the implementation is different. The ST-MSS algorithm does not work with individual state pairs but rather with subsets of states that are stored in a *splitting tree*. The use of the splitting tree together with an optimization trick improve the time complexity to $O(np \log n)$.

A splitting tree (ST) is a successor tree such that (i) each node is labelled with a subset of states, (ii) the root is labelled with the set of all states S , (iii) internal nodes also contain a separating sequence of the related subset of states, and (iv) the label of every parent is the union of disjoint sets of states labelling its children. A complete ST has its leaves labelled with singletons, that is, there are n leaves, each corresponds to one state. The splitting tree that is constructed for the Mealy machine defined in Figure 4.1 is shown in Figure 4.2.

The algorithm builds an ST in two phases that correspond to the ones of the SSS algorithm. In the first phase, an input symbol is assigned to each node of ST if it separates a state pair formed of states labelling the node. Initially, there is just the root of ST that contains all states. After a separating input x is chosen for a node r_i , the states in r_i are divided according to the response to x and they form labels of the children of r_i . The responses to the separating input label the related edges between r_i and its children. All nodes separated in the first phase are called 1-candidates as their separating sequences have the length of 1. The second phase checks all ‘predecessors’ of each k -candidate on every input. It starts with $k = 1$, then increases k by 1 after all k -candidates were checked and it stops if $k = n - 1$. Notice that there could be a better stop condition like n_u in the SSS algorithm. The check of predecessor is done using the precomputed inverse transition function δ^{-1} defined for each state s and input x as $\delta^{-1}(s, x) = \{s_i \in S \mid \delta(s_i, x) = s\}$. It is used such that the states of $\delta^{-1}(s, x)$ are located in a leaf of ST for each state s of the k -candidate r_i and each input x . Such a leaf is then possibly separated with $x \cdot w$ where w is the separating sequence of r_i . An optimization trick that lowers the time complexity from $O(n^2p)$ to $O(np \log n)$ is employed to find out if $x \cdot w$ is a separating sequence. Instead of traversing all states of r_i , the algorithm looks at the states in the children of r_i but the child with the most states is omitted. Each processed child r_c of r_i is then a basis for a temporary child of the leaves that contain the predecessor state of a state of r_c ; the label of edge from r_i to r_c becomes the label of edge connecting the temporary child to the related leaf. If a temporary child contains all states of its parent, then the sequence $x \cdot w$ is not separating and the temporary child is deleted. Otherwise, temporary children of the leaf r_j are made permanent as $x \cdot w$ is a separating sequence of r_j . A child needs to be added if the temporary children did not cover all states of r_j . Such states form the label of the added child and they are not covered by the temporary children because they lead on x to the states of the omitted child of r_i .

The algorithm of a splitting tree with minimal separating sequences was previously described in [Gil62, Section 4.10] which is not mentioned in [SMJ16]. The contribution of [SMJ16] is thus the efficient implementation of the construction of separating sequences based on a partial splitting tree that is already constructed. The name of splitting tree was introduced in [LY94] that uses the splitting tree for the construction of adaptive distinguishing sequence so the separating sequences do not have to be minimal. It will be discussed in a greater detail in the next chapter that introduces another algorithm based on splitting trees.

A separating sequence of a state pair (s_i, s_j) is obtained from the splitting tree by finding the lowest common ancestor of the leaves labelled with both states. This can be done by traversing the ST or by bookkeeping separating nodes for all state pairs as it is described in the next chapter.

can be done in $O(n^2)$ as V contains $\frac{n(n-1)}{2}$ sequences to collect. Together with the SSS algorithm a CSet is thus built in $O(n^2p + n^2)$.

The CSet of the Mealy machine in Figure 4.1 is formed of sequences ‘a’, ‘aa’, ‘cb’, ‘b’ that are collected from the state pair array shown in Figure 4.2.

3.2.4 From ST-MSS

Two approaches based on a splitting tree (ST) with minimal separating sequences (MSS) in its internal nodes (Section 3.1.3) were proposed with the same strategy to construct a CSet from the ST. Both [SMJ16] and [Gil62] form a CSet as a collection of all separating sequences stored in the ST. This can be done during the construction of ST so the time complexity remains $O(np \log n)$ in the case of ST-MSS algorithm [SMJ16].

The ST-MSS algorithm builds the splitting tree shown in Figure 4.2 for the Mealy machine in Figure 4.1 so that the CSet from that ST is ‘a’, ‘b’, ‘aa’, ‘cb’, which is the same as in the case of the construction from SSS.

3.2.5 Reduction of Characterizing Set

Characterizing sets constructed by any of the described methods are usually not minimal in terms that a sequence could be eliminated and the rest of the sequences would still form a characterizing set. This is mostly caused by adding a separating sequence that separates all state pairs as a shorter separating sequence that is already in the CSet. A step towards smaller CSet was proposed in [Gil62] where elimination of sequences that are prefixes of other sequences in the CSet is discussed. However, it is usually not sufficient.

There are several ways how a minimal CSet can be seen but in fact it is always a trade off between the minimal number of separating sequences in the CSet and the minimal length of sequences. Focusing on the minimal number of sequences would lead to the search for a preset distinguishing sequence or several incomplete preset distinguishing sequences that is a PSPACE-complete problem [LY94, HT15]. Hence, the described methods build CSets of the shortest separating sequences. The reduction task is then to choose a subset of these sequences such that the chosen sequences can still separate all state pairs. Unfortunately, this task is NP-complete as the NP-complete task Set Cover Problem (SCP) can be reduced to it. The universe to cover in SCP represents state pairs and the given subsets represent state pairs distinguished by a separating sequence, that is, each subset relates to one separating sequence. The task is to choose minimum number of subsets and still cover the universe by their union. NP-completeness of SCP is shown in [Kar72]. As a consequence, two approximation algorithms were proposed in [Sou14, Sou15] to reduce a CSet at least suboptimally in a reasonable time.

The first reduction algorithm called LS-SL is captured in Algorithm 2 that is an amended version of the original one proposed in [Sou14]. The algorithm passes through the given CSet W twice, from the longest to the shortest sequences at first and then in the reverse order, that is, from the shortest to the longest. In both passes it handles a set of *undistinguished*

state pairs that initially includes all state pairs not distinguished by their state outputs. If a sequence u does not separate any of undistinguished state pairs, then it is removed from W . Otherwise, it is checked whether the sequence u can be shortened such that it would still separate the same undistinguished state pairs. These state pairs separated by u are eliminated from *undistinguished* and the next sequence of W is processed. Finally, the *stOut* symbol is prepended to any sequence of W or added to W if it is needed, that is, there is a state pair that is not separated by any sequence of W . Note that this can happen as *undistinguished* is initialized such that \uparrow is assumed to be in W and if it is there, it is removed from W in the first pass as it would not separate any undistinguished state pair.

Algorithm 2: Reduction LS-SL of CSet

input : A characterizing set W to reduce

- 1 $undistinguished \leftarrow \{(s_i, s_j) \in S \times S \mid i < j \wedge \lambda(s_i, \uparrow) = \lambda(s_j, \uparrow)\}$
- 2 **foreach** $u \in W$ in the order from the longest to the shortest **do**
- 3 $separated_u \leftarrow \{(s_i, s_j) \in undistinguished \mid \lambda^*(s_i, u) \neq \lambda^*(s_j, u)\}$
- 4 **if** $separated_u$ is empty **then** erase u from W
- 5 **else**
- 6 get a prefix w of u that separates all state pairs of $separated_u$
- 7 **if** $w = u$ **then** $undistinguished \leftarrow undistinguished \setminus separated_u$
- 8 **else** replace u with w in W // w will be processed again
- 9 $undistinguished \leftarrow \{(s_i, s_j) \in S \times S \mid i < j \wedge \lambda(s_i, \uparrow) = \lambda(s_j, \uparrow)\}$
- 10 **foreach** $u \in W$ in the order from the shortest to the longest **do**
- 11 $separated_u \leftarrow \{(s_i, s_j) \in undistinguished \mid \lambda^*(s_i, u) \neq \lambda^*(s_j, u)\}$
- 12 **if** $separated_u$ is empty **then** erase u from W
- 13 **else**
- 14 get a prefix w of u that separates all state pairs of $separated_u$
- 15 $undistinguished \leftarrow undistinguished \setminus separated_u$
- 16 **if** $w \neq u$ **then** replace u with w in W
- 17 **if** $\exists s_i \neq s_j \forall w \in W : \lambda^*(s_i, w) = \lambda^*(s_j, w)$ **then**
- 18 prepend \uparrow to any $w \in W$ or add \uparrow to W if $W = \emptyset$

The LS-SL algorithm checks responses to each sequence u of W at most for all state pairs. All the presented algorithms create a CSet W with at most $\frac{n \cdot (n-1)}{2}$ sequences that are shorter than n . Therefore, the time complexity of the LS-SL algorithm is $O(2 \cdot \frac{n \cdot (n-1)}{2} \cdot (n-1) \cdot \frac{n \cdot (n-1)}{2}) = O(n^5)$. Nevertheless, the complexity was much lower in experimental evaluation.

The second reduction algorithm called EqualLen is described in Algorithm 3 that amends the original algorithm proposed in [Sou15]. It was developed in a reaction to the finding that the LS-SL reduction algorithm does not optimise well the sequences of the same length. The EqualLen algorithm thus improves the first pass of the LS-SL algorithm by dealing with all sequences of a particular length instead of just individual sequences. As

all of the described algorithms construct the shortest separating sequences, the given W is required to contain the shortest separating sequence for each state pair. For each sequence u of the CSet W , information regarding *undistinguished* state pairs is collected. In addition to observing which state pairs are separated by u , the algorithm counts state pairs that are separated by the last input of u . If there is no such state pair, u is eliminated from W . If arbitrary W was given, u would be needed to be shortened to the shortest prefix that separates all state pairs in *undistinguished* as it is done in the LS-SL algorithm. After the information for all sequences of the same length is obtained, sequences that separate the most states are gradually selected to remain in W . They are first compared on the number of undistinguished state pairs separated by the last input. If they equal, they are compared on the number of undistinguished state pairs separated by the entire sequence, and if they are equal even on this number, then the lexicographically lowest is selected to remain in W . The chosen sequence then updates the set of undistinguished state pairs which also influences information held for the rest of the sequences. Some sequences can become unnecessary as they do not separate any undistinguished state pair by the last input. Hence, they are eliminated from W . Finally, \uparrow is prepended to a sequence of W or added to W if W is empty. It can be needed because the set *undistinguished* was initialized with state pair that are not distinguished by \uparrow . The input \uparrow is handled differently than the other input symbols because the state is not changed after \uparrow is applied.

Compared to the LS-SL algorithm, the EqualLen algorithm passes the given CSet W just once. Nevertheless, the time complexity remains in $O(n^5)$ as again the responses to all separating sequences in W are observed for all state pairs.

This section is about CSet but sometimes one needs state characterizing sets (SCSet), for example, the Wp-method (Section 9.4) uses them to build a test suite. These sets can be created as harmonized state identifiers (HSI) described in the next section, however, they can then be reduced because they do not have to meet the requirements posed on the HSIs. Both described reduction algorithms can be easily adapted for SCSets. It is sufficient to work just with state pairs relating to the particular state for which the given SCSet is constructed.

3.3 Construction of Harmonized State Identifiers

Harmonized state identifiers (HSI) were shown to produce smaller test suites than using a charactering set (CSet) [Sou15]. Therefore, it is good to know how to construct them. This section describes four such construction approaches.

3.3.1 From CSet

One of the first formally described construction algorithms of HSIs was proposed in [LPB95, Appendix II]. It derives HSIs from the given characterizing

Algorithm 3: Reduction EqualLen of CSet

input : A characterizing set W such that \forall state pair $(s_i, s_j) \exists u \in W : u$ is the shortest separating sequence of s_i and s_j

- 1 $undistinguished \leftarrow \{(s_i, s_j) \in S \times S \mid i < j \wedge \lambda(s_i, \uparrow) = \lambda(s_j, \uparrow)\}$
- 2 **for** $len \leftarrow \max(\{|u| \mid u \in W\})$ **to** 0 **do**
- 3 $U_{len} \leftarrow \{u \in W \mid |u| = len\}$ such that $u = vx$ and $x \in X_{\uparrow}$
- 4 **foreach** $u \in U_{len}$ **do**
- 5 $separated_u \leftarrow \{(s_i, s_j) \in undistinguished \mid \lambda(s_i, u) \neq \lambda(s_j, u)\}$
- 6 $byLastInput_u \leftarrow \{(s_i, s_j) \in separated_u \mid \lambda(s_i, v) = \lambda(s_j, v)\}$
- 7 **if** $byLastInput_u$ is empty **then**
- 8 erase u from W and U_{len}
- 9 **while** U_{len} is not empty **do**
- 10 pop w from U_{len} with maximal $|byLastInput_w|$, if several comply, then with maximal $|separated_w|$, then the lexicographically lowest
- 11 $undistinguished \leftarrow undistinguished \setminus separated_w$
- 12 **foreach** $u \in U_{len}$ **do**
- 13 $separated_u \leftarrow separated_u \setminus separated_w$
- 14 $byLastInput_u \leftarrow byLastInput_u \setminus separated_w$
- 15 **if** $byLastInput_u$ is empty **then**
- 16 erase u from W and U_{len}
- 17 **if** $\exists s_i \neq s_j \forall w \in W : \lambda^*(s_i, w) = \lambda^*(s_j, w)$ **then**
- 18 prepend \uparrow to any $w \in W$ or add \uparrow to W if $W = \emptyset$

set W . At first, it constructs a HSI_0 of the initial state s_0 such that for each state $s_j, j \neq 0$, there is $w_j \in HSI_0$ that separates s_0 from s_j and $w_j \in \text{pref}(W)$ where $\text{pref}(W)$ is the prefix-closed set of W . Then, it builds HSI_i gradually for each state s_i such that: (i) $HSI_i \subseteq \text{pref}(W)$, (ii) for each state $s_j, j < i$, there is $w_j \in \text{pref}(HSI_i) \cap \text{pref}(HSI_j)$ that separates s_i from s_j , and (iii) for each state $s_j, j > i$, there is $w_j \in HSI_i$ that separates s_i from s_j . The given CSet constructed by any construction algorithm from Section 3.2 can have at most $\frac{n \cdot (n-1)}{2}$ sequences of length up to $n - 1$. As all sequences of W can be tried for each state, the time complexity of the construction of HSIs from the CSet is $O(n^4)$.

Harmonized state identifiers of all states of the Mealy machine defined in Figure 4.1 are captured in the comparison table in Figure 4.2. At first, ‘aa’ and ‘cb’ are chosen from $W = \{aa, b, cb\}$ to form the HSI_A . The input ‘a’ is sufficient to separate B from the other states and it is a prefix of ‘aa’ included in HSI_A , therefore, $HSI_B = \{a\}$. The state C is separated with ‘aa’ from A, B and D, but ‘cb’ also needs to be included in HSI_B as no other sequence separates C from E. HSIs of D and E are the same as HSIs of A and C, respectively.

3.3.2 From SSS

The harmonized state identifier of a state s is obtained from the state pair array of the shortest separating sequences (Section 3.1.2) by selection of all the sequences stored for state pairs (s, s_i) such that s_i is any state different from s [Sou15]. As the sequences are collected in a set, all duplicates are eliminated. If there is a sequence that is a prefix of another one in the constructed HSI, then such a sequence is eliminated as well. The construction of all HSIs from the state pair array of the shortest separating sequences runs in $O(n^2)$ as just $n - 1$ entries of the state pair array are visited for each state.

Figure 4.2 shows HSIs of all states of the Mealy machine defined in Figure 4.1. It also captures how the HSI of state C is formed from the state pair array of the shortest separating sequences.

3.3.3 From ST-MSS

Section 3.1.3 described how a splitting tree (ST) is built to contain minimal separating sequences (MSS) in its internal nodes. Besides the ST-MSS algorithm, the original paper [SMJ16] proposed how to form a HSI from the ST. As the HSI of a state s needs to contain a separating sequence of s and s_i for all states s_i different from s , all sequences of the internal nodes that include s in their label are collected to form the HSI of s . There are at most $n - 1$ such internal nodes and the procedure is repeated for all n states so that the construction of HSIs from the ST runs in $O(n^2)$.

HSIs of all states of the Mealy machine (Figure 4.1) are the same as in the case of the construction from SSS if the prefixes are eliminated. It is captured together with the corresponding splitting tree in Figure 4.2. There is another procedure for construction HSIs from an ST that constructs smaller HSIs (the third column of the comparison table in Figure 4.2). It will be described in the next chapter after the related restrictions on the ST are introduced.

3.3.4 From Incomplete Adaptive Distinguishing Sequences

A completely different approach to the construction of harmonized state identifiers (HSI) is proposed in [HT15]. The authors noticed a correspondence between HSIs and an adaptive distinguishing sequence (ADS) and used it to construct HSIs from so-called incomplete ADSs. A lot of machines do not have an ADS but they always have separating sequences that can form an *ADS of a subset of states*. As it does not distinguish all states, it is called an incomplete adaptive distinguishing sequence (IADS). A set of IADSs is thus needed to distinguish all states. Moreover, HSIs can be constructed only from a set of IADSs that is fully distinguishing which means that for each state pair (s_i, s_j) the set contains an IADS that separates s_i and s_j .

Unfortunately, the proposed greedy algorithm [HT15] is incorrect in some cases and its description is ambiguous. The inconsistencies of the algorithm are described in Appendix D.1. This section sketches the idea behind the

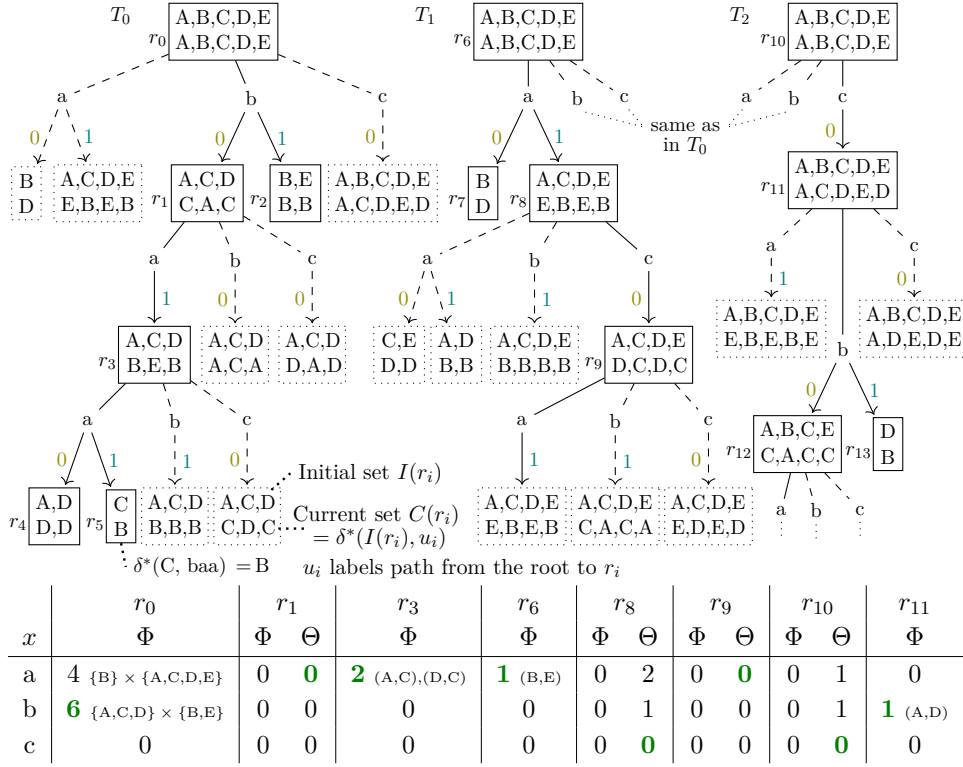


Figure 3.1: Construction of incomplete adaptive distinguishing sequences

into nodes r_1 and r_2 such that $C(r_1) = \{A,C\}$ and $C(r_2) = \{B\}$. Both nodes cannot be refined by any input, however, $|C(r_1)| > 1$ means that r_1 may be separated. As both the functions Φ and Θ return 0 for all inputs, the lowest input in the lexicographical order, input ‘a’, is chosen for r_1 . Fortunately, next input ‘a’ refines the new node r_3 . The successor cannot be further refined and so a new IADS T_1 is initialized because state pairs (A,D) and (B,E) are not distinguished. The set F of current sets that belong to nodes which cannot be refined should contain $\{B\}$ and $\{D\}$ as r_2, r_4 and r_5 cannot be refined (this is not specified in the proposed algorithm). Input ‘a’ is the only one that can refine the root r_6 of IADS T_1 such that an undistinguished state pair is separated. The remaining undistinguished state pair (A,D) obviously cannot be separated in T_1 as both states lead to state E on the chosen input ‘a’. Nevertheless, the algorithm commands to analyse both successors. Node r_7 cannot be refined so it should update F . Node r_8 could be refined but the last state pair cannot be separated. Hence, Θ estimates the most promising input; both ‘a’ and ‘b’ lead to nodes with the current states in F so that ‘c’ is chosen. Similarly to node r_1 , input ‘a’ is chosen for the new successor r_9 as it is lexicographically the lowest. However, the current state of the successor $\{E,B\}$ was observed on the path from the root and so the successor is not added to T_1 ; this is a safety check for cycles. The last IADS T_2 is sketched in Figure 3.1 up to the point when the separating sequence ‘cb’ of the last undistinguished state pair (A,D) is formed. The algorithm does not specify

how HSIs are obtained from the constructed IADSs. There can be several IADSs that separate a pair of states. If one adds the sequence separating of a particular state pair into the HSIs of both states when the state pair is distinguished for the first time, then HSIs formed for the Mealy machine in Figure 4.1 are listed in the comparison table in Figure 4.2.

The reported complexity of the algorithm is $O(nl(n^2p + n^2)) = O(n^4p)$ if each constructed IADS distinguishes two or more states not previously distinguished. The parameter l is an upper bound on the length of constructed IADSs; it is set to $n - 1$ to be possible to construct a fully distinguishing set of IADSs. The idea of IADSs was adopted in a parallel algorithm that has exponential time complexity in the worst case as it tries all sequences until all state pairs are distinguished [HT16].

Chapter 4

Splitting Tree with Invalid Inputs

Separating sequences for state identification are essential elements in testing and active learning. Both testing methods and learning algorithms need sequences that distinguish a particular state from the given subset of states and such sequences should be together minimal in some sense to be efficient in the testing or learning task. If the state can be separated by a sequence from the other states, then it is obvious that the aim is to have the shortest such sequence. However, if several sequences are needed to separate the state from others, then there is usually a trade-off between the number of separating sequences and their total length. This chapter proposes an extension to the existing algorithm such that one is able to construct efficiently a small number of separating sequences for any subset of states. The existing algorithm constructs a splitting tree (ST) for machines that possess an adaptive distinguishing sequence (ADS) that can be built from the constructed splitting tree [LY94]. The extension called ST-IADS constructs a splitting tree even for machines with no ADS. Before the extension is proposed in Section 4.4 and described on a running example in Section 4.5, its benefits are sketched in Section 4.1. Section 4.2 introduces the structure of splitting tree and what a valid input means. Algorithms for the construction of state identification sequences from the splitting tree are proposed in Section 4.3.

4.1 Motivation Example

The primary motivation to develop a new construction algorithm of separating sequences was that the new testing method (Chapter 11) and learning algorithm (Chapter 15) require sequences that distinguish the given state from the subset of states. Such sequences cannot be easily constructed by any method introduced in Chapter 3; the most suitable method for this task would be the algorithm for the construction of incomplete adaptive distinguishing sequences (Section 3.3.4), however, it was not known to the author of this thesis at the time when the proposed extension was developed. As the new method should construct separating sequences for any subset of states and the number of such sequences should be as small as possible, the task would reduce to the construction of an adaptive distinguishing sequence (ADS) if the given subset of states included all states. Fortunately, there is an algorithm

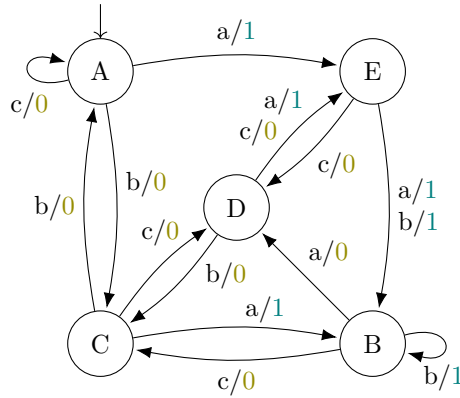


Figure 4.1: Mealy machine without an ADS

of complexity $O(pn^2)$ that constructs an adaptive distinguishing sequence of length up to $\frac{n(n-1)}{2}$ if there exists one [LY94]. The algorithm does not have to produce the shortest ADS but it is fast and based on a splitting tree that can easily provide required separating sequences for a subset of states. The only issue to solve is that the algorithm works only for machines with an ADS.

The structure of a splitting tree (ST) and the procedure constructing separating sequences for a subset of states led to a discovery that harmonized state identifiers (HSI) can be easily formed based on the splitting tree. Moreover, the number of sequences in HSIs is implicitly minimized as the approach tries to minimize the number of separating sequences for all subsets of states. This also follows from the relation between ADS and HSIs that a DFSM has an ADS if and only if it has HSIs that are singletons (Section 1.6.1). Note that HSIs are formed from the ST in a different way to how Section 3.3.3 describes.

The HSI construction methods described in Section 3.3 are compared with the approach based on the new extension ST-IADS on the Mealy machine defined in Figure 4.1. The machine has 5 states, A–E, 3 inputs, a–c, and 2 outputs, 0 and 1. It has no ADS and so some HSIs need to contain more than one sequence. Figure 4.2 shows the comparison of the constructed HSIs and two data structure used to construct HSIs. The SSS algorithm (Section 3.1.2) fills the state pair array with the shortest separating sequences of all state pairs as depicted in the top left of Figure 4.2. The orange lines show how the HSI of state C is formed from SSS. The splitting tree on the right of Figure 4.2 is created by both the ST-MSS algorithm (Section 3.1.3) and the new ST-IADS method. They follow different designs but for this small example the resulting splitting tree is the same. Nevertheless, HSIs constructed from the ST are different. HSIs from ST-MSS according to Section 3.3.3 (without prefixes) are equal to the ones formed from SSS. HSIs from ST-IADS are equal to the ones constructed from the CSet $\{aa, b, cb\}$ according to Section 3.3.1. The approach based on incomplete adaptive distinguishing sequences (Section 3.3.4) builds different HSIs than all four

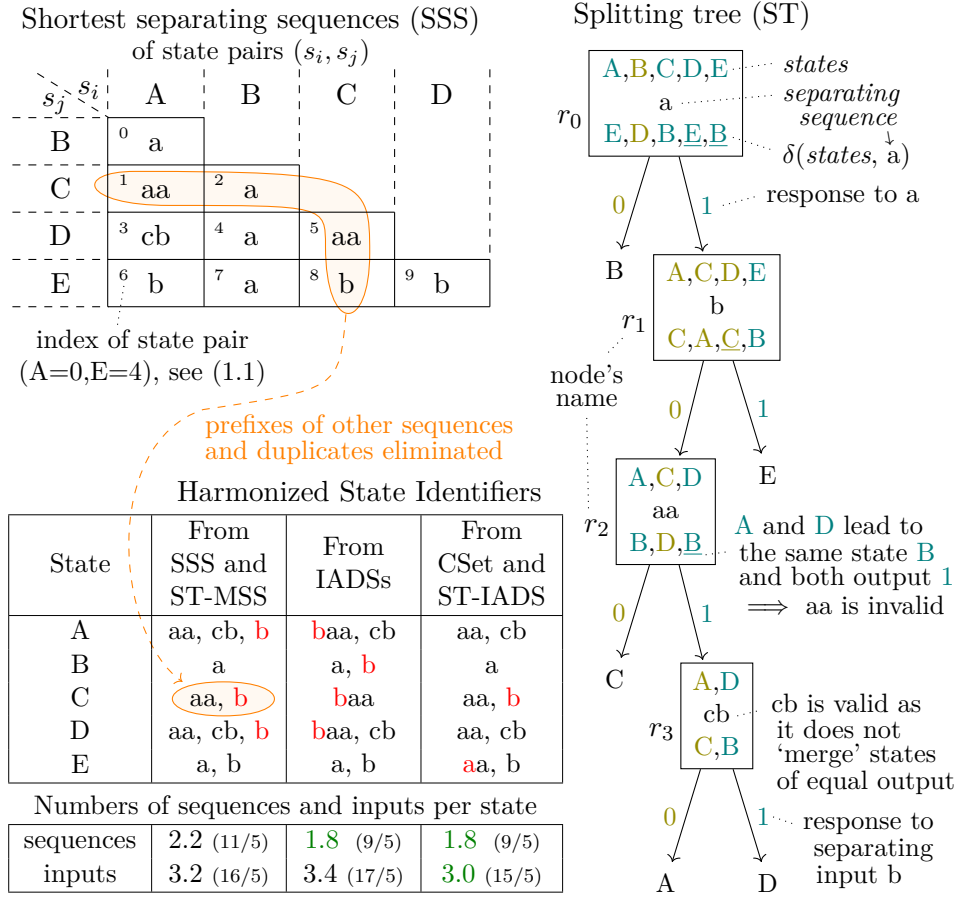


Figure 4.2: Construction of HSIs – a comparison of five different approaches

other methods. These three different HSIs constructed by five different approaches are listed in the comparison table in the bottom left of Figure 4.2; their differences are highlighted in red. The HSIs are compared on the average numbers of sequences and inputs per state. The numbers are below each of the three HSIs and the best values, that is, the minimal ones, are highlighted in green.

What does the primary motivation of a new construction method mean in the context of the example? Assume that state C needs to be separated from states A and E, and then state E is to be distinguished from states A and B. In the former case, one would like to obtain the state verifying sequence ‘baa’ of state C that uniquely identifies C amongst all states. Although state E has no state verifying sequence, sequence ‘aa’ separates it from A and B which is the requirement of the latter case. No HSI construction method produces both sequences for these two sets of states, nevertheless, they can be easily constructed from ST-IADS as Section 4.3 will describe. To sketch the construction of sequences from the splitting tree, consider a related learning scenario such that the machine is expected to be in one of states A, C or E and one wants to identify the state. The node of the ST that contains all

the states A,C and E, and has the smallest number of states is found and its separating sequence is applied to the machine. According to the observed response, a subset of states in which the machine can be after that sequence is considered in the next step. In the example, the separating sequence ‘b’ of node r_1 is queried and let assume that the response is 0. Then the machine is in state C or A if it was in A or C, respectively. The same procedure is repeated with this subset of two states, that is, r_2 is used as a separating node of C and A and so ‘aa’ is queried. According to the response, one can thus derive the state the machine was in at the beginning of the identification process and the state it entered at the end.

4.2 Splitting Tree for Incomplete Adaptive Distinguishing Sequences

Splitting tree (ST) as a data structure holding refinement of the set of states is used in many algorithms. It was sketched in Section 3.1.3 but this section describes it in detail and introduces the requirements that need to hold to create an adaptive distinguishing sequence (ADS) based on the ST.

The notion of splitting tree was introduced in [LY94], however, such a tree without the requirements on included sequences was previously described in [Gil62] as well. A splitting tree (ST) is a successor tree such that:

- each node is labelled with a subset of states S' ,
- the root is labelled with the set of all states S ,
- each internal node has assigned a sequence w that separates S' ,
- the label of every parent is the union of disjoint sets of states labelling its children, and
- each edge to child r is labelled with the response of states of r to the sequence w assigned to the parent of r .

An ST is complete if all states are separated, that is, if there are exactly n leaves and each corresponds to one state. Note that the labels of leaves of an ST always form a partition of all states. For easier access in the implementation, each internal node also possesses the set of next states $\delta^*(S', w)$. If the separating sequences in internal nodes are minimal, then it is sufficient to label the edges by an output symbol in response to the separating input that is the last input of w .

4.2.1 Input Validity Types

An adaptive distinguishing sequence cannot be constructed from an arbitrary splitting tree. All sequences of the ST need to be composed of valid inputs. An input x is *valid* for a subset of states S' if every two distinct states s_i and s_j of S' either respond differently to x , $\lambda(s_i, x) \neq \lambda(s_j, x)$, or they lead to

different states on x , $\delta(s_i, x) \neq \delta(s_j, x)$. Therefore, an input x is *invalid* for S' if there are two different states in S' such that both respond equally to x and both lead to the same state. There are three types of a valid input x with respect to the partition π given by the leaves of ST as proposed in [LY94]:

- a) Two or more states of S' respond with different outputs to input x , that is, $|\lambda(S', x)| > 1$.
- b) All states of S' respond with the same output, $|\lambda(S', x)| = 1$, but they lead to more than one block of π on x , that is, states of $\delta(S', x)$ are not in a single block of π .
- c) Neither of the above, that is, all states produce the same output and lead to the same block of π .

Inputs can also be divided by their ability to separate some states regardless of their validity. An input x is called *separating* if two or more states of the subset of states S' respond differently, that is, $|\lambda(S', x)| > 1$. Otherwise, the input x is called *transferring*. This corresponds to the notion of the shortest separating sequence of a state pair. The shortest separating sequence is always formed of transferring inputs followed by a single separating input. Note that every valid input of type a) is separating and valid inputs of type b) and c) are transferring.

An example of a splitting tree with valid and invalid separating sequences is shown on the right of Figure 4.2. If a sequence contains an invalid input, it is called *invalid*. The separating sequences of nodes r_0, r_1 and r_2 are invalid because they *merge* some states. Two states are merged by a sequence u if they respond equally and lead to the same state on u . The second and other occurrences of a state are underlined in the set of next states in each node with an invalid sequence in Figure 4.2. Only the sequence ‘cb’ of node r_3 is valid as it does not merge any states. Notice that the first symbols of sequences ‘aa’ and ‘cb’ of nodes r_2 and r_3 are transferring which will be more obvious in Section 4.5 where the construction of the ST is described on the example.

A splitting tree that contains an invalid sequence cannot be the basis for an ADS but several incomplete adaptive distinguishing sequences (IADS) can be formed from it. Therefore, such a splitting tree is referred to as ST-IADS.

4.3 State Identification Sequences From ST-IADS

Splitting tree that is complete can be a basis for the construction of separating sequences of all state pairs (Section 3.1.3), characterizing sets (Section 3.2.4) or harmonized state identifiers (Section 3.3.3), however, there is alternative construction approach than just collecting particular sequences of ST nodes. The approach simply follows the chosen separating sequence and extends it by another one that separated the reached next states. This is repeated until no

states can be separated. By doing this, new longer separating sequences can be formed and subsequently the number of needed sequences can be reduced. This is significant for testing of finite-state machines because HSIs are often applied at the end of every test sequence, so halving their number also halves the amount of testing.

Algorithm 4 describes how to obtain a sequence from a given ST that separates the given state s from all other states of the given subset of states S' . The state s is separated by the returned sequence from all the other states in S' in the best case which depends on separating sequences stored in the ST. At first, the lowest common ancestor of the leaves corresponding to states in S' including s is located in the ST by calling the function `GETSEPARATINGNODE` defined in Algorithm 5. A state pair array *separatingNodes* that is a part of the ST in the implementation helps with this search. It stores the node of ST that separates the associated state pair, for example, r_1 is stored for state pair (A,E) and $separatingNodes[(C,D)] = r_2$ of the ST in Figure 4.2. It is sufficient to compare separating nodes of $(|S'| - 1)$ state pairs that relate to a chosen state $s_k \in S'$. The lowest common ancestor r for S' is the one of compared nodes that contains the most states because it is the root of subtree that contains the leaves corresponding to all the states of S' . This follows from the property of ST that the label of every parent is the union of disjoint sets of states labelling the children. The separating sequence of r is then appended to w that stores the resulting separating sequence for s ; w is initially empty. The procedure of locating the lowest common ancestor for S' and appending its separating sequence is then repeated with updated S' and s until S' contains just s . The subset of states S' is replaced with the states reached by the separating sequence of r from the states of S' that responded with the same output as the reference state s to the separating sequence of r . The reference state s is then transferred to its next state as well. Now it should be clear how the separating sequences ‘baa’ and ‘aa’ for the subsets of states $\{A,C,E\}$ and $\{A,B,E\}$ mentioned in the end of Section 4.1 are obtained.

Algorithm 4: `GETSEPARATINGSEQUENCEFROMST($s \in S', S' \subseteq S, ST$)`

```

1  $w \leftarrow \varepsilon$ 
2 while  $|S'| > 1$  do           // there is  $s_i \in S'$  not separated from  $s$ 
3    $r \leftarrow \text{GETSEPARATINGNODE}(S')$ 
4    $v \leftarrow r.separatingSequence$ 
5    $w \leftarrow w \cdot v$ 
6    $S' \leftarrow \{\delta^*(s_i, v) \mid s_i \in S' \wedge \lambda^*(s_i, v) = \lambda^*(s, v)\}$ 
7    $s \leftarrow \delta^*(s, v)$ 
8 return  $w$ 

```

The time complexity of Algorithm 4 depends on the number n_d of states in the given S' that are separated from the given s ; $n_d < |S'|$. If all different states of S' are distinguished from s , $n_d = |S'| - 1$. The algorithm separates at least one state of S' in every main cycle (lines 2–9) so it does at most

Algorithm 5: GETSEPARATINGNODE($S' \subseteq S$)

```

1 select a pivot  $s_k$  from  $S'$ 
2 return the node of  $ST.separatingNodes[(s_k, s_i)]$ ,  $s_k \neq s_i \in S'$ , with the
   most states

```

n_d cycles, and it compares at most $|S'|$ separating nodes to find the lowest common ancestor r by GETSEPARATINGNODE(S'). Therefore, it runs in $O(n_d \cdot |S'|) = O(|S'|^2)$. Considering that the next states are stored in r , updating S' does not increase time complexity.

It is easy to build harmonized state identifiers with Algorithm 4 at hand. The HSI of a state s_k is formed by successive calls GETSEQUENCEFROMST(s_k, S', ST) for subset S' of states that are not separated from s_k by a sequence which was already added to the HSI. In such a way, Algorithm 6 constructs an HSI for each state. If the constructed HSI of a state contains just one sequence, then its construction takes $O(n^2)$ which follows from the time complexity of GETSEQUENCEFROMST. The complexity does not increase even if the constructed HSI contains several sequences because the sum of the numbers n_d of distinguished states for each call of GETSEQUENCEFROMST is equal to n . Therefore, Algorithm 6 constructs HSIs of all states from the ST in $O(n^3)$.

Algorithm 6: GETHARMONIZEDSTATEIDENTIFIERSFROMST(ST)

```

1 foreach  $s_k \in S$  do
2    $HSI_k \leftarrow \emptyset$ 
3    $S' \leftarrow S$ 
4   while  $|S'| > 1$  do // there is  $s_i \in S'$  not separated from  $s_k$ 
5      $w \leftarrow$  GETSEQUENCEFROMST( $s_k, S', ST$ )
6     add  $w$  to  $HSI_k$ 
7      $S' \leftarrow \{s_i \in S' \mid \lambda^*(s_i, w) = \lambda^*(s_k, w)\}$ 
8 return  $HSI$  as a collection of  $HSI_k$  of all states

```

The construction method of an adaptive distinguishing sequence (ADS) from a complete ST was proposed in [LY94] and Algorithm 7 extends it to work even if the ST contains invalid separating sequences. A set of incomplete adaptive distinguishing sequences (IADS) is thus returned in general instead of a single ADS. The algorithm returns an ADS if the complete ST has no invalid sequence. The idea is the same as in GETSEQUENCEFROMST but the separating sequences are stored in nodes of the tree representing IADS instead of appending them one after another. All responses (different branches) are handled as there is no reference state s .

The tree structure of IADS is recalled before Algorithm 7 is described in detail. An IADS can be represented by a successor tree such that:

- each node r_j is labelled with the initial set $I(r_j)$, the current set $C(r_j)$ and an input x_j ,

- all edges leading from an internal node r_j are labelled with distinct output symbols produced by states of $C(r_j)$ in response to x_j , and
- if $\lambda^*(s, u)$ labels the path from the root r_0 to a node r_j where $s \in I(r_0)$ and u is the input sequence formed of x_i 's on the path (without x_j of r_j), then the state s is in $I(r_j)$ and $\delta^*(s, u)$ is in $C(r_j)$.

This definition implies that for every root r_0 holds $I(r_0) = C(r_0)$. Nevertheless, $I(r_0) = S$ is not required here for IADSs compared to the description in Section 3.3.4. It is required only for ADSs that need to distinguish all states and so they have exactly n leaves. The omitted requirement on IADSs is transferred to the following property of a set of IADSs. A set D of IADSs is *fully distinguishing* if every pair of distinct states is distinguished by some IADS from D [HT15]. This enforces a construction algorithm to include every state in at least one root of IADS. The length of an IADS is the depth of its tree representation.

Algorithm 7: GETIADSSFROMST(ST)

```

1 push  $S$  into undistinguished
2 foreach  $S' \in$  undistinguished do
3   create a new IADS  $T_i$  with the root  $r$  such that  $I(r) = C(r) = S'$ 
4   push  $r$  into unprocessedNodes
5   foreach  $r_j \in$  unprocessedNodes do
6      $r^{ST} \leftarrow$  GETSEPARATINGNODE( $C(r_j)$ )
7      $u \cdot x_j \leftarrow r^{ST}.separatingSequence$ 
8     foreach  $x_k \in u$  in their order do
9        $r_j.input \leftarrow x_k, \quad y \leftarrow \lambda(C(r_j), x_k)$ 
10      create a successor  $r_k$  with the edge from  $r_j$  labelled with  $y$ 
11       $I(r_k) \leftarrow I(r_j), \quad C(r_k) \leftarrow \delta(C(r_j), x_k)$ 
12       $r_j \leftarrow r_k$ 
13     $r_j.input \leftarrow x_j$ 
14    foreach  $y \in \lambda(C(r_j), x_j)$  do
15      create a successor  $r_k$  with the edge from  $r_j$  labelled with  $y$ 
16       $C(r_k) \leftarrow \{\delta(s_j, x_j) \mid s_j \in C(r_j) \wedge \lambda(s_j, x_j) = y\}$ 
17       $I(r_k) \leftarrow \{s_j \in I(r_j) \mid \delta^*(s_j, d_k) \in C(r_k) \text{ where } d_k \text{ is the}$ 
           sequence of  $x_i$ 's along the path from the root of  $T_i$  to  $r_k\}$ 
18      if  $|C(r_k)| > 1$  then push  $r_k$  into unprocessedNodes
19      else if  $|I(r_k)| > 1$  then push  $I(r_k)$  into undistinguished
20 return IADSs  $T_i$ 's
  
```

Algorithm 7 uses two queues to construct a fully distinguishing set of IADSs. The first queue called *undistinguished* includes subsets of states that are not distinguished from each other by any constructed IADS; it contains initially the set of all states. The second queue called *unprocessedNodes* is for the leaves r_j of the current IADS T_i such that their current sets $C(r_j)$ contain several

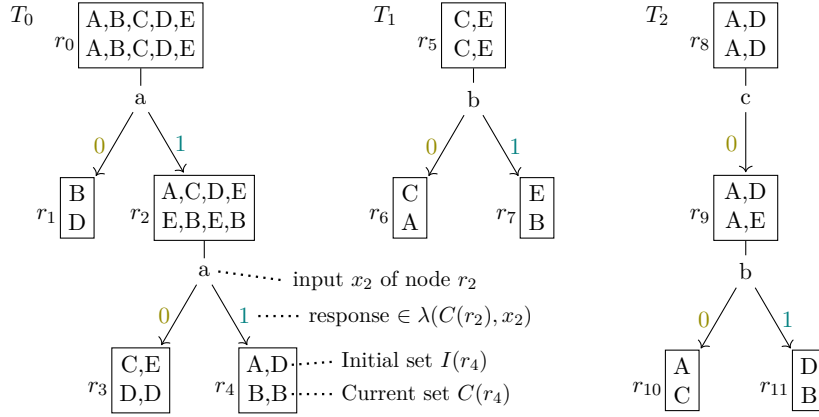


Figure 4.3: Incomplete adaptive distinguishing sequences from ST-IADS

states which means that they can be separated. Each root of a new IADS T_i is initialized with a subset of states from *undistinguished* and pushed into *unprocessedNodes* to start the construction of T_i . For each unprocessed node r_j of T_i the lowest common ancestor is found by `GETSEPARATINGNODE` in ST and its separating sequence is then divided into the transferring sequence u and the separating input x_j . If u is not empty, a chain of successors representing u is appended to r_j and r_j then points to the new leaf. All these successors have the same initial state and differ in the current sets that are updated by each input of u . The separating input x_j is assigned to the leaf r_j and divides its initial and current sets according to the responses. Each successor r_k corresponds to the states of $C(r_j)$ that produce the same output to x_j . Their initial and current sets are updated accordingly (lines 16 and 17 of Algorithm 7). The initial and current sets are implemented as arrays so the correspondence between initial and current states is easily accessible, and the node r^{ST} with its successors in the splitting tree provides enough information to form successors of r_j and their current sets. Finally, if the successor r_k can be further separated, then it is added to *unprocessedNodes*. If r_k cannot be separated but its initial set contains several states, then these states, that were not distinguished, are pushed into *undistinguished*. A fully distinguishing set of IADSs constructed by Algorithm 7 from the ST in Figure 4.2 is shown in Figure 4.3. There is usually no need to store the chain of successors representing the transferring inputs. Therefore, a shortened version of IADSs can be introduced such that each node stores a separating sequence instead of a single input as proposed in [Sou14]. Then, lines 8–12 of Algorithm 7 would be omitted and lines 13–17 would work with the entire separating sequence $u \cdot x_j$ instead of just with x_j .

At most $n - 1$ separating sequences are needed to distinguish all states. A suitable separating sequence is found in the ST by `GETSEPARATINGNODE` in $O(n)$. Therefore, Algorithm 7 runs in $O(n^2)$ if the shortened version of IADSs is constructed. Otherwise, it also depends on the length of separating sequences in the ST because the chain of successors corresponding to each

sequence needs be created; the time complexity is $O(n^3)$ if all separating sequences have the length at most n .

All three algorithms constructing state identification sequences work with any splitting tree, however, `GETIADSSFROMST` can build an ADS only from the ST that has valid separating sequences. The ST-MSS algorithm (Section 3.1.3) thus cannot be used to prove the existence of an ADS and generally it results in more sequences than by the use of ST-IADS. A characterizing set can be formed from the ST as the collection of all separating sequences (Section 3.2.4) and also as a union of HSIs or sequences of IADSSs, which is equivalent.

4.4 ST-IADS Construction Algorithm

This section proposes the extension to the existing algorithm [LY94] such that the existing algorithm constructing a splitting tree with valid separating sequences is not disrupted by the extension. It means that the extension is not employed if the given machine has an adaptive distinguishing sequence (ADS) and so a splitting tree with only valid sequences is constructed. As the extension allows one to create incomplete adaptive distinguishing sequences (IADS), it is referenced as the ST-IADS algorithm and its part corresponding to the existing algorithm from [LY94] is referenced by the ST-ADS algorithm.

The ST-IADS algorithm is described in Algorithms 8–12 such that Algorithm 8 captures the main part and the others handle the case when no valid separating input is found for a subset of states. The algorithm is first roughly sketched and then discussed in detail in the following paragraphs. **The idea of the algorithm is to gradually refine the partition of states given by the labels of leaves until all leaves contain only one state.** Compared to the ST-MSS algorithm (Section 3.1.3) that processes the leaves containing several states in the order of the length of their expected separating sequence, the ST-ADS orders the leaves for processing according to the number of states in their labels. This helps with the search for a valid separating sequence of a subset of states. If the states of a leaf have a separating input, then it is used to refine the subset of states by appending the corresponding successors to the leaf. Otherwise, the leaf is stored in a collection of leaves that also need a separating sequence of several inputs. These leaves are then processed in a similar manner as the SSS algorithm (Section 3.1.2) does, that is, links between them are found first and subsequently if one is separated, it can separate the others that have a link to it. In other words, when a subset r_j of states is separated with a sequence w , another subset r_i can be separated with the sequence $x \cdot w$ where x labels the link from r_i to r_j , that is, states of r_i transfer to states of r_j on x . In the case of the ST-ADS algorithm, the links, or transitions between subsets of states, are restricted to valid inputs. Therefore, some leaves do not have to be distinguished as they do not have a valid separating sequence. The extension is thus employed such that even invalid inputs are considered for the links from the undistinguished leaves. As the extension is designed to

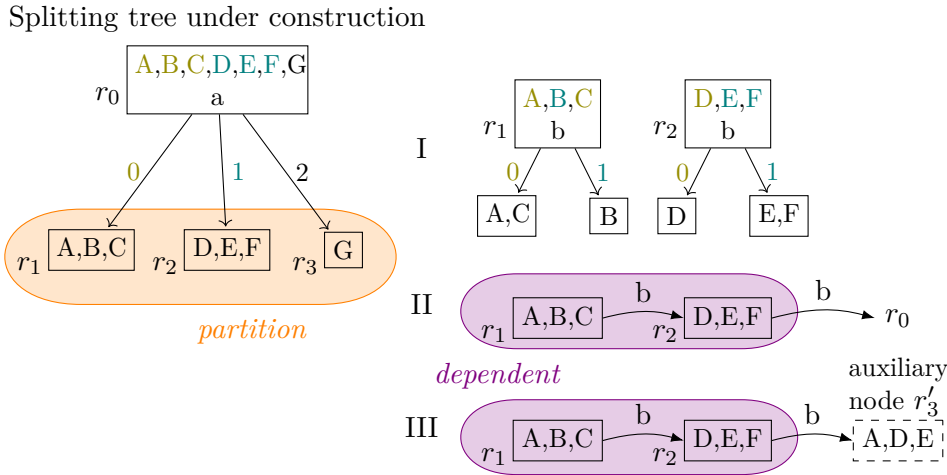


Figure 4.4: Cases I–III of splitting a node during the construction of a ST

find the best invalid separating sequence according to a particular score if there is no valid one, it explores all the shortest invalid separating sequences. Hence, auxiliary nodes representing the subsets of states reached by these sequences are created; they are also stored after they are analysed as they may be reused later.

Figure 4.4 shows a hypothetical construction of a splitting tree for a machine with 7 states A–G. Three possible ways how a separating sequence can be formed and a node be split are explained on this small example. After the input ‘a’ separates some states of the root of the splitting tree, there are three leaves included in a set called *partition*, see the splitting tree on the left of Figure 4.4. The nodes r_1 and r_2 are processed first as they have the most states amongst the leaves. If there is a valid input of type a) that separates some states, the node is split immediately as the case I on the right of Figure 4.4 shows. If there is no separating input for a node, the node is pushed in the set *dependent*. The function INITTRANSONVALIDINPUTS then creates links between nodes according to the transition function. Assume that the next states of A, B and C on the input ‘b’ are states D, E and F, respectively, and D, E, F lead to A, D, E on ‘b’. Therefore, the cases II and III show a link ‘b’ from r_1 to r_2 . The link ‘b’ from r_2 first points to r_0 because it includes all states A, D, E. If the sequence assigned to r_0 was invalid, then an auxiliary node r'_3 only with states A, D, E would be created in order to maximise the chance of discovery of a valid separating sequence. In the case II, ‘a’ is a valid input for r_0 and so the link ‘b’ points from r_2 to r_0 . The function PROCESSDEPENDENT forms the separating sequence ‘ba’ for r_2 and then ‘bba’ for r_1 and splits the nodes accordingly. In the case III, separating sequences for nodes r_2 and r_1 would be constructed in a similar way using a separating sequence for the auxiliary node r'_3 . It can happen that there is no valid input for r'_3 and thus no valid sequence for r_1 and r_2 . Both nodes then remain in *dependent* and the function INITTRANSONINVALIDINPUTS creates links

between nodes even for invalid inputs. The next call of `PROCESSDEPENDENT` then finds the best invalid sequences for the nodes and splits them.

There are several structures that the algorithm handles. Every node r of the splitting tree ST contains a set of *states*, a *separatingSequence* and the associated next states $\delta^*(r.states, r.separatingSequence)$. A state pair array *separatingNodes* as a part of ST stores for each state pair (s_i, s_j) a node r of ST that separates s_i and s_j , that is, both s_i, s_j are in $r.states$ but they are in different *states* of the children of r . A set *partition* represents the leaves of ST ; it is implemented as a priority queue such that the leaves with the most states in *states* are on the top. Nodes with *states* that do not have a separating input are stored in an array *dependent*. An array of lists *transitionsTo* is filled with the links between these nodes before they are processed in the order given by a priority queue *dependentPriorityQueue*. For each node r in *dependent*, $best_r$ stores an *input*, the node *next* reached by the *input* and a *score* reflecting how good is to start the separating sequence with *input*. The $best_r.score$ is then used when r is pushed into *dependentPriorityQueue* that favours nodes with the lowest score. Besides a minimal DFSM M , the ST-IADS algorithm takes an input parameter *validOnly* that when true forces the algorithm to follow the existing ST-ADS algorithm. Hence, null representing no splitting tree is returned if M has no ADS and *validOnly* is true.

Algorithm 8 starts with separating all states in the root of ST using the `stOut` input \uparrow if the given machine M produces state outputs. Separating a leaf r consists of appending new nodes to r and updating *partition* and $ST.separatingNodes$ if r is a part of ST . It is described by the function `SEPARATE` on lines 5–9 of Algorithm 8. New successors of r are formed of states of r that respond to the separating sequence of r in the same way; the last output symbol of their response, that is, the response to the separating input, labels the edge from r to the successors. The condition on line 7 of Algorithm 8 ensures that only successors of r present in ST lead to updates to *partition* and $ST.separatingNodes$. This is necessary because the update is not desired when auxiliary nodes are separated during the search for the best invalid separating sequence. After *partition* is initialized with the current leaves of ST , the main cycle (lines 11–35) starts. In each cycle, one leaf r with the most states is processed such that it is separated if it has a separating input. Otherwise, it is stored in *dependent*. An exception is if r has no valid input and valid inputs are required by *validOnly*. Then the ST-IADS algorithm returns null as the sign that there is no ADS for the given machine. To find out if r has a valid separating input, r is first analysed. The function `ANALYSE(r)` defined on lines 14–20 checks each input x and stores it as the separating sequence of r if it is a valid separating input. Otherwise, all x and the related next states $\delta(r.states, x)$ are remembered if x is valid or invalid inputs are allowed. Invalid transferring inputs that merge all states into one, that is, $|\delta(r.states, x)| = 1$, are not stored as they cannot begin a separating sequence. The leaf r could have already been analysed as an auxiliary node so the analysis is not repeated due to the condition on line 13. After all leaves of ST with the same number of states were analysed and checked, those

Algorithm 8: Construction of a splitting tree (ST-IADS)

```

input : A minimal DFMSM  $M$  with  $n$  states
input : validOnly allows to use only valid inputs if true
output: A splitting tree  $ST$  of  $M$  or null if validOnly and  $M$  has no ADS

1  $r.states \leftarrow$  all  $n$  states of  $M$  //  $r$  is initially the root of  $ST$ 
2  $ST.separatingNodes[(s_i, s_j)] \leftarrow$  null for all state pairs  $(s_i, s_j)$ 
3 if  $M$  has state outputs then
4    $r.separatingSequence \leftarrow \uparrow$ 
5   SEPARATE( $r$ ):
6     create successors of  $r$  by grouping states of  $r.states$  with the same
       response to  $r.separatingSequence$ 
7     if  $r$  is in  $ST$  then
8       add the successors to partition
9        $ST.separatingNodes[(s_i, s_j)] \leftarrow r \ \forall s_i, s_j$  in different successors
10 else  $partition \leftarrow \{r\}$  // if  $M$  is a Mealy machine
11 while  $|partition| \neq n$  do
12    $r \leftarrow$  pop a node from partition with the most states
13   if  $r$  is not analysed then
14     ANALYSE( $r$ ):
15     foreach input  $x \in X$  do
16       if  $x$  is a valid separating input then
17          $r.separatingSequence \leftarrow x$ 
18         break
19       else if  $x$  is a valid transferring input or (not validOnly
20         and ( $|\lambda(r.states, x)| > 1 \vee |\delta(r.states, x)| > 1$ )) then
21         store  $x$  and next states  $\delta(r.states, x)$ 
22   if validOnly and  $r$  has no valid input then return null // no ADS
23   if  $r.separatingSequence$  is assigned then SEPARATE( $r$ )
24   else push  $r$  into dependent
25   if  $|dependent| > 0$  and  $\forall p \in partition: |p.states| < |r.states|$  then
26     foreach  $r \in dependent$  do
27       INITTRANSITIONSONVALIDINPUTS( $r$ )
28     PROCESSDEPENDENT()
29     if  $|dependent| > 0$  then
30       if not validOnly then
31         foreach  $r \in dependent$  do
32           INITTRANSITIONSONVALIDINPUTS( $r$ )
33           if  $r$  has no valid separating sequence then
34             INITTRANSITIONSONINVALIDINPUTS( $r$ )
35           PROCESSDEPENDENT()
36   else return null // no ADS
37 return  $ST$ 

```

pushed into *dependent* are separated by sequences of several inputs (lines 25–35). At first, only valid separating sequences are used such that links on valid inputs are prepared by INITTRANSITIONSONVALIDINPUTS (Algorithm 9) for every node in *dependent* and then they are gradually processed by PROCESSDEPENDENT (Algorithm 10). If some of them are still not separated after that (they remain in *dependent*), either the process is repeated using invalid inputs or the algorithm exits with null if only valid inputs are allowed. The function INITTRANSITIONSONVALIDINPUTS on line 31 is called for every node of *dependent* but the implementation calls it only for new auxiliary nodes that are added to *dependent* by INITTRANSITIONSONVALIDINPUTS or by INITTRANSITIONSONINVALIDINPUTS (Algorithm 11). As the ST-IADS algorithm has the property that each node is separated by a valid sequence if there is a valid one, INITTRANSITIONSONINVALIDINPUTS is called only if no valid separating sequence was found so far for r .

Algorithm 9: INITTRANSITIONSONVALIDINPUTS(r)

```

1 ( $best_r.input, best_r.next, best_r.score$ )  $\leftarrow$  (null, null,  $\infty$ )
2 foreach valid transferring input  $x$  of  $r$  do
3    $r_x \leftarrow$  GETSEPARATINGNODE( $\delta(r.states, x)$ )
4   if  $r_x \in dependent$  then
5     | add  $(r, x)$  to  $transitionsTo[r_x]$ 
6   else if separating sequences of  $r_x$  and  $best_r.next$  are not valid and
7     |  $|r.states| < |r_x.states|$  then
8     |  $r_x \leftarrow$  a (stored or new) node with states equal to  $\delta(r.states, x)$ 
9     | if  $r_x$  is not analysed or  $r_x.separatingSequence$  is not set then
10    |   | if  $r_x$  is not analysed then
11    |   |   | ANALYSE( $r_x$ )
12    |   |   | if  $r_x.separatingSequence$  is assigned then
13    |   |   |   |  $best_r \leftarrow (x, r_x, SCORE(r, x, r_x))$ 
14    |   |   |   | push  $r_x$  into dependent if it is not there
15    |   |   |   | add  $(r, x)$  to  $transitionsTo[r_x]$ 
16    |   |   |   | else if  $SCORE(r, x, r_x) < best_r.score$  then
17    |   |   |   |   |  $best_r \leftarrow (x, r_x, SCORE(r, x, r_x))$ 
18    |   |   |   | else if  $SCORE(r, x, r_x) < best_r.score$  then
19    |   |   |   |   |  $best_r \leftarrow (x, r_x, SCORE(r, x, r_x))$ 
19 store  $best_r$ 
20 if  $best_r.next$  has a valid separating sequence then
21 | push  $r$  with  $best_r.score$  into dependentPriorityQueue

```

Algorithm 9 describes the function INITTRANSITIONSONVALIDINPUTS that initializes $best_r$ of the given node r and links from r on each valid transferring input x . In the case of the ST-ADS algorithm, this function chooses the best valid input of type b) and stores the links on valid inputs of type c), see Section 4.2.1 for input validity types. A valid input x of type b) means

that the next states $\delta(r.states, x)$ are covered by more than one block of the current partition. Therefore, there is a node r_x with the sequence that separates the next states. The best input out of those of type b) should lead to a node with the shortest separating sequence. This is exactly what is done on lines 17–18 of Algorithm 9 as the function SCORE (Algorithm 12) returns the length of separating sequence of r_x if it is valid. A valid input x of type c) transfers the states of r into another block of the current partition. It means that the node r_x representing such a block of states is a leaf in *dependent* because a valid input does not merge states, that is, $|r.states| = |\delta(r.states, x)|$, and the leaves are processed in the order of the size of *states*. Therefore, a link from r to r_x on x is stored into *transitionsTo* (lines 4–5) so that r can be separated based on r_x when a separating sequence is found for r_x . The node r_x that includes all next states is located in the *ST* as the lowest common ancestor of the leaves corresponding to the next states; the function GETSEPARATINGNODE is defined in Algorithm 5. In the case of the ST-IADS algorithm, r_x can have an invalid separating sequence. Lines 6–16 optimizes the choice of invalid separating sequence starting with a valid transferring input x of type b). If r_x has invalid separating sequence and more states than r , then there could be a better separating sequence for the next states $\delta(r.states, x)$. Therefore, an auxiliary node including just these states is created and analysed (if it was not). However, once a r_x with a valid separating sequence is observed, the condition on line 6 is false and thus no other auxiliary nodes are created. Auxiliary nodes created for the previous inputs are already analysed and stored but do not have a separating sequence assigned as they are not processed unless another node leads to them; the condition on line 8 allows these nodes to be used again. If the condition on line 8 is satisfied, then the auxiliary node is added to *dependent* and a link to it from r is stored to *transitionsTo*. Analysed auxiliary nodes with a separating sequence are like the internal nodes of *ST*, therefore, the score for them is calculated and compared against the best score (lines 15–16). If one does not want to optimize the choice of invalid separating sequences, hence the number of sequences in HSIs can be larger, then lines 6–16 can be omitted. The function SCORE favours valid sequences so *best_r.next* gets the node with the shortest valid separating sequence if there is one. Finally, if a node r_x with a valid separating sequence is found, then r is sorted into *dependentPriorityQueue* according to the score calculated for the best such r_x .

Nodes in *dependent* are processed using PROCESSDEPENDENT described in Algorithm 10 after their links were initialized either by INITTRANSITIONSONVALIDINPUTS or by INITTRANSITIONSONINVALIDINPUTS. Besides the links both functions fill *dependentPriorityQueue* with nodes r for which the separating sequence can be constructed based on the chosen *best_r*'s. Algorithm 10 goes through all nodes r in *dependentPriorityQueue* that is sorted according to the scores given by *best_r*'s. If r is not yet separated, its separating sequence is set to the one of the *best_r.next* prepended by x leading to *best_r.next* from r . After separating r , r is removed from *dependent*. As r is now separated,

Algorithm 10: PROCESSDEPENDENT()

```

1 while dependentPriorityQueue is not empty do
2   pop r from dependentPriorityQueue with the lowest score
3   if r is not separated then
4     r.separatingSequence  $\leftarrow$  bestr.input · bestr.next.separatingSequence
5     SEPARATE(r)
6     foreach  $(p, x) \in \textit{transitionsTo}[r]$  do
7       if p is not separated and SCORE(p, x, r) < bestp.score then
8         bestp  $\leftarrow$  (x, r, SCORE(p, x, r))
9         push p with bestp.score into dependentPriorityQueue
10    pop r from dependent

```

it can help other nodes in *dependent* that lead to it. Therefore, all links (p, x) in *transitionsTo* leading to *r* are checked and if a predecessor *p* is better separated based on *r*, it is pushed to *dependentPriorityQueue* with its new SCORE(*p*, *x*, *r*).

Algorithm 11: INITTRANSITIONSONINVALIDINPUTS(*r*)

```

1 foreach invalid separating input x of r do
2   if SCORE(r, x, null) < bestr.score then
3     bestr  $\leftarrow$  (x, null, SCORE(r, x, null))
4 foreach invalid transferring input x of r such that  $|\delta(r.states, x)| > 1$  do
5   rx  $\leftarrow$  GETSEPARATINGNODE( $\delta(r.states, x)$ )
6   if rx.separatingSequence is not assigned or not valid then
7     rx  $\leftarrow$  a (stored or new) node with states equal to  $\delta(r.states, x)$ 
8     if rx is not analysed then
9       ANALYSE(rx)
10    push rx into dependent if it is not there
11    add (r, x) to transitionsTo[rx]
12   if rx.separatingSequence is assigned then
13     if SCORE(r, x, rx) < bestr.score then
14       bestr  $\leftarrow$  (x, rx, SCORE(r, x, rx))
15 push r with bestr.score into dependentPriorityQueue

```

Algorithm 11 checks all invalid inputs after all valid transferring inputs are checked by INITTRANSITIONSONVALIDINPUTS and no valid separating sequence was observed for the given node *r*. At first, all invalid separating inputs are checked if any of them can improve the best separating score initialized in INITTRANSITIONSONVALIDINPUTS. Then all invalid transferring inputs *x* that do not merge all states are processed in a similar way as the valid ones were in Algorithm 9. If the lowest common ancestor *r_x* of the

leaves corresponding to the next states $\delta(r.states, x)$ was not processed or has an invalid separating sequence, an auxiliary node relating just to the next states is considered as r_x instead. It is analysed if it was not, and pushed into *dependent* as its separating sequence may be needed to obtain the best invalid sequence for r . The link from r to r_x is stored as well. If r_x already has a separating sequence, it is checked whether r_x is a better basis for the best separating sequence of r and so whether $best_r.score$ can be improved. Finally, r is pushed into *dependentPriorityQueue* with the best score encountered so far. Note that r can be added to *dependentPriorityQueue* with a better score later in PROCESSDEPENDENT after one of its r_x 's that is not separated gets a separating sequence.

Algorithm 12: SCORE(r, x, r_x)

```

1  $w \leftarrow x \cdot r_x.separatingSequence$  if  $r_x$  is not null else  $x$ 
2 if  $w$  is a valid separating sequence of  $r.states$  then return  $|w|$ 
3  $(a, b, c, d, e, n_r) \leftarrow (0, 0, 0, 0, |w|, |r.states|)$ 
4 foreach response  $z \in \lambda^*(r.states, w)$  do
5    $S' \leftarrow \{s \in r.states \mid \lambda^*(s, w) = z\}$  // states of successor
6   if  $|S'| = |\delta^*(S', w)|$  then //  $w$  valid for  $S'$ 
7      $b \leftarrow b + 1$  // number of valid successors
8   else
9      $a \leftarrow a + |S'|$  // number of states in invalid successors
10     $d \leftarrow d + |S'| - |\delta^*(S', w)|$  // undistinguished states
11     $c \leftarrow c + 1$  // number of successors
12 return  $((a \cdot n_r - b) \cdot n_r - c) \cdot n_r + d) \cdot n_r + e$ 

```

The last part of the ST-IADS algorithm to describe is the scoring function in Algorithm 12. The function SCORE(r, x, r_x) analyses how the states of r would be separated by the sequence $w = x \cdot r_x.separatingSequence$, or only by x if the given r_x is null. If w is a valid sequence for $r.states$, then the length of w is returned. Otherwise, a higher score is returned so that valid sequences are favoured. As the ultimate aim is to have the smallest number of separating sequences for a given subset of states, this scoring function prioritises invalid sequences that are valid for the maximum total number of states in the successors for which the sequence does not merge any two states. A successor r_i of r is valid if $r.separatingSequence$ is valid for $r_i.states$, that is, $|r_i.states| = |\delta^*(r_i.states, r.separatingSequence)|$. A r_i is a successor of r , $|\lambda^*(r_i.states, r.separatingSequence)| = 1$. If some states of r_i are merged by $r.separatingSequence$, r_i is an invalid successor of r . The number of undistinguished states of an invalid successor r_i is the difference between the number of states in r_i and the number of their next states on $r.separatingSequence$, that is, $|r_i.states| - |\delta^*(r_i.states, r.separatingSequence)|$. There are five parameters a – e to compare invalid separating sequences. The parameter a represents the total number of states in invalid successors, b is the number of valid successors, c is the number of all successors of r , d is

the total number of undistinguished states, and e is the length of w . As the score of the best sequence is the lowest, the parameters b and c that are signs of a good separating sequence decrease the score and the parameters a , d and e are rather bad signs so that they increase the score. The priority of parameters how they influence the score is given by their alphabetical order. The parameter a estimates for how many states another separating sequence will be needed, for example in the construction of HSIs. Therefore, the higher a the less likely the sequence is chosen to be the separating sequence of r . The parameters b and c provide a ratio of the number of ‘good’ successors to their total number and c also represents how well the states of r are divided by w ; the higher c the more state pairs are likely to be separated by w . The parameter d estimates how many states will remain undistinguished when a state is asked to be separated from the others in $r.states$. The parameters are connected together by the formula on line 12 of Algorithm 12 to get one number evaluating the invalid sequence w . The number n_r of states in r is employed in the formula to order the parameters in the resulting score by their priority. Note that $d < a \leq n_r$ and $b < c < n_r$ as there is always an invalid successor that contains at least two states. Therefore, only e could interfere with d if $e \geq n_r$, but it is acceptable as d provides just an estimate that does not have to be precise and this interference thus penalizes sequences that are too long. The scoring function could be implemented differently which influences the choice of invalid separating sequences and not the correctness of the algorithm.

The time complexity of the existing ST-ADS algorithm is $O(n^2p)$ where $n = |S|$ and $p = |X|$ ([LY94, Theorem 3.2]). There are at most $n - 1$ refinements of the partition and so the resulting ST has at most $2n - 1$ nodes (n leaves and at most $n - 1$ inner nodes). Each node r is analysed in $O(np)$ as all states of r and at most all inputs are checked. The function `INITTRANSITIONSONVALIDINPUTS` (Algorithm 9) prepares links in $O(np)$ and it is called at most n times as *dependent* contains less than n nodes in total during the entire construction. All valid transferring inputs are checked when links are formed and for each input the lowest common ancestor of the next states is found by `GETSEPARATINGNODE` in $O(n)$. Each node of *dependent* is processed by Algorithm 10 once and it allows one to check all links stored in *transitionsTo*. The total number of links is in $O(np)$ which is also the complexity of `PROCESSDEPENDENT` in Algorithm 10. Therefore, the ST-ADS algorithm runs in $O(n^2p)$. Moreover, all separating sequences are of length at most $n - 1$ so the space complexity of the splitting tree is $O(n^2)$ if there is an ADS [LY94]. The proposed pseudocode is based on the implementation of the ST-ADS algorithm proposed in [Sou14] rather than the original one from [LY94] as [Sou14] simplifies dealing with valid inputs of types b) and c).

The proposed extension potentially increases the time and space complexity a lot. In the worst case $O(2^n)$ (auxiliary) nodes representing all subsets of states could be analysed. This can be easily avoided by omitting lines 6–16 of Algorithm 9 and lines 6–11 of Algorithm 11 that try to find a better invalid separating sequences by introducing auxiliary nodes. Without these lines, the

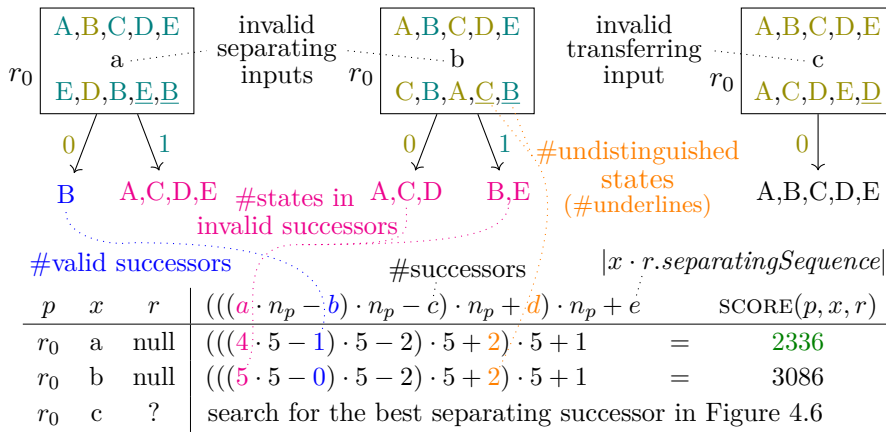


Figure 4.5: Construction of splitting tree – analysis of the root node

ST-IADS algorithm still works but may result in more sequences in HSIs built from the resulting ST. Nevertheless, the number of created auxiliary nodes is restricted by the transition and output functions of the given machine so that it hardly reaches huge numbers. A possible improvement that could result in lower number of separating sequences in HSIs is the use of a scoring function also on valid sequences. This would mean adjusting the existing ST-ADS algorithm, not just extending it as was done by the proposed ST-IADS algorithm.

4.5 Running Example

The ST-IADS algorithm is described in this section how it builds the splitting tree showed in Figure 4.2 for the Mealy machine M defined in Figure 4.1. An example describing just the ST-ADS algorithm can be found in [Sou14, LY94].

The algorithm starts with the root r_0 of ST that contains all states A–E. As M does not have state outputs, r_0 is the only node in *partition* (line 10 of Algorithm 8). The root is then popped from *partition* and analysed. The analysis of all three input symbols is captured in Figure 4.5 where the scoring function is also explained. Inputs ‘a’ and ‘b’ are separating as the states respond on them with 2 different outputs; input ‘c’ is transferring. Notice that states and next states in the root r_0 have the colour of the corresponding output. As all inputs merges some states, there is no valid input for r_0 and null would be returned as the sign that M has no ADS if *validOnly* was true. The root is thus added to *dependent* that is immediately processed by *PROCESSDEPENDENT* because *partition* is empty and there is no valid transferring input that could be checked by *INITTRANSITIONSONVALIDINPUTS*. However, *dependentPriorityQueue* is empty so that *PROCESSDEPENDENT* exits and r_0 is still in *dependent*. Hence *INITTRANSITIONSONINVALIDINPUTS* is called to prepare links from r_0 .

INITTRANSITIONSONINVALIDINPUTS first calculates the score for the invalid separating input ‘a’ such that r_0 with 5 states would have 4 states in the

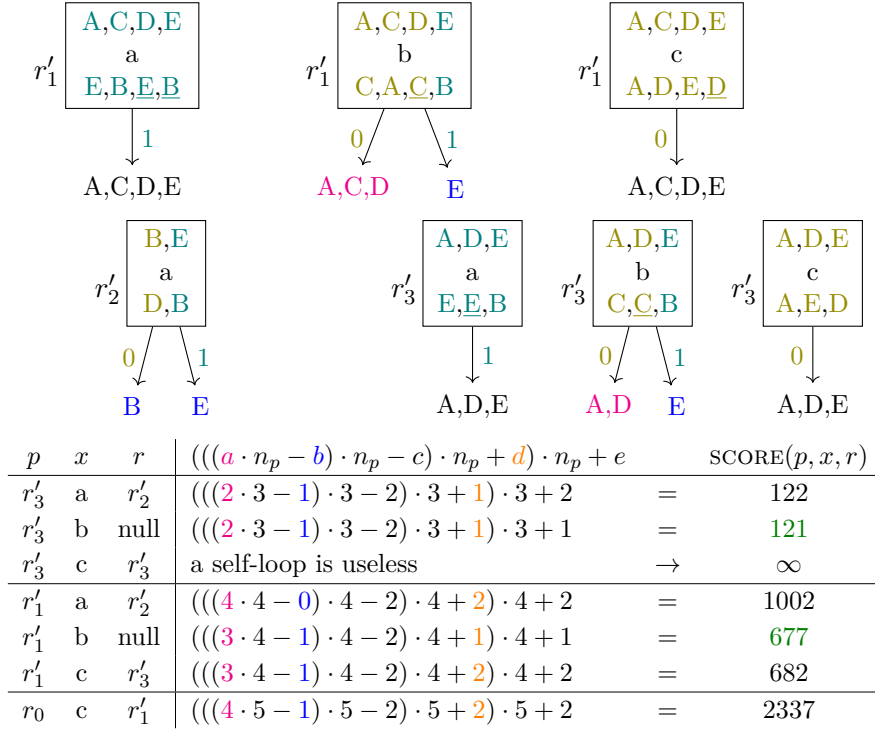


Figure 4.6: Search for the best successor on invalid transferring input ‘c’

invalid successor, 1 valid successor out of 2 successors, 2 undistinguished states and the separating input has the length of 1. $\text{SCORE}(r_0, a, \text{null})$ is thus 2336. States A, D and B, E merge in their corresponding successors on input ‘b’, therefore, there is no valid successor of r_0 on ‘b’ and so the corresponding score 3086 is worse than on ‘a’. The invalid transferring input ‘c’ does not merge all states and so it is checked if it can begin a better invalid separating sequence than ‘a’. The lowest common ancestor r_x of the leaves relating to the next states $\delta(r_0.\text{states}, c)$ is the root itself and as it has no separating sequence assigned yet, an auxiliary node r'_1 is created; $r'_1.\text{states} = \{A, C, D, E\}$. The analysis of r'_1 for all inputs is shown in Figure 4.6; ‘a’ and ‘c’ are invalid transferring for r'_1 and ‘b’ is invalid separating. The node r'_1 is then added to *dependent* and a link (r_0, c) is stored into *transitionsTo*[r'_1]. As r'_1 does not have a separating sequence yet, it cannot improve the best score for r_0 and so r_0 with the score of 2336 relating to the invalid separating input ‘a’ is pushed into *dependentPriorityQueue*.

The next cycle of ‘foreach’ loop on line 30 of Algorithm 8 chooses to initialize links from the node r'_1 . The node has no valid input so again *INITTRANSITIONSONINVALIDINPUTS* is called. After the score of 677 is calculated for the invalid separating input ‘b’, both transferring inputs are processed. In both cases auxiliary nodes are created; $r'_2.\text{states} = \delta(r'_1.\text{states}, a) = \{B, E\}$ and $r'_3.\text{states} = \delta(r'_1.\text{states}, c) = \{A, D, E\}$. A valid separating input ‘a’ is found for the auxiliary node r'_2 during the analysis of inputs. Only ‘a’ is thus analysed. Therefore, $\text{SCORE}(r'_1, a, r'_2)$ can be immediately

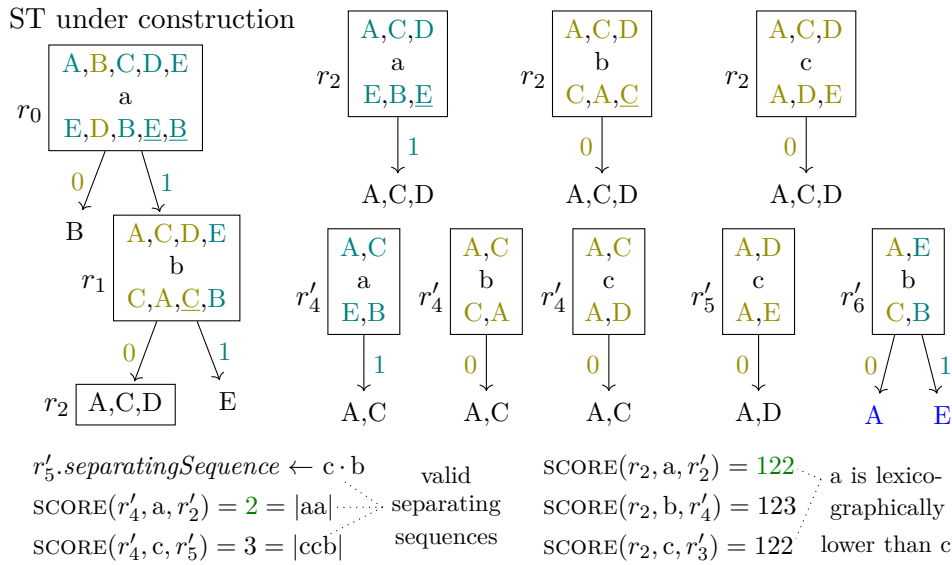


Figure 4.7: Analysis and separating node r_2 with states A,C,D

calculated but the value of 1002 does not improve the best score for r'_1 relating to input 'b'. The node r'_2 is then reused to calculate the score for the auxiliary node r'_3 on invalid transferring input 'a'. The score is worse than the best one set by the invalid separating input 'b'; the scores differ just by 1 because of the length of separating sequence 'aa'. The states of node r'_3 are transferred to themselves on input 'c' which means that 'c' cannot start the shortest separating sequence of a good score; such nodes can be thrown away already during the analysis of inputs. After r'_3 is processed by `INITTRANSITIONSONINVALIDINPUTS`, `dependentPriorityQueue` is filled up with r'_3 (score 121), r'_1 (677) and r_0 (2336). `PROCESSDEPENDENT` pops r'_3 first and sets 'b' as its separating sequence. Then, it tries to improve the best of r'_1 but $\text{SCORE}(r'_1, c, r'_3) = 682$ is not better. The same happens to r'_1 that is popped from `dependentPriorityQueue` next. It gets 'b' as the separating sequence and $\text{SCORE}(r_0, c, r'_1) = 2337$ does not improve the best so that r_0 is separated with 'a'. Analyses of inputs for r'_2 and r'_3 and the calculation of scores are shown in Figure 4.6. Notice that separating sequence 'cb' has the score worse than the selected separating input 'a' just because of its length.

Separating the root r_0 with 'a' results in two new leaves and the update of `partition` and `ST.separatingNodes`. One leaf represents state B and so it will not be further processed. The second leaf r_1 represents the other states A, C, D, E. `ST.separatingNodes` is thus updated to point to r_0 for all state pairs associated with state B. Both leaves form the current partition, however, r_1 is popped from `partition` right at the start of the second cycle of the main loop (line 12 of Algorithm 8). Fortunately, r_1 is the same as the auxiliary node r'_1 and as r'_1 was analysed and has a separating sequence 'b', r_1 is separated with 'b' directly. Node r'_1 with its successors just replaces r_1 in the `ST` in the implementation.

The current partition is updated to contain two singletons representing states B and E, and the leaf r_2 that includes states A, C, D. The splitting tree in the current form is shown on the left of Figure 4.7. The node r_2 is popped from *partition* and analysed as it is not stored amongst the auxiliary nodes. It has no valid separating sequence which leads to storing it to *dependent*. Right after that the algorithm tries to find a separating sequence as there is no other leaf with the same number of states in *partition*. This time there is a valid transferring input and a separating sequence is already assigned to the node to which the input leads. Thus, $best_r$ for r_2 is initialized in INITTRANSITIONSONVALIDINPUTS with input ‘c’ leading to r'_3 and the score of 122. Node r_1 as the lowest common ancestor of the leaves containing states A, C, E is first considered instead of r'_3 but as r_1 also contains state D and its sequence ‘b’ is invalid, r'_3 is chosen. As the separating sequence ‘b’ of r'_3 is invalid, r_2 is not pushed into *dependentPriorityQueue* and so the first call of PROCESSDEPENDENT (line 27) does not change anything. All the shortest invalid separating sequences for r_2 need to be compared to choose the one that separates it. The auxiliary nodes created during the search for the best invalid sequence are shown in Figure 4.7. In the case of r'_5 relating to states A and D, inputs ‘a’ and ‘b’ are not visualized as they merge the states so that they cannot begin a separating sequence. The node r'_6 has a separating input ‘b’ so that only this input is used and shown. After all the needed auxiliary nodes are created, analysed and connected by links, nodes of *dependent* are processed by Algorithm 10 as sketched at the bottom of Figure 4.7. The implementation checks all inputs in one pass so that alphabetically lower inputs with the minimal score are favoured. Six auxiliary nodes were created and two of them were later reused. As $n = 5$ and the total number of explored nodes is $9 + (6 - 2) = 13$, the space complexity seems to be closer to $\Theta(n^2)$ than in $\Theta(2^n)$.

Chapter 5

Experiments

The new extension proposed in the previous chapter is evaluated on the construction of harmonized state identifiers (HSI) in order to address the research question RQ I.2. As the aim of the thesis is to design an efficient learning technique, the experiments in this chapter only show the additional value of the proposed extension and do not aim to provide a thorough evaluation. HSIs are first constructed for randomly generated machines and then for models of real systems.

Harmonized state identifiers are formed in the experiments using three approaches. The first one follows Section 3.3.2 and forms HSIs from the shortest separating sequences (SSS). The second approach gets separating sequences of all state pairs from the splitting tree using Algorithm 4 and then collects the sequences that relate to each state. The third approach follows Algorithm 6 and forms HSIs from the splitting tree.

5.1 Randomly Generated Machines

The suite of 13 600 randomly generated machines is described in Appendix C.1. There are 4 machine types, each represented equally with 3 400 machines such that a half of them has 5 inputs and the other 1 700 machines have 10 inputs. The number of states ranges from 10 to 1000 and there are 17 state groups of 100 machines. For each machine it is recorded the total number of sequences in the constructed HSIs of all states, the total length of sequences in the HSIs, and the time that is needed for the construction of HSIs of all states by each method.

The results of construction of HSIs for deterministic finite automata are shown in Figure 5.1. The three figures on the left present the results for machines with 5 inputs and those on the right for machines with 10 inputs. All six figures show 1 and 3 quartiles that are calculated from 100 values for each of 17 state groups. In order to compare the total number and the total length of sequences when the number of states varies, the values are divided by the number of states and the average number of sequences, respectively. The first two figures thus capture the number of sequences per a single state. The figures in the middle show the length that each separating sequence would have if the constructed HSIs had the average number of sequences

(calculated from the 100 values for each state group and each method). The last two figures show the construction time.

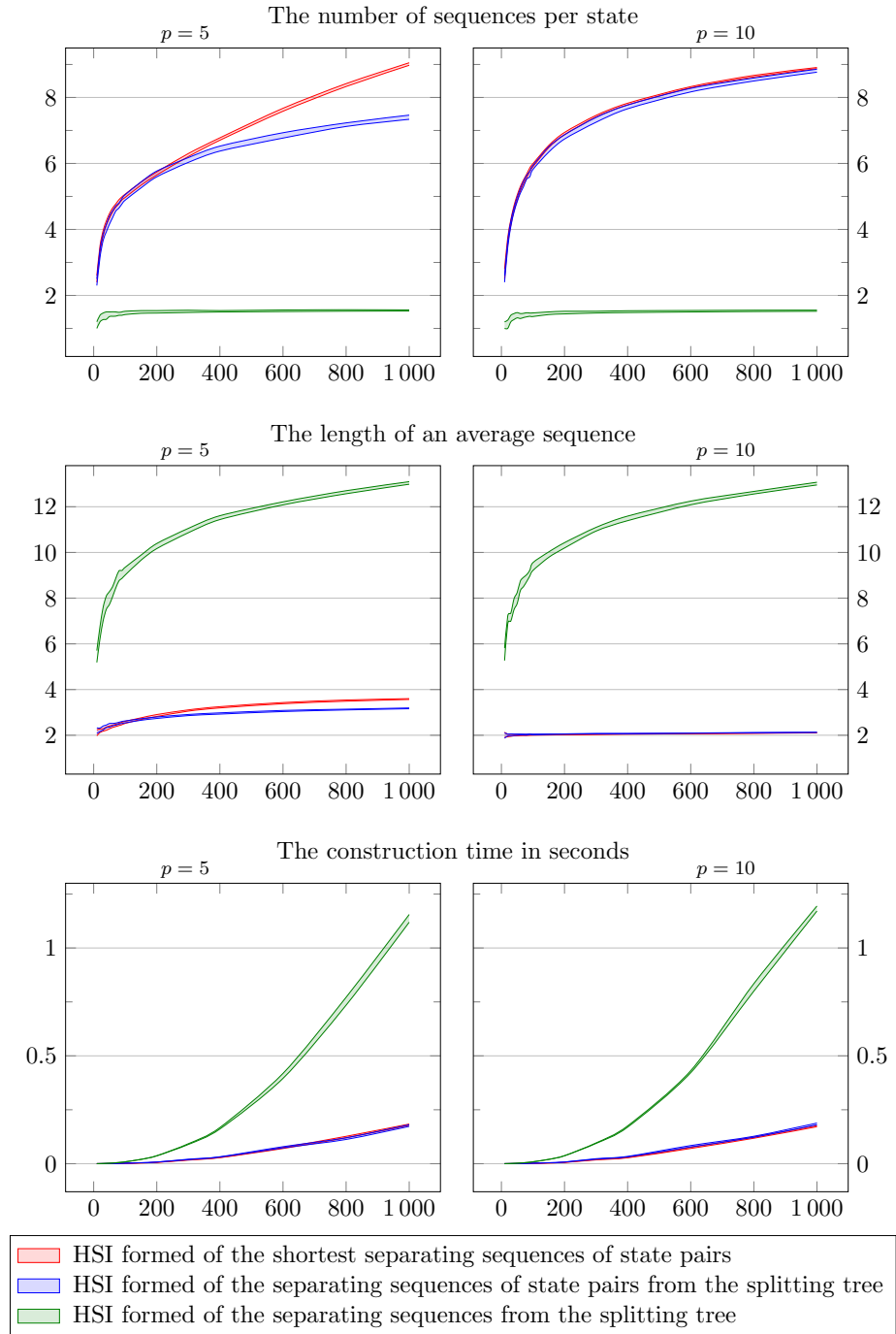


Figure 5.1: Comparison of the HSI construction methods on DFA with 5 inputs on the left and on DFA with 10 inputs on the right. 1 and 3 quartile calculated for 100 machines per each state group

The first two approaches form HSIs out of the (shortest) separating sequences of all state pairs which results in many sequences that consist of 2 symbols on average. On the other hand, the third approach constructing HSIs directly from the splitting tree builds a small number of sequences that are longer than 5 symbols on average. As both the second and the third approaches create the same splitting tree, the higher construction time of the third approach is thus due to the construction of separating sequences from the splitting tree using Algorithm 6 that calls `GETSEPARATINGSEQUENCEFROMST` (Algorithm 4) repeatedly. Moreover, it means that the construction time of a splitting tree is similar to the running time of the SSS algorithm, that is, the time complexity of the proposed extension ST-IADS is $O(n^2p)$ for randomly generated machines. Note that just few machines have adaptive distinguishing sequence and so the new extension is employed for most machines.

Comparison of the three approaches is very similar as shown in Figure 5.1 for the other machine types. In fact, only the results for DFA with 5 inputs significantly distinguish the first and second approaches. The results for the other three machine types and both numbers of inputs look like the results for DFA with 10 inputs, that is, the three figures on the right of Figure 5.1. All the results with visualizations can be found in the repository `FSMmodels v1.3`¹.

The real contribution of the proposed extension can be observed when the constructed HSIs are used. Section 12.1 presents the comparison of testing methods on the randomly generated machines (the same as in these experiments) and two methods significantly improve their performance by the use of HSIs constructed from the splitting tree instead of from SSS. These two methods are the HSI-method (Section 9.5) and the SPY-method (Section 9.7), and they are only two that use harmonized state identifiers. Their ‘improved’ versions are referred by ‘HSI/ST’ and ‘SPY/ST’, respectively.

5.2 Models of Real Systems

Harmonized state identifiers are constructed for the models of real systems in order to compare the three construction approaches. The models `peterson2`, `sched4` and `sched5` are deterministic finite automata with 50, 97 and 241 states and 18, 12 and 15 inputs, respectively. Appendix C.2 describes the models in more detail.

Table 5.1 shows the parameters of HSIs constructed by the three approaches for the three DFA models. There are two numbers to describe each collection of HSIs: the total number of sequences (Seqs) and the total number of input symbols (Syms) that equals to the total length of sequences in HSIs of all states. The construction time is missing in Table 5.1 but it can be summarized for all three models as follows. The second approach is twice as slow as the

¹<https://github.com/Soucha/FSMmodels/releases/tag/v1.3>

first one and the third approach needs four times more time than the first approach.

HSI construction method	peterson2		sched4		sched5	
	Seqs	Syms	Seqs	Syms	Seqs	Syms
From SSS	439	881	760	1601	2176	4609
From ST-StatePairSepSeqs	301	865	672	1425	2080	4417
From ST	268	914	537	1589	1649	4883

Table 5.1: Comparison of three approaches for the construction of HSIs on the total number of sequences (Seqs) and symbols (Syms) in the constructed HSIs for the models of real systems

The results show that the third approach that uses the new extension ST-IADS constructs HSIs with fewer sequences in total than the other two approaches. However, the difference is much smaller than in the case of randomly generated machines (Section 5.1). The same holds for the total length of sequences that is high enough to have the longest sequences on average but the average length of sequences for all the approaches is between 2 and 3 symbols. Therefore, the constructed HSIs are very similar in terms of the parameters. This is reflected in the performance of testing methods that use HSIs. Such methods are the HSI-method and the SPY-method. Section 12.2 presents the comparison of testing methods on the models of real systems (the same ones as used in this experiment). Both the HSI- and SPY- methods perform very similarly regardless of whether they use HSIs from SSS (the first approach in this experiment) or from the splitting tree (the third approach here).



Chapter 6

Conclusion

This part of thesis summarized different approaches for the construction of state identification sequences such as separating sequences of state pairs, characterizing set and harmonized state identifiers (HSI). Chapter 4 described the new ST-IADS algorithm that extends the existing ST-ADS algorithm in the construction of a splitting tree with invalid inputs. The new extension allows one to construct a splitting tree for any deterministic finite-state machine, not only for those having an adaptive distinguishing sequence. Moreover, HSIs derived from such a splitting tree consist of a lower number of sequences that are generally longer than in the case of HSIs formed of the shortest separating sequences of all state pairs. This was evaluated experimentally as Chapter 5 reported. The evaluation is however not sufficient to answer the research question (RQ) I.2 because the influence of the number and the lengths of sequences in a constructed HSI on the total length of test sequences is demonstrated in the next part of the thesis.

The main purpose why the new ST-IADS algorithm was designed is not just for the construction of HSIs but to easily construct a sequence separating a state from a subset of states (RQ I.1). Such a sequence is returned by the function `GETSEPARATINGSEQUENCEFROMST` defined in Algorithm 4. This function is a key point of the new testing method, the S-method, that is proposed in the next part of thesis.



Part II

Testing Methods

Chapter 7

Introduction

Deterministic finite-state machines (DFSM) can be tested for the presence of faults as was sketched in Section 1.2 in the introduction chapter of the thesis. Each fault revealed by testing can be corrected which results in a more reliable system and thus testing improves the quality of a system under development. This is particularly useful in software development. For instance, any fault in a safety-critical application does not mean just the loss of money but may result in death or serious injuries to people. Therefore, it is very important to make an effort and test the system to reduce the risk of unpleasant effects.

This part of thesis deals with *testing* of systems that:

- have a specification in the format of a *deterministic finite-state machine* that is *minimal* and *completely specified*, and
- can be reliably *reset* to their unique initial states.

The first condition on systems allows the use of a testing method that creates a test suite of several test sequences based on the specification of the system. The second condition enables one to apply these test sequences in the same state of the system. If the reset is not available, the constructed test suite is a singleton and the test sequence is called a checking sequence for the given specification. Nevertheless, the machine is then required to be strongly connected so that every state of the machine can be reached and tested. In addition, most testing methods constructing a checking sequence work only for machines with a preset or adaptive distinguishing sequence or with a verifying set, however, many machines do not have these sequences. This part about testing methods for resettable DFSMs is based on the overview of this field proposed in [Sou15] that describes standard testing methods for both resettable machines and machines without reset that require a checking sequence. Systems that include a fault differ from their specifications but the type of fault influences the difference between the specification and the implementation that represents the system. There are several types of faults and also many testing methods that deal with different types of faults. Therefore, this part focuses only on testing methods that tackle the problem when the implementation can have more states than the specification.

The next chapter gives basic definitions used in testing and specifies requirements on testing methods to give a guarantee about the absence of

more states in the implementation. Then, the standard testing methods are described in Chapter 9 and novel testing methods are proposed in Chapter 10 and Chapter 11. The new testing methods are based on a new sufficient condition that guarantees the absence of a particular number of additional states in the implementation if the constructed test suite meets particular requirements. The new sufficient condition combines two existing ones and it is described and proven in Chapter 8. Chapter 12 describes experiments that compare the new testing methods against the standard ones. Chapter 13 concludes this part of thesis.

■ 7.1 Research Questions

- II.1 Is there a sufficient condition for test suite completeness weaker than the most advanced conditions on which the H- and SPY- methods are based?
- II.2 Is there a technique constructing m -complete test suites smaller than the standard testing methods?

Chapter 8

Test Suite Completeness

The process of testing was sketched in Section 1.2 and this chapter specifies the used notions and defines conditions that are usually employed to test implementations that may comprise more states than their specifications. Deterministic finite-state machines (DFSM) as introduced in Section 1.6 are used to specify the following key notions that relate to the testing task.

- The *specification* is a DFSM $M = (S, X, Y, s_0, D_M, \delta_M, \lambda_M)$. The numbers of states, inputs and outputs are denoted by n, p, q , respectively, that is, $n = |S|$, $p = |X|$ and $q = |Y|$.
- The *implementation* is a DFSM $N = (Q, X, Y', q_0, D_N, \delta_N, \lambda_N)$ with $m = |Q|$. It usually holds that $Y \subseteq Y'$, otherwise some outputs expected by the specification cannot be produced by the implementation which corresponds to output faults.
- *Extra states* (ES) is a notion used to describe that the implementation has more states than the specification. If both the specification and the implementation are minimal, then there are l extra states where $l = m - n$.
- *Test suite* (TS) is a set T of *test sequences*, or just *tests*, t_i such that T aims to test whether a property of the specification holds in the implementation.
- A *testing method* creates a test suite based on the specification.
- The implementation N *passes* a test suite T if it responds to all test sequences with the outputs that are expected according to the specification, that is, $\forall t_i \in T : \lambda_N^*(q_0, t_i) = \lambda_M^*(s_0, t_i)$. Otherwise, N *fails* T .
- The *fault domain* F_T is a set of all possible implementation that pass the test suite T .

Other requirements on the specification, the implementation and a testing method will be described after two definitions are proposed. The first one defines a relation between two machines and the second specifies a property of a test suite. M and N range over DFSMs with the above given definition

and the subscripts of the transition and output functions are omitted when they are clear from the context.

Definition 8.1. Two DFSMs M and N are *output-different*, or *distinguishable* by w , if there is an input sequence $w \in X_{\uparrow}^*$ that separates the initial states s_0 of M and q_0 of N , that is, $\lambda_M^*(s_0, w) \neq \lambda_N^*(q_0, w)$.

DFSMs M and N are said to be *U -distinguishable* if there is a $w \in U$ that separates their initial states.

Definition 8.2. A test suite T is *m -complete* with respect to a reduced DFSM M with n states if and only if for each DFSM N such that M and N are output-different and N has at most m states there is a test sequence $t_i \in T$ that separates the initial states s_0 of M and q_0 of N , that is, M and N are distinguishable by t_i . An *m -complete test suite* is said to be *closed for l extra states* where $l = m - n$.

The following proposition directly follows from Definition 8.2.

Proposition 8.3. *If the implementation N passes an m -complete test suite for the specification M but M and N are output-different, then N must have more than m states.*

Proposition 8.3 specifies Theorem 2 of [Moo56] for *m -complete test suites*. The proof follows from Definition 8.2 as all output-different N with up to m states are distinguished by a test sequence of the *m -complete T* so that N needs to have more than m states. The use of Proposition 8.3 is a standard way to give a precise guarantee about the number of states in the implementation. The restrictions on the related machines and testing methods with which this part of thesis deals are given as follows.

- The specification M is *minimal*, that is, M is initially connected and every two states $s_i \neq s_j \in S$ are distinguishable.
- Both the specification M and the implementation N are *completely specified*, that is, $D_M = S \times X$ and $D_N = Q \times X$.
- The implementation is *resettable* so that all test sequences can be tested from the same unique state.
- All considered testing methods construct *m -complete test suites*, that is, the constructed test suites are closed for the given number of extra states $l = m - n$.

Key notions related to testing and the restriction considered in this thesis were described. Before the sufficient conditions of test suite completeness are introduced, some basic notions are defined as follows. The notions of convergence and state domains are described in the next two sections.

- A set of sequences U is *initialized* if it contains the empty sequence ε .
- The *access sequence* \bar{s}_i of state s_i is a sequence that leads from the initial state to state s_i , that is, $\delta^*(s_0, \bar{s}_i) = s_i$.

- A *state cover* \bar{S} for M is a set of access sequences of all states of M , that is, $\forall s_i \in S \exists \bar{s}_i \in \bar{S} : \delta_M^*(s_0, \bar{s}_i) = s_i$. State cover \bar{S} is usually initialized so that the initial state is reached by the empty sequence. It is also often prefix-closed and minimal, that is, there is exactly one access sequence for each state.
- A *transition cover* U for M is a set of sequence such that each defined transition is included in a sequence of U , that is, $\forall (s, x) \in D \exists uv \in U : \delta_M^*(s_0, u) = s$. Transition cover U is usually the one-input extension of the minimal prefix-closed state cover \bar{S} , that is, $U = \bar{S} \cdot X$.
- The letter T is overloaded to represent both a test suite T as a set of sequences and the related output function $T : X^* \times X_{\uparrow}^* \rightarrow Y^*$ that is defined as $T(u, v) = \lambda^*(\delta^*(s_0, u), v)$ for all $uv \in T$ and thus it describes the response to the suffix v from state reached by the prefix u .
- Two test sequences u and v in T are *T -separable* if both are extended by a w in T such that w separates the states reached by u and v , that is, $\exists w : uw, vw \in T \wedge T(u, w) \neq T(v, w)$.

Notice that whenever a sequence w is considered for the purpose of a transfer from one state to another, the stOut input is not included as it does not change the target state, that is, $w \in X^*$. However, if w is considered for the purpose of its response, for example, w is a separating sequence, then it is reasonable to append \uparrow to each input $x \in X$ in the case of DFSMs, that is, $w \in X_{\uparrow}^*$.

8.1 Convergence of Sequences

Convergence and divergence of test sequences with respect to a set of machines are important notions that enable one to use properties of regular languages in testing. Regular languages are equivalent to finite-state machines, however, this fact was not utilized for testing before the notions of convergence and divergence were proposed in [SP09a, SPY12]. The structure of a DFSM was described only by the sets of access and separating sequences but the relations between the sequences were missing.

Definition 8.4. Given a set of FSMs F , two tests are *F -convergent*, if they converge in each FSM of F , that is, both test sequences lead from the initial state to the same state in each FSM of F ; and two tests are *F -divergent*, if they diverge in each FSM of F , that is, both test sequences lead from the initial state to different states.

Definition 8.4 can be applied to different sets of machines but the two most important are F representing the fault domain F_T of DFSMs that pass the test suite T and F corresponding only to the specification M . Two sequences u and v are thus *F_T -convergent* if for each $N \in F_T$, it holds that $\delta_N^*(q_0, u) = \delta_N^*(q_0, v)$; u and v *F_T -converge* [SP09a]. Similarly, u and v are

M -convergent in state s if $\delta_M^*(s_0, u) = \delta_M^*(s_0, v) = s$. The curly brackets representing a set are omitted in the case of a single machine for simplicity, that is, M -convergent or M -divergent is used instead of $\{M\}$ -convergent or $\{M\}$ -divergent. Moreover, the set is omitted totally when it is clear from the context. A necessary condition for two test sequences to be F_T -convergent (F_T -divergent) is that they need to be M -convergent (M -divergent). This follows from the fact that the specification M always passes its test suite T and so M is always in F_T . There is also a sufficient condition for F_T -divergence. Two test sequences u and v are F_T -divergent if they are T -separable.

The convergence relation is reflexive, symmetric and transitive, which means that it is an equivalence relation over the set of tests [SPY12]. Tests $u_i \in T$ can be thus partitioned into corresponding equivalence classes

$$[u_i] = \{u_j \in T \mid u_i \text{ and } u_j \text{ are } F_T\text{-convergent}\}.$$

The following properties hold for any $u, v \in T$ such that $[u] = [v]$ [SPY12, Lemma 1]:

$$(i) \forall w \in X_{\uparrow}^* : [uw] = [vw] \quad \text{and} \quad (ii) \forall t \in T : [u] \neq [t] \implies [v] \neq [t].$$

As the equivalence classes group sequences that lead to the same state, the behaviour on all extensions of these sequences is the same and so T can be extended to work with the equivalent classes as follows. If there is $u' \in [u]$ that is in T , then $[u] \in T$. The function T is defined for a class $[u]$ and a sequence w as $T([u], w) = T(u', w)$ where $u' \in [u]$ and $u'w \in T$. If there is a sequence w such that $T([u], w) \neq T([v], w)$, then $[u], [v]$ are T -separable.

Equivalence classes are derivable from a pairwise comparison of tests based on their convergence and divergence, however, other notions are needed to describe relations between those classes.

Definition 8.5. Given a test suite T for the specification M , a set of tests in T is *F_T -convergence-preserving* if all its M -convergent tests are F_T -convergent; a set of tests in T is *F_T -divergence-preserving* if all its M -divergent tests are F_T -divergent.

For example, a set of two T -separable tests is F_T -divergence-preserving because all machines that pass T respond differently to each of the two tests and so the two test sequences are F_T -divergent. With the convergence it is a little harder as there is no simple sufficient condition for two or more tests to be F_T -convergent. The next theorem states a sufficient condition of two tests to be F_T -convergent.

Theorem 8.6. Given a test suite T for an FSM M and $l = m - n \geq 0$, let u and v be M -convergent tests in T , such that, for any sequence w of length l , there exist tests $u' \in [u], v' \in [v]$ and an F_T -divergence-preserving state cover for M in T containing $\{u', v'\} \cdot \text{pref}(w)$. Then, u and v are F_T -convergent.

Theorem 8.6 is the same as [SPY12, Theorem 2] except the denotation. Its proof is based on two lemmas [SPY12, Lemma 3 and Lemma 4]. The former lemma contains a mistake in [SPY12] as an assumption is not strong

enough to cover all possible cases. Hence, a revised version is proposed in the following lemma to correct the original one. The changed assumption is in bold and an explanation of the flaw in [SPY12, Lemma 3] is discussed in Appendix D.2.

Lemma 8.7. (revised [SPY12, Lemma 3]) *Given a test suite T and an FSM M , let u and v be tests in T which are M -convergent in state s and $N \in F_T$ be an FSM, such that u and v are N -divergent. Let also w be a shortest input sequence, such that w **separates states reached by u and v in N** . Then, for each proper prefix w' of w such that $\{u, v\} \cdot \text{pref}(w')$ is F_T -divergence-preserving, the tests in $\{u, v\} \cdot \text{pref}(w')$ reach at least $|w'| + |\delta_M^*(s, \text{pref}(w'))| + 1$ distinct states in N .*

Lemma 8.7 is proven by induction on the length of prefix w' (equally to the proof of [SPY12, Lemma 3]). The base case of the induction ($w' = \varepsilon$) holds as $|\varepsilon| + |\{s\}| + 1 = |\{q, q'\}|$. Let $w = w_k \cdot w'_k$ for all $0 \leq k < |w|$ such that $|w_k| = k$. There are two cases for the inductive step in which the bound on distinct states in N is proven based on the bound that holds for shorter prefixes. The cases are if state s_{k+1} reached by w_{k+1} from s is in the set of states $\delta_M^*(s, \text{pref}(w_k))$, and if it is not in the set.

Case 1: If s_{k+1} was not reached by any prefix of w_k from s , then uw_{k+1} is M -divergent with all $u \cdot \text{pref}(w_k)$. The same holds for the prefix v . As all these tests are in an F_T -divergence-preserving set, the M -divergent sequences are also F_T -divergent and so N -divergent as well. In addition, the states q_{k+1}, q'_{k+1} reached by w_{k+1} from q and q' are different because the suffix w'_{k+1} (of $w = w_{k+1} \cdot w'_{k+1}$) separates them. Therefore, the states q_{k+1}, q'_{k+1} are different from all states in $\delta_N^*(\{q, q'\}, \text{pref}(w_k))$ are reached by w_{k+1} which correspond to the rise of the bound by considering a longer prefix of one more symbol and one new state reached by w_{k+1} in M compared to $\delta_M^*(s, \text{pref}(w_k))$;

$$|w_k| + |\delta_M^*(s, \text{pref}(w_k))| + 1 + \mathbf{2} = |w_{k+1}| + |\delta_M^*(s, \text{pref}(w_k)) \cup \delta_M^*(s, w_{k+1})| + 1.$$

Case 2: If $s_{k+1} = \delta_M^*(s, w_{k+1}) \in \delta_M^*(s, \text{pref}(w_k))$, then the bound rises just by 1 compared to the bound related to w_k . The rise by 1 corresponds to the last input symbol x of $w_{k+1} = w_k \cdot x$. Let U_{k+1} be a set of all proper prefixes of w_{k+1} that are in the form of extensions of u M -convergent with uw_{k+1} , that is, $U_{k+1} = \{w_j \in \text{pref}(w_k) \mid uw_j, uw_{k+1} \text{ are } M\text{-convergent}\}$. If there was a prefix $w_j \in U_{k+1}$ such that the states reached by it in N from q, q' responded to the separating suffix w'_{k+1} differently, then it would contradict the assumption that w is the shortest that separates q, q' ; $w_j \cdot w'_{k+1}$ would separate them and $|w_j| < |w_{k+1}|$. Therefore, both states reached by each prefix of U_{k+1} respond equally to w'_{k+1} , that is, $\forall w_j \in U_{k+1} : |\lambda_N^*(\delta_N^*(\{q, q'\}, w_j), w'_{k+1})| = 1$. Note that the states reached by a prefix w_j of w in N from q, q' are N -divergent, that is, $\delta_N^*(q, w_j) \neq \delta_N^*(q', w_j)$. If all the responses of state pairs reached by $w_j \in U_{k+1}$ to w'_{k+1} are different from the response of q_{k+1} or q'_{k+1} where $q_{k+1} = \delta^*(q, w_{k+1})$ and $q'_{k+1} = \delta^*(q', w_{k+1})$, then the bound holds as at least one of q_{k+1}, q'_{k+1} is distinct from all states reached by $\{u, v\} \cdot \text{pref}(w_k)$. Otherwise, that is, both responses of q_{k+1}, q'_{k+1} to w_{k+1} are produced by some states reached by sequences of $\{u, v\} \cdot \text{pref}(w_k)$, there are two cases.

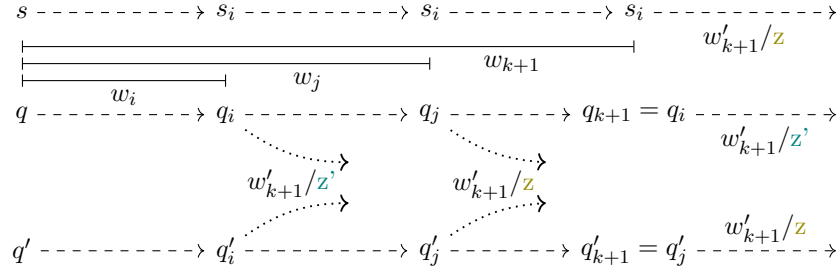


Figure 8.1: The bound from Lemma 8.7 for convergent sequences

The case when at least one of q_{k+1}, q'_{k+1} is different from all these states is not interesting as the bound holds. The other case when both q_{k+1}, q'_{k+1} are reached by some sequences of $\{u, v\} \cdot \text{pref}(w_k)$, is proven based on the following Lemma 8.8. Let V_z be a set of all sequences of U_{k+1} that lead from q to a state that responds to w'_{k+1} with output sequence z , that is, $V_z = \{w_j \in U_{k+1} \mid \lambda_N^*(\delta_N^*(q, w_j), w'_{k+1}) = z\}$, and let w_z be the shortest sequence of V_z . If w_{k+1} is the shortest prefix that reaches states previously reached by $\{u, v\} \cdot \text{pref}(w_k)$, then sequences of V_z reach from q and q' at least $|V_z| + 1$ distinct states as w_z reaches two distinct states and the other prefixes reach at least one new distinct state. Therefore, the total number of distinct states reached by prefixes of w_k is $\sum_{z \in Z} (|V_z| + 1) = |U_{k+1}| + |Z|$ where $Z = \{\lambda_N^*(\delta_N^*(q, w_j)) \mid w_j \in U_{k+1}\}$. There are at least two different responses to w'_{k+1} , that is, $|Z| \geq 2$. Hence, there is at least one more distinct state in N than the bound set for w_k . A minimal example is sketched in Figure 8.1 where $w_i \in V_{z'}$, $w_j \in V_z$, $i < j$, $q_{k+1} = q_i$ and $q'_{k+1} = q'_j$. Both q_i, q'_i are needed to satisfy the bound for w_i , however, q_j, q'_j are also different from all previous so that $|w_j| + |\delta_M^*(s, \text{pref}(w_j))| + 1 < |\delta_N^*(\{q, q'\}, \text{pref}(w_j))|$. One of q_i, q'_i and one of q_j, q'_j correspond to those whose responses are in Z . As the bound is not tight, it also holds for w_{k+1} that does not reach any 'new' distinct state; one of q_j, q'_j could be considered that is used instead. Lemma 8.8 below is used to prove that there cannot be another prefix that would lead to states reached by sequences of U_{k+1} . Therefore, each additional distinct state, like the one of q_j, q'_j , is considered for at most one longer prefix, like the w_{k+1} , to prove that the bound holds.

Both cases describe that the bound holds if a longer prefix w_{k+1} is considered. Therefore, Lemma 8.7 is proven.

Lemma 8.8. *Let w be the shortest separating sequence of states s and s' in a minimal completely specified DFSM M , and let S' be a set of any n' different states of M , that is, $|S'| = n' > 1$. Then, there are at most $n' - 1$ proper prefixes of w that reach a pair of states of S' , that is, $|\{w_k \in (\text{pref}(w) \setminus \{w\}) \mid \delta^*(s, w_k), \delta^*(s', w_k) \in S'\}| \leq n' - 1$.*

Lemma 8.8 is based on the well-known fact that the shortest separating sequence of any state pair has length up to $n - 1$.

Proof. Lemma 8.8 is proven by induction on the size of S' . Given w according to Lemma 8.8, let $w = w_k \cdot w'_k$ for all $0 \leq k < |w|$ such that $|w_k| = k$.

Base case: $n' = 2$. If there are $i < j$ such that $\delta^*({s, s'}, w_i) = \delta^*({s, s'}, w_j)$, then w would not be the shortest as $w_i \cdot w'_j$ would separate s and s' as well. Hence, every two states can be reached by proper prefixes of w at most once.

Inductive step: Let the result hold for all S' of size up to n' . Assume S'' of $n' + 1$ states and w_k as the longest prefix of w that transfer s, s' to a pair of states s_i, s_j of S'' . Then, w'_k separates s_i and s_j , and it also partitions all states of S'' according to their responses z to w'_k , that is, the partition is formed of disjoint block of states $B_z = \{s \in S'' \mid \lambda_M^*(s, w'_k) = z\}$. If any two states from different blocks of the partition were reached by a proper prefix w_i of w_k , then w would not be the shortest as $w_i \cdot w'_k$ would separate s and s' as well. Therefore, if a proper prefix w_i of w_k reaches a pair of states of S'' , the states are from a single block B_z . Each block B_z has at most n' states because there are $n' + 1$ states and at least two blocks that are not empty. As the result holds for all n' and only one pair of states of S'' is reached by w_k , the result holds for $n' + 1$ as well. \square

Based on Lemma 8.7, one can easily derive that for each proper prefix w' of w such that $\{u, v\} \cdot \text{pref}(w')$ is covered in an F_T -divergence-preserving state cover \bar{S} of M , the tests in \bar{S} reach at least $|w'| + n + 1$ distinct states in N . This is described formally in [SPY12, Lemma 4] followed by a proof. If w' has length $m - n$, then $m + 1$ distinct states are reached in N that can be distinguished from M . However, as N is considered to have at most m states, this is a contradiction and it follows that u and v are F_T -convergent. Theorem 8.6 is proven.

The flaw in [SPY12, Lemma 3] was pointed out in [KK15], however, no correction was provided. A more general framework was proposed instead. Its proof thus also proves Theorem 8.6 that is a specification of the framework proposed in [KK15]. The framework called deduction pattern allows to bound the number of states that are reached by a set of sequences. An amended version for completely specified machines is proposed in the following theorem.

Theorem 8.9. (amended [KK15, deduction pattern 13]) *For each completely specified DFSM N , a set $U \subseteq T$ and natural number k with $0 < k \leq |U|$, a sufficient condition for $|\delta_N^*(q_0, U)| \leq k$ is that there exists a function f and $l = \max(0, m - |\{f(u) \mid u \in T\}| - k + 1)$ such that*

(i) $f(u) = f(u')$ for every subset $\{u, u'\}$ of T with $\delta_N^*(q_0, u) = \delta_N^*(q_0, u')$,

(ii) $\{(u', f(uu')) \mid |u'| \leq l\}$ is the same for every $u \in U$.

If $k = 1$, then Theorem 8.9 gives a sufficient condition of convergence of sequences in U . Unfortunately, Theorem 8.9 is not constructive and it is not shown how to find a function f to prove the convergence of sequences. Therefore, a different approach was developed as the next section describes.

8.2 State Domains of Test Sequences

The previous section indicated that one can partly encode the transition function of a particular machine by proving convergence of test sequences. One usually needs to know which test sequences are F_T -divergent and which are already proven to be F_T -convergent. Instead of handling just sets of sequences, this section describes a more convenient structure to store information about divergence of test sequences.

The structure is called a *state domain*, or simply a *domain*, and it is described by a function that assigns to each test sequence t of T a subset of states that could be reached by t according to the comparison with the fixed access sequences \bar{s} of states. The domain function $\phi_{T,\bar{s}} : X_{\uparrow}^* \rightarrow \mathcal{P}(S)$ is defined for all test sequences t in T such that $\phi_{T,\bar{s}}(t) = \{s \in S \mid t, \bar{s} \text{ are not } F_T\text{-divergent}\}$. The domain function can be immediately extended to classes of convergent sequences as $\Phi_{T,\bar{S}}([t]) = \{s \in S \mid [t], [\bar{s}] \text{ are not } F_T\text{-divergent}\}$. Note that the subscripts are used in the definitions to indicate the dependency on the particular test suite T and state cover \bar{S} , however, the subscripts will be omitted as these sets of sequences are clear from the context. The domain functions on tests and their equivalence classes are denoted deliberately by different symbols, lower- and upper-case of a Greek letter, respectively. For each test $t \in T$, it holds that the domain of $[t]$ is the intersection of domains of individual tests in $[t]$ with respect to any state cover formed of sequences convergent with the access sequences of \bar{S} , that is,

$$\Phi_{T,\bar{S}}([t]) = \bigcap_{t' \in [t], \bar{U} \in U} \phi_{T,\bar{U}}(t') \subseteq \bigcap_{t' \in [t]} \phi_{T,\bar{S}}(t')$$

where $U = \{\bar{U} \mid \forall \bar{s} \in \bar{S} \exists u \in [\bar{s}] : u \in \bar{U}\}$.

Figure 8.2 captures an example of domain functions for test sequences ‘aaa’, ‘ab’, ‘ba’, ‘caa’ and ‘cbb’. Responses to the sequences are given by the Mealy machine in Figure 4.1 and they identify only the states A, B and E. The test sequences with responses are visualized as a *testing tree* on the left of Figure 8.2. A testing tree is a prefix tree such that each node corresponds to the state reached from the initial state by the sequence labelling the path from the root of the tree; root corresponds to the initial state. Nodes representing identified states are coloured and are called *state nodes*. All nodes are arbitrarily numbered and the state domains corresponding to their access sequences are shown at their bottom right (except state nodes). For instance, the domain of test sequence ‘b’ (node 5) contains only A and E because state B responds to ‘a’ differently; ‘a’ is a separating sequence of B (node 2) and the state reached by ‘b’ (node 5). The state reached by ‘c’ (node 7) is distinguished from both B (node 2) and E (node 1) so that $\phi(c) = \{A\}$. This indicates the convergence of ‘c’ and ‘ε’ that is the access sequence of state A. Assume that all conditions are fulfilled and the convergence is confirmed. Then, convergent classes of the five test sequences can be visualized as a *convergent graph* on the right of Figure 8.2. A convergent graph is created from a testing tree such that the subtree of a node reached by sequence u is merged

into the state node r_s of state s if u is proven to be convergent with \bar{s} . Nodes of a convergent graph represent convergent classes and are called *convergent nodes*. In Figure 8.2, the nodes are labelled with the number of testing tree nodes that correspond to convergent sequences. For example, nodes 1 and 8 represent convergent class [a] (and state E) in the convergent graph. Access sequences ‘a’ and ‘ca’ of nodes 1 and 8 are convergent because the sequences ‘ ε ’ and ‘c’ represented by nodes 0 and 7 (predecessors of nodes 1 and 8) were confirmed to be convergent. Similarly, nodes 5 and 10 represent convergent class [b]. Notice that the domain $\Phi([b])$ provides better information about the correspondence to a particular state node than the state domains ϕ of both sequences separately; $\Phi([b]) \subseteq \phi(b) \cap \phi(cb) = \{A,E\} \cap \{A,B\} = \{A\}$.

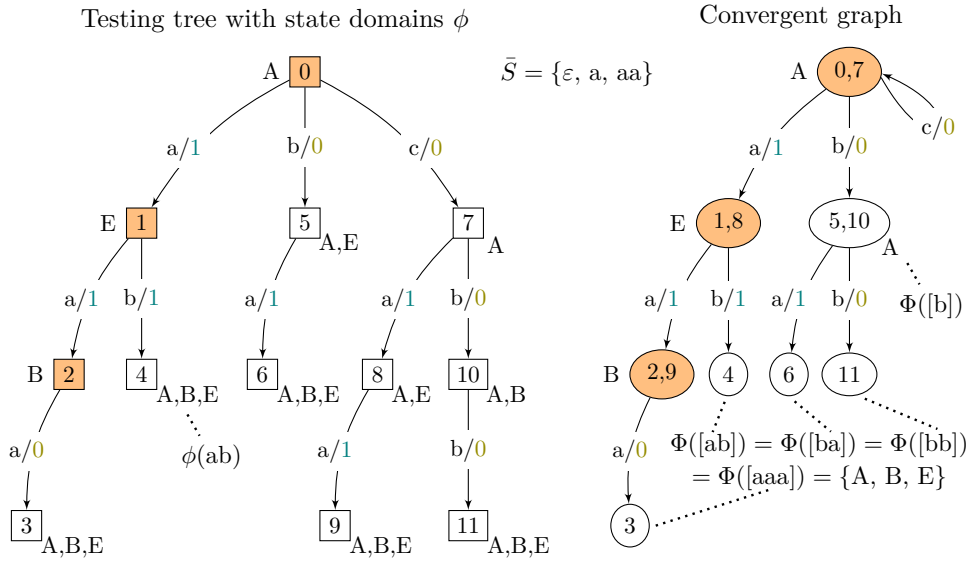


Figure 8.2: State domains of test sequences ‘aaa’, ‘ab’, ‘ba’, ‘caa’ and ‘cbb’

How can domain functions help? Domains keep track of F_T -divergence. It is really easy because as soon as a new test t is added to T , each prefix t' of t in T is compared against the access sequences of states and state s is eliminated from the domains of t' and $[t']$ if t' and \bar{s} are separated by the related suffix of t . This is nice but domains are really helpful because they can be employed to prove F_T -convergence of two tests as well as completeness of the entire test suite. Before Theorem 8.6 is rephrased using domains, let us describe a condition when a subset of test sequences U is F_T -divergence-preserving. As a reminder, two tests are F_T -divergent if they are T -separable. A set of tests $U \subseteq T$ is F_T -divergence-preserving if every two tests in U that lead to different states in M have F_T -convergent tests that are T -separable, that is, $\forall u, v \in U : \delta_M^*(s_0, u) \neq \delta_M^*(s_0, v) \implies \exists u' \in [u], v' \in [v] : u', v'$ are T -separable.

Theorem 8.10. Given a test suite T for a DFSM M , a minimal state cover \bar{S} for M and $l = m - n \geq 0$, let t_i and t_j be M -convergent tests in T , t_i and

t_j are F_T -convergent if the following conditions hold for all sequence w of length up to l , that is, for $w = \varepsilon$ as well.

- 1) There are $t'_i \in [t_i]$ and $t'_j \in [t_j]$ such that $t'_i w, t'_j w \in T$
- 2) States reached by the access sequences in \bar{S} are identified in T , that is,

$$\forall s \in S : \Phi([\bar{s}]) = \{s\}.$$

- 3) States reached by $t_i w, t_j w$ are identified in T , that is,

$$\exists s \in S : \Phi([t_i w]) = \{s\} = \Phi([t_j w]).$$

- 4) For each prefix w' of w that leads to a different state than w , the M -divergence of the related access sequences is preserved in F_T by distinguishing them with a separating suffix, that is,

$$\begin{aligned} & \forall w' \in \text{pref}(w) : \delta_M^*(s_0, t_i w') \neq \delta_M^*(s_0, t_j w') \implies \\ & (\forall ([u], [v]) \in \{[t_i w'], [t_j w']\} \times \{[t_i w], [t_j w]\} : [u], [v] \text{ are } T\text{-separable}). \end{aligned}$$

Proof. Theorem 8.10 is proven by showing that all requirements of Theorem 8.6 are met. The condition 1 of Theorem 8.10 ensures that T contains all extensions of $[t_i]$ and $[t_j]$ that have length up to l . It satisfies the first part of Theorem 8.6 that such sequences exist in T . The second part of Theorem 8.6 is that these sequences need to be included in an F_T -divergence-preserving state cover. The conditions 2, 3 and 4 are in Theorem 8.10 for this purpose. The condition 2 ensures that there is an F_T -divergence-preserving state cover because a domain is a singleton containing state s only if the related sequence is T -separable from the access sequences of all states different from s . Let U be an F_T -divergence-preserving state cover such that $U = \bar{S}$. U is gradually extended with sequences of $\{t_i, t_j\} \cdot \text{pref}(w)$. If U is to remain F_T -divergence-preserving, then each sequence t_a that should be added needs to be T -separable from all sequences in T with respect to the F_T -convergent sequences. At first, each sequence t_a is distinguished from all M -divergent access sequences of \bar{S} . This is ensured by the condition 3 of Theorem 8.10. Note that t_a does not have to be T -separable with exactly \bar{s}_i if they are M -divergent but with at least one sequence u that is proven to be F_T -convergent with \bar{s}_i . The same holds for t_a itself, hence, any $t'_a \in [t_a]$ and any $u \in [\bar{s}_i]$ can prove F_T -divergence of t_a and \bar{s}_i . The only thing that remains to ensure is what the condition 4 captures. As U is extended with t_a 's of $\{t_i, t_j\} \cdot \text{pref}(w)$, U begins to contain sequences that are not yet proven to be F_T -convergent with the access sequence of a state. These sequences also need to be shown to be divergent if they are M -divergent. As t_i, t_j are M -convergent, any extension of them is convergent as well. Therefore, it is sufficient to compare extensions of only one of those to find out if they are M -divergent; for a prefix w' of w if $t_i w'$ and $t_j w'$ are M -divergent, then $t_j w'$

and $t_j w$ are also M -divergent. If w' is a prefix of w and $t_i w', t_i w$ lead to different states, then there are four pairs of M -divergent sequences that need to be proven to be F_T -divergent. The pairs are $(t_i w', t_i w)$, $(t_j w', t_j w)$, $(t_i w', t_j w)$ and $(t_j w', t_i w)$. Again, any sequence F_T -convergent with these four sequences can be extended by a separating sequence instead of these four sequences to show their T -separability. If all four conditions of Theorem 8.10 are fulfilled, all M -divergent sequences of U are T -separable and so F_T -divergent. Hence, U is F_T -divergence-preserving state cover for M containing $\{t_i, t_j\} \cdot \text{pref}(w)$ which gives F_T -convergence of t_i and t_j according to Theorem 8.6. \square

Using domains thus allows one to derive F_T -convergence but domains also provide a different view on test sequences. Domains were used to construct a checking sequence and so-called *state recognition patterns* (SRP) reduced the domains based on tests in a test suite under construction in [KK12]. One of these SRPs is described in the following theorem. It works only if no extra state is assumed, that is, $m = n$.

Theorem 8.11. *Given a test suite T for a reduced DFSM M with n states, $n = m$, and an F_T -divergence-preserving minimal state cover \bar{S} , for any tests u and v in T if there is a sequence w such that $uw, vw \in T$ and $\Phi([uw]) \cap \Phi([vw]) = \emptyset$, then u and v are F_T -divergent.*

Proof. Each test sequence needs to be convergent with exactly one access sequence of \bar{S} as \bar{S} is F_T -divergence-preserving and minimal, and $m = n$. Empty intersection of domains $\Phi([uw]), \Phi([vw])$ means that every access sequence $\bar{s} \in \bar{S}$ is F_T -divergent from $[uw]$ or $[vw]$. Hence, there would not be a state that would correspond to the state reached by uw if uw and vw were convergent, that is, $[uw] = [vw]$. As this holds for any $N \in F_T$, uw and vw need to be F_T -divergent. No single sequence w can lead from a state (reached by convergent sequences u, v) to different states (reached by divergent sequences uw, vw). Therefore, u and v are F_T -divergent. \square

Note that there are (so far) two ways to reduce the domain of u , that is, to show that $[u]$ is F_T -divergent with an access sequence \bar{s} of \bar{S} . Either there is a separating extension of u and \bar{s} so that $[u], [\bar{s}]$ are T -separable, or Theorem 8.11 is used if $n = m$. Theorem 8.11 contains a possible self-reference as the intersection of $\Phi([uw]), \Phi([vw])$ can be empty due to the use of Theorem 8.11. As all test sequences have finite length and this ‘self-reference’ needs to start somewhere, separating sequences need to reduce domains before Theorem 8.11 can be used.

8.3 Sufficient Conditions

Completeness of a test suite provides a guarantee on the number of states in the tested implementation as Proposition 8.3 describes. There are several sufficient conditions of an m -complete test suite but also some necessary conditions, for example, an m -complete test suite needs to include a transition cover, that is, all transitions are traversed during the testing [PBY96]. Sufficient conditions usually specify properties or types of test sequences such that the

test suite T is m -complete if they hold or are present in T . The goal is to have conditions that are both necessary and sufficient. If this is hard to achieve, the weaker sufficient conditions are available, the smaller m -complete test suite can be constructed. This section proposes a new sufficient condition that combines two most advanced sufficient conditions which makes it the weakest sufficient condition of an m -complete test suite. The first advanced sufficient condition, called here the H-condition, covers most of other sufficient conditions as it is more general. The second advanced sufficient condition, called here the SPY-condition, employs the convergence of sequences. Both conditions are defined and explained before the new sufficient condition called the S-condition is proposed.

The H-condition was proposed first just for n -complete test suites in [PBY96] and then generalized for m -complete test suites in [DEFY05]. The following theorem defines its revised version for reduced completely specified machines.

Theorem 8.12. (H-condition) *A test suite T is m -complete with respect to the reduced completely specified machine M with n states if T contains the set P of sequences $\bar{S} \cdot X^{\leq m-n+1}$ where \bar{S} is a minimal state cover of M and the following conditions hold:*

1. *for each $u \in \bar{S}$ and each $v \in P$ such that $\delta^*(s_0, u) \neq \delta^*(s_0, v)$, there should be two sequences $uw, vw \in T$ such that $\lambda^*(\delta^*(s_0, u), w) \neq \lambda^*(\delta^*(s_0, v), w)$; and*
2. *if $m > n$, then for each $u \in P \setminus \bar{S}$ and each $v \in \text{pref}(u) \setminus \bar{S}$ such that $\delta^*(s_0, u) \neq \delta^*(s_0, v)$, there should be two sequences $uw, vw \in T$ such that $\lambda^*(\delta^*(s_0, u), w) \neq \lambda^*(\delta^*(s_0, v), w)$.*

The first condition of Theorem 8.12 is divided into two cases in the literature. At first, all access sequences $u \in \bar{S}$ are made T -separable from each other and then each sequence of P that is not in \bar{S} is distinguished from all access sequences $u \in \bar{S}$ that lead to a different state. The second condition of Theorem 8.12 describes that every two sequences of P that are not in \bar{S} , one is a prefix of the other, and lead to different states, need to be made T -separable by extending with a separating sequence. If no extra state is considered, the second condition is not needed as it verifies transitions between potential extra states. As the new S-condition proposed below is a generalization of the H-condition, its proof also proves the H-condition and so the proof of Theorem 8.12 is not presented here. The original proof was proposed in [DEFY05] and the proof of the S-condition is largely based on it.

The SPY-condition was proposed [SPY12] and its revised version is captured in the following theorem.

Theorem 8.13. (SPY-condition) *Suppose that T is a test suite for a DFSM M and F_T is the corresponding fault domain of machines with up to m states. If T contains an F_T -convergence-preserving initialized transition cover for M , then T is an m -complete test suite for M .*

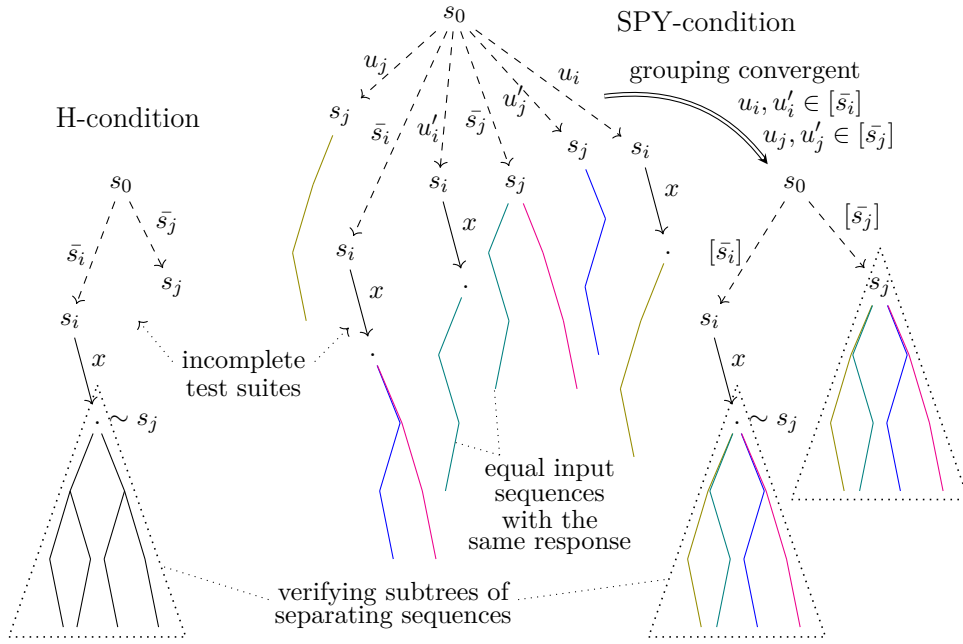


Figure 8.3: Tests verifying $\delta(s_i, x) = s_j$ according to the H- and SPY- conditions

The proof of Theorem 8.13 in [SPY12] shows that each machine N in F_T needs to be equivalent to M due to the isomorphism between states of M and N . The initial states correspond to each other as T contains their access sequence, that is, $\varepsilon \in T$ as T is initialized. All M -convergent sequences of the transition cover are F_T -convergent and so they lead to the same state in every N , that is, there is one to one correspondence between states of M and states of N . All transitions of M are tested as T contains transition cover. Therefore, Theorem 8.13 holds.

Both the H- and SPY-conditions have the same approaches to m -completeness as both verify each transition if it is implemented correctly. The difference is how they verify the correct implementation of a transition. Consider test sequences that are in a test suite according to both conditions for the purpose of verification that the transition on input x leads from state s_i to state s_j . These tests are sketched in Figure 8.3; test sequences required by the H-condition are on the left and those required by the SPY-condition are in the middle and also on the right where they are grouped according to their F_T -convergence that was already proven. The test suites are depicted as a tree. A tree is a suitable representation because it eliminates the redundancy of common prefixes; each test labels a path from the root to a leaf and a common prefix of any two tests lead to the same node.

The H-condition verifies the transition by extensions of $\bar{s}_i x$ that form a *verifying subtree*. A verifying subtree contains all sequences of length $m - n$, all domains related to these sequences are singletons, and each prefix of these sequences that leads to a different state is T -separable. A verifying subtree has depth at least $m - n$ but mostly it is more than that because there are separating sequences that extends those fixed sequences of length $m - n$.

The SPY-condition aims to create the verifying subtree, however, it uses the verifying subtree to prove F_T -convergence of sequences $\bar{s}_i x$ and \bar{s}_j . According to Theorem 8.6, there needs to be two verifying subtrees, one extending $\bar{s}_i x$ and one \bar{s}_j . In addition, not only prefixes of sequences in the subtree are T -separable from their extensions that lead to different states but also the same prefixes in the other subtree need to be T -separable. This may result in a bigger verifying subtree than in the case of the H-condition. Moreover, there are two such subtrees. On the other hand, the SPY-condition allows one to distribute the sequences of a verifying subtree over convergent sequences which may lead to a smaller test suite.

To sum up the comparison of the two sufficient conditions, the H-condition requires one verifying subtree to verify the target state of a transition (s_i, x) but all of its sequences need to extend the fixed $\bar{s}_i x$. The SPY-condition proves the convergence of the target state $\delta(s_i, x)$ and the related state s_j by extending each with a verifying subtree that can be distributed over the convergent sequences but all sets including the same sequence of length $m - n$ from both verifying subtrees need to be F_T -divergence-preserving. Note that verifying subtrees extend all \bar{s}_j implicitly in the case of the H-condition because they are formed of the subtrees that verify target states of transitions leading from s_j . Nevertheless, subtrees of $\bar{s}_i x$ and \bar{s}_j are not related by the H-condition and there is no requirement on the sequences from different subtrees to be separated.

The new sufficient condition combines both described conditions in a way that one can choose transitions which will be proven just by a verifying subtree appended to the target state and which will be verified by the convergence, that is, by Theorem 8.6 or by Theorem 8.10. In addition, as some sequences are proven to be F_T -convergent, sequences of any verifying subtree can be then distributed over the convergent sequences. Therefore, the H-condition is generalized by allowing one to extend convergent sequences instead of fixed access sequences $\bar{s}_i x$ and the SPY-condition is weakened that some sequences $\bar{s}_i x$ do not have to be proven to be F_T -convergent with the related \bar{s}_j . Nevertheless, all sequences $\bar{s}_i x$ are F_T -convergent with the related \bar{s}_j in the end (without being proven to be convergent during the construction of T using Theorem 8.6) because the entire test suite T is m -complete. The new sufficient condition follows.

Theorem 8.14. (S-condition) *A test suite T is m -complete with respect to the reduced completely specified DFSM M with n states and the corresponding fault domain F_T if \bar{S} is an initialized minimal state cover of M and the following conditions hold:*

1. *for each state $s \in S$ and each sequence v of length up to $m - n + 1$, T contains a sequence uv such that u is convergent with \bar{s} , that is,*

$$\forall s \in S \forall v \in X^{\leq m-n+1} \exists u \in [\bar{s}] : uv \in T,$$

2. *for each access sequence $\bar{s}_i \in \bar{S}$ and its extension v of length up to $m - n + 1$ that leads to state s_k , T secures that the sequence $\bar{s}_i v$ is*

F_T -divergent with the access sequences $\bar{s}_j \in \bar{S}$ that are different from \bar{s}_k , for example, by making them T -separable, that is,

$$\forall s_i, s_j \in S \forall v \in X^{\leq m-n+1} \exists u_i \in [\bar{s}_i] \exists u_j \in [\bar{s}_j] \exists w \in X_{\uparrow}^* : \\ \delta^*(s_i, v) \neq s_j \implies (u_i v w, u_j w \in T \wedge T(u_i v, w) \neq T(u_j, w)),$$

3. if $m > n$, then for each access sequence $\bar{s} \in \bar{S}$ and its extension v of length up to $m - n + 1$ that leads to state s_k , T secures that the sequence $\bar{s}v$ is F_T -divergent with all sequences $\bar{s}v'$ that lead to states different from s_k and where v' is a prefix of v , for example, by making them T -separable, that is,

$$\forall s \in S \forall v \in X^{\leq m-n+1} \forall v' \in \text{pref}(v) \exists u, u' \in [\bar{s}] \exists w \in X_{\uparrow}^* : \\ \delta^*(s, v) \neq \delta^*(s, v') \implies (u v w, u' v' w \in T \wedge T(u v, w) \neq T(u' v', w)).$$

Theorem 8.14 requires by the first condition that all extensions appear in the test suite T , and the other two conditions correspond to the ones of the H-condition defined in Theorem 8.12. As the definition of the S-condition is the same as the one of the H-condition except the use of convergence, the proof is an amendment of the original proof of the H-condition that was proposed in [DEFY05].

Proof. Assume a test suite T that satisfies all the requirements of Theorem 8.14, and a DFSM $N \in F_T$ with m states such that N and M are distinguishable. Let w be the shortest input sequence that distinguishes some state $s_i \in S$ from $\delta_N^*(q_0, \bar{s}_i)$, that is, $\lambda_M^*(\delta_M^*(s_0, \bar{s}_i), w) \neq \lambda_N^*(\delta_N^*(q_0, \bar{s}_i), w)$. Note that there is no prefix u of w such that $\bar{s}_i \cdot u \in \bar{S}$ because w would not be the shortest separating sequence; the extension of u in w would separate $s_j = \delta_M^*(s_i, u)$ from $\delta_N^*(q_0, \bar{s}_j)$. Obviously, $\bar{s}_i \cdot w$ cannot be in T as N would be distinguished from M and so $N \notin F_T$ which contradicts the assumption. The access sequence \bar{s}_i is extended with all sequences of length up to $m - n + 1$ in T according to the condition 1 of Theorem 8.14. Hence, w consists of a prefix w_k of length $m - n + 1$; $w = w_k \cdot w'_k$ and $|w_k| = m - n + 1$.

Let R be a set of the access sequences of all states and sequences formed of \bar{s}_i extended with the prefixes of w_k , that is, $R = \bar{S} \cup \bar{s}_i \cdot \text{pref}(w_k)$. The number of sequences in R is $m + 1$ because there is n states and $m - n + 1$ non-empty prefixes of w_k . As N has at most m states, there are at least two N -convergent sequences in R . Let $u, v \in R$ be N -convergent, that is, $\delta_N^*(q_0, u) = \delta_N^*(q_0, v)$. They are also M -convergent because all M -divergent sequences in R are F_T -divergent as well due to the conditions 2 and 3 of the S-condition. There are three possible cases for u and v with respect to the state cover \bar{S} .

- 1) $u, v \in \bar{S}$. All access sequences in \bar{S} are extended with a separating sequence in T according to the condition 2 of Theorem 8.14. Therefore, this case is not possible if N passes T .

- 2) $u \in \bar{S}$ and $v \notin \bar{S}$. Let v' be the extension of v in $\bar{s}_i w$ and $u = \bar{s}_j$. Then, v' distinguishes s_j from $\delta_N^*(q_0, \bar{s}_j)$ because u, v are M - and N -convergent, that is, $\lambda_M^*(s_j, v') = \lambda_M^*(\delta_M^*(s_0, v), v') \neq \lambda_N^*(\delta_N^*(q_0, v), v') = \lambda_N^*(\delta_N^*(q_0, u), v')$. As v' is a proper suffix of w and so a shorter sequence, it contradicts the assumption that w is the shortest that distinguishes states of M and N reached by an access sequence of \bar{S} .
- 3) $u, v \notin \bar{S}$. Let $u = \bar{s}_i \cdot u'$ and u be shorter than v , that is, u is a prefix of v , and let v' be the extension of v in $\bar{s}_i w$. Then, $u'v'$ distinguishes s_i from $\delta_N^*(q_0, \bar{s}_i)$ because u, v are M - and N -convergent. Again, it is a contradiction as $|u'v'| < |w|$.

Neither of the three cases above is possible so that there cannot be such a separating sequence w . For each distinguishable DFSM with at most m states, there is a sequence in T that distinguishes it from M , hence, T is m -complete. \square

Note that if $m = n$, then Theorem 8.11 about reasoning based on empty intersection of domains can be employed to secure F_T -divergence in the condition 2 of Theorem 8.14. Otherwise, there is (so far) only one way to prove F_T -divergence in general for $m \geq n$ and that is making sequences T -separable.

Chapter 9

Existing Methods

Testing methods tend to produce test suites of several sequences if the test suite is to be m -complete. Therefore, the system under test (the implementation) needs to be resettable in order to apply all test sequences and observe their responses. There are several testing methods creating tests that begin in the initial state. This chapter describes the standard testing methods in the following sections and sketches others in Section 9.8. The description of methods is based on their overview proposed in [Sou15]. Comparisons of some standard testing methods can be found, for example, in [SL89, Ura92, DEFM⁺05, ES13]. The author of the thesis has implemented most of the testing methods and so they can be employed for an experimental comparison with the new testing methods proposed in the subsequent chapters.

All standard testing methods verify each transition of the reduced completely specified specification M according to the H-condition (Theorem 8.12) or the SPY-condition (Theorem 8.13). Any test suite needs to include a transition cover U for M and each sequence of U is extended with all sequences of length $m - n$ if the H-condition is to be satisfied. Therefore, let \bar{S} be a minimal initialized state cover for M , P be a set of all access sequences extended with all sequences of length $m - n + 1$, and R be a set of the maximal sequences of P (a maximal sequence has no extension in the set), that is,

$$P = \bar{S} \cdot X^{\leq m-n+1} \quad \text{and} \quad R = P \setminus (\bar{S} \cdot X^{\leq m-n}).$$

Note that P is a minimal transition cover if $m = n$. The sets P and R simplify the description of most standard testing methods. The methods differ mainly in the way how they verify the reached state that is the target state of a transition. For this purpose, different sequences for state identification introduced in Section 1.6.1 are used. As some of these sequences (or sets of sequences) correspond to a particular state, a special concatenation operator \circ is introduced to simplify the description when, for example, a harmonized state identifier H_i of state s_i should extend each sequence u that leads to s_i . The special concatenation operator \circ is defined for a set U of sequences and a set of sets V_i such that there is a set of sequences V_i for each state s_i ,

$$U \circ V_i = \{u \cdot v \mid u \in U, v \in V_i \text{ such that } V_i \text{ corresponds to } s_i = \delta^*(s_0, u)\}.$$

The definition can be easily adjusted if V_i is a single sequence, for example, a state verifying sequence.

9.1 ADS-method

The ADS-method proposed in [Sou15] uses an adaptive distinguishing sequence (ADS) for the verification of the target state. Given an ADS in the form of a verifying set as defined in Section 1.6.1, let d_i be the state verifying sequence of state s_i . Then, a test suite T is formed as follows:

$$T = P \circ d_i$$

Note that an ADS does not have to exist for M so that this method fails if there is no ADS.

9.2 SVS-method

The SVS-method proposed in [Sou15] is similar to the ADS-method as it uses state verifying sequences (SVS) to verify the target state. However, the entire verifying set (VSet) needs to be appended to each sequence of P that is not maximal because otherwise the divergence of different sequences would not be confirmed. The SVS-method is similar to the UIOV-method proposed in [CVO89] but it does not aim to construct a checking sequence that is just n -complete. A test suite is created as follows:

$$T = (P \setminus R) \cdot VSet \cup R \circ SVS_i$$

If a state s_i does not have an SVS_i , then a state characterizing set W_i is used instead so that the SVS-method can be used for all reduced completely specified machines.

9.3 W-method

The W-method is the oldest testing method. It was proposed in [Vas73, Cho78] and it simply appends a characterizing set W to each sequence of P .

$$T = P \cdot W$$

9.4 Wp-method

The Wp-method, or the partial W-method, was introduced in [FKA⁺91] and similar to the W-method it appends a characterizing set W to each sequence of P except the maximal ones. The target state of a maximal sequence is verified with the corresponding state characterizing set W_i . Nevertheless, each state characterizing set W_i must be a subset of the characterizing set W . Then, a test suite T is formed as follows:

$$T = (P \setminus R) \cdot W \cup R \circ W_i$$

9.5 HSI-method

The HSI-method proposed in [Pet91] uses harmonized state identifiers (HSI) for the verification of transitions. As for any two states s_i, s_j there is a common separating prefix in the HSIs H_i and H_j for states s_i and s_j respectively, the HSIs are sufficient to verify the target state of each sequence of P , that is,

$$T = P \circ H_i$$

9.6 H-method

The H-method proposed in [DEFY05] is similar to the HSI-method but it chooses separating sequences on the fly instead of using fixed HSIs. It basically follows the H-condition (Theorem 8.12) so that it extends P with separating sequences of two different states reached by particular sequences of P . Formally, the stepwise construction of a test suite T is described in the following four steps:

- 1) $T = P$.
- 2) For each $\bar{s}_i, \bar{s}_j \in \bar{S}$ such that $s_i \neq s_j$, if T does not contain a common separating extension w of \bar{s}_i and \bar{s}_j (that is, $\neg \exists \bar{s}_i \cdot w, \bar{s}_j \cdot w \in T : \lambda^*(s_i, w) \neq \lambda^*(s_j, w)$), then add such $\bar{s}_i \cdot w$ and $\bar{s}_j \cdot w$ to T .
- 3) For each $\bar{s}_i \in \bar{S}, v \in P \setminus \bar{S}$ such that $s_i \neq s_v = \delta^*(s_0, v)$, if T does not contain a common separating extension w of \bar{s}_i and v (that is, $\neg \exists \bar{s}_i \cdot w, v \cdot w \in T : \lambda^*(s_i, w) \neq \lambda^*(s_v, w)$), then add such $\bar{s}_i \cdot w$ and $v \cdot w$ to T .
- 4) If $m > n$, then for each $u, v \in P \setminus \bar{S}$ such that $u \in \text{pref}(v)$ and $s_u = \delta^*(s_0, u) \neq s_v = \delta^*(s_0, v)$, if T does not contain a common separating extension w of u and v (that is, $\neg \exists u \cdot w, v \cdot w \in T : \lambda^*(s_u, w) \neq \lambda^*(s_v, w)$), then add such $u \cdot w$ and $v \cdot w$ to T .

The H-method is one of advanced testing methods that produce small test suites compared to the other methods. Nevertheless, it is not clear how the separating extensions w 's are chosen in the steps 2–4. Hence, the following sections describe an implementation of the H-method and a running example of the construction of a test suite by the H-method. The original paper does not provide the implementation details, hence, the implementation is based on the version in [Sou15] developed by the author of the thesis.

9.6.1 Implementation

Algorithm 13 captures all four steps of the H-method such that the work of step 2 is delegated to the function `DISTINGUISH` (Algorithm 14) and the work of steps 3 and 4 to the recursive functions `DISTINGUISHFROMSTATECOVER` (Algorithm 15) and `DISTINGUISHFROMSET` (Algorithm 16), respectively.

Each two access sequences u, v in \bar{S} are to be separated by a common extension in the step 2. This is done by the function `DISTINGUISH` defined in Algorithm 14. It first obtains a common extension w' in T by the function

Algorithm 13: H-method

input : A minimal DFSM M with n states
input : A number of extra states l ; $m = n + l$
output : An m -complete test suite T for M

```

1  $T \leftarrow \bar{S} \cdot X^{m-n+1}$  // step 1
2 foreach  $u, v \in \bar{S}$  such that  $u \neq v$  do // step 2
3    $\lfloor$  DISTINGUISH( $u, v$ )
4 foreach  $v \in (\bar{S} \cdot X \setminus \bar{S})$  do // step 3
5    $\lfloor$  DISTINGUISHFROMSTATECOVER( $v, \bar{S}, l$ )
6 if  $l > 0$  then // step 4
7   foreach  $v \in (\bar{S} \cdot X \setminus \bar{S})$  do
8      $\lfloor$  DISTINGUISHFROMSET( $v, \emptyset, l$ )
9 return  $T$ 

```

GETBESTPREFIXOFSEPARATINGSEQUENCE (Algorithm 17) such that the shortest separating sequence w of the states reached by uw' and vw' is estimated to enlarge T the least. Then, $uw'w$ and $vw'w$ are added to T . Note that one of $uw'w, vw'w$ can already be in T . If u and v were already separated in T , then GETBESTPREFIXOFSEPARATINGSEQUENCE would return the estimated number e of new symbols extending T equal to 0 and so no sequence is added to T .

Algorithm 14: DISTINGUISH(u, v)

```

1  $(e, w') \leftarrow$  GETBESTPREFIXOFSEPARATINGSEQUENCE( $u, v$ )
2 if  $e > 0$  then
3    $w \leftarrow$  shortest separating sequence of  $\delta^*(s_0, uw'), \delta^*(s_0, vw')$ 
4    $\lfloor$  add  $uw'w, vw'w$  to  $T$  if not there already

```

The function DISTINGUISH does the main work also in the steps 3 and 4. Algorithm 15 describes the recursive function DISTINGUISHFROMSTATECOVER that is called for each $v \in P \setminus \bar{S}$ as step 3 requires. Every such v is then distinguished from each access sequence in \bar{S} that leads to a state different from the one reached by v . Similarly, step 4 is implemented by Algorithm 16 such that the recursive function DISTINGUISHFROMSET is called for each $v \in P \setminus \bar{S}$. All prefixes u of such v that are not in \bar{S} are collected in a set V so that u, v can then be easily distinguished and the condition of step 4 is fulfilled.

The function GETBESTPREFIXOFSEPARATINGSEQUENCE described in Algorithm 17 is used in the function DISTINGUISH to check extensions of the given sequences u and v if there is a common extension in T that separates states reached by u and v . If there is not such an extension, then the function returns the prefix of a separating sequence of the states reached by u and v such that the prefix is a common extension of both u and v in T and the

Algorithm 15: DISTINGUISHFROMSTATECOVER($v, \bar{S}, depth$)

```

1 if  $depth > 0$  then
2   foreach  $x \in X$  do
3      $\lfloor$  DISTINGUISHFROMSTATECOVER( $vx, \bar{S}, depth - 1$ )
4 foreach  $u \in \bar{S}$  such that  $\delta^*(s_0, u) \neq \delta^*(s_0, v)$  do
5    $\lfloor$  DISTINGUISH( $u, v$ )

```

Algorithm 16: DISTINGUISHFROMSET($v, V, depth$)

```

1 if  $depth > 0$  then
2   add  $v$  to  $V$ 
3   foreach  $x \in X$  do
4      $\lfloor$  DISTINGUISHFROMSET( $vx, V, depth - 1$ )
5   pop  $v$  from  $V$ 
6 foreach  $u \in V$  such that  $\delta^*(s_0, u) \neq \delta^*(s_0, v)$  do
7    $\lfloor$  DISTINGUISH( $u, v$ )

```

entire separating sequence adds minimal number of symbols to T when it is appended to u, v in T . The function compares all extensions of u, v in T by trying to start with each input x . If ux and vx are included in T , then a recursive call is made to check extensions of ux, vx in T unless x separates states reached by u, v or x transfers these states to the same state. In the former case (line 5 of Algorithm 17), the function returns that the given sequence are already distinguished in T . In the latter case (line 6), another input x is considered as no separating sequence can start with x . The recursive call of the function returns two values, an estimate e of symbols that enlarge T and a sequence w' that is an extension of the given ux and vx in T and also is a prefix of the separating sequence of states reached by ux, vx . If e is 0, then ux and vx are distinguished in T and so u, v are distinguished as well. Otherwise, e is compared with $minEstimate$ that stores the minimal estimate amongst inputs x considered so far. $minEstimate$ is updated if e is lower or equal (line 9). Then, also $bestPrefix$ is updated with xw' . Note that the equality in the condition on line 9 means that longer extensions are favoured in the selection. This aims to select extensions that are not proper prefixes of other extensions and so the number of sequences in T does not increase when the chosen separating sequence is appended to u and v .

If an input x does not start extensions of both u and v in T , then the function ESTIMATEGROWTHOFT estimates the number of symbols that would be added to T to distinguish the given states s_u and s_v if the separating sequence began with x . Algorithm 18 defining ESTIMATEGROWTHOFT assumes that one of sequences u, v is extended with x in T . Therefore, it returns 1 if x separates the given states. If x cannot begin the shortest separating sequence because the states go on x to themselves or to a single

Algorithm 17: GETBESTPREFIXOFSEPARATINGSEQUENCE(u, v)

```

1  $s_u \leftarrow \delta^*(s_0, u), s_v \leftarrow \delta^*(s_0, v)$ 
2  $minEstimate \leftarrow 2n, bestPrefix \leftarrow \varepsilon$ 
3 foreach  $x \in X$  do
4   if there are  $w_u, w_v$  such that  $uxw_u, vxw_v \in T$  then
5     if  $\lambda^*(s_u, x \uparrow) \neq \lambda^*(s_v, x \uparrow)$  then return  $(0, \varepsilon)$ 
6     if  $\delta(s_u, x) = \delta(s_v, x)$  then continue
7      $(e, w') \leftarrow$  GETBESTPREFIXOFSEPARATINGSEQUENCE( $ux, vx$ )
8     if  $e = 0$  then return  $(0, \varepsilon)$ 
9     if  $e \leq minEstimate$  then  $minEstimate \leftarrow e, bestPrefix \leftarrow xw'$ 
10  else
11     $e \leftarrow$  ESTIMATEGROWTHOFT( $s_u, s_v, x$ )
12    if there is no  $w$  such that  $uxw \in T$  or  $vxw \in T$  then  $e \leftarrow e + 1$ 
13    if  $e < minEstimate$  then  $minEstimate \leftarrow e, bestPrefix \leftarrow x$ 
14 return  $(minEstimate, bestPrefix)$ 

```

state, then $2n$ is returned. Note that twice the number of states is always greater than twice the length of the shortest separating sequence of a state pair. Otherwise, ESTIMATEGROWTHOFT estimates that T would be enlarged by $2 \cdot |w| + 1$ symbols where w is the shortest separating sequence of next states of s_u, s_v on x such that w would be appended to both ux, vx and $+1$ stands for one x appended to either u or v . The function assumes that no prefix of w is an extension of ux or vx in T , therefore, the estimate is always higher than or equal to the actual number of symbols that enlarge T by appending w to ux, vx . Note that instead of constructing w for every call of ESTIMATEGROWTHOFT the H-method possesses a state pair array (SPA) of all separating sequences encoded by connections between cells of the SPA. This SPA provides for each state pair both the length of the shortest separating sequence and information if an input can begin a separating sequence [Sou15]. The separating sequence w is then obtained based on the connections between the cells of SPA only once for each call of DISTINGUISH. The estimate e returned by ESTIMATEGROWTHOFT is increased by 1 if both u, v are not extended with x (line 12 of Algorithm 17). Then, it is compared with $minEstimate$ and if e is lower, then $bestPrefix$ is updated with x . Finally, $minEstimate$ and $bestPrefix$ are returned as the values capturing the best way to distinguish the given u and v .

Algorithm 18: ESTIMATEGROWTHOFT(s_u, s_v, x)

```

1 if  $\lambda(s_u, x) \neq \lambda(s_v, x)$  then return 1
2 if  $\delta(\{s_u, s_v\}, x) = \{s_u, s_v\}$  or  $|\delta(\{s_u, s_v\}, x)| = 1$  then return  $2n$ 
3 let  $w$  be the shortest separating sequence of  $\delta(s_u, x)$  and  $\delta(s_v, x)$ 
4 return  $2 \cdot |w| + 1$ 

```

9.6.2 Running Example

This section describes how the H-method creates a test suite of the 5-state Mealy machine defined in Figure 4.1. A test suite usually contains sequences with common prefixes, therefore, it is suitable to represent the test suite as a successor tree called here a *testing tree*. As the H-method and other advanced testing methods compare the responses of individual test sequences, it is good to store also outputs in the testing tree. Therefore, a testing tree can be thought of as a deterministic finite-state machine such that every state except the initial one has exactly one incoming transition. An example of testing tree is in Figure 9.1 where nodes are numbered in the order when they extended the test suite T .

The H-method starts with a transition cover that is then extended to form $\bar{S} \cdot X^{m-n+1}$. The transition cover is constructed gradually according to the length of access sequences and the alphabetical order. At first, sequences of 1 input are added to the test suite T . Sequences ‘a’ and ‘b’ are the access sequences of states E and C, respectively. Therefore, they are extended in the next step of the transition cover construction as captured in Figure 9.1. By adding sequences ‘aa’, ‘ab’ and ‘ac’, T contains access sequences to all states; the initial state A is reached by the empty sequence ε . A transition cover is included in T after states C, B and D are extended with all three inputs. The testing tree then has 16 nodes; note that the root is labelled with 0. Consider 1 extra state, that is, $l = 1$ and $m = n + 1$ where $n = 5$. All the maximal sequences of the current test suite T are thus extended with each input. It is done in the order of the number of the related leaves, that is, all sequences added to T are in a queue that directs their further extensions. The step 1 of the H-method ends with 49 nodes; the last added sequence is ‘acc’.

The step 2 distinguishes pairs of access sequences in \bar{S} . First, it calls the function `GETBESTPREFIXOFSEPARATINGSEQUENCE` on sequences ε and ‘a’. They reach states A and E that are already distinguished in T by the common extension ‘aa’. Note that the states are also separated by ‘b’ but the function searches in depth first so that ‘aa’ is observed earlier. Similarly, the sequence ε is distinguished from the access sequence ‘b’ of state C and the access sequence ‘aa’ of state B by the extensions ‘aa’ and ‘a’, respectively. The function `GETBESTPREFIXOFSEPARATINGSEQUENCE` has more work with the access sequences of states A and D. Both extensions ‘a’ and ‘b’ transfer states to a single state, hence, they cannot start a separating sequence. The next extension to check is ‘ca’ that transfer states A, D into states E, B respectively. The corresponding nodes 16 and 46 of the testing tree are leaves so that `GETBESTPREFIXOFSEPARATINGSEQUENCE(‘ca’, ‘acca’)` gets estimates for each input at first. Then, it returns (2, ‘a’) because ‘a’ is the shortest separating sequence of E, B and it needs to be appended to both sequences so T would grow by 2 symbols. Nevertheless, this extension is not needed as the next extension ‘cb’ distinguishes states A, D so that T is again not enlarged. Similarly, the other pairs of access sequences in \bar{S} are found to be already distinguished in T . The step 2 thus does not add any node to the testing tree.

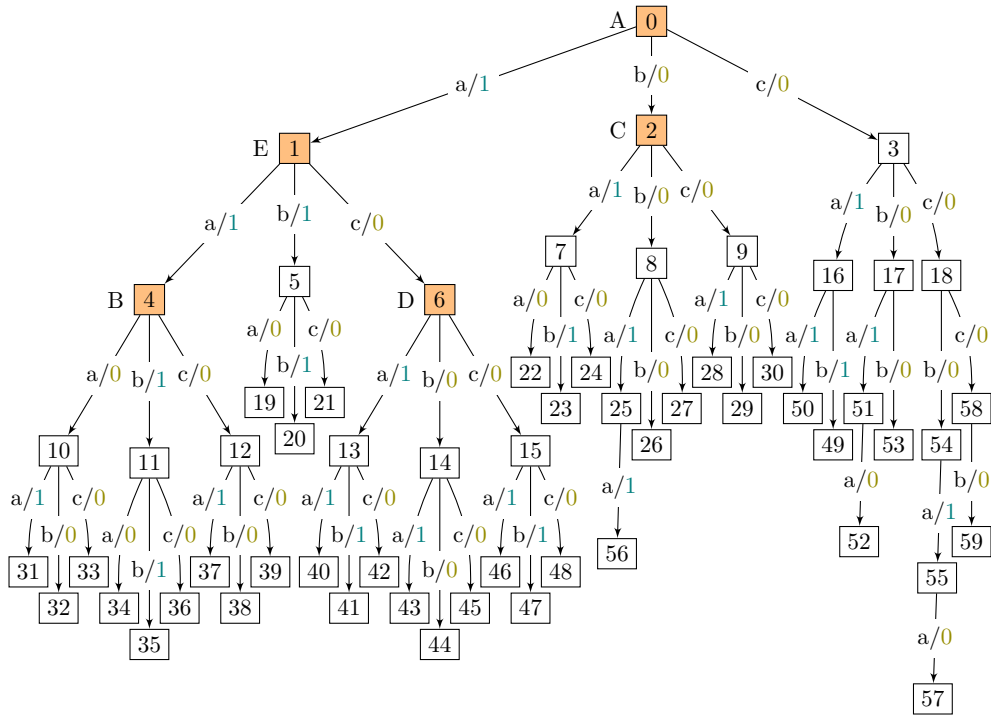


Figure 9.1: H-method: testing tree after verifying transition (A,c)

Figure 9.1 captures processing of the first sequence ‘c’ of P by the step 3. The function `DISTINGUISHFROMSTATECOVER` first recursively calls itself with one-input extensions of ‘c’. Therefore, sequences ‘ca’, ‘cb’, ‘cc’ and ‘c’ are distinguished from the fixed access sequences of states in this order. The sequence ‘ca’ reaches state E that can be easily distinguished from A by the extension ‘b’ (node 49 of the testing tree). The sequence ‘b’ also separates E from C and D but ‘a’ needs to be appended to ‘ca’ to distinguish it from the access sequence of B. It is similar with the extension of ‘cb’; ‘aa’ distinguishes C from A, B and D, and ‘b’ distinguishes C from E. An interesting case comes when ‘cc’ leading to A is to be distinguished from the access sequences of other states. First, it is distinguished from E by appending ‘b’. To distinguish ‘cc’ from ‘b’ associated with state C, the function `DISTINGUISHFROMSTATECOVER` checks the extensions in T as follows. Input ‘a’ is missing after ‘cc’ but it can start a separating sequence ‘aa’ such that w on line 3 of Algorithm 18 would be ‘a’ and the estimate e gets 3 as ‘a’ is an extension of ‘b’ in T . Therefore, $minEstimate$ is updated to 3 and $bestPrefix$ to ‘a’ on line 13 of Algorithm 17. The next input is ‘b’ that extends both ‘cc’ and ‘b’ in T . The function is thus called on their successors that correspond to states C and A. No input extends ‘ccb’ so that only the estimates are calculated for each input. The best estimate is 3 that corresponds to the separating sequence ‘aa’ (it is not 4 because ‘a’ extends ‘bb’ in T). Algorithm 18 thus returns (3, ‘a’) and $bestPrefix$ is updated to ‘ba’ because of the equality in the condition on line 9. There is no better estimate and so ‘aa’ is appended to ‘ccb’ (nodes 55 and 57 of the testing tree) and ‘a’ extends ‘bba’ (node 56); the separating

sequence is appended to both sequences simultaneously, hence the alternation of node numbers. Note that if the separating sequence ‘aa’ was chosen, T would grow only by 2 symbols instead of estimated 3 as the entire sequence already extends ‘b’ in T . However, the number of test sequences and the total number of inputs in T would be greater than in the case of separating sequence ‘baa’; ‘caa’ would be a new test sequence, that is, plus 4 symbols, in contrast to the chosen case that just extends current sequences by 3 symbols. The condition on line 9 thus tries to optimize locally the number of sequences in the resulting test suite. The step 3 also appends ‘cb’ to ‘cc’ before it starts to process the sequence ‘ab’. Notice that ‘c’ is distinguished from all access sequences of other states due to the extensions appended to its successors, that is, sequences ‘ca’, ‘cb’ and ‘cc’. Moreover, ‘c’ is also distinguished from all its successors so that the step 4 will have no work with this sequence.

9.7 SPY-method

The SPY-method proposed in [SPY12] employs the convergence relation (Definition 8.4) to reduce test branching and thus the number of sequences in the resulting test suite T . Test branching is the branching of the testing tree that groups test sequences with common prefixes. The method aims to meet the SPY-condition (Theorem 8.13) that requires an F_T -convergence-preserving initialized transition cover included in T . It is accomplished by verification of all transitions. A transition from state s_i to state s_j on input x is *verified* if $\bar{s}_i \cdot x$ and \bar{s}_j are proven to be F_T -convergent. The SPY-method first creates an F_T -divergence-preserving, initialized, prefix-closed and minimal state cover \bar{S} and then employs Theorem 8.6 to verify all transitions. Note that transitions included in the state cover are verified as \bar{S} is prefix-closed so that $\bar{s}_i \cdot x = \bar{s}_j$.

Harmonized state identifiers H_i are used like in the HSI-method to verify the reached state. However, compared to the HSI-method the SPY-method distributes sequences of a HSI over convergent sequences. This way the number of sequences in the resulting test suite T can be reduced. A separating sequence of H_i extends a current test sequence such that minimal symbols are appended and no new test sequence is added to T (if it is possible). The following two sections describe the implementation explaining the details of the SPY-method and a running example that shows how a test suite is created by this method.

9.7.1 Implementation

Algorithm 19 is an adapted version of the SPY-method proposed in the original paper [SPY12] with a small space optimization. The original version stores all classes of convergent sequences. The author of the thesis found out that only convergent classes of the fixed access sequences of states are needed in the algorithm and so only those are handled [Sou15].

The SPY-method starts with the construction of a state cover \bar{S} and harmonized state identifiers H_i . Those are then concatenated accordingly to form an F_T -divergence-preserving state cover which is a precondition in Theorem 8.6 to prove the convergence of two sequences. Each transition (s, x) such that $\bar{s} \cdot x$ is not proven to be convergent with \bar{s}_x , where $s_x = \delta(s, x)$, is verified by appending HSIs to the extensions of $\bar{s} \cdot x$ and \bar{s}_x . A queue U helps to traverse the extensions from the shortest ones to the ones of the given maximal length l that represents the number of extra states. For each extension u and each separating sequence w from the corresponding HSI, a suitable sequence to extend is chosen using the function APPENDSEPARATINGSEQUENCE called on lines 8 and 9 of Algorithm 19. When all extensions of the length up to l are appended to both convergent classes and all reached states are verified by appending the related HSIs, sequences $\bar{s} \cdot x$ and \bar{s}_x are F_T -convergent and their convergent classes can be merged. Note that the classes of their successors are merged as well; for instance, sequences of $[\bar{s} \cdot x \cdot x']$ enlarge the convergent class $[\bar{s}_i]$ if $\delta(s_x, x') = s_i$ and transition (s_x, x') is already verified. As a reminder, the special concatenation operator \circ is defined to extend each sequence u of the first set with the set of sequences H_i that is associated with the state $\delta^*(s_0, u)$.

Algorithm 19: SPY-method

input : A minimal DFSM M with n states
input : A number of extra states l ; $m = n + l$
output : An m -complete test suite T for M

- 1 $H_i \leftarrow$ harmonized state identifier of s_i , for all $s_i \in S$
- 2 $T \leftarrow \bar{S} \circ H_i$
- 3 **foreach** unverified transition (s, x) such that $s_x = \delta(s, x)$ **do**
- 4 $U \leftarrow \{\varepsilon\}$ // a queue of sequences $X^{\leq l}$
- 5 **while** U is not empty **do**
- 6 pop u from U
- 7 **foreach** $w \in H_i$ such that the related $s_i = \delta^*(s_x, u)$ **do**
- 8 APPENDSEPARATINGSEQUENCE($[\bar{s}], xuw$)
- 9 APPENDSEPARATINGSEQUENCE($[\bar{s}_x], uw$)
- 10 **if** $|u| < l$ **then** $U \leftarrow U \cup u \cdot X$
- 11 merge $[\bar{s} \cdot x]$ and $[\bar{s}_x]$, and the convergent classes of their successors
- 12 **return** T

Algorithm 20 describes the function APPENDSEPARATINGSEQUENCE that chooses a sequence u_{best} of the given convergent class such that it is the most suitable for the extension by the given sequence w . The objective function is to minimize branching of the testing tree and so the total number of sequences in T . Therefore, the default value of u_{best} is the shortest sequence of the given class $[u]$. If there is u' in $[u]$ with a maximal extension w' that is a prefix of w , then the function chooses such u' that has the longest w' . A sequence is maximal in a set if it is not a proper prefix of another sequence

in the set. Notice that the function checks whether w is already in T (line 4 of Algorithm 20). The choice of u_{best} thus minimizes the number of added symbols to T even if there is no other option than to add a new test sequence to T . Finally, the function APPENDSEPARATINGSEQUENCE either enlarges T with a new test sequence $u_{best} \cdot w$ or appends the related suffix of $u_{best} \cdot w$ to the test sequence in T that is maximal and is a prefix of $u_{best} \cdot w$.

Algorithm 20: APPENDSEPARATINGSEQUENCE($[u]$, w)

```

1  $u_{best} \leftarrow$  the shortest  $u' \in [u]$ ,  $maxLength \leftarrow -1$ 
2 foreach  $u' \in [u]$  do
3    $w' \leftarrow$  the longest prefix of  $w$  such that  $u'w' \in T$ 
4   if  $w' = w$  then return
5   if there is no  $v$  such that  $u'w'v \in T$  and  $|w'| > maxLength$  then
6      $u_{best} \leftarrow u'$ ,  $maxLength \leftarrow |w'|$ 
7 add  $u_{best} \cdot w$  to  $T$ 

```

■ 9.7.2 Running Example

This section describes how the SPY-method constructs a test suite for the 5-state Mealy machine defined in Figure 4.1. The testing tree in Figure 9.2 captures the test suite with verified transitions (A,c) and (E,b) and the four transitions verified by the state cover. Nodes of the testing tree are numbered in order of their creation, that is, when the input labelling the incoming edge was added to T . Consider 1 extra state and that the HSIs are obtained from the splitting tree using Algorithm 6. The HSIs for the machine are listed in Figure 4.2.

The method first builds a minimal prefix-closed state cover \bar{S} (nodes 0–4 of the testing tree). The access sequences are then extended with the related separating sequences of the fixed harmonized state identifiers. The separating sequences in HSIs are sorted first in the increasing order of their length and then alphabetically, therefore, for example, the access sequence ‘a’ of state E is extended with ‘b’ before ‘aa’ (nodes 7 and 8). An F_T -divergence-preserving state cover is formed of the first 16 nodes of the testing tree. Transitions for verification are ordered according to time when they were explored during the construction of the state cover; the first transition is (A,c), the second (E,b), and then transitions from state C, B and D in this order.

The first transition to verify is always verified by appending all separating sequences to a single access sequence because the convergent classes are singletons initially. In the example, the separating sequences ‘aa’ and ‘cb’ of harmonized state identifier of state A extend both ‘c’ and ε (in fact, only ‘c’ is extended as both sequences are already extensions of ε). The first transition (A,c) leads back to A, hence, the sequences ‘c’ and ε . One extra state is considered so that the target states of extensions of one input need to be verified as well. Therefore, ‘ca’ and ‘cb’ are extended with the

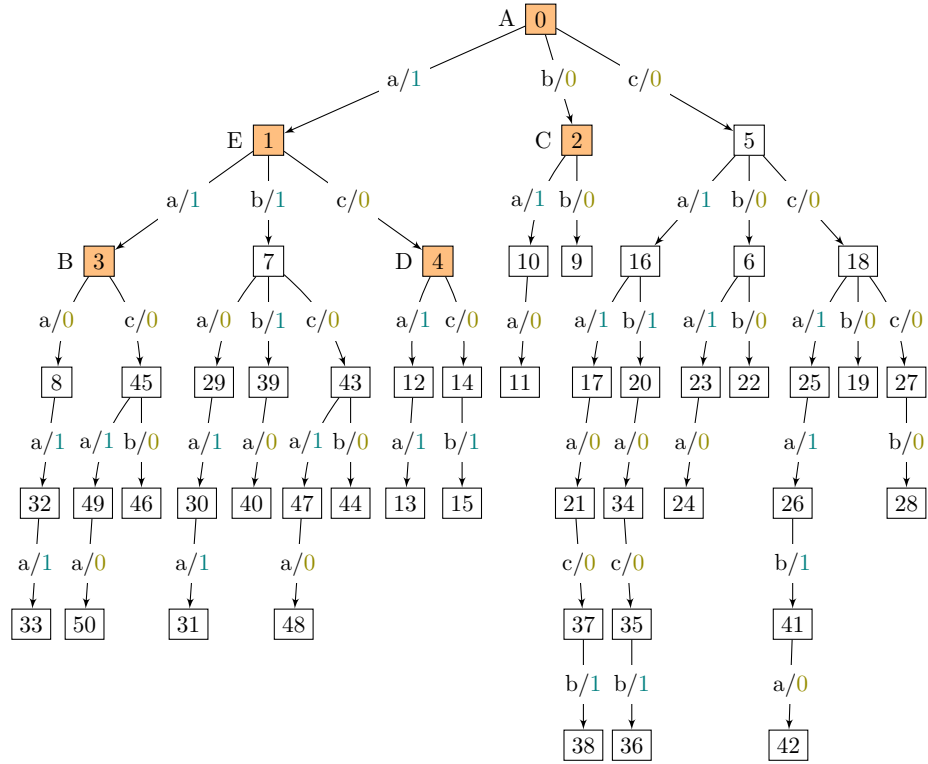


Figure 9.2: SPY-method: testing tree after verifying transitions (A,c) and (E,b)

separating sequences ‘b’ and ‘aa’, and ‘cc’ is extended with ‘aa’ and ‘cb’. The corresponding extensions of ε are already in T . The testing tree with 29 nodes satisfies Theorem 8.6 so that ‘c’ and ε are F_T -convergent. The convergent classes are thus: state A: $[\varepsilon] = \{\varepsilon, c, cc, ccc\}$ (nodes 0, 5, 18, 27), state B: $[aa] = \{aa, caa, ccaa\}$ (nodes 3, 17, 26), state C: $[b] = \{b, cb, ccb, cccb\}$ (nodes 2, 6, 19, 28), state D: $[ac] = \{ac\}$ (node 4) and state E: $[a] = \{a, ca, cca\}$ (nodes 1, 16, 25).

The second transition is (E,b) that leads to state B. Therefore, the sequence ‘ab’ and the access sequence ‘aa’ of B are considered for the extension. First, the HSI of B is appended to both sequences. It is just the separating sequence ‘a’ and it already extends ‘aa’ so only node 29 (sequence ‘aba’) is added to the testing tree. This sequence is extended immediately after with ‘aa’ as the separating sequence of state D reached by ‘aba’; ‘aa’ also extends ‘aaa’. The HSI of D also contains sequence ‘cb’. In its case, the function APPENDSEPARATINGSEQUENCE is called first with the arguments [a], ‘bacb’ and then with [aa], ‘acb’. The function finds better sequences to extend than the shortest ones in both cases; sequence ‘ca’ is chosen from [a] and ‘caa’ from [aa]. Therefore, T grows only by 5 inputs that correspond to the suffixes of test sequences ‘cabacb’ and ‘caaacb’. The testing tree would get only 4 new nodes if the shortest sequences were extended but T would increase by 2 new test sequences and 10 new symbols in total. Similarly, the extension ‘b’ with the separating sequence ‘a’ of the reached state B extends ‘ccaa’ instead

of ‘aa’ for the price of saving some symbols. After appending a few more separating sequences, the testing tree with 51 nodes satisfies Theorem 8.6 and so transition (E,b) is verified as ‘ab’ and ‘aa’ are F_T -convergent. Nevertheless, only two sequences in total become convergent with the access sequence of a state, particularly, the sequences ‘ab’ and ‘cab’ are merged into the convergent class related to state B; $[aa] = \{aa, caa, ccaa, ab, cab\}$ (nodes 3, 17, 26, 7, 20).

9.8 Other Methods

The PDS-, FF-, SC-, P- and KK- methods are amongst the other testing methods that create complete test suites.

The *PDS-method* proposed in [Sou15] uses a preset distinguishing sequence (PDS) to verify the target state of each sequence of P , that is, it constructs an m -complete test suite $T = P \cdot \text{PDS}$. The drawback is that the construction of a PDS has exponential time complexity as the shortest one can have length that is exponential in the number of states [LY94]. Fortunately, an adaptive distinguishing sequence (ADS) can substitute PDS and so broaden the range of machines for which the testing method creates a test suite [HJUY09]; all machines having a PDS have an ADS and there are machines with an ADS that have no PDS.

The *FF-method*, or the Fault Function-based method, produces a test suite in accordance to the chosen fault function that describes how the implementation can look. It was proposed in [PY92] and it generalizes the HSI-method such that a smaller test suite can be constructed if just particular types of faults can occur. Therefore, it coincides with the HSI-method if all faults are possible and the implementations in the fault domain are restricted just by the assumed number of states m .

The *SC-method*, or the State-Counting method, proposed in [PY05] creates m -complete test suites for unreduced partially-specified DFMS. As partial machines can have several minimal forms, the method needs to deal with more general properties of an FSM so that it is unnecessarily complex for minimal completely-specified DFMS.

The *P-method* proposed in [SP09b] is the only method that aims to be efficient in the construction of a p -complete test suite where $p \leq n$. It was proven that a test suite T is p -complete if either $p < n$ and T contains an F_T -divergent set with $p + 1$ tests, or $p = n$ and T contains an F_T -convergence-preserving initialized transition cover for M which corresponds to the SPY-condition (Theorem 8.13). The construction approach is different from all other methods as it is based on finding a maximal clique in a so-called divergence graph. Nevertheless, the method is not defined for a general m -completeness so that it is not included in the experiments.

The *KK-method* proposed in [KK15] provides a generic framework to construct m -complete test suites. It is a goal-oriented search where goals are properties of the specification that need to be secured to hold in the implementation. Several so-called deduction patterns were proposed to check if a goal is satisfied or not. Unfortunately, the method is too general and some

parts are not clear how to implement them, therefore, it was not implemented and so it is missing in the experiments.

Chapter 10

SPYH-method

The idea of a new testing method emerged from the analysis of the most advanced testing methods described in the previous chapter, the SPY-method and the H-method. Their implementations favour different parts of the design. The H-method uses a fixed state cover that is extended with separating sequences chosen on the fly. On the contrary, the SPY-method uses separating sequences of fixed harmonized state identifiers but appends them to different access sequences that were proven to be convergent with the one in the fixed state cover. This chapter describes a new testing method called the SPYH-method in the same way as was recently proposed in [SB18]. The method is a straightforward combination of the SPY- and H- methods. After the idea of the method is sketched, its implementation is described in the following section and the chapter is concluded with a running example of how the SPYH-method creates a test suite.

The SPYH-method is very similar to the SPY-method as it aims to satisfy the SPY-condition (Theorem 8.13). It gradually verifies transitions by proving the convergence but for the verification of the reached state it uses the approach of the H-method, that is, the separating sequences are chosen on the fly. The approach of choosing a separating sequence was also adapted to work with classes of convergent sequences.

The order in which unverified transitions are processed influences the size of the resulting test suite. Therefore, a small optimization is proposed. The optimization sorts unverified transitions according to the sum of the lengths of the related access sequences, in particular, the value $|\bar{s}| + |\bar{s}_x|$ is used for transition (s, x) leading to state s_x where \bar{s}, \bar{s}_x are the fixed access sequences in \bar{S} . The convergent classes related to states that are closer to the initial one increase in their sizes sooner than the others and so there are more convergent sequences to choose from when transitions from these states are to be verified. Hence, when a separating sequence is to extend an access sequence, there is a higher chance that only a few symbols are appended to a current test sequence than that the entire test sequence enlarges the test suite.

10.1 Implementation

The SPYH-method is implemented as a combination of approaches and functions from the implementations of the SPY- and H- methods. The main difference is handling convergent sequences as the SPYH-method works with all convergent classes, not only those related to the access sequences of fixed state cover like in the SPY-method. Classes are stored and represented as *convergent nodes* (CN) of a *convergent graph*. A convergent graph is a transition diagram of a DFSM. Initially, it corresponds to the testing tree that captures traces of test sequences. Then, two subtrees are merged when the sequences leading to the roots of the subtrees are proven to be convergent. Such a merge can create a cycle in the convergent graph, therefore, it has no longer a tree structure. The corresponding merge is done every time two sequences become F_T -convergent. Finally, the convergent graph represents the specification M on which the construction of the test suite T is based. The example of a convergent graph (still in a tree-like structure) is depicted on the right of Figure 8.3. Convergent nodes thus group prefixes of test sequences such that these prefixes are convergent if they reach the same convergent node. Whenever any following algorithm uses a convergent class $[\cdot]$, the implementation works with the corresponding convergent node.

Algorithm 21: SPYH-method

input : A minimal DFSM M with n states
input : A number of extra states l ; $m = n + l$
output : An m -complete test suite T for M

- 1 $T \leftarrow \bar{S}$
- 2 **foreach** $\bar{s} \in \bar{S}$ **do**
- 3 \lfloor SPYH-DISTINGUISH($[\bar{s}]$, \bar{S})
- 4 sort unverified transitions according to $|\bar{s}| + |\bar{s}_x|$ calculated for each transition (s, x) and $s_x = \delta(s, x)$; the first one has minimal value
- 5 **foreach** unverified transition (s, x) such that $s_x = \delta(s, x)$ **do**
- 6 add $\bar{s} \cdot x$ to T if not there
- 7 SPYH-DISTINGUISHFROMSET($[\bar{s} \cdot x]$, $[\bar{s}_x]$, a copy of \bar{S} , l)
- 8 merge $[\bar{s} \cdot x]$ and $[\bar{s}_x]$, and the convergent classes of their successors
- 9 **return** T

Algorithm 21 captures the main flow of the SPYH-method. It first builds a minimal prefix-closed state cover \bar{S} that is then made F_T -divergence-preserving using the function SPYH-DISTINGUISH. Every sequence of \bar{S} needs to be distinguished from each other in order to construct an F_T -divergence-preserving \bar{S} .

The proposed optimization sorts the unverified transitions that are then processed in the sorted order. An unverified transition (s, x) first needs to be covered in T (line 6) and then both $[\bar{s} \cdot x]$, $[\bar{s}_x]$, where $s_x = \delta(s, x)$, need to be extended to satisfy Theorem 8.6. SPYH-DISTINGUISHFROMSET takes care of

suitable extensions satisfying all requirements so that both convergent classes (and CNs as well) can then be merged as they become convergent. Finally, the m -complete test suite is returned.

SPYH-DISTINGUISH described in Algorithm 22 follows the version in the H-method but it works with convergent classes. Therefore, the function for the comparison of extensions was rewritten and the separating sequence $w'w$ are not added directly to T but using APPENDSEPARATINGSEQUENCE defined in Algorithm 20 for the SPY-method.

Algorithm 22: SPYH-DISTINGUISH($[u], V$)

```

1 foreach  $[v] \in V$  such that  $\delta^*(s_0, u) \neq \delta^*(s_0, v)$  do
2    $(e, w') \leftarrow$  SPYH-GETBESTPREFIXOFSEPSSEQ( $[u], [v]$ )
3   if  $e > 0$  then
4      $w \leftarrow$  shortest separating sequence of  $\delta^*(s_0, uw')$ ,  $\delta^*(s_0, vw')$ 
5     APPENDSEPARATINGSEQUENCE( $[u], w'w$ )
6     APPENDSEPARATINGSEQUENCE( $[v], w'w$ )

```

Algorithm 23 specifies the function SPYH-DISTINGUISHFROMSET that is also inspired by its version in the H-method. There are several changes compared to Algorithm 16 of the H-method. SPYH-DISTINGUISHFROMSET extends two sequences simultaneously as both $\bar{s} \cdot x$ and \bar{s}_x (and their extensions) are to be in an F_T -divergence-preserving state cover according to Theorem 8.6. The required extensions of the length up to l do not have to be already in T , therefore, lines 7 and 8 of Algorithm 23 adds them gradually by one symbol. A change is also that SPYH-DISTINGUISH is called (lines 1 and 2) before the recursive call of SPYH-DISTINGUISHFROMSET (line 9). It means that the separating sequences appended to u can cover some of the required extensions and sequences appended to these extensions can have these separating sequences as prefixes. Hence, the total number of test sequences could be reduced by this small change. The condition on lines 2, 5 and 10 checks if the given class $[v]$ is convergent with the fixed access sequence of a state. If it is, then the class is already in V as V gets a copy of \bar{S} when SPYH-DISTINGUISHFROMSET is called. The set V is used to store all convergent classes from which the given $[u], [v]$ are to be distinguished, that is, the classes of fixed access sequences and proper prefixes of u, v .

The last part of the SPYH-method is the function that checks all common extensions of the given classes and chooses the best prefix of a separating sequence that should extend T . SPYH-GETBESTPREFIXOFSEPSSEQ in Algorithm 24 has the same flow as GETBESTPREFIXOFSEPARATINGSEQUENCE of the H-method but it was adapted to work with convergent classes, that is, sets of sequences. There are again variables *minEstimate* and *bestPrefix* that hold information about the best way found so far to distinguish the given sequences. The calculation of estimate was changed. The length of access sequences are included as well if a new test sequence is to enlarge T . The function HASLEAF defined on lines 5–6 checks whether the given class

Algorithm 23: SPYH-DISTINGUISHFROMSET($[u]$, $[v]$, V , $depth$)

```

1 SPYH-DISTINGUISH( $[u]$ ,  $V$ )
2 if  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then SPYH-DISTINGUISH( $[v]$ ,  $V$ )
3 if  $depth > 0$  then
4   add  $[u]$  to  $V$ 
5   if  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then add  $[v]$  to  $V$ 
6   foreach  $x \in X$  do
7     APPENDSEPARATINGSEQUENCE( $[u]$ ,  $x$ )
8     APPENDSEPARATINGSEQUENCE( $[v]$ ,  $x$ )
9     SPYH-DISTINGUISHFROMSET( $[ux]$ ,  $[vx]$ ,  $V$ ,  $depth - 1$ )
10  if  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then pop  $[v]$  from  $V$ 
11  pop  $[u]$  from  $V$ 

```

contains a sequence that is maximal in T and so it provides information if the chosen separating sequence can easily extend a current test sequence. The function is implemented as a property of convergent nodes that keep track which sequences reach the nodes but do not continue to the successors. The initial estimate of the number of symbols that extend T assumes that the shortest separating sequence w will extend both classes and it also assumes that no prefix of w extends any sequences of the convergent classes. Therefore, $minEstimate$ gets twice the length of w plus the lengths of the shortest sequences u, v of the given classes if they do not contain a maximal sequence. All inputs are then compared if they begin a common extension that lead to a better estimate. If both classes have an extension starting with an input x , then it is the same as in the H-method (lines 12–16 of Algorithm 24), that is, SPYH-GETBESTPREFIXOFSEPSEQ is recursively called on the successors. Otherwise, it depends on which of the classes have such an extension (lines 18–23 and 25–30). The function ESTIMATEGROWTHOFT defined in Algorithm 18 initializes e with an estimate of the number of symbols added to T by appending a separating sequence. If just x separates the reached states ($e = 1$), then only the class that is not extended with x is checked whether it contains a maximal sequence and if not, the length of the shortest access sequence increases e (lines 22 and 29). Otherwise, the other class needs to be checked as well (lines 19–21 and 26–28). If e is lower than $minEstimate$, both $minEstimate$ and $bestPrefix$ are updated accordingly. Finally, both variables are returned as the values capturing the best way to distinguish the given $[u]$ and $[v]$.

Meaningful average time and space complexity are not easy to derive as they are really dependent on the structure of the machine under test, that is, access sequences of states and their separating sequences, the number of states n , the number of inputs p and others. The space complexity for the SPYH-method includes the resulting testing tree, the convergent graph and a state pair array of all separating sequences. The convergent graph represents the machine under test in the end so that it takes $O(n)$ space. The SPA has

Algorithm 24: SPYH-GETBESTPREFIXOFSEPSEQ($[u], [v]$)

```

1 Let  $u, v$  be the shortest sequences of the given classes
2  $s_u \leftarrow \delta^*(s_0, u)$ ,  $s_v \leftarrow \delta^*(s_0, v)$ 
3  $minEstimate \leftarrow 2|w|$  where  $w$  is the shortest separating sequence of  $s_u, s_v$ 
4  $bestPrefix \leftarrow \varepsilon$ 
5 HASLEAF( $[u]$ ):
6    $\lfloor$  returns true if there is  $u' \in [u]$  that is maximal in  $T$ 
7 if not HASLEAF( $[u]$ ) then  $minEstimate \leftarrow minEstimate + |u|$ 
8 if not HASLEAF( $[v]$ ) then  $minEstimate \leftarrow minEstimate + |v|$ 
9 foreach  $x \in X$  do
10   if there are  $u' \in [u]$  and  $w_u \in X_{\uparrow}^*$  such that  $u'xw_u \in T$  then
11     if there are  $v' \in [v]$  and  $w_v \in X_{\uparrow}^*$  such that  $v'xw_v \in T$  then
12       if  $\lambda^*(s_u, x \uparrow) \neq \lambda^*(s_v, x \uparrow)$  then return  $(0, \varepsilon)$ 
13       if  $\delta(s_u, x) = \delta(s_v, x)$  then continue
14        $(e, w') \leftarrow$  SPYH-GETBESTPREFIXOFSEPSEQ( $[ux], [vx]$ )
15       if  $e = 0$  then return  $(0, \varepsilon)$ 
16       if  $e \leq minEstimate$  then  $minEstimate \leftarrow e$ ,  $bestPrefix \leftarrow xw'$ 
17     else
18        $e \leftarrow$  ESTIMATEGROWTHOFT( $s_u, s_v, x$ )
19       if  $e \neq 1$  then
20         if HASLEAF( $[u]$ ) then  $e \leftarrow e + 1$ 
21         else if not HASLEAF( $[ux]$ ) then  $e \leftarrow e + |u| + 1$ 
22       if not HASLEAF( $[v]$ ) then  $e \leftarrow e + |v|$ 
23       if  $e < minEstimate$  then  $minEstimate \leftarrow e$ ,  $bestPrefix \leftarrow x$ 
24   else if there are  $v' \in [v]$  and  $w_v \in X_{\uparrow}^*$  such that  $v'xw_v \in T$  then
25      $e \leftarrow$  ESTIMATEGROWTHOFT( $s_u, s_v, x$ )
26     if  $e \neq 1$  then
27       if HASLEAF( $[v]$ ) then  $e \leftarrow e + 1$ 
28       else if not HASLEAF( $[vx]$ ) then  $e \leftarrow e + |v| + 1$ 
29     if not HASLEAF( $[u]$ ) then  $e \leftarrow e + |u|$ 
30     if  $e < minEstimate$  then  $minEstimate \leftarrow e$ ,  $bestPrefix \leftarrow x$ 
31 return  $(minEstimate, bestPrefix)$ 

```

space of $O(n^2)$. However, the testing tree depends on test sequences. Its size is bounded by the total length of test sequences, that is, the size of test suite, but it is usually much smaller because each common prefix of several test sequences is stored in the testing tree just once. The upper bound of the size of test suite is possible to derive by considering the W-method. Each test sequence has three parts: the access sequence of a state, the input of the tested transition, the extension of length up to the given l and a separating sequence. Its length is thus at most $(n - 1) + 1 + l + (n - 1)$ which is in $O(2n + l)$. The number of test sequences is bounded by $n \cdot p^{l+1} \cdot (n - 1)$ because there is n access sequences that are extended with all sequences of the

length up to $l + 1$ and at most $n - 1$ separating sequences are then appended. Together, the size of test suite is in $O((2n + l)(n^2 p^{l+1}))$ which is in $O(n^3 p^{l+1})$ if n is strictly greater than l . This is important bound as the standard testing methods have the same (worst case) space complexity $O(n^3 p^{l+1})$.

The worst case time complexity can be calculated based on Algorithms 21–24. The most time is spent in the function SPYH-DISTINGUISH. It is called approximately $(n + np \cdot p^l \cdot 2)$ times; for each of n access sequences (line 3 of Algorithm 21) one call plus for each of at most np unverified transitions and each of their p^l extensions there are two calls. The exact number of extensions is $\frac{p^{l+1}-1}{p-1}$ as the sum of a geometric progression. Inside the function, the given class $[u]$ is distinguished from particular classes in V . For the first n calls of SPYH-DISTINGUISH the size of V is n and for the other calls $|V|$ is at most $n + l$. All common extensions of the given classes are checked by SPYH-GETBESTPREFIXOFSEPSEQ (line 2 of Algorithm 22). There are very different numbers of extensions for different classes during the construction, therefore, let bound them by the (worst case) size of test suite, that is, $n^3 p^{l+1}$. The separating sequence w can be obtained proportionally to its length so that at most in $O(n)$. The last bit is two calls of APPENDSEPARATINGSEQUENCE (Algorithm 20). This function chooses one sequence of the given class and appends w to it. Let assume that every class has the same number of convergent sequence in the end, that is, there are n classes so that each has $n^2 p^{l+1}$ sequences. However, as each sequence of the class is checked for an extension that is a prefix of the given w , APPENDSEPARATINGSEQUENCE runs in $O(n^3 p^{l+1})$. Altogether, the SPYH-method spends $O((n^2 + 2np^{l+1}(n + l))(n^3 p^{l+1} + n + n^3 p^{l+1}))$, or $O((n^5 + n^4 l)p^{2l+2} + (n^5 + n^3 + n^2 l)p^{l+1} + n^3)$, time with the function SPYH-DISTINGUISH. The other parts like the construction of state cover, sorting unverified transitions or merging convergent classes, do not change the estimated complexity so that the worst case time complexity of the SPYH-method is $O((n^5 + n^4 l)p^{2l+2})$.

10.2 Running Example

This section explains how the SPYH-method creates a test suite assuming 1 extra state for the 5-state Mealy machine defined in Figure 4.1. The process is very similar to the SPY-method. First, a minimal prefix-closed state cover \bar{S} is obtained; the related nodes of the testing tree are highlighted in Figure 10.1. The access sequences in \bar{S} are then made T -separable and so F_T -divergent. States A, B are separated by ‘a’ that already extends the access sequence ε of A but it needs to be appended to ‘aa’ that relates to B. Similarly, ‘aa’ extends ‘b’ to separate states A and C. After C, E are separated by appending ‘b’ to both access sequences (nodes 14 and 15), all state pairs are distinguished so that the testing tree with 16 nodes contains an F_T -divergence-preserving state cover.

Unverified transitions are then sorted such that transition (A,c) with the value of 0 ($|\varepsilon| + |\varepsilon|$ as it leads back to A) will be verified first, the second

transition will be (C,b) leading to A (value $1 = |b| + |\varepsilon|$), the third and the fourth are (C,a) and (B,c) with the value 3 ($|b| + |aa|$ and vice versa respectively), and so on. Equally to the SPY-method, all sequences that verify the first transition (A,c) are appended to the fixed access sequences as the convergent classes are initially singletons. The state reached by 'c' is first distinguished from B by appending 'a' (node 16), then from C by 'aa' (node 17) and from D by 'cb' (nodes 18 and 19). The verification of the states reached by 'ca', 'cb' and 'cc' is done in a similar way but using different separating sequences. All the appended separating sequences also extend the access sequence ε , however, no node is added to the testing tree as the sequences are already in T . The test sequence 'cccb' is the last one needed to verify transition (A,c); Theorem 8.6 is satisfied by the testing tree with 29 nodes. In this moment, the convergent graph ends to look like a tree. Node 8 merges into node 0 which implies that also nodes 18 and 27 become convergent with the access sequence of state A. Similarly, nodes 1, 16 and 25 representing state E are merged together, nodes 2, 9, 19 and 28 relate to C, nodes 3, 17, 26 relate to B, and only node 4 relate to D.

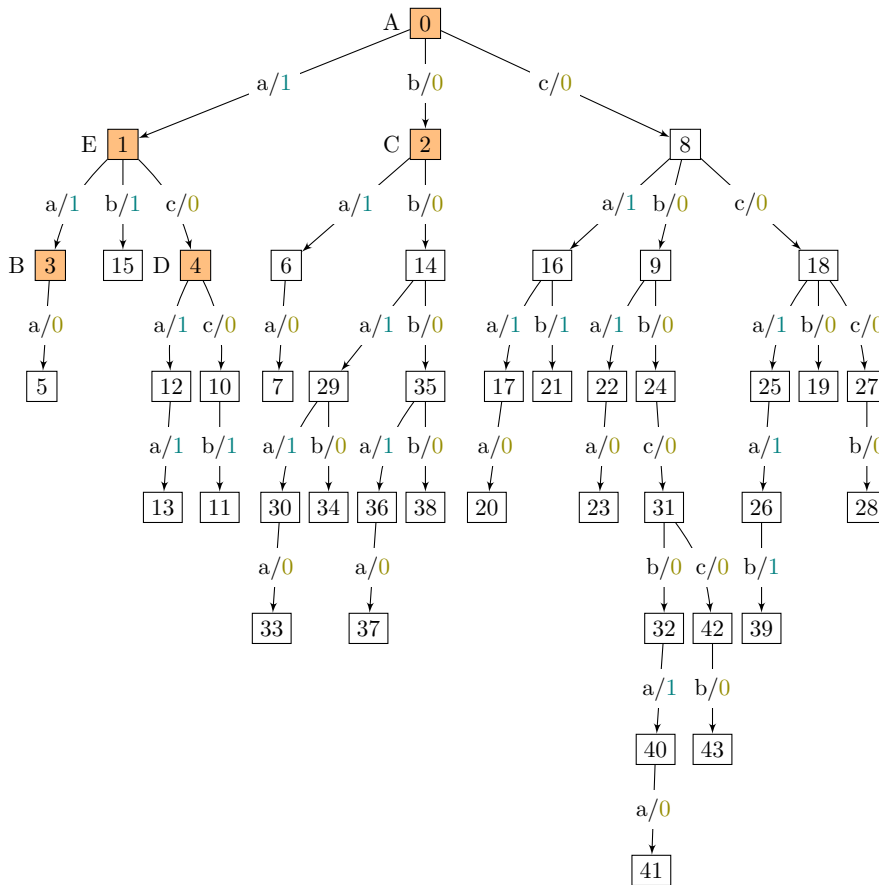


Figure 10.1: SPYH-method: testing tree after verifying transitions (A,c), (C,b)

The second transition to verify is (C,b) that leads to A. After appending 'aa' to 'bb' (nodes 29 and 30) that separates the reached A from both B

and C, the separating sequence ‘cb’ is to be appended to [bb]. The function APPENDSEPARATINGSEQUENCE chooses ‘cbb’ to be extended instead of ‘bb’ which saves some symbols as ‘cbb’ is maximal in T and can be extended without branching. It is a benefit from the SPY-method. Then, the sequences ‘bba’ and ‘bbb’ are verified by the separating sequences ‘aa’ and ‘b’. Note again that these separating sequences are checked if they extend the corresponding access sequence in T ; in this case the presence of ‘aaa’, ‘ab’, ‘baa’ and ‘bb’ is required in T . The last extension to verify is ‘c’, that is, [bbc] is to be shown to reach state A. As the sequence ‘cbbc’ from [bbc] is already in T , ‘bbc’ is not needed to be added so far. The first benefit of the H-method is that the function SPYH-GETBESTPREFIXOFSEPSEQ estimates that it is better to extend a sequence related to B with the separating sequence ‘b’ (node 39) than append the separating sequence ‘a’ to ‘cbbc’. It is because ‘cbbc’ is extended with ‘b’ and there is a maximal sequence convergent with the access sequence ‘aa’ of state B. The second benefit comes immediately after when the function advises to extend ‘cbbcb’ with ‘aa’ instead of appending a separating sequence to ‘cbbc’. Again, it saves several symbols in the resulting test suite. The next separating sequence ‘cb’ is appended to ‘cbbc’ to distinguish it from the access sequence ‘ac’ of state D. This reveals inefficiency of the implementation. The sequence ‘cbbc’ was not maximal so that the entire sequence ‘cbbccb’ enlarged T , that is, plus 6 symbols. If ‘cb’ was appended to ‘bbc’, the test suite would grow just by 3 symbols. The problem is that the implementation works with current convergent classes/convergent nodes and does not consider the ‘predecessors’. In the example, the sequences ‘bb’ and ‘cbb’ are convergent but only ‘cbb’ is extended with ‘c’ and so the algorithm looks just at ‘cbbc’ when it works with the related class and does not take advantage of ‘bb’ that could be extended with ‘c’ and thus be in the handled class. This efficiency issue is solved by the S-method proposed in the following chapter. The sequences ‘bba’, ‘bbb’ and ‘bbc’ also need to be distinguished from ‘bb’ and from ε as it is a requirement for proving the convergence of ‘bb’ and ε . It is usually fulfilled by the separating sequences that are appended to distinguish individual extensions from the fixed access sequences. After transition (C,b) is verified, node 14 is merged into node 0 and the successive merge updates the convergent graph as it is shown in Figure 10.2. Convergent nodes are labelled with the numbers of nodes of the testing tree and the leaves (maximal sequences in T) are underlined.

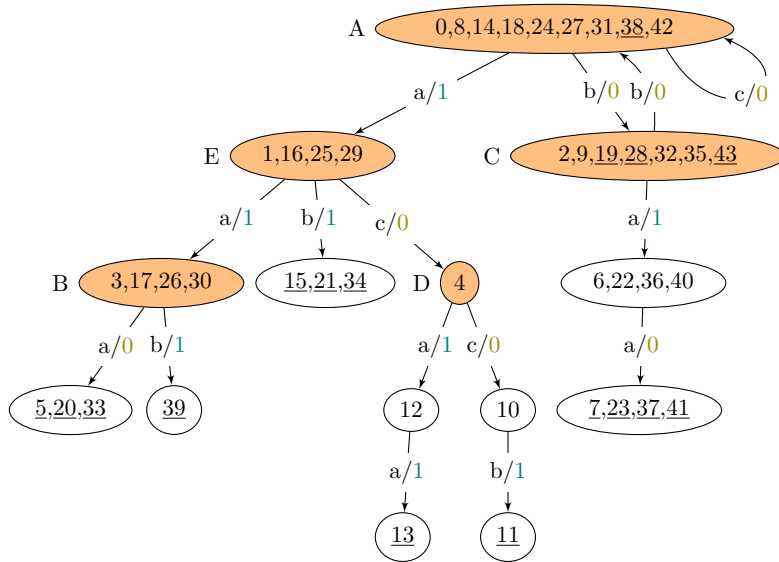


Figure 10.2: Convergent graph after verifying transitions (A,c), (C,b)

Chapter 11

S-method

A novel method called the S-method is proposed in this chapter. It was developed in the reaction to the drawbacks of the advanced testing methods, namely the H-, SPY- and SPYH- methods, and due to the requirements of a new active-learning approach that will be proposed in Chapter 15. The idea and novelty of the testing method is described first, followed by a section about the implementation of the method and a section explaining the method on an example.

The S-method employs the concept of convergence so that it is very similar to the SPY- and SPYH- methods. Nevertheless, there are several changes compared to these two methods.

- *S-condition*: The method creates an m -complete test suite based on Theorem 8.14 that describes the weakest sufficient condition, the S-condition.
- *State domains*: The domain function Φ introduced in Section 8.2 is used to keep track of the divergence of convergent classes. It has two benefits. The conditions specified in Theorem 8.6 to prove the convergence of two sequences are easily checked using domains (Theorem 8.10). If no extra state is considered, empty intersection of domains reduces domains of predecessors which eliminate the need of appending some separating sequences and thus it minimizes the size of the resulting test suite (Theorem 8.11).
- *Separating sequences from the splitting tree*: Instead of using fixed separating sequence (the SPY-method) or constructing a separating sequence to distinguish a state pair (the H- and SPYH- methods), the S-method uses the splitting tree (Chapter 4) to obtain separating sequences that reduce the given domain the most.
- *Predecessor convergent nodes*: The choice of a sequence to extended with the chosen separating sequences is not restricted to only the related convergent class. If there is no maximal sequence in the class, then the convergent classes of predecessors are checked because it is usually better to extend their maximal sequence than add a completely new test sequence to T .

Algorithm 25: S-method

input : A minimal DFSM M with n states
input : A number of extra states l ; $m = n + l$
input : Incomplete test suite T and state cover \bar{S} ; empty sets by default
output : An m -complete test suite T for M

- 1 $ST \leftarrow$ a splitting tree for M by Algorithm 8 (*validOnly* is false)
- 2 S-CREATEDIVERGENCEPRESERVINGSC(T, \bar{S})
- 3 sort unverified transitions according to $|\bar{s}| + |\bar{s}_x|$ calculated for each transition (s, x) and $s_x = \delta(s, x)$; the first one has minimal value
- 4 **foreach** unverified transition (s, x) such that $s_x = \delta(s, x)$ **do**
- 5 $proveConvergence \leftarrow$ estimate if proving convergence of $\bar{s}x, \bar{s}_x$ helps
- 6 **foreach** $u \in X^l$ **do**
- 7 S-APPENDSEPARATINGSEQUENCE($[\bar{s}], xu$)
- 8 **if** $proveConvergence$ **then**
- 9 S-APPENDSEPARATINGSEQUENCE($[\bar{s}_x], u$)
- 9 S-DISTINGUISHFROMSET($[\bar{s} \cdot x], (proveConvergence ? [\bar{s}_x] : \text{null}), \emptyset, l$)
- 10 **if** $proveConvergence$ **and** $[\bar{s}x], [\bar{s}_x]$ are not merged yet **then**
- 11 merge $[\bar{s}x]$ and $[\bar{s}_x]$ so that $\Phi([\bar{s}x] = [\bar{s}_x])$ gets $\Phi([\bar{s}x]) \cap \Phi([\bar{s}_x])$,
do the same merge and update of domains for their extensions
- 12 **if** $l = 0$ **then** merge $[v], [\bar{s}_v]$ if $\Phi([v]) = \{s_v\}$ and $s_v = \delta^*(s_0, v)$

13 **return** T

Following the S-condition (Theorem 8.14) has its pitfalls. The S-condition allows one to choose if the verifying subtree will extend either only $[\bar{s}x]$ or both $[\bar{s}x]$ and $[\bar{s}_x]$, see Figure 8.3 that sketches these two choices. However, it does not say when one choice is better than the other. The S-method thus estimates the usefulness of having the convergence proven, that is, whether $[\bar{s}x] = [\bar{s}_x]$ helps. The boolean variable *proveConvergence* is true if there is an unverified transition (s_u, x_u) such that s_u is reached by a sequence of length up to l from s_x , that is, $\exists u \in X^{\leq l} : \delta^*(s_x, u) = s_u$. In other words, there is a chance that proving convergence of $\bar{s} \cdot x$ and \bar{s}_x provides new convergent sequences that can be extended to prove another unverified transition. At least the last unverified transition thus takes the advantage of the S-condition and the verifying subtree extends only $[\bar{s}x]$. The variable *proveConvergence* controls if the convergent class $[\bar{s}_x]$ is to be extended with the verifying subtree (lines 8 and 9 of Algorithm 25) and if the classes are to be merged (line 10). Note that the condition on line 10 also checks if the classes were not merged. It is because they could be merged in the function S-APPENDSEPARATINGSEQUENCE if no extra state was considered ($l = 0$) and the domain $\Phi([\bar{s}x])$ became $\{s_x\}$. This is essentially the condition to which Theorem 8.10 reduces if $l = 0$. Such a merge of convergent classes is made also on line 12 of Algorithm 25.

Algorithm 26 defines S-CREATEDIVERGENCEPRESERVINGSC and specifies thus how an F_T -divergence-preserving state cover is created. The function

takes two parameters, a test suite T and the related state cover $\bar{S} \subseteq T$. Both T and \bar{S} are empty sets unless the S-method is provided with an incomplete test suite T and the related \bar{S} . At first, consider the case when \bar{S} is empty and so it needs to be constructed. The reason for a new construction method of a state cover is explained on an example. Assume that the machine has an adaptive distinguishing sequence that begins with input ‘b’ for all states. If the ADS is appended to the access sequence of every state, one gets a divergence-preserving state cover. The choice of access sequence thus influences the resulting test suite. Consider the access sequence ‘a’ of state A and both ‘aa’ and ‘ab’ lead to state B. Sequence ‘aa’ would be chosen as the access sequence of B if the standard construction algorithm was used because it checks 1-input extensions in the alphabetical order. A sequence starting with ‘ab’ would be added to T later as the ADS needs to extend the access sequence ‘a’ of A. Therefore, if ‘ab’ is chosen as the access sequence of B, ‘aa’ may not be added to T at all; transition (A,a) could be verified by extending sequences convergent with ‘a’ except ‘a’ itself. This is solved by S-CREATEDIVERGENCEPRESERVINGSC that first checks if a sequence in T can be the access sequence of a state that is not covered by \bar{S} yet. As only sequences that are 1-input extensions of an access sequence in \bar{S} are checked, all access sequences in \bar{S} are the shortest for each state and \bar{S} is prefix-closed. The shortest access sequences are chosen because it is likely that they will be extended with several sequences in T which would lead to a bigger test suite if longer access sequences were selected. Access sequences are chosen in two steps. First, all access sequences of a particular length are checked if their 1-input extensions are in T and can be access sequences of states uncovered by \bar{S} (lines 8–16 of Algorithm 26). Then, the same check is processed but for all 1-input extensions (or rather only for those that are not in T). Both steps are controlled by two queues, U stores access sequences of length k that are already in \bar{S} and V stores access sequences of length $k + 1$ that are currently added to \bar{S} . After all extensions of sequences in U are checked, k is increased so that U gets V . This is repeated until all states are covered by \bar{S} . When a sequence u is chosen to be the access sequence of state s , the ‘main’ separating sequence w is obtained from the splitting tree and uw is added to T (lines 13–14). This is also the case of the initial state s_0 and its access sequence ε (lines 1–4). The ‘main’ separating sequence has the same prefix for all states because it starts with the sequence of the root of the splitting tree. If an incomplete \bar{S} is given, then all 1-input extensions are still checked by traversing the access sequences in \bar{S} (lines 15–16) and thus \bar{S} is updated to be a state cover. When the state cover is constructed, the convergent graph coincide with the testing tree and the domains of all convergent nodes are initialized such that a state s is in $\Phi([u])$ only if $[u]$ and $[\bar{s}]$ are not T -separable. Finally, S-CREATEDIVERGENCEPRESERVINGSC calls the function S-DISTINGUISH for each state which makes \bar{S} F_T -divergence-preserving.

Algorithm 27 describes the function S-DISTINGUISH that is adapted to work with separating sequences from the splitting tree compared to its versions in the H- and SPYH- methods. First, the function obtains *domain* of convergent

Algorithm 26: S-CREATEDIVERGENCEPRESERVINGSC(T, \bar{S})

```

1 if  $\varepsilon \notin \bar{S}$  then
2   | add  $\varepsilon$  to  $\bar{S}$  as the access sequence of  $s_0$ 
3   |  $w \leftarrow \text{GETSEPARATINGSEQUENCEFROMST}(s_0, S)$ 
4   | add  $w$  to  $T$ 
5  $U \leftarrow \{\varepsilon\}$ 
6 while  $|U| > 0$  do
7   |  $V \leftarrow \emptyset$ 
8   | foreach  $\bar{s} \in U$  and each  $x \in X$  such that  $\bar{s}x \in T$  do
9     | CHECKTARGETSTATE( $s, x$ ):
10    |   | if  $\delta(s, x)$  is not covered by  $\bar{S}$  then
11    |   |   | add  $\bar{s}x$  to  $\bar{S}$  as the access sequence of  $\delta(s, x)$ 
12    |   |   | add  $\bar{s}x$  to  $V$ 
13    |   |   |  $w \leftarrow \text{GETSEPARATINGSEQUENCEFROMST}(\delta(s, x), S)$ 
14    |   |   | add  $\bar{s}xw$  to  $T$ 
15    |   |   | else if  $\bar{s}x \in \bar{S}$  then
16    |   |   |   | add  $\bar{s}x$  to  $V$ 
17   | foreach  $\bar{s} \in U$  and each  $x \in X$  such that  $\bar{s}x \notin T$  do
18   |   | CHECKTARGETSTATE( $s, x$ )
19   |  $U \leftarrow V$ 
20 update domains  $\Phi$ 
21 foreach  $s \in S$  do
22   | S-DISTINGUISH( $[\bar{s}], \emptyset$ )

```

classes which the given class $[u]$ is to be distinguished from based on the domain $\Phi([u])$ and the given V . States associated with the classes in *domain* are considered for the construction of a separating sequence w from the splitting tree (lines 5 and 6). The sequence w is then appended to $[u]$ and to all classes in *domain* that produce a different response to w than u (lines 8–10). These classes are removed from *domain* as they are distinguished by w from $[u]$. Subsequently, all remaining classes in *domain* are compared against $[u]$ whether they are not distinguished because a part of w appended to a different class could form a sequence that separates $[u]$ from a particular class in *domain*. If there is still a class in *domain*, the process of getting a separating sequence w and appending it is repeated until all classes in *domain* are distinguished from $[u]$. The choice of w does not consider extensions of the related classes in T as both the H- and SPYH- methods do. This could optimize the resulting test suite even more but the S-method puts this time effort rather into the search for the best sequence to extend. This new feature will be described by the function S-APPENDSEPARATINGSEQUENCE.

The function S-DISTINGUISHFROMSET defined in Algorithm 28 is similar to its version in the SPYH-method. It is a recursive function that passes all

Algorithm 27: S-DISTINGUISH($[u], V$)

```

1  $s_u \leftarrow \delta^*(s_0, u)$ 
2  $U \leftarrow \{[v] \in V \mid s_u \neq \delta^*(s_0, v) \wedge \text{not S-AREDISTINGUISHED}([u], [v], \emptyset)\}$ 
3  $domain \leftarrow (\{[\bar{s}] \mid s \in \Phi([u])\} \cup U) \setminus \{[u]\}$ 
4 while  $domain$  is not empty do
5    $S' \leftarrow \{\delta^*(s_0, v) \mid [v] \in domain\}$ 
6    $w \leftarrow \text{GETSEPARATINGSEQUENCEFROMST}(s_u, S')$ 
7   S-APPENDSEPARATINGSEQUENCE( $[u], w$ )
8   foreach  $[v] \in domain$  such that  $\lambda^*(s_u, w) \neq \lambda^*(\delta^*(s_0, v), w)$  do
9      $w' \leftarrow$  the shortest prefix of  $w$  that separates  $s_u$  and  $\delta^*(s_0, v)$ 
10    S-APPENDSEPARATINGSEQUENCE( $[v], w'$ )
11    remove  $[v]$  from  $domain$ 
12  $domain \leftarrow \{[v] \in domain \mid \text{not S-AREDISTINGUISHED}([u], [v], \emptyset)\}$ 

```

extensions of the length up to l and creates thus the verifying subtrees of the given two classes. As the S-condition allows one to create only one verifying subtree for the price of not proving the convergence, the second class $[v]$ does not have to be given; ‘null’ is passed instead. This is reflected by the conditions on lines 2, 5, 7 and 8 when the function wants to deal with the class $[v]$. Lines 4–5 and 8–9 updates V such that V contains convergent nodes (convergent classes) that are not reference nodes but are predecessors of the current CN $[u], [v]$ on the path from the original CN for which S-DISTINGUISHFROMSET was called from Algorithm 25. Note that classes $[ux], [vx]$ used on line 7 are already in T because of the third small optimization mentioned above (all extension of the length up to l are appended in advance).

Algorithm 28: S-DISTINGUISHFROMSET($[u], [v], V, depth$)

```

1 S-DISTINGUISH( $[u], V$ )
2 if  $v \neq \text{null}$  and  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then S-DISTINGUISH( $[v], V$ )
3 if  $depth > 0$  then
4   add  $[u]$  to  $V$ 
5   if  $v \neq \text{null}$  and  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then add  $[v]$  to  $V$ 
6   foreach  $x \in X$  do
7     S-DISTINGUISHFROMSET( $[ux], (v \neq \text{null} ? [vx] : \text{null}), V, depth - 1$ )
8   if  $v \neq \text{null}$  and  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then pop  $[v]$  from  $V$ 
9   pop  $[u]$  from  $V$ 

```

Appending the chosen separating sequence w to the convergent class $[u]$ is done by S-APPENDSEPARATINGSEQUENCE defined in Algorithm 29. It starts the same as in the SPY-method, that is, the best sequence u_{best} is chosen from $[u]$ such that it is extended with the longest prefix w' of w and $u_{best}w'$ is maximal in T (lines 2–6). If $[u]$ is already extended with w , the function does not add anything to T and exits (line 4). If there is no such sequence in $[u]$,

Algorithm 29: S-APPENDSEPARATINGSEQUENCE($[u], w$)

```

1  $u_{best} \leftarrow$  the shortest  $u' \in [u]$ ,  $maxLength \leftarrow -1$ 
2 foreach  $u' \in [u]$  do
3    $w' \leftarrow$  the longest prefix of  $w$  such that  $u'w' \in T$ 
4   if  $w' = w$  then return
5   if there is no  $v$  such that  $u'w'v \in T$  and  $|w'| > maxLength$  then
6      $u_{best} \leftarrow u'$ ,  $maxLength \leftarrow |w'|$ 
7 if  $maxLength = -1$  and  $u_{best} \neq \varepsilon$  then
8    $minLength \leftarrow |u_{best}|$ 
9    $visited \leftarrow \{[u]\}$ 
10   $t_{best} \leftarrow \varepsilon$ 
11  push  $([u], \varepsilon)$  into the queue predecessors
12  while predecessors is not empty do
13     $([v], t_v) \leftarrow$  pop first from predecessors
14    foreach  $v' \cdot x \in [v]$  such that  $v'$  is the shortest in  $[v']$  do
15      if  $[v'] \notin visited$  then
16        if there is  $v'_{leaf} \in [v']$  that has no extension in  $T$  then
17           $u_{best} \leftarrow v'_{leaf}$ ,  $t_{best} \leftarrow xt_v$ 
18          clear predecessors
19          break
20        if  $|v'xt_v| < |u_{best}t_{best}|$  then
21           $u_{best} \leftarrow v'$ ,  $t_{best} \leftarrow xt_v$ 
22        if  $|xt_v| < minLength$  and  $v' \neq \varepsilon$  then
23          push  $([v'], xt_v)$  into predecessors
24          push  $[v']$  into visited
25        if  $\forall \bar{s} \in \bar{S} : \bar{s} \notin [v]$  then break
26   $w \leftarrow t_{best} \cdot w$ 
27 add  $u_{best} \cdot w$  to  $T$  and update domains accordingly
28 if  $l = 0$  then merge  $[v], [\bar{s}_v]$  if  $\Phi([v]) = \{s_v\}$  and  $s_v = \delta^*(s_0, v)$ 

```

u_{best} is initialized with the shortest sequence of the class. The difference from the SPY-method is that the function searches through the predecessors of CN associated with $[u]$ (lines 8–26). A predecessor of CN r means a node r_p of the convergent graph such that there is a directed path from r_p to r . Note that the condition on line 7 checks if $[u]$ does not relate to the initial state. Basically, the search tries to find a maximal sequence v'_{leaf} in a predecessor $[v']$ such that the path t_v from $[v']$ to $[u]$ is the shortest possible and it is shorter than u_{best} that is the shortest sequence of $[u]$ initially. The function uses a queue *predecessors* to traverse the predecessors and a set *visited* to keep track of the visited CNs and not to repeat the search from them. The condition on line 20 checks if the predecessor can form a shorter sequence than found so far in the case that no maximal sequence is discovered and thus a new test sequence

needs to be added to T . Such a shorter sequence can exist because the access sequences do not have to be the shortest when an incomplete test suite is given to the S-method. If $[v]$ is not a reference node, then the cycle checking all sequences of $[v]$ stops (line 25) because only reference nodes can have several different predecessors. When a predecessor with a maximal sequence is found or there is no such predecessor, u_{best} is extended with w (and added to T if it is a new test sequence). Subsequently, the domains Φ are updated in the following way. Nodes of the testing tree on the path of $u_{best}w$ are traversed from the leaf to the root and the related suffix of $u_{best}w$ is checked if it separates any state of the related domain. Domains Φ are implemented to store references to particular convergent nodes instead of states. Therefore, it is possible to use the domains of reference nodes differently than just to store the reference to themselves as each RN is distinguished from the others. The domain Φ of an RN stores the references to convergent nodes that are not distinguished from the RN, except the reference to itself. This way, the update of domains along the path of $u_{best}w$ really updates all domains in the convergent graph that are affected by $u_{best}w$. If no extra state is considered, S-APPENDSEPARATINGSEQUENCE checks whether any CNs can be merged due to their identification as a particular RN (line 28).

The last part of the method to describe is the function S-AREDISTINGUISHED comparing the given convergent classes if there is a different response to a common extension. As the function is recursive and it works with a possibly cyclic graph, it uses a set C to store visited convergent nodes and so the stop condition is ensured.

Algorithm 30: S-AREDISTINGUISHED($[u], [v], C$)

```

1  $s_u \leftarrow \delta^*(s_0, u), s_v \leftarrow \delta^*(s_0, v)$ 
2 if  $[u] = [v]$  or  $s_u = s_v$  then return false
3 if  $\lambda(s_u, \uparrow) \neq \lambda(s_v, \uparrow)$  then return true
4  $isRN_u \leftarrow \exists \bar{s} \in \bar{S} : \bar{s} \in [u], isRN_v \leftarrow \exists \bar{s} \in \bar{S} : \bar{s} \in [v]$ 
5 if  $isRN_u$  or  $isRN_v$  then
6   if  $isRN_u$  and  $isRN_v$  then return true
7   if  $isRN_u$  then
8     if  $(s_u, [v]) \in C$  then return false
9     add  $(s_u, [v])$  to  $C$ 
10  else
11    if  $(s_v, [u]) \in C$  then return false
12    add  $(s_v, [u])$  to  $C$ 
13 else if  $l = 0$  and  $\Phi([u]) \cap \Phi([v]) = \emptyset$  then return true
14 foreach  $x \in X$  such that  $[ux], [vx] \in T$  do
15   if  $T([u], x) \neq T([v], x)$  or S-AREDISTINGUISHED( $[ux], [vx], C$ ) then
16     return true

```

The space complexity of the S-method is similar to the one of the SPYH-method, that is, the size of test suite that was estimated to $O(n^3 p^{l+1})$ where

p is the number of inputs. In addition to the testing tree, every convergent node possesses a domain of size bounded by the size of the convergent graph if node is a reference node, and with size up to n otherwise. The convergent graph does not grow much during the construction because most of convergent nodes that are not reference nodes are used to prove the convergence and so they are merged when a transition is verified. Nevertheless, before the verifying subtrees are merged, the size of the convergent graph, or the number of convergent nodes, can be estimated to $O(n^3p^l)$ as there are $p^l \cdot (n - 1)$ sequences of length $n + l + n^2$; the length of separating sequences from the splitting tree is theoretically in $O(n^2)$. Only reference nodes can contain almost any convergent node in their domains, therefore, the space complexity of the convergent graph in the implementation (CN + domains) is $O(n^4p^l)$. Hence, the worst case space complexity of the S-method is $O(n^4p^l + n^3p^{l+1})$.

The time complexity can be estimated based on Algorithms 25–30. Let start with S-APPENDSEPARATINGSEQUENCE and assume that every of the n reference nodes corresponds to at most n^2p^{l+1} sequences in the end and so any convergent class can be bounded by this number. The separating sequence w obtained from the splitting tree has the length in $O(n^2)$. The first part of Algorithm 29 (lines 2–6) thus runs in $O(n^4p^{l+1})$. It is not so hard to pass through at most $n + l$ predecessors as CNs keep track of the related maximal sequences. Much harder is the update of domains (line 27). Assume that w extends uv where u is an access sequence of length in $O(n)$ and v is an extension of length l . First, the algorithm checks $n^2 + l$ convergent nodes with at most n RNs in their domains. They are compared with the RNs on the related suffix of length at most $n^2 + l$. Then, at most n RNs are checked with respect to the sequence of the length up to $n^2 + n + l$. The domain of an RN can contain $O(n^3p^l)$ convergent nodes. The most dominant factor is thus n^6p^l and so S-APPENDSEPARATINGSEQUENCE runs in $O(n^4p^{l+1} + n^6p^l)$. This is immediately used to derive the time complexity of S-DISTINGUISH (Algorithm 27) that calls S-APPENDSEPARATINGSEQUENCE $n(1 + n + l)$ times in the worst case. In addition, it calls S-AREDISTINGUISHED (Algorithm 30) $l + n(n + l)$ times and at most n separating sequences are obtained from the splitting tree. The function S-AREDISTINGUISHED compares the successors of the given CNs so that it is bounded by the size of convergent graph, that is, $O(n^3p^l)$. S-DISTINGUISH thus runs in $O(n(1 + n + l)(n^4p^{l+1} + n^6p^l) + (l + n(n + l))(n^3p^l) + n \cdot n^2)$ and so in $O(n^6p^{l+1} + n^8p^l)$. A divergence-preserving state cover is created (or updated) by S-CREATEDIVERGENCEPRESERVINGSC such that a separating sequence is obtained for each state, domains are then updated in $O(n^7)$ and finally S-DISTINGUISH is called n times. The testing tree and the convergent graph are bounded by $O(n^3)$ before the domains are updated in Algorithm 26 because each access sequence of state has the length at most $n - 1$ and is extended with the separating sequence of the length in $O(n^2)$. Therefore, the comparison of n RNs with all other $O(n^3)$ convergent nodes takes $O(n^7)$ time as the successors to compare are also bounded by $O(n^3)$. The worst case time complexity of S-CREATEDIVERGENCEPRESERVINGSC is $O(n \cdot n^2 + n^7 + n(n^6p^{l+1} + n^8p^l))$ that is $O(n^7p^{l+1} + n^9p^l)$. Algorithm 25 puts all together. First, the splitting

tree is constructed in $O(2^n)$ but usually it is in $O(n^2)$, see Section 4.4. Then, S-CREATE DIVERGENCE PRESERVING SC is called and transitions are sorted in $O(n^2 p^2)$. For each unverified transition, every extension of the length up to l is appended to both related classes and the classes are merged when the transition is verified. S-DISTINGUISH is called from S-DISTINGUISH FROM SET for each such extension to verify the transition. For these extensions, S-APPEND SEPARATING SEQUENCE takes just $O(ln^2 p^{l+1} + n^5 p^l)$ time because the sequences have at most $n + l$ symbols. The merge of two classes can be estimated to run in $O(n^3 p^l)$ as there are $p^l \cdot n$ sequences of the length up to $l + n^2$ and l is assumed to be smaller than n . Therefore, the S-method runs in $O(2^n + (n^7 p^{l+1} + n^9 p^l) + n^2 p^2 + np((ln^2 p^{l+1} + n^5 p^l) + p^l(n^6 p^{l+1} + n^8 p^l) + n^3 p^l))$ which corresponds to $O(n^7 p^{2l+2} + n^9 p^{2l+1})$ plus $O(2^n)$ to construct the splitting tree.

11.2 Running Example

This section describes how the S-method creates a test suite for the 5-state Mealy machine defined in Figure 4.1. The beginning of the construction is captured by the testing tree in Figure 11.1. Nodes of the testing tree are numbered according to the time of their creation. After the method obtains the splitting tree shown in Figure 4.2, it creates the testing tree with 10 nodes that captures all 5 access sequences extended with the separating sequence ‘aa’ (or just ‘a’ in the case of the access sequence ‘aa’ of state B). The state cover is then made divergence-preserving by appending the separating sequence ‘cb’ to the access sequences of A and D (nodes 10–13), and the separating sequence ‘b’ to the access sequences of C and E (nodes 14 and 15). The unverified transitions are sorted like in the SPYH-method, hence, the first one to verify is (A,c) and the second is (C,b). Their verification is captured in Figure 11.1 and described in the following paragraphs.

The verification of (A,c) starts with appending 1-input extensions to the classes $[c]$ and $[\varepsilon]$ because the transition leads back to A. As the convergent classes are still singletons, there is only one convergent sequence to extend. After appending the 1-input extensions (nodes 16 and 17), node 10 reached by ‘c’ is distinguished from reference nodes 1 (state E) and 2 (state B) so that the domain $\Phi([c]) = \{A,C,D\}$. The separating sequence ‘aa’ is obtained from the splitting tree as the best for distinguishing A from the other two states. When it is appended (node 18), the domains $\Phi([caa])$, $\Phi([ca])$, $\Phi([c])$ and $\Phi([\varepsilon])$ are updated in this order by the related suffix of ‘caa’. The domains of CNs that are not yet distinguished from the RN of state A would be also updated if they were distinguished from the RN of A by ‘caa’. Sequence ‘aa’ reduces $\Phi([c])$ to $\{A,D\}$ so that another separating sequence is needed. The splitting tree provides sequence ‘cb’ that is appended just to ‘c’ (node 19) because the access sequence ‘ac’ of state D is already extended by ‘cb’. The function S-DISTINGUISH FROM SET then transfers the verification process to $[ca]$. The domain $\Phi([ca]) = \{A,C,D,E\}$, hence, appending the separating sequences ‘b’ to ‘ca’ and to the access sequence ‘ac’ of B is sufficient to identify

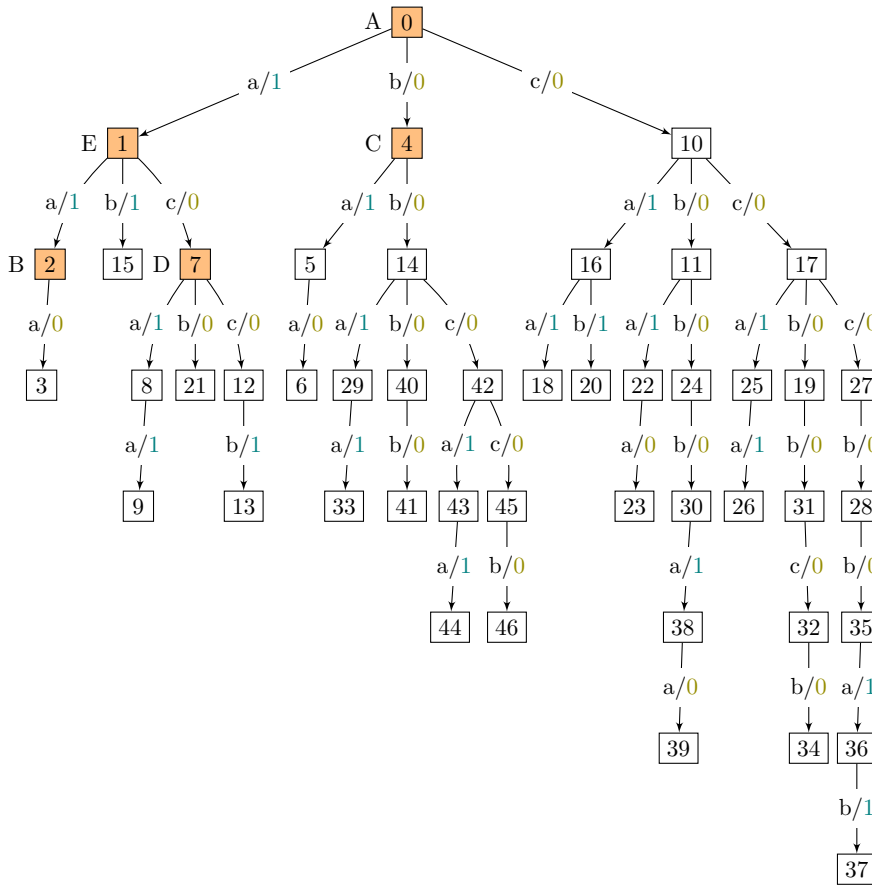


Figure 11.1: S-method: testing tree after verifying transitions (A,c) and (C,b)

the related CN ($\Phi([ca]) = \{E\}$) and also distinguish it from $[c]$ that is in V when S-DISTINGUISH is called. Similarly, the separating sequences ‘aa’ and ‘b’ are appended to ‘cb’ (nodes 22–24) and the sequences ‘aa’ and ‘cb’ to ‘cc’ (nodes 25–28). Then, transition (A,c) is verified and the convergent node corresponding to node 10 of the testing tree (with 29 nodes) can be merged into the reference node of state A. Reference nodes group these nodes of the testing tree: nodes 0, 10, 17 and 27 relate to state A, nodes 2, 18 and 26 to B, nodes 4, 11, 19 and 28 to C, only node 7 to D, and nodes 1, 16 and 25 to E.

Transition (C,b) as the second transition to verify is also extended with each input at first. Input ‘a’ extends ‘bb’ (node 29), input ‘b’ extends ‘cbb’ that was already in [bb] (node 30), and input ‘c’ employs the improvement of the S-method. As there are only ‘bb’ and ‘cbb’ in [bb] and they are not maximal, the method searches in the predecessors and it finds ‘ccb’ in [b]. Input ‘c’ is thus prepended with ‘b’ and such a sequence is appended to ‘ccb’ (nodes 31 and 32). This local optimization enlarged T just by 2 inputs instead of addition a new test sequence of length 3. The extensions ‘a’ and ‘b’ distinguish [bb] from the reference nodes of states B and E so that $\Phi([bb]) = \{A,C,D\}$. Hence, the separating sequences ‘aa’ and ‘cb’ are appended to ‘bb’ and ‘ccbb’ that are in [bb]. This verifies [bb] and so S-DISTINGUISH is called on [bba]

that is already distinguished from RN of state B. The sequence ‘bba’ leads to state E that can be separated from A, C and D by ‘b’. As [bba] has no maximal sequence, its predecessors are checked and ‘cccb’ is found in [b]. It means that ‘cccb’ is first extended with ‘ba’ as a transfer sequence and then ‘b’ is appended (nodes 35–37). This time, the improvement saves 1 symbol and 1 test sequence; ‘bbab’ would be added to T if ‘bab’ did not extend ‘cccb’. After appending ‘aa’ to ‘cbbb’ (nodes 38 and 39), all maximal sequences of the predecessors were extended and so the other separating sequences are appended to the shortest sequence of the related classes. The test suite is thus enlarged with new test sequences ‘bbbb’, ‘bbcaa’ and ‘bbccb’. Then, transition (C,b) is verified because the testing tree with 47 nodes satisfies the conditions of Theorem 8.10. The convergent node including node 14 is then merged into the reference node of state A. The convergent graph after the merge is shown in Figure 11.2; nodes are labelled with the related nodes of the testing tree and the leaves are underlined.

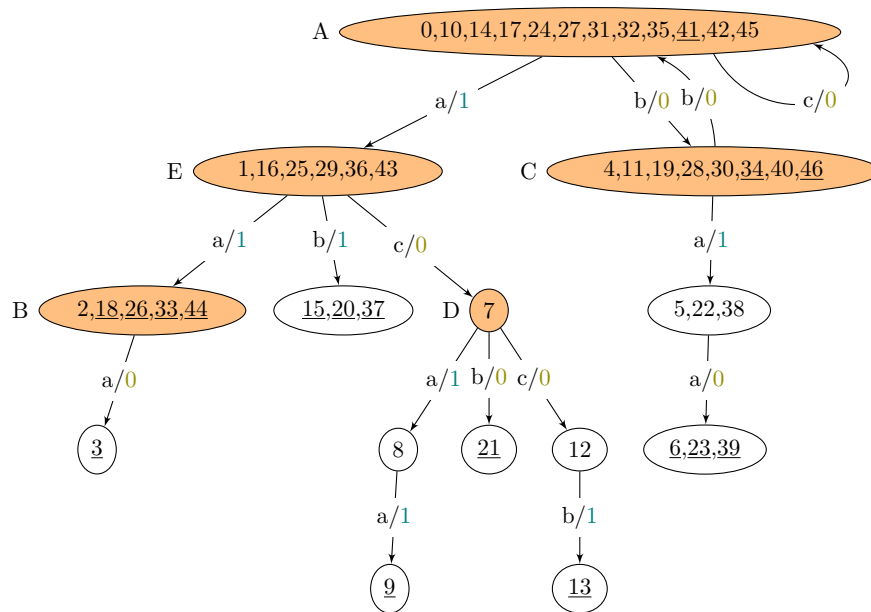


Figure 11.2: Convergent graph after verifying transitions (A,c), (C,b)

Chapter 12

Experiments

The three previous chapters described the standard and two new testing methods. The methods were implemented by the author of the thesis in the FSMLib (Appendix A) in order to compare their performances. The experimental evaluation is done on two sets of machines as the following sections describe. The first set comprises randomly generated finite-state machines (described in Appendix C.1) and the second one consists of three models of real systems (described in Appendix C.2).

Most machines in the experiments do not have an adaptive distinguishing sequence (ADS) because machines with ADS are rare amongst all possible DFSMs. Hence, the ADS-method is not included in the comparison. The experiments thus compare the SVS-, W-, Wp-, HSI-, H-, SPY-, SPYH- and S-methods. In addition, there are two versions of the HSI- and SPY- methods according to the harmonized state identifiers (HSI) that they use. The results labelled with ‘HSI’ and ‘SPY’ belong to the methods using the HSIs constructed from the shortest separating sequences (SSS) as Section 3.3.2 describes. The labels ‘HSI/ST’ and ‘SPY/ST’ relate to the methods that use the HSIs constructed from the splitting tree ST-IADS using Algorithm 6. The characterizing set and state characterizing sets for the W- and Wp- methods are constructed from SSS (Section 3.2.3) and then reduced using the LS-SL algorithm defined in Algorithm 2.

Each method constructs an m -complete test suite T for each machine. A constructed T is described by four values: the number of tests ($|T|$), the total number of symbols (the sum of the lengths of all tests), the construction time and the exploration efficiency. The first two measures the size of constructed test suite and so they address the research question (RQ) II.2, the third measure captures the performance of a testing method, and the *exploration efficiency* (EE) is a new objective developed by the author of the thesis. The EE is calculated as the number of edges in the testing tree of T divided by the total number of symbols in T . As it is based on the testing tree, it permits one to evaluate how much of the implementation will be explored by tests even in the implementation with much more states than the specification. The exploration efficiency also captures how much effort would be put in the exploration of the implementation because the size of the testing tree is divided by the total number of symbols. In addition, it captures how

many prefixes of tests are overlapping with other tests, for example, the fixed access sequences are covered by several tests. In other words, the value of 1 minus the value of exploration efficiency represents the duplicated sequences that transfer the machine from the initial state to some other states. The exploration efficiency is thus higher (and better) if a testing method constructs longer sequences that do not overlap much.

Testing Method	No extra state			1 extra state			2 extra states		
	$ T $	TL	EE	$ T $	TL	EE	$ T $	TL	EE
Wp	20	76	0.57	59	280	0.46	174	1015	0.38
HSI	23	88	0.55	68	330	0.44	203	1192	0.37
HSI/ST	20	80	0.59	59	293	0.48	174	1055	0.40
H	19	72	0.60	56	279	0.48	168	1003	0.40
SPY	12	67	0.82	39	257	0.68	124	913	0.56
SPY/ST	10	62	0.85	37	243	0.68	108	859	0.61
SPYH	17	70	0.61	44	254	0.52	130	893	0.44
S	10	66	0.86	28	231	0.77	84	807	0.68

Table 12.1: Testing methods compared on the Mealy machine (Figure 4.1)

The testing methods are first compared on the 5-state Mealy machine defined in Figure 4.1. Each method constructs 3 test suites depending on the number l of extra states that is 0, 1 or 2. Table 12.1 shows the results for most methods. The columns labelled with ‘TL’ contain the total numbers of symbols. The values that are the best for each column are highlighted in green. The S-method thus shows promising improvement in the size of constructed test suites. Notice that ‘SPY/ST’ is the second due to the use of HSIs constructed from the splitting tree.

12.1 Randomly Generated Machines

Randomly generated machines are employed in order to evaluate the new testing methods. Appendix C.1 describes the suite of 13 600 machines that were generated using the generator in the FSMlib (Appendix B). There are 4 machine types, each represented equally with 3 400 machines such that a half of them has 5 inputs and the other 1 700 machines have 10 inputs. The number of states ranges from 10 to 1000 and there are 17 state groups of 100 machines. The testing methods construct 3 m -complete test suites for each machine. The three test suites differ as the number l of extra states ranges from 0 to 2. The parameters of the constructed test suites are grouped in order to describe each state group of 100 machines. All machines and the results are available in the repository FSMmodels v1.3¹.

Figure 12.1 and Figure 12.2 show the result for Moore machines with 5 inputs. On the right of each figure, there is a comparison of the testing methods on the state group of machines with 1 000 states. The boxplots

¹<https://github.com/Soucha/FSMmodels/releases/tag/v1.3>

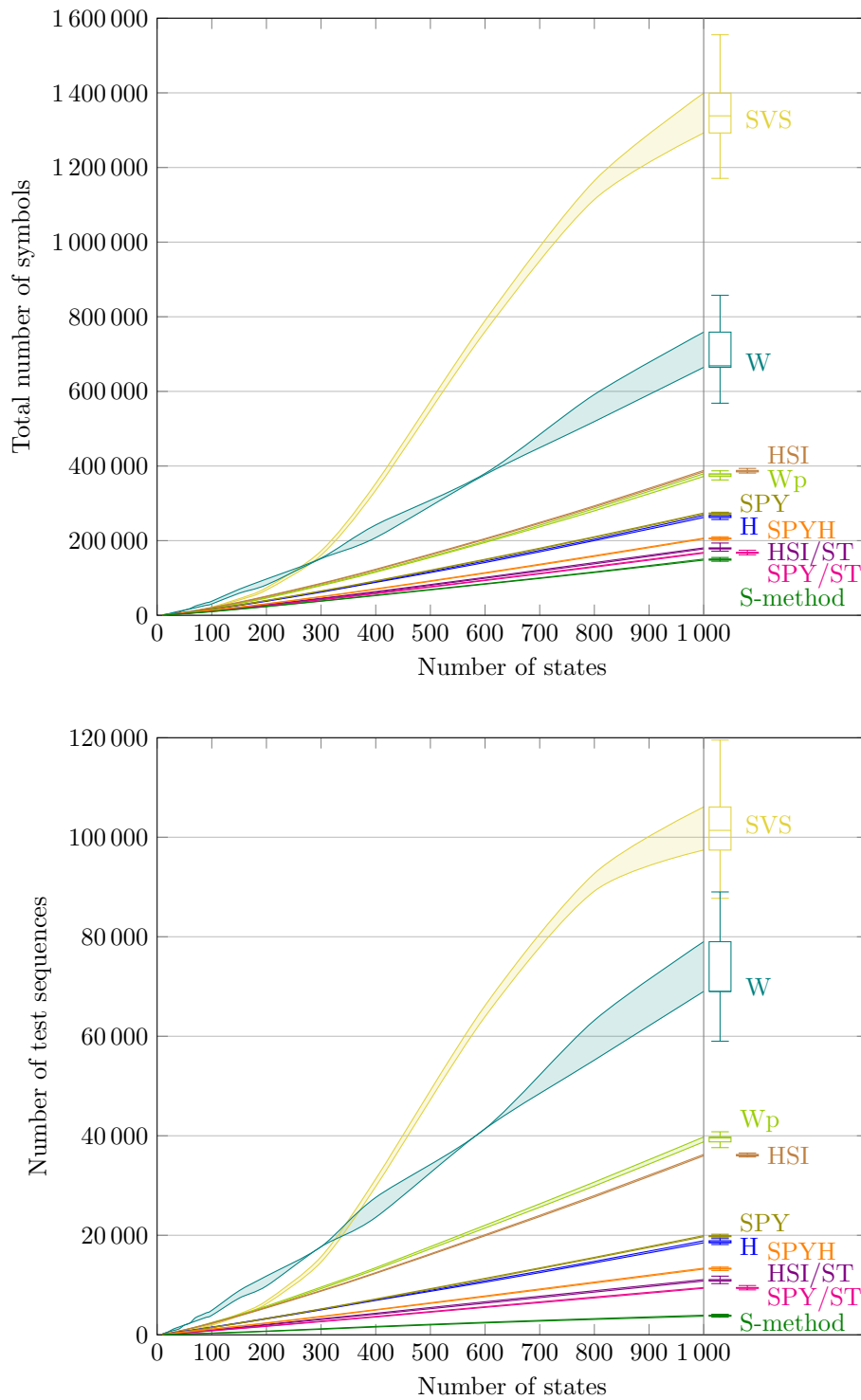


Figure 12.1: Moore machines with 5 inputs and no extra state: the size of a test suite and the number of test sequences, 1 and 3 quartile calculated for 100 machines per each state group

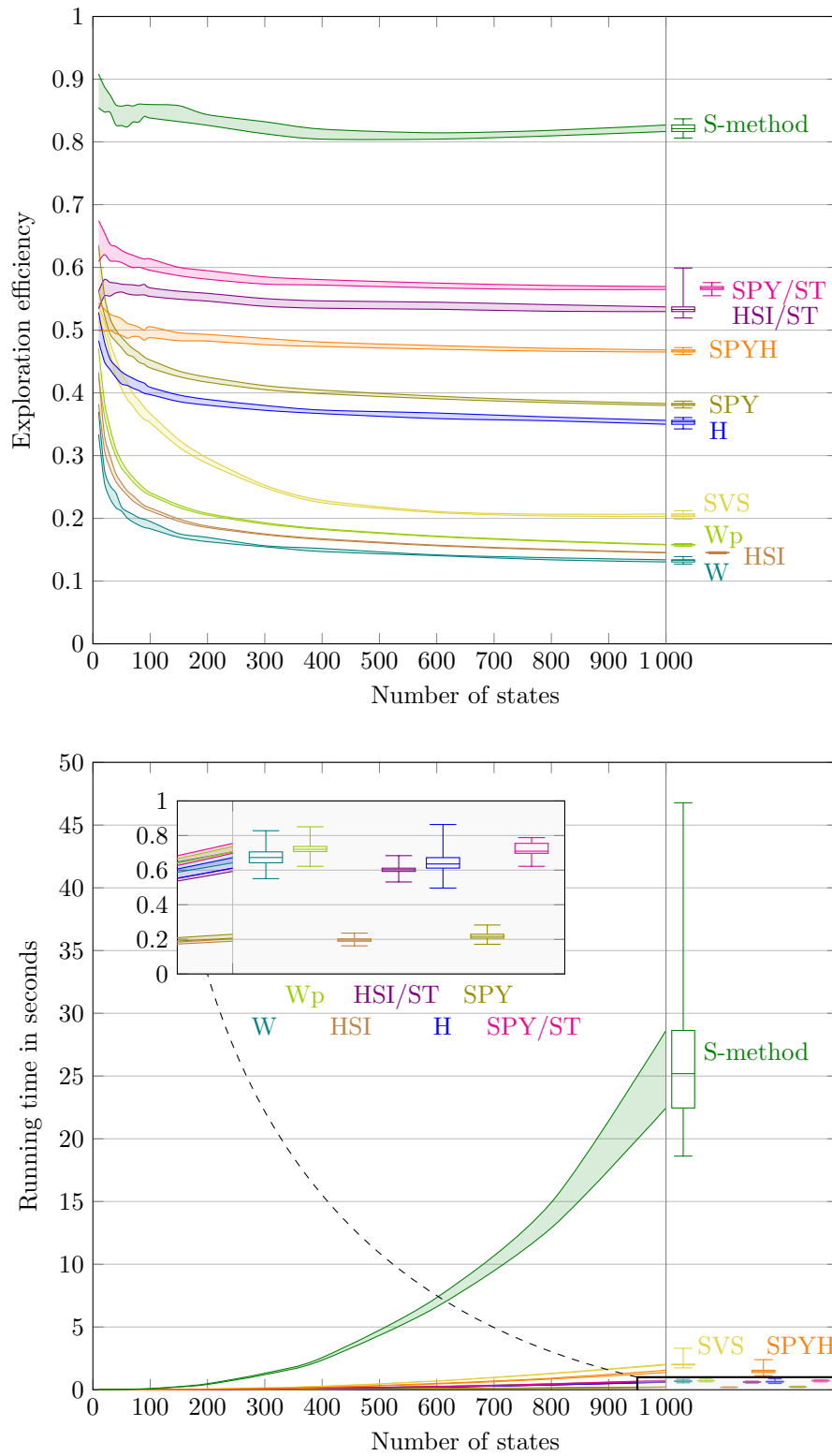


Figure 12.2: Moore machines with 5 inputs and no extra state: the exploration efficiency of the test suite and the running time of the testing methods, 1 and 3 quartile calculated for 100 machines per each state group

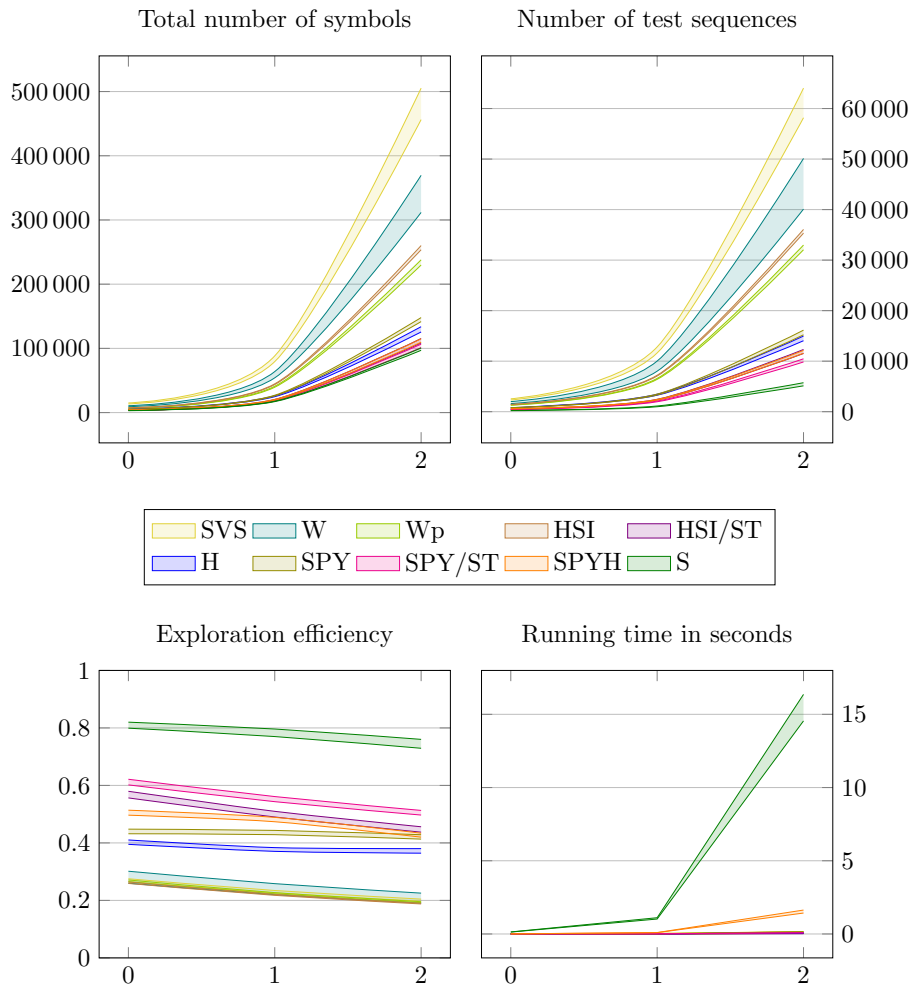


Figure 12.3: Mealy machines with 100 states and 5 inputs: values for 0, 1 and 2 extra states

show the median, 1 and 3 quartiles, and whiskers capturing minimum and maximum values for the 100 Moore machines.

The SPYH-method creates smaller test suites than the standard testing methods including the H- and SPY- methods. However, if the splitting tree ST-IADS is used to construct HSIs, both the HSI- and SPY- methods create even smaller test suites. Nevertheless, the smallest test suites are constructed by the S-method. There is significant gap between the S-method and the others in the number of tests. The exploration efficiency just emphasizes the improvement by the S-method. The relative order of the SPY, H, SPYH, HSI/ST, SPY/ST and S methods holds for the other three machine types and for machines with 10 inputs as well.

The main drawback of the S-method is its running time. Less than a minute for the construction of n -complete test suite of Moore machine with 1 000 state and 5 inputs is not that bad but the trend shows that scaling up to bigger models would be hard. Moreover, assuming an extra state in

the construction also increases the running time as shown in Figure 12.3. A better choice thus would be the use of the SPY-method with the HSIs constructed from ST-IADS if one had fixed time for the construction of an m -complete test suite.

Figure 12.3 captures the results of Mealy machines with 100 states and 5 inputs. The objectives are shown with respect to the increasing number of extra states considered during the construction of test suites. The total number of symbols, the number of test sequences as well as the running time grow exponentially for all methods. It corresponds to p^l extensions that need to be included in order to prove the absence of l extra states. Note that the running time of all the standard methods is close to 0, hence, they are not visible on the bottom right plot of Figure 12.3.

12.2 Models of Real Systems

The testing methods are also evaluated on three models of real systems. The models are described in Appendix C.2. They are referred as `peterson2`, `sched4` and `sched5`. All three are deterministic finite automata and they have 50, 97 and 241 states and 18, 12 and 15 inputs, respectively.

The results capturing the constructed test suites for each model are shown in Figure 12.4. The testing methods assumed three numbers of extra states, 0, 1 and 2. As the size of test suites grows exponentially with the increasing number l of extra states, the total number of symbols and the number of test sequences are divided by p^l where p is the number of inputs.

The new SPYH- and S- methods perform well if no extra state is assumed. However, if one or two extra states are considered, both methods construct just a little smaller test suites than the H-method. The best method for these three models of real systems is the SPY-method. It performs slightly worse than the S-method when no extra state is assumed but otherwise, it creates the smallest test suites. Moreover, it does not depend on the type of harmonized state identifiers (HSI) that it works with. Both SPY and SPY/ST that obtain HSIs from the shortest separating sequences (SSS) and from the splitting tree ST-IADS, respectively, construct almost the same test suites in terms of their sizes. In contrary, the HSI-method that also uses these two different HSIs creates quite different test suites. HSI/ST constructs bigger test suites than the HSI-method using HSIs from SSS. This may reflect that HSIs from ST-IADS consist of more input symbols in total than HSIs from SSS as Table 5.1 shows for the three models of real systems. The SPY-method employs the convergence of sequences and so the sequences of HSIs can overlap each other in order to reduce the size of resulting test suite. However, the HSI-method only appends HSIs to particular sequences so that it does not take an advantage of the longer separating sequences constructed from the splitting tree.

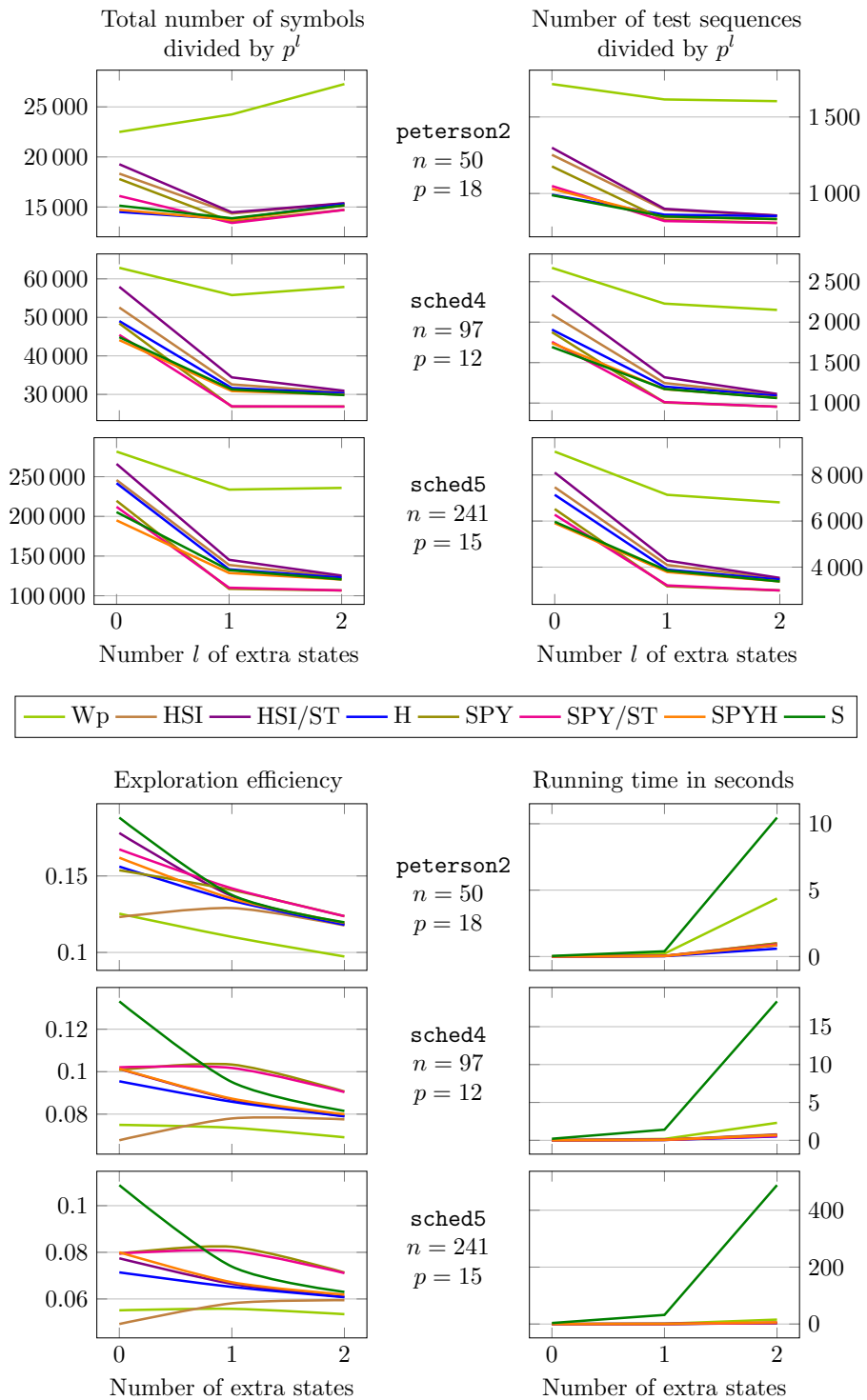


Figure 12.4: DFA models of three real systems with different numbers of states and inputs: values for 0, 1 and 2 extra states

Chapter 13

Conclusion

The second part of this thesis is about testing of resettable deterministic finite-state machines. It provided both theoretical and algorithmic insight into this research field with long history. First, sufficient conditions for m -completeness that ensure a guarantee of the absence of particular faults were discussed (Section 8.3). Then, an overview of the standard testing methods was given in Chapter 9. The H- and SPY- methods as the most advanced ones were accompanied with a running example which their construction approach of test suites was described on.

There are several contributions done by the author of the thesis. In the testing theory, a flaw was discovered and so an amended Lemma 8.7 was proposed in order to prove that two sequences are F_T -convergent. The use of state domains provides new means of the identification of states as Theorem 8.11 defines. A new sufficient condition for m -completeness called the S-condition was proposed and thus the research question (RQ) II.1 is answered positively. The condition is more general than the standard ones and so it allows one to construct even smaller test suites.

Two new testing methods were proposed, namely the SPYH-method (Chapter 10) and the S-method (Chapter 11). They were experimentally evaluated against the standard testing methods on randomly generated machines and on models of real systems. The result of experiments are described in Chapter 12. The S-method significantly outperforms the other testing methods on the randomly generated machines in terms of the size of constructed test suites. Hence, RQ II.2 has also a positive answer that there is a testing method constructing small test suites. Harmonized state identifiers (HSI) constructed from the splitting tree ST-IADS as proposed in Chapter 4 improve the performance of both the HSI- and SPY- methods on randomly generated machines. In contrast, they do not help the HSI-method on the models of real systems. RQ I.2 can be thus answered that the ST-IADS is a new technique that helps to create smaller test suites via constructed HSIs. If one has fixed time for the construction of a test suite, then the SPY-method using HSIs constructed from ST-IADS seems to be a good choice according to the experimental evaluation.

Overall, this part of thesis proposed new ways how the testing of finite-state machine can be improved.



Part III

Active-Learning Algorithms

Chapter 14

Introduction

Interaction with a system can supply enough information to create a model of any system. Such a model then provides not just a description of how the system works but also a valuable insight into the behaviour of the system. An analysis of the model can reveal a fault in the system design or an improvement how to work with the system more efficiently. In addition, if the model has particular properties, it can be used for testing the system as was described in Part II.

This part of thesis deals with *active learning* of systems that:

- can be modelled as a *deterministic finite-state machine* that is *completely specified* and *initially connected*, and
- can be reliably *reset* to their unique initial states.

These two conditions simplify the learning and ensure that it is possible to derive a correct model of the system from an interaction of finite length. It is based on the theorem by Gold that each finite automaton is identifiable in the limit from positive and negative examples [Gol72]. This theorem initiated the field of active learning, particularly *automata active learning* as the concern was the learning of deterministic finite automata (DFA), or regular languages. Deterministic finite-state machine (DFSM) is a superclass of DFA but it is still in the class of regular languages and so there is a finite automaton for any DFSM. Therefore, the theorem holds even for DFSM (the first condition). Machines need to be resettable (the second condition) because the theorem requires one to observe all possible outputs which also relates to connectedness. One can learn only states that are reachable from the initial state, therefore, the model of a system is at least initially connected. The condition of having specified a transition on each input from all states has two purposes. First, it simplifies the learning as working with incomplete machines has several obstacles, for example, partially specified machines do not have a minimal form in general or there does not have to be a separating sequence of a state pair. Second, it allows one to explore any part of the system by trying arbitrary sequence of inputs which can reveal a rare anomaly in the system. Most systems are not completely specified but there are techniques to extend them so that the active-learning approaches described in this part of the thesis can be used even for such systems.

The following section introduces terms used in active learning of finite-state machines. Chapter 15 proposes the observation tree approach and three new learning algorithms, in particular the H-learner (Section 15.3), the SPY-learner (Section 15.4) and the S-learner (Section 15.5). Standard learning algorithms are described in Chapter 16 and all algorithms are compared on three experiments in Chapter 17. A summary of the algorithms and their interpretation of the General Learning Framework proposed in Section 15.6 are described in Section 16.8. Chapter 18 concludes this part of the thesis.

14.1 Active Learning

This section introduces terms used in active learning of finite-state machines including a black box, a teacher, types of queries, and a counterexample. All terms are described and then explained on an example in Figure 14.1.

Active learning is an interaction of a learning algorithm with a system usually called a *black box*. A learning algorithm, or a *learner*, tries to infer the inner representation of the black box. The representation constructed by a learner is called a *conjectured model*, or simply a *conjecture* or a *hypothesis*. The interaction between the learner and the black box is provided by a *teacher* that abstracts and simplifies the learning task. This is the standard setting of active learning of finite-state machines adopted from the field of *active automata learning*. The difference between these two field will be discussed in the following chapter.

A *black box* (BB) is a system whose inner representation is not available but it provides observable responses to applied inputs. In the case of active learning, a black box is assumed to be modelled by a finite-state machine. The type of FSM is known in advance and the input alphabet X is given. A *conjectured model* should correspond to the BB in the end of learning, hence, it has the same type and the same input alphabet as the black box. Formally, the conjectured model is a DFMSM $M = (S, X, Y, s_0, D_M, \delta_M, \lambda_M)$ and the black box is a DFMSM $N = (Q, X, Y', q_0, D_N, \delta_N, \lambda_N)$. The numbers of states, inputs and outputs of M are denoted by n, p, q , respectively, that is, $n = |S|$, $p = |X|$ and $q = |Y|$. The black box has m states and it holds that $Y \subseteq Y'$ as the conjecture model can produce only outputs that were observed as a response of the black box. Both models can differ in transition and output functions during the learning as the sets of states do; $|S| \leq |Q|$. However, when the learning ends, both models should be output-equivalent. It means that there is a bijective function γ mapping states of M to states of the black box, that is, $\gamma : S \rightarrow Q$, and the correspondence of transition and output functions, $\gamma(\delta_M(s, x)) = \delta_N(\gamma(s), x)$ and $\lambda_M(s, x) = \lambda_N(\gamma(s), x)$, hold for all defined transitions $(s, x) \in D_M \cup \{S \times \{\uparrow\}\}$. Moreover, $D_N = \{(\gamma(s), x) \mid (s, x) \in D_M\}$ and $Y = Y'$. Notice the correspondence between the conjectured model and the specification (defined for testing of finite-state machines in Chapter 8), and between the black box and the implementation in testing. They are defined equally because they have very similar task in both testing and learning as was depicted in Figure 1.4.

A learning algorithm, or a *learner*, aims to create a conjectured model that is output-equivalent to the black box. As the black box is unknown to the learner, a completely specified conjectured model is constructed based only on the observed responses to the input sequences that the learner asked. The learner analyses these traces, that is, input-output sequences, and finds separating sequences that identify different states of the black box. Separating sequences are then used to define all transitions from revealed states of the black box and thus to create a completely specified conjectured model. States of the black box are identified both by separating sequences that show their uniqueness and by their access sequences. As the black box is assumed to be resettable, once the responses to separating sequences distinguish a state from the others already revealed, its access sequence is remembered and the learner can reach this state and test transitions leading from it by resetting the system and asking this access sequence. Learning algorithms differ in the way they store and analyse traces, which separating sequences they use and how they use them, and what they can say about the correctness of the conjectured model with respect to the black box.

A *teacher* for active-learning algorithms, also called a *minimally adequate teacher* [Ang86], is an abstraction that simplifies the learning. The teacher provides a learning algorithm with information about the black box. A learner can ask two essential types of queries that the teacher answers. The first type of query is asking for an output to the given input. The second type of query requires a check of equivalence between the black box and the conjectured model inferred by the learning algorithm. The teacher can also cover a *mapper* and a *query filter*. A mapper translates abstract symbols used by the learner to more complex messages required by the system. A query filter answers queries that were asked previously or the response is known to the teacher because of a special property of the black box, such as prefix-closedness of the language described by the BB.

The first query type depends on the type of DFSM that models the black box. If the black box specifies a regular language, usually modelled by a DFA, or the black box is a classifier, usually modelled by a Moore machine, then the teacher provides a *membership query* (MQ) to decide which class the queried sequence belongs to. Otherwise, an *output query* (OQ) is in question. The teacher replies to a MQ with the last output symbol and to an OQ with a sequence of all output symbols observed along the path following the queried input sequence. That is, for all input sequence u and $x \in X_{\uparrow}$:

$$MQ(u \cdot x) = \lambda_N(\delta_N^*(q_0, u), x) \quad \text{and} \quad OQ(u) = \lambda_N^*(q_c, u),$$

where q_c is the current state of the black box. Instead of OQ taking only one sequence, a specified OQ providing only output to the given suffix is more appropriate for active-learning algorithms:

$$OQ(u, v) = \lambda_N^*(\delta_N^*(q_0, u), v).$$

Both types of output queries are covered in the definition of output-providing function T :

$$T(u, v) = \begin{cases} \text{OQ}(u, v) & \text{if OQs are allowed,} \\ \text{MQ}(u \cdot v) & \text{if only MQs are allowed.} \end{cases} \quad \forall u, v \in X_{\uparrow}^*$$

Note that the machine is reset before each query T as both MQ and OQ are defined to start in the initial state. Therefore, the learning algorithm can ask another version of T if it wants to continue querying without reset. A query $T(v)$ does not reset the black box and the teacher replies either with $\text{OQ}(v)$ or with $\text{MQ}(u \cdot v)$ according to which type of output query is allowed. The previous queried sequence u starting in the initial state is remembered for the purpose of MQ by the teacher. Formally,

$$T(v) = \begin{cases} \text{OQ}(v) & \text{if OQs are allowed (without reset),} \\ \text{MQ}(u \cdot v) & \text{if only MQs are allowed (} u \text{ remembered).} \end{cases} \quad \forall v \in X_{\uparrow}^*$$

In testing, T represents both a test suite and a function returning the responses to the test sequences. This is adopted in learning. As soon as an input sequence uv is queried by $T(uv)$ or by $T(u, v)$, the response is obtained by the particular type of function T and the sequence uv is added to the set T which could be called a *query suite*.

The second query type is called an *equivalence query* (EQ). The learner provides the teacher with the conjectured model M and asks to check it against the black box. The teacher compares the model and the black box and if they produce the same response to each input sequence, that is, they are output-equivalent, then the teacher replies with success. However, if there is an input sequence w for which the outputs differ, then the teacher returns this sequence w that is called a *counterexample* (CE).

Counterexamples play a vital role for most active-learning algorithms. Therefore, they were analysed thoroughly. Let s_u be the state reached by u from the initial state of the conjectured model M , that is, $s_u = \delta_M^*(s_0, u)$. Each state s is identified by a unique access sequence \bar{s} . Hence, \bar{s}_u denotes the access sequence of state reached by u in M . The following theorem is an adjusted version of Theorem 2 in [SHM11] where a proof can be found.

Theorem 14.1. (Counterexample Decomposition). *For every counterexample w there exists a decomposition $w = u \cdot x \cdot v$ into a prefix u , an input x , and a suffix v such that $T(\bar{s}_u \cdot x, v) \neq T(\bar{s}_{ux}, v)$.*

Theorem 14.1 says that a ‘new’ state of the black box that has not yet been identified and captured by the conjectured model can be revealed by a suffix v queried from particular states. Suffix v is a separating sequence of the state reached by $u \cdot x$, that is, $\delta_M^*(s_0, u \cdot x)$, and the new state that is the next state of the state $\delta_M^*(s_0, u)$ on x , that is, $\delta_M(\delta_M^*(s_0, u), x)$ after updating δ_M . Theorem 14.1 states that there exists a decomposition but there can be more than one such decomposition. This would be the case when the counterexample reveals more than one state or it contains a cycle.

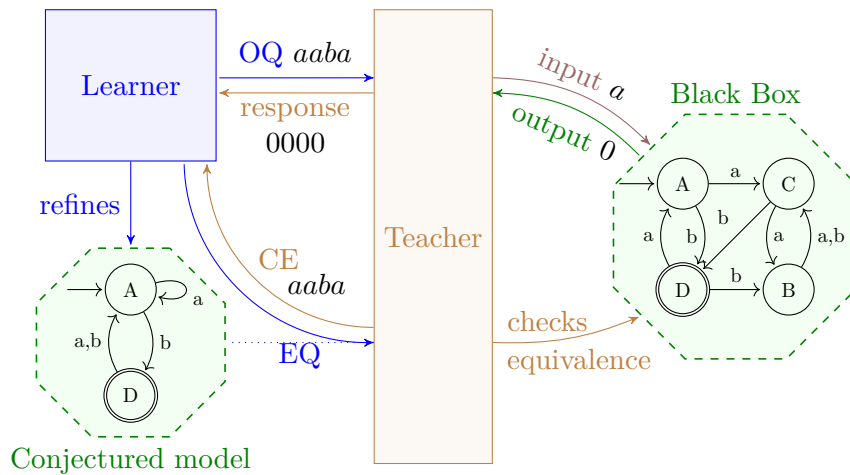


Figure 14.1: Active learning of finite-state machines

Figure 14.1 shows a general process of active learning on a specific machine. All general notions are coloured whilst all relating to the particular example are in black. In the example, the black box is modelled by a 4-state DFA with the accepting state D . Consider an intuitive learner, it starts with resetting the black box and asking for the state output of the initial state, that is, it queries $T(\varepsilon, \uparrow)$ with the response of 0 as state A is rejecting. Then, it tries both inputs. First, $T(a)$ returns 0 as the reached state C is rejecting. The learner considers the reached state equal to the initial one as they are not separated, hence, it defines the transition (A, a) to lead back to A . The next query is $T(\varepsilon, b)$. The output of 1 reveals a new state that is identified by the access sequence of 'b' and is separated from A by \uparrow . Let the state be called D to correspond with the black box. The learner then wants to define transitions from D , therefore, it asks $T(a)$ and $T(b, b)$. Both queries return 0 and so the learner assumes that both transitions (D, a) and (D, b) lead to state A . The conjectured model is thus completely specified as depicted in Figure 14.1 and the learner decides to ask an EQ. The teacher checks the provided model against the black box and replies with the counterexample 'aaba'. The learner then asks an OQ on the CE, that is, $T(\varepsilon, aaba)$. In order to answer, the teacher resets the black box and then applies gradually all four input symbols and accumulates the observed outputs as shown in Figure 14.1. The response 0000 to the OQ captures the inconsistency between the conjectured model and the black box as the output of the conjecture model to 'aaba' is 0010. Therefore, the learner needs to refine the conjectured model to be consistent with the observed responses. This is done with the help of further output queries which reveal a new state and the learning cycle of creating a completely specified conjectured model repeats. Notice that the outputs from the black box and the conjectured model to a CE do not have to differ in the last symbol if T implements OQs. Sequence 'aab' would be a counterexample for the 2-state conjectured model in Figure 14.1.

If the system is not completely specified, that is, transitions on some inputs are missing, then there are several approaches how to deal with this type of machines. A teacher that acts as a mapper can respond with the ‘invalid’ output symbol to any input that is not allowed in the current state of the system and either let the system remain in the current state or reset it into its initial state. All learning algorithms in the following chapters thus relax on this and assume that the black box is completely-specified.

■ 14.2 Research Questions

- III.1 Is there an active-learning framework more general than Observation Pack [BDGW96] that the standard learning algorithms implement? A framework is more general if a learner implementing the framework can utilize techniques decreasing the number of queries compared to techniques available in a less general framework.
- III.2 Is there a learner that can learn a black box system using less amount of interaction with the black box than the standard learning algorithms?

Chapter 15

Observation Tree Approach

The aim of this thesis is to show that it is possible to learn finite-state machines from **reasonable interaction** with them and at the same time with a **minimal help of the teacher**. This is a slightly different objective than the field of active learning has had for the last three decades. This chapter starts with a brief summary of the development in the field of active learning of finite-state machines. Drawbacks are emphasized as they are addressed by the observation tree approach that is gradually introduced along with the related work. After the key structure, the observation tree, is described, a learner based on the observation tree approach is presented in Section 15.1. Three novel learning algorithms that implement the observation tree approach are proposed along with a description of their implementation and a running example in Sections 15.3–15.5. The chapter is concluded with a new learning framework that describes a general flow such that any active-learning algorithm implements it (Section 15.6). A brief description of the approach with a sketch of novel learners is published in [SB19].

The field of automata active learning began with the well-known L^* algorithm in 1986 [Ang86]. It learns a deterministic finite automaton in polynomial time of the number of states of the black box and the maximal length of a counterexample returned to an equivalence query. The focus of the research reduced to the questions of how to derive a completely specified conjectured model using additional membership queries and how to process a counterexample (see Theorem 14.1 for the decomposition of a CE). Trying to reveal states intentionally was omitted and passed to the teacher as its responsibility. This is most obvious in the second oldest learning algorithm, the Discrimination tree algorithm [KV94], that needs a counterexample to reveal almost every state of the black box so that the number of equivalence queries is proportional to the number of states of the black box. Besides the number of EQs, the main measure to compare the learning algorithms was the number of membership queries. Unfortunately, this measure was adopted to compare algorithms that learn reactive systems usually modelled by Mealy machines [Nie03]. The number of output queries represents how many times such a reactive system is reset but it does not say anything about the time spent on processing a query. It means that the length of queries was not considered as an important factor.

The aforementioned goal of efficient learning has two parts that depend on each other. Interaction with the system is characterized both by the number of resets during the learning and by the total number of symbols that are queried. On the one hand, one can reduce the interaction with the black box by a proper analysis of observed traces. The standard learning algorithms (Chapter 16) do not analyse the traces and so they often query sequences that do not add any new knowledge. A particular reason is that they use a fixed set of separating sequences. On the other hand, elimination of the dependency on the counterexamples relates to an exploration of the black box just by the learner itself. It means more output (or membership) queries that aim to reveal states of the black box that are not included in the conjectured model. Imagine a learner that queries less symbols than the standard learning algorithms to construct a completely specified conjectured model; it could be due to the analysis of the observed traces. The difference of the symbols queried by the standard learning algorithms and by the learner could then be used by the learner to explore the black box using random output queries. The chance of finding a new state grows with the number of symbols that were queried. However, it can happen that no additional knowledge about the black box is gained. If the learner does not provide any guarantee about the correctness of the conjectured model, it is no better than the standard learning algorithms and the gain due to the analysis of the observed traces is lost. Therefore, the exploration should be guided by the learner to provide a guarantee if no new state is revealed.

A guarantee on the conjectured model can be based on the testing theory.

Active learning is like testing of an evolving specification.

The correspondence of testing and learning can be seen in several aspects. Both tasks work with access and separating sequences to identify states. In the basic case when a testing method does not consider extra states, separating sequences are applied after each transition to identify it in the implementation. This is the same for all learning algorithms when they try to define each transition and thus construct a completely specified conjectured model. The correspondence of testing and active learning was discussed in [BGJ⁺05] and recently a category-theoretic formalism that establishes formal relations between algorithms for learning and testing was published in [vHSS17]. Nevertheless, learning utilizing the extensive testing theory was not proposed.

The first attempt to use testing in active learning is the recent learning algorithm called in this thesis the Quotient algorithm (Section 16.5). It is inspired by the oldest testing method, the W-method (Section 9.3). The only difference from the oldest learning algorithm, the L* algorithm, is that the Quotient algorithm uses the most suitable learning structure, an *observation tree* (OTree). An observation tree groups observed traces into a prefix tree in the same way as test sequences are grouped in the testing tree. This provides another correspondence between testing and active learning because this chapter demonstrates that an OTree plays a key role in efficient active

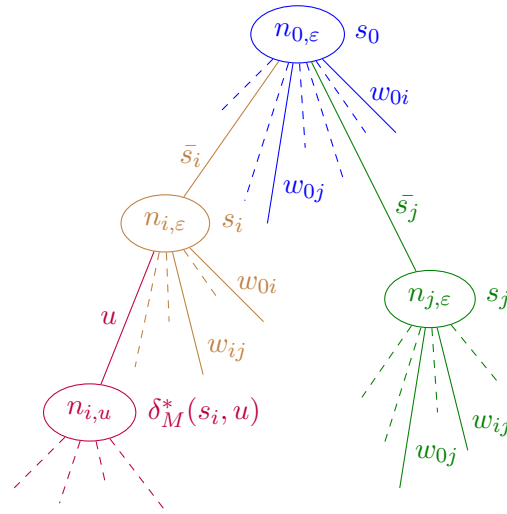


Figure 15.1: A sketch of observation tree with some state-relating parts coloured

learning and Part II showed that the state-of-the-art testing methods use directly the testing tree to construct the smallest test suites. OTree is the most suitable structure to store observed traces because it provides an easy access to all traces and thus eliminates duplicate queries, and observed traces can be easily analysed, for example, to find out which separating sequences are queried from particular states. Moreover, it is like a machine of the same type as the black box and the conjectured model. Hence, the correspondence between an OTree and the two models is obvious as all handle inputs in the same way. As observation trees are in fact equal to the testing trees, an example of observation tree can be found in Figure 10.1 or Figure 11.1. A general structure of OTree is sketched in Figure 15.1. Subtrees relating to states s_0 , s_i and s_j together with their fixed access sequences are highlighted with different colours to show which sequences identify particular states of the black box. All three subtrees contain common extensions w_{0i} , w_{0j} and w_{ij} that indicate separating sequences (without the corresponding outputs that are different). The state reached by $\bar{s}_i u$ from the initial state is also depicted in Figure 15.1 to show how nodes of OTree are labelled. If a node represents an identified state of the black box, it is called a *state node* (SN) and is denoted $n_{i,\epsilon}$ where i corresponds to the related state s_i in the conjectured model and the fixed access sequence \bar{s}_i . Otherwise, the node is reached by an extension u from an SN $n_{i,\epsilon}$, therefore, it is denoted $n_{i,u}$. Note that for each node $n_{i,u}$ only the closest SN is considered; there is no SN $n_{j,\epsilon}$ such that $\bar{s}_j = \bar{s}_i \cdot v'$ and $u = v' \cdot v$. Any node reached by v from the initial state can be alternatively denoted n_v , that is, $n_{i,u} = n_{\bar{s}_i u}$.

How the testing theory can help to learn more efficiently and achieve the given goal? The answer is simple: use a sufficient condition for an m -complete test suite. For example, the S-condition (Theorem 8.14). First, it states what one needs to observed to construct a completely specified conjectured model M , that is, the observation tree needs to include an n -complete test suite for

M . Second, it can direct the exploration and give a guarantee after responses to all sequences of an $(n + l)$ -complete test suite for M were observed where l is the number of considered extra states. The guarantee would be in this case that the reduced conjectured model with n states is either a correct representation of the black box or the black box has more than $n + l$ states. There is just one obstacle. In the first phase when no extra state is considered, a testing method can construct separating sequences based on the completely specified specification and knows the target state of each transition. This is missing in the learning. Fortunately, this does not limit the learning because there is at least one separating sequence for each state pair, otherwise the states would not be revealed if they were not distinguished from the others. In the case of undefined transitions, these separating sequences can be gradually queried until the target state is identified as one of the states. This idea is not new. The GoodSplit algorithm (Section 16.6) also queries observed separating sequences to identify target states of each transition. This process is called making the observation tree *closed* (for 0 extra states). As indicated, the term can be extended to work with extra states. The observation tree is *closed* for l extra states if it contains an $(n + l)$ -complete test suite for the conjectured machine M with n states. From this perspective, all the standard learning algorithms always make their learning structure closed for 0 extra states and then ask an equivalence query.

The idea of extra states in active learning is not completely new. A theoretic framework called an *observation pack* (OP) was proposed in 1996 [BDGW96]. It basically describes what is captured in Figure 15.1. There are components for each state such that these components groups sequences with the common prefix of the related access sequence. It was shown that both the L* algorithm and the Discrimination tree algorithm implement the framework and that the number of equivalence queries can be reduced by checking states reached by extensions of all access sequences such that these suffixes have length up to l . It comes with the price of increasing the number of membership queries (the concern was just regular languages and DFA). As all extensions of length up to l are checked, the complexity of the number of MQs contains p^l where p is the number of inputs. If l is chosen to be $\log n$, then there is a polynomial-time learning algorithm that needs $O(\frac{n}{\log n})$ equivalence queries to learn a DFA. Further lower and upper bounds on the number of EQs and MQs were derived in [BDGW94b]. Nevertheless, those bounds are based on the worst case scenarios that are hardly possible if at all. As will be shown in Chapter 17, a clever learner using just one extra state needs a very few EQs (if any) to learn a model of real systems. The L* algorithm also does not need so many equivalence queries even though it does not consider any extra state. It is because L* queries many sequences (that are not necessary) to construct a complete model but the sequences reveal new states unintentionally. The concept of observation pack with a ‘lookahead’ was not implemented to show its performance experimentally but there is an important difference from the observation tree approach. The OP was defined to compare reached states using fixed sets of separating sequences. Therefore, its idea corresponds to

the HSI-method (Section 9.5). The observation tree approach is more general as it allows one to follow any sufficient condition for an m -complete test suite. It also means that if there will be a better sufficient condition for an m -complete test suite than the S-condition, it can be directly used to improve an active-learning algorithm. Note that the authors of OP also described a possible parallelization of the L* algorithm [BDGW94a].

15.1 General Idea

This section summarizes the observation tree approach and discusses aspects in which the approach differs from the standard active-learning setting.

The observation tree approach consists of two phases. The first phase corresponds to the work of the standard learning algorithms, that is, it constructs a completely specified conjectured model by asking output (or membership) queries. The second phase explores the black box in order to reveal other states. Both phases aim to meet a sufficient condition for an $(n + l)$ -complete test suite for the conjectured model M with n states and l as the number of considered extra states. In the first phase, l equals to 0 and it is greater than 0 in the second phase. Both phases are repeated several times during the learning. The second phase always starts right after the first one constructs a complete conjectured model M . Whenever an inconsistency between the observation tree T and M appears in the second phase, it stops and the learning continues by the first phase. An inconsistency means that there is a sequence u with the observed response z that differs from the expected output given by λ_M , that is, $T(\varepsilon, u) = z \neq \lambda_M^*(s_0, u)$. The second phase also stops if the observation tree is closed for the given maximal number of extra states and either there is no counterexample returned on an equivalence query or EQs are not allowed.

Why would EQs not be allowed? Equivalence query is an abstract construct that requires detailed knowledge about the black box to check if the given conjectured model differs. Such a perfect oracle answering any EQ is not available in practice. Therefore, the teacher answering EQs is usually approximated by a testing method or a random walk that tries to find a counterexample. Hence, the user may want the learning algorithm to learn completely itself and so equivalence queries do not have to be implemented by the provided teacher; in fact, then there is no ‘teacher’ as output queries are just responses of the black box and thus the teacher does not teach anything. Another case is that the user can be just satisfied with the guarantee provided by the observation tree approach that the resulting conjectured model is correct with respect to the given maximal number of extra states. Note that any number l of considered extra states cannot guarantee the equivalence of the conjectured model and the black box because there always can be a DFSM that passes $(n + l)$ -complete test suite but has more than $n + l$ states [Moo56]. Therefore, the concept of extra states cannot substitute equivalence queries.

How does the observation tree approach differ from the standard learning algorithms if the equivalence queries are approximated by a testing method? There are conceptual and practical differences. On the one hand, the observation tree approach does not eliminate the concept of a teacher but in this particular case, it shifted the workload from the teacher to the learner. It means that the number of EQs is lowered and they can be implemented by a different way than by a testing method; an expert as a human teacher would also be happy to analyse the model fewer times. On the other hand, the observation tree approach does not require a teacher as was mentioned in the previous paragraph. If the teacher approximates the equivalence queries by a testing method, it queries several test sequences before it finds a counterexample, however, these queries are not available to the learner. The learner then can query completely different sequences to gain the same knowledge that the learner would get out of the test sequences used by the teacher. Moreover in this case, both the learner and the teacher should be considered as one learner because both interact with the black box to learn its model. It is thus reasonable that they should share the observed responses to minimize the interaction with the black box and not doing the same thing twice. A teacher is for the observation tree approach the last (optional) check that the black box has or has not more that $n + l$ states.

The observation tree approach is specified in Algorithm 31 that describes a general structure of a learner based on the approach.

Both phases of the observation tree approach aim to meet the same sufficient condition which means satisfying conditions for each transition. Therefore, the phases are wrapped in a cycle that processes each unverified transition (lines 3–14 of Algorithm 31). An inconsistency can occur in both phases (lines 9–14) but after the inconsistency is resolved, the learner always enters the first phase; the number of considered extra states l gets 0 on line 13. Note that any inconsistency results in the increase of the number of states in the conjectured model. After the observation tree is made closed for the given *maxExtraStates*, the stop conditions are checked. If no counterexample is obtained, the conjectured model with the guarantee of its correctness is returned and the learning stops.

15.2 Common Properties and Optimizations

The next three sections propose new learning algorithms that implement the observation tree approach but aim to satisfy different sufficient conditions for an m -complete test suite. In particular, the *H-learner* in Section 15.3 is based on the H-condition (Theorem 8.12), the *SPY-learner* in Section 15.4 on the SPY-condition (Theorem 8.13), and the *S-learner* in Section 15.5 on the S-condition (Theorem 8.14). All three learners have several common properties and optimizations. These are described in this section.

All three learners work with the observation tree (OTree) as the main learning structure. As shown in Figure 15.1, a node of OTree is denoted $n_{i,u}$ where u is the sequence labelling the path from the closest predecessor

Algorithm 31: Learner based on the observation tree approach

input : A teacher providing information about the black box through output and equivalence queries

input : $maxExtraStates$ as the maximal number of extra states to be considered during learning

output : A conjectured model M

```

1 repeat
2   for  $l \leftarrow 0$  to  $maxExtraStates$  do
3     // make the observation tree  $T$  closed for  $l$  extra states
4     while there is an unverified transition do
5        $(s, x) \leftarrow$  choose an unverified transition
6       if  $l = 0$  then //  $t$  is not defined in  $M$ 
7         | identify state  $\delta(s, x)$  using adaptive separating sequences
8       else
9         | verify transition  $(s, x)$  using testing theory,  $l$  extra states
10      if observed responses and outputs of  $M$  differ then
11        RESOLVEINCONSISTENCY( $T, M$ ):
12          | query appropriate sequences that reveal a new state
13          | update the conjectured model  $M$ 
14       $l \leftarrow 0$  // assume 0 extra states again as
15      break // some transitions are not defined in  $M$ 
16   if equivalence query returns a counterexample  $w$  then
17     | query  $w$  and RESOLVEINCONSISTENCY( $T, M$ )
18 until  $M$  is correct or EQs are not allowed or the user is satisfied with  $M$ 

```

$n_{i,\varepsilon}$ that is a *state node* (SN) corresponding to the state s_i . For an easier description, let $E_{i,u}$ denote the subtree with node $n_{i,u}$ as its root, that is, $E_{i,u} = \{w \mid \bar{s}_i u w \in T\}$. The subtrees $E_{i,u}$ called distinguishing sets are repeatedly compared in all learners in order to find separating sequences and identify the correspondence between nodes and an SN. To reduce the number of such comparisons and to keep track during the identification of nodes, the learners employ domains as were introduced in Section 8.2. A domain ϕ is implemented as a set tied to every node of OTree rather than a function of sequences. For each node $n_{i,u}$, the domain ϕ is defined as $\phi(\bar{s}_i u) = \{s_j \in S \mid \neg \exists w \in E_{i,u} \cap E_{j,\varepsilon} : T(\bar{s}_i \cdot u, w) \neq T(\bar{s}_j, w)\}$. A node $n_{i,u}$ is called *consistent* with a state s_j , or equivalently with a state node $n_{j,\varepsilon}$, if s_j is in its domain, that is, they are not separated by an observed common extension. Note that the terms ‘separating’ and ‘be separated’ are extended to nodes of OTree in the meaning of the states that they represent.

Section 8.2 introduced the domain function Φ of classes of convergent sequences besides the domain function ϕ . The SPY-condition and the S-condition use the convergence of test sequences that allows one to construct smaller test suites. Hence, both the SPY-learner and the S-learner employ

a convergent graph as the secondary learning structure and domains of convergent nodes as well. A convergent node (CN) denoted as $[u]$ is a node of the convergent graph such that it groups sequences that are proven to be convergent (with u). Initially, each convergent node $[u]$ relates only to u , therefore, the convergent graph corresponds to the observation tree. Instead of grouping sequences, CNs are implemented to store the related nodes of OTree, that is, a CN $[u]$ contains at least node n_u . A convergent node that contains a state node is called a *reference node* (RN); a RN $[\bar{s}_i]$ contains the SN $n_{i,\varepsilon}$ relating to \bar{s}_i . Similarly to the implementation of ϕ , Φ is not implemented as a function but as a set tied to particular convergent node. Such a set does not contain states but references to CNs. The content of Φ of a CN depends on whether the CN is an RN or not as follows:

- If a CN is an RN $[\bar{s}_i]$, then Φ of $[\bar{s}_i]$ includes all CNs that are consistent with $[\bar{s}_i]$.
- If a CN $[u]$ is not an RN, then its Φ contains references to RNs that are consistent with $[u]$.

Similarly to the nodes of OTree, convergent nodes $[u], [v]$ are consistent if there is no sequence w such that $T(u', w) \neq T(v', w)$ for all $u' \in [u]$ and $v' \in [v]$. Note that the same is used in the testing S-method. This definition of Φ helps to reach and update all CNs that can be affected by the last query. For example, if a CN $[u]$ is in Φ of an RN $[\bar{s}_i]$ and the last queried sequence w extends $E_{i,\varepsilon}$, then the CN $[u]$ is checked if it is separated from $[\bar{s}_i]$ by the corresponding suffix of w . If $[u]$ and $[\bar{s}_i]$ become separated, they are removed from the domain of each other.

One part of the goal is to minimize both the number of symbols queried during the learning and the number of resets of the black box. It is reflected by the way how the observation tree is extended. The observation tree approach focuses on reactive system that are usually modelled by DFSMs or Mealy machines rather than on regular languages represented by DFA. Therefore, it benefits from the possibility to query a sequence u one by one symbol using the output query $T(u)$ that does not reset the black box. There are two advantages. First, if an inconsistency is observed in the middle of a requested query, the rest of the sequence does not have to be applied to the black box and some interaction is saved. Second, the learner can extend the last queried sequence and so save a reset of the black box. All three new learners benefit from both advantages. It means that they analyse the traces after the response to (almost) each symbol is observed and thus they can choose a suitable symbol to query next and whether the black box is to be reset first. The sufficient conditions that the learners aim to meet ask for the verification of each transition. Hence, after the learner queries a sequence u to verify a particular transition, it then checks if an extension of u can help to verify another transition. Algorithm 32 describes how the observation tree is extended with one node. The learners remember the current state of the black box by the corresponding node *nodeBB* of the observation tree. Therefore, they know if the black box needs to be reset before input x is queried from node $n_{i,u}$, that is, x extends sequence $\bar{s}_i u$. Note that the domain

$\phi(\bar{s}_i ux)$ is initialized with all states that have the same state output as the reached state because there is no sequence from $n_{i,ux}$ except the stout input \uparrow (line 11 of Algorithm 32). The variable $nodeBB$ points to the root of OTree when the learning starts as the black box is reset into its initial state.

Algorithm 32: QUERY($n_{i,u}, x$)

```

1 Let  $y_t$  be the output of the incoming transition and  $y_s$  be the state
  output of the successor node  $n_{i,ux}$ 
2 if OQs are allowed and BB is DFMSM then
3   |  $(y_t, y_s) \leftarrow (nodeBB = n_{i,u}) ? T(x \uparrow) : T(\bar{s}_i u, x \uparrow)$ 
4 else
5   |  $y_t \leftarrow (nodeBB = n_{i,u}) ? T(x) : T(\bar{s}_i u, x)$ 
6   | if BB is DFMSM then
7     |  $y_s \leftarrow T(\uparrow)$ 
8   | else if BB is DFA or Moore then
9     |  $y_s \leftarrow y_t$ 
10  | else  $y_s \leftarrow \varepsilon$  // BB is Mealy
11  $\phi(\bar{s}_i ux) \leftarrow$  all states with the same state output  $y_s$ 
12  $nodeBB \leftarrow n_{i,ux}$ 

```

There are two other aspects that are common for all three learners. The first one is a so-called adaptive separating sequence that is used to identify the target state of a transition in the first phase of the observation tree approach. The second aspect is the inconsistency between the observation tree T and the conjectured model M . Both aspects are introduced in the following two subsections.

15.2.1 Adaptive Separating Sequences

The target state of a transition needs to be identified when the transition is to be defined. Node $n_{i,x}$ representing the target state of transition (s_i, x) is identified if its domain Φ is singleton because then it is consistent with a single reference node. If the convergence of sequences is not considered, such as by the H-learner, Φ corresponds to ϕ . How to make domain Φ (or ϕ) of a node $n_{i,u}$ singleton? It is sufficient to query the observed separating sequences of RNs that are in $\Phi([\bar{s}_i u])$ from $[\bar{s}_i u]$. In the case of the H-learner, it reduces to querying the observed separating sequences of SNs related to states in $\phi(\bar{s}_i u)$ from $n_{i,u}$.

The choice of separating sequences can be optimized to minimize the interaction with the black box. The best case would be if particular separating sequences were grouped into an adaptive distinguishing sequence (ADS). Then, only one sequence would be necessary to identify any reached state. However, a lot of machines do not have an ADS or the state verifying sequences of ADS are not queried from the related reference nodes. Therefore, the aim is to choose the best so-called *adaptive separating sequence* that is like an ADS but it does not have to separate all the given RNs. In fact, an adaptive separating

sequence is like incomplete adaptive distinguishing sequence (IADS), see Section 3.3.4. Both sequences try to distinguish the most states of a subset of all states but an IADS is constructed based on the completely specified specification and an adaptive separating sequence considers only extensions of the given RNs captured in the observation tree.

An adaptive separating sequence can be represented as a successor tree with nodes labelled with input symbols and edges by outputs. For simplicity, consider first the case without the convergence of sequences, that is, an adaptive separating sequence is to be constructed to distinguish the most states of $\phi(\bar{s}_i u)$. Each node of the adaptive separating sequence is thus tied to a set of states that are consistent under the separating sequence leading to the node. It means that the root of adaptive separating sequence is tied with the domain $\phi(\bar{s}_i u)$. The set of states in any other node then represents the domain $\phi(\bar{s}_i u)$ after the separating sequence leading to the node is queried and corresponding outputs are observed.

Figure 15.2 shows two adaptive separating sequences. Let the domain $\phi(\bar{s}_i u)$ of a node $n_{i,u}$ contain five states A–E of a Mealy machine. There are only two input symbols, ‘a’ and ‘b’, and three outputs, 0–2. States A and D respond to ‘a’ with unique outputs, 1 and 2 respectively. Therefore, they are separated by ‘a’ from the other states. States B, C and E output 0 on ‘a’ but there is input ‘b’ that separates their next states. Unfortunately, the sequence ‘ab’ from $n_{B,\varepsilon}$ is not observed in the OTree yet. Therefore, B remains in the domain $\phi(\bar{s}_i u)$ regardless of the output on ‘b’. Thus, $n_{i,u}$ would relate to either E or B if $n_{i,u}$ produced 0 on ‘a’ and then $n_{i,u,a}$ responded with 1 to ‘b’. The second adaptive separating sequence captures the scenario if the first input is ‘b’. Both states D and E respond to ‘b’ with 2. There is no observed separating sequence for their next states, therefore, another sequence would need to be queried from $n_{i,u}$ to identify the node if the node produced 2 on ‘b’. Notice that an input does not have to separate the related set of states immediately. The purpose of ‘a’ after ‘b’ produced 0 is the transfer of the next states of A, B and C to the states that can be separated by another ‘a’.

A *distinguishing score* is introduced to compare adaptive separating sequences. The score represents the probability that the node will not be fully identified and how many states will not be separated. Therefore, the lower the score, the better. It assumes that the node really corresponds to a state of its domain. This means that the score does not consider extra states. The score is calculated for each node of an adaptive separating sequence as follows:

$$score_i = \begin{cases} \frac{\#states_i - 1}{\#states_{parent} - 1} & i \text{ is a leaf node,} \\ \sum_{j \in successors} \frac{1}{\#successors} \cdot score_j & \text{otherwise.} \end{cases}$$

The score of the root then represents the distinguishing score of the sequence. If the distinguishing score is the same for more sequences, the shortest is selected. The length of an adaptive separating sequence is $\sum_{i \in leaves} \#states_i \cdot |w_i|$, where w_i is the separating sequence leading to the leaf i .

The distinguishing score depends only on leaves with more than one state as can be seen in Figure 15.2. Only the branch on 0 of the first adaptive

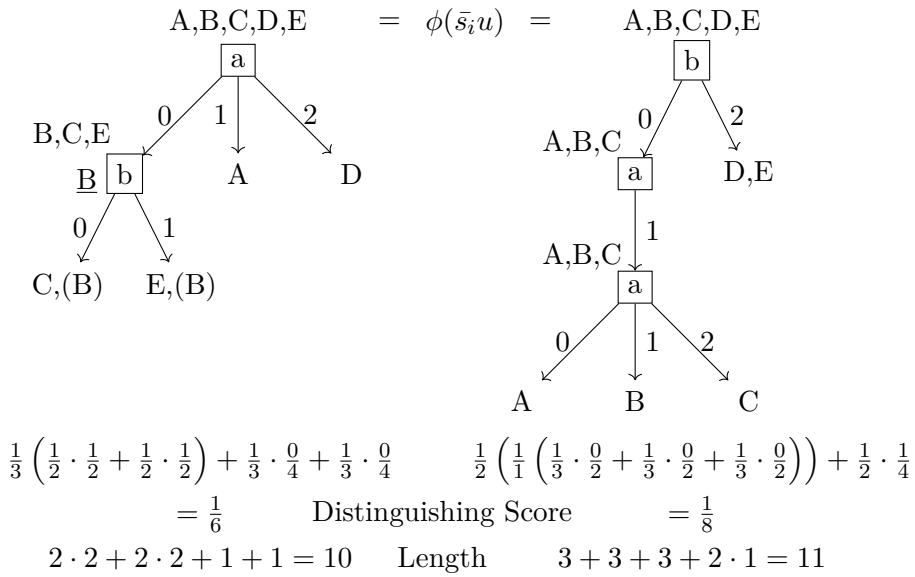


Figure 15.2: Calculating distinguishing score of adaptive separating sequences

separating sequence contributes to the score. Both leaves have two states and three states (B, C, E) relate to their parent. Therefore, the score of the leaves is the same, $\frac{2-1}{3-1} = \frac{1}{2}$. Both leaves are equally probable so the score of their parent is $\frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}$. As there are three successors of the root, the calculated score is divided by 3 and the distinguishing score is thus $\frac{1}{6}$. It is similar for the second adaptive separating sequence in Figure 15.2 where only the second branch from the root matters. Notice that the second sequence is better even if it is longer.

Adaptive separating sequences are implemented to work with nodes of OTree instead of states in the sets tied to each node. If the convergence of sequences is not employed, each state in the set tied to a node of adaptive separating sequence is represented by at most one node of OTree. If a node of OTree has no successor on a particular input, then the number of undistinguished states is increased; an example is captured on the left adaptive separating sequence in Figure 15.2 where the state B is in both lowest leaves because the OTree node $n_{B,a}$ has no successor on ‘b’. The number of undistinguished states is then added to the number of OTree nodes in the set tied to a node of adaptive separating sequence which gives the total number of states used to calculate the distinguishing score. In the case of the H-learner (no convergence of sequences), the root of an adaptive separating sequence is thus tied with a set of state nodes that correspond to the states in $\phi(\bar{s}_i u)$. If the convergence of sequences is employed, each state in the set tied to a node of adaptive separating sequence can be represented by several nodes of OTree. This reflects that the root is tied with $\Phi([\bar{s}_i u])$ which is a set of reference nodes and each RN as well as each CN consists of the nodes of OTree that are convergent. Adaptive separating sequences do not work directly with the CNs because the convergent graph can contain a cycle

and the chosen separating sequence would not have to be queried from the corresponding RNs.

The learners choose the best adaptive separating sequences by a recursive function. The function compares scores for all inputs that can be applied to the given set of nodes (or set of set of nodes). The root is initialized by the related state nodes (or the reference nodes). The function then calls itself with the corresponding successor nodes of OTree according to a chosen input. The distinguishing score is accumulated and whenever a local score is higher than the best score found so far, the search for a better input is pruned. A local score is employed for the comparison of different inputs and the related subtrees of OTree.

There is a small optimization related to the choice of separating sequence. If the target state of transition (s_i, x) is to be identified (no extra state is considered), it can happen that the state output uniquely identifies the target state amongst the states of the conjectured model. There is no need to query any other symbol from the reached node. However, it can help to reveal a new state, especially when the learning starts and the next states of the initial state are not distinguished from the initial state. Let $a \in X$ be the very first queried input and the reached state $\delta_M(s_0, a)$ cannot be distinguished from s_0 , that is, the state outputs are the same (or the black box is a Mealy machine). Then, a is the most likely input that reveals new information. In the case of DFA and Moore machine that have outputs tied to states, there is no difference between inputs. However, transition outputs can be compared for DFSA and Mealy machine only if the input symbols are the same. Therefore, the learners choose the input x which was applied the last. Repeating a thus allows one to compare the initial state with the next state $\delta_M(s_0, a)$ by the outputs of the transitions on a and the outputs of reached states. Consider that a did not distinguish $\delta_M(s_0, a)$ from s_0 , that is, $T(\varepsilon, a) = T(a, a)$. The next input is queried from the initial state. Let b be the second input. Then, there are two most likely inputs that could reveal new information. Both a and b applied in $\delta_M(s_0, b)$ could distinguish the state from the initial one. Nevertheless, a had its opportunity to show whether it is a separating input. In addition, if b appears to be separating, then it can identify the next state $\delta_M^*(s_0, b \cdot b)$ by querying b again. The learners follow the mentioned rule that if the target state is identified directly by its state output then the last input is repeated. In the example, b would be queried again, that is, $T(b, b)$.

■ 15.2.2 Types of Inconsistency

There are four basic types of inconsistency of the observation tree T and the conjectured model M .

- I *Empty domain ϕ* : A node $n_{i,u}$ with $\phi(\bar{s}_i u) = \emptyset$ represents a state of the black box that is not modelled in the conjectured model M .
- II *Inconsistent domain ϕ* : A node $n_{i,u}$ is identified as a state s_j , that is, $\delta_M^*(s_i, u) = s_j$, but then it is separated from $n_{j,\varepsilon}$ and so $s_j \notin \phi(\bar{s}_i u)$.

- III *Empty CN domain* Φ : A convergent node $[\bar{s}_i u]$ is separated from all reference nodes so that $\Phi([\bar{s}_i u]) = \emptyset$. There is a CN $[v]$ that groups sequences that are divergent in the black box.
- IV *Wrong merge* of CNs: In the first phase when 0 extra states are considered and some transitions are not defined, a convergent node $[u]$ can be identified as a state s_j but during merging it into the RN $[\bar{s}_j]$ it is found out that $[u]$ and $[\bar{s}_j]$ are divergent.

Any inconsistency indicates that the black box has more states than the conjectured model. Therefore, the learners ask additional queries to observe the inconsistency I that reveals a node $n_{i,u}$ representing a new state. It is important to find such node $n_{i,u}$ that is a successor of a state node, that is, u is just an input x . Then, the new state can be easily connected to the current states of the conjectured model, in particular a transition (s_i, x) is defined to lead to the new state s_j . It means that sometimes it is not sufficient to observe the inconsistency I and the inconsistency II needs to be resolved first to reveal another instance of inconsistency I that meets the mentioned criterion. Note that the types of inconsistencies are numbered by their priority and the learners resolve first the inconsistency with a higher priority (a lower Roman numeral).

The inconsistency II of a node $n_{i,u}$ is easy to resolve if there is a predecessor $n_{i,v}$, $u = v \cdot v'$, that is identified as a state s_k , that is, $\delta_M^*(s_i, v) = s_k$. See an illustrating sketch in Figure 15.3. Let $n_{i,u}$ be identified as s_j and w be a separating sequence of $n_{i,u}$ and $n_{j,\varepsilon}$. Then, the query $\bar{s}_k v' w$ reveals an inconsistency of either $n_{i,v}$ if $T(\bar{s}_k, v' w) \neq T(\bar{s}_i v, v' w)$, or $n_{k,v'}$ if $T(\bar{s}_k v', w) \neq T(\bar{s}_j, w)$. As both v, v' are shorter than u , the inconsistency is revealed closer to a state node. Therefore, resolving the inconsistency II results in the inconsistency I of a node $n_{i,x}$ for an input x unless the convergence of sequence is considered and there is no such predecessor $n_{i,v}$. It can happen that the closest predecessor that is identified as a state is the state node $n_{i,\varepsilon}$. It is only possible if $n_{i,u}$ was identified based on its Φ . If $n_{i,u}$ was identified using its ϕ and it was separated from the identified state, then the domain ϕ would be empty and so the inconsistency I would be resolved first. The case when the closest identified predecessor is the state node is resolved like the inconsistency III because $\Phi([\bar{s}_i u])$ is empty.

Figure 15.4 shows a part of reasoning behind resolving the inconsistency III. An empty domain $\Phi([\bar{s}_i u])$ means that for each RN $[\bar{s}_k]$ there is a separating sequence w from a node in $[\bar{s}_i u]$ that separates $[\bar{s}_k]$ and $[\bar{s}_i u]$. These sequences are used to reveal the inconsistency II or even the inconsistency I. First, similarly to resolving the inconsistency II, the closest identified predecessor is found. Let $[\bar{s}_i]$ be the RN of the closest identified predecessor and consider $n_{i,u}$ as the witness of the empty CN domain. Note that $n_{i,u}$ does not have to exist, hence, u would be queried from $n_{i,\varepsilon}$. For each state s_k in $\phi(\bar{s}_i u)$ a separating sequence w as mentioned is queried both from $n_{i,u}$ and $n_{k,\varepsilon}$. There are three cases that can happen with respect to the observed responses to the separating sequence. The first one is that all states are eliminated from $\phi(\bar{s}_i u)$

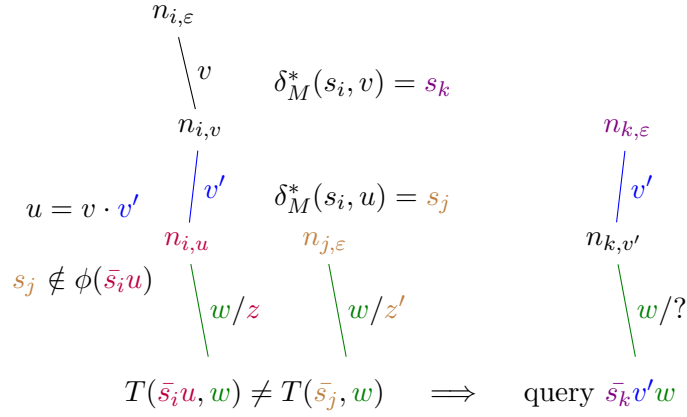


Figure 15.3: Resolving inconsistency II: inconsistent state domain of node $n_{i,u}$

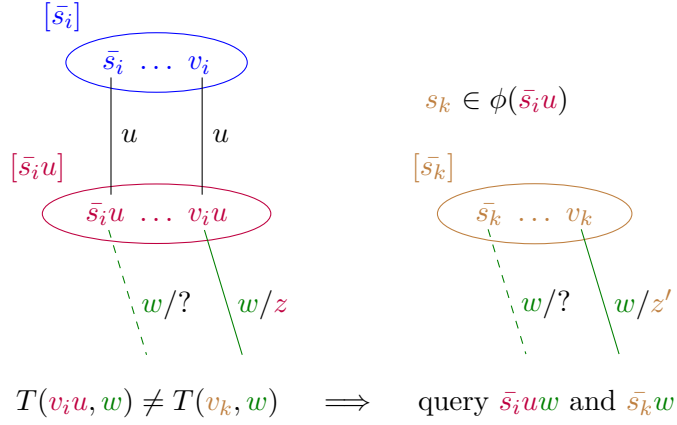


Figure 15.4: Resolving inconsistency III: empty domain of convergent node $[\bar{s}_i u]$

as $T(\bar{s}_i u, w) \neq T(\bar{s}_k, w)$. It means the inconsistency I is observed in $n_{i,u}$. The second and third cases result in the inconsistency II because the separating sequence w reveals that $[\bar{s}_i]$ or $[\bar{s}_k]$ contain divergent sequences, that is, nodes corresponding to different states of the black box. In particular as Figure 15.4 sketches, if $T(\bar{s}_i u, w) \neq T(v_i u, w)$, then $s_i \notin \phi(v_i)$, and if $T(\bar{s}_k, w) \neq T(v_k, w)$, then $s_k \notin \phi(v_k)$.

The inconsistency IV happens when the domain $\Phi([u])$ is $\{[\bar{s}_j]\}$ so that $[u]$ is to be merged into $[\bar{s}_j]$, and the following hold. There is a cycle (or the self-loop of a state) in the ‘subtrees’ of $[u]$ and $[\bar{s}_j]$ such that a CN $[uv']$ is identified as a state s_j in the first round and in the next round its successor $[uv'v']$ should be merged into the same RN $[\bar{s}_j]$ but they are separated by w . Note that there can be several cycles of v' . Let v be the sequence that leads to the separated CNs that cannot be merged and v'' is a suffix of v without the cycles v' . It is sufficient to query vw from $n_{j,\epsilon}$ and n_u and in some cases $v''w$ from $n_{j,\epsilon}$ to separate $[u]$ from $[\bar{s}_j]$ and so reveal the inconsistency III of $[u]$.

Resolving inconsistencies II, III and IV is formalized in Algorithm 33 that is used by the SPY- and S- learners. The H-learner does not consider the convergence of sequences, therefore, it has much easier work as only the inconsistencies I and II can be observed. Each node $n_{i,u}$ of OTree has in the implementation the attribute *state* that represents the corresponding state if it is known, that is, if the node was identified.

Algorithm 33: RESOLVEINCONSISTENCY(T, M)

```

1 let  $n_{i,u}$  be a witness of inconsistency
2 if  $n_{i,u}.state$  is set and  $n_{i,u}.state \notin \phi(\bar{s}_i u)$  then // Incons. domain
3   let  $n_{i,u'}$  be the closest predecessor of  $n_{i,u}$  such that  $n_{i,u'}.state$  is set
   and is in  $\phi(\bar{s}_i u')$ , and let  $u = u' \cdot v$  and  $s_k = n_{i,u}.state$ 
4    $n_{i,u}.state \leftarrow \text{null}$ 
5   if  $n_{i,u'}$  is an SN, that is,  $u' = \varepsilon$  and  $u = v$  then
6     foreach  $s_j \in \phi(\bar{s}_i u)$  do
7       if there is  $v' \in [\bar{s}_i]$  such that  $v'v \in T$  and a sequence  $w$ 
       separates  $n_{v'v}$  from  $[\bar{s}_j]$  then
8         if  $\bar{s}_j w \notin T$  then QUERYSEQANDCHECK( $n_{j,\varepsilon}, w$ )
9         if  $\bar{s}_i u w \notin T$  then QUERYSEQANDCHECK( $n_{i,u}, w$ )
10    else
11       $w \leftarrow$  separating sequence of  $n_{i,u}$  and  $n_{k,\varepsilon}$ 
12      let  $s_j$  be  $n_{i,u'}.state$ 
13      if  $\bar{s}_j v w \notin T$  then QUERYSEQANDCHECK( $n_{j,\varepsilon}, v w$ )
14  else if  $\Phi([\bar{s}_i u])$  is empty then // Empty CN domain
15    let  $n_{i,u'}$  be the closest predecessor of  $n_{i,u}$  such that  $n_{i,u'}.state$  is set
    and is in  $\phi(\bar{s}_i u')$ , and let  $u = u' \cdot v$  and  $s_{k'} = n_{i,u'}.state$ 
16     $n_{i,u}.state \leftarrow \text{null}$ 
17    if  $n_{i,u'}$  is not an SN, that is,  $u' \neq \varepsilon$  then
18      if  $\bar{s}_{k'} v \notin T$  then QUERYSEQANDCHECK( $n_{k',\varepsilon}, v$ )
19      consider  $n_{k',v}$  instead of  $n_{i,u}$  in the following
20    foreach  $s_j \in \phi(\bar{s}_i u)$  do
21      if there is a separating sequence  $w$  of  $[\bar{s}_i u]$  and  $[\bar{s}_j]$  then
22        if  $\bar{s}_i u w \notin T$  then QUERYSEQANDCHECK( $n_{i,u}, w$ )
23        if  $\bar{s}_j w \notin T$  then QUERYSEQANDCHECK( $n_{j,\varepsilon}, w$ )
24  else // Wrong merge
25    let  $[\bar{s}_j]$  be the RN to which  $[\bar{s}_i u]$  should merge
26    let  $v$  be the transfer sequence that revealed the merge to be wrong
27     $w \leftarrow$  separating sequence of  $[\bar{s}_j v]$  and  $[\bar{s}_i u v]$ 
28    if  $\bar{s}_i u v w \notin T$  then QUERYSEQANDCHECK( $n_{i,u}, v w$ )
29    if  $\bar{s}_j v w \notin T$  then QUERYSEQANDCHECK( $n_{j,\varepsilon}, v w$ )
30    if there is a prefix  $v'$  of  $v$  such that  $[\bar{s}_j v'] = [\bar{s}_j]$  then
31      let  $v'$  be the longest one that loops back to  $[\bar{s}_j]$  and  $v = v' v''$ 
32      if  $\bar{s}_j v'' w \notin T$  then QUERYSEQANDCHECK( $n_{j,\varepsilon}, v'' w$ )

```

15.3 H-learner

The H-learner aims to make the observation tree closed for the specified number of extra states l according to the H-condition (Theorem 8.12). It means that each state node is extended with all sequences of length up to $l + 1$ where l is the currently-considered number of extra states. Each such reached node $n_{i,u}$ needs to be identified, that is, $|\phi(\bar{s}_i u)| = 1$ for all s_i and $|u| \leq l + 1$. Moreover, nodes $n_{i,uv}$ and $n_{i,u}$ such that $|uv| \leq l + 1$ and $\delta_M^*(s_i, uv) \neq \delta_M^*(s_i, u)$ need to be observably separated. The idea of the H-learner is based on the H-method (Section 9.6).

How an adaptive separating sequence is chosen to identify the target state of a transition was described in Section 15.2.1. A similar procedure is also followed in the case of verification of transitions in the second phase of the observation tree approach. The only difference is that the target state is known and thus the adaptive separating sequence is not adaptive because a particular response is expected. The chosen sequence is a state verifying sequence in the best case as then no other sequence needs to be queried to verify the given transition. Note that all transitions covered by sequences of length up to $l + 1$ need to be verified; the chosen separating sequences aim to reduce the domains ϕ of nodes $n_{i,u}$ with $u \leq l + 1$. If the domain does not become a singleton after the chosen separating sequence is queried, another sequence is selected for the undistinguished states in $\phi(\bar{s}_i u)$.

The H-learner makes the observation tree closed for an increasing number l of extra states. For each l it verifies each transition one by one. As each transition is verified for l extra states, in order to make the transition verified for $l + 1$ it is sufficient to reduce the domains of all successors $n_{i,ux}$ where $|u| = l$, to singletons and check that predecessors of $n_{i,ux}$ are separated from $n_{i,ux}$ if they correspond to a different state. Assume that the domains are singletons and there is $n_{i,v}$ that corresponds to a different state than $n_{i,ux}$ and v is a prefix of ux . Let $s_j = \delta^*(s_i, v)$ and $s_k = \delta^*(s_i, ux)$. There is $w_j \in E_{i,v} \cap E_{k,\varepsilon}$ and $w_k \in E_{i,ux} \cap E_{j,\varepsilon}$ such that w_j separates $n_{i,v}$ from $n_{k,\varepsilon}$ and w_k separates $n_{i,ux}$ from $n_{j,\varepsilon}$. If $n_{i,v}$ and $n_{i,ux}$ are not separated in the OTree, then there is not $w \in E_{i,v} \cap E_{i,ux}$ that would distinguish $n_{i,v}$ and $n_{i,ux}$. Therefore, it is sufficient to query either w_j from $n_{i,ux}$ or w_k from $n_{i,v}$. The H-learner chooses the latter option so it queries $\bar{s}_i \cdot v \cdot w_k$. The sequence is usually shorter than $\bar{s}_i \cdot ux \cdot w_j$ and additionally it can help to reduce the domain of a successor.

15.3.1 Implementation

This section provides a more detailed description of the H-learner. The learning algorithm is specified using Algorithms 34–39 and then described on a running example in the following section.

The H-learner maintains three structures: the observation tree T , the conjectured model M and the queue *Unchecked* used to keep track of nodes of the OTree that is to be checked. A node $n_{i,u}$ is *checked* if all its successors

Algorithm 34: H-learner

```

input : A teacher
input : maxExtraStates
output: A conjectured model M

1  $l \leftarrow 0$ 
2 add  $n_{0,\varepsilon}$  that corresponds to the initial state  $s_0$  to the queue Unchecked
3 while Unchecked is not empty do
4    $n_{i,u} \leftarrow$  the first node in unchecked
5   if  $|u| > l$  then
6      $l \leftarrow l + 1$ 
7     if  $l > \text{maxExtraStates}$  then
8       if EQ allowed then
9          $l \leftarrow l - 1$ 
10        if there is a previous CE  $w$  and  $\delta^*(s_0, w) \neq T(w)$  then
11          H-UPDATEWITHCE( $w$ )
12          continue
13        if there is a counterexample  $w$  obtained on EQ then
14          H-APPLYANDCHECK( $n_{0,\varepsilon}, w$ )
15          if no SN was revealed then H-UPDATEWITHCE( $w$ )
16          continue
17        break // stop learning
22 if there is  $x \in X$  that was not queried from  $n_{i,u}$  then
23   QUERY( $n_{i,u}, x$ )
24   if the response reveals a new state then
25     H-UPDATEDOMAINS( $n_{i,ux}$ )
26     H-TRYEXTENDQUERIEDPATH( $n_{i,ux}$ )
27   else H-IDENTIFY( $n_{i,ux}$ )
28 else if there is a successor  $n_{i,ux}$  with  $|\phi(\bar{s}_i ux)| > 1$  then
29   H-IDENTIFY( $n_{i,ux}$ )
30 else if there is  $n_{i,v}$  undistinguished from a successor  $n_{i,ux}$  such that
31    $v$  is a prefix of  $ux$  and  $s_k = \delta^*(s_i, v) \neq \delta^*(s_i, ux)$  then
32      $w \leftarrow$  the shortest separating sequence of  $n_{i,ux}$  and  $n_{k,\varepsilon}$ 
33     H-APPLYANDCHECK( $n_{i,v}, w$ )
34 else
35   add all successors  $n_{i,ux}$  that are not state nodes to Unchecked
36   remove  $n_{i,u}$  from Unchecked
37 if user is satisfied with  $M$  then break
38 return the conjecture  $M$ 

```

are consistent with a single state node, that is, $\phi(\bar{s}_i u \cdot x)$ is singleton for all $x \in X$, and they are separated from their predecessors that correspond to a different state. Algorithm 34 describes the processing of nodes in *Unchecked* and the stop condition. The learning stops if l exceeds *maxExtraStates* and there is no counterexample if EQs are allowed (lines 7–17). If a node $n_{i,u}$ has a successor on each input x (lines 18–23), all domains $\phi(\bar{s}_i u \cdot x)$ are singletons (lines 24–25), and divergent predecessors are separated (lines 26–28), then the node is checked and all its successor are pushed back in *Unchecked* (lines 30–31). *Unchecked* is also changed inside the function H-UPDATEDOMAINS if a new state is revealed. Notice that if EQs are allowed, l is decreased (line 9) so that the algorithm always assumes at most l extra states.

Algorithm 35: H-UPDATEDOMAINS(n_u)

```

1 foreach  $n_v$  such that  $u = v \cdot w$  do
2   if  $n_v$  is an SN, that is,  $n_v = n_{i,\varepsilon}$  then
3     | remove  $s_i$  from the domains of all nodes that are separated by  $w$ 
4   else
5     | remove from  $\phi(v)$  all states which SNs are separated from  $n_v$  by  $w$ 
6 if there is a  $n_{j,v}$  with empty domain then // new state found
7   | add a new state  $s_k$  to the conjecture  $M$ ;  $n_{j,v} = n_{k,\varepsilon}$ 
8   | add  $s_k$  to the domains of all nodes that are not separated from  $n_{k,\varepsilon}$ 
9   | fill Unchecked with all state nodes  $n_{i,\varepsilon}$  sorted by  $|\bar{s}_i|$ ; the shortest first

```

The function H-UPDATEDOMAINS in Algorithm 35 checks all parents up to the root from the given node n_u whether they are separated by the corresponding suffix of u . The node n_u is a leaf as u is the last queried sequences. If a predecessor n_v , $u = v \cdot w$, is a state node $n_{i,\varepsilon}$ (lines 2–3), then all consistent nodes are compared under the related suffix w . Nodes are consistent if there is no common sequence that would separate them. If a consistent node $n_{j,u'}$ is separated by w from $n_{i,\varepsilon}$, s_i is removed from its domain $\phi(\bar{s}_j u')$. Only nodes $n_{j,u'}$ with $|u'| \leq l$ are sufficient to consider as ‘nodes near leaves’ have so sparse distinguishing set $E_{j,u'}$ that they are unlikely to be separated from all state nodes and become new states of the conjectured model. If the predecessor n_v is not a state node and w separates it from a state node $n_{j,\varepsilon}$ in $\phi(v)$ (lines 4–5), then s_j is removed from the domain.

A state s_k is added to the conjectured model if a domain becomes empty (lines 6–9 of Algorithm 35). All nodes of OTree are then compared with the new state node $n_{k,\varepsilon}$. If a node $n_{j,u'}$ is consistent with $n_{k,\varepsilon}$, s_k enlarges its domain $\phi(\bar{s}_j u')$. The detection of a new state invalidates transitions of the conjectured model so that all previously checked nodes need to be checked again. Therefore, the queue *Unchecked* is filled up with the state nodes only. The state nodes are sorted by the length of their access sequences for the reason of efficiency. The closer to the root a new state node is revealed, the less amount of queried symbols is spent.

The H-learner usually needs to apply a sequence of several inputs to obtain intended information. Therefore, Algorithm 36 proposes the function H-APPLYANDCHECK that repeats the function QUERY (Algorithm 32) for all inputs of the given sequence w and then checks for any changes. Not all inputs are applied if the observed response is unexpected so a new state is revealed. In such a case, the function stops querying and calls H-UPDATEDOMAINS that identifies the new state node. Then, the learning continues from the reached node $n_{i,u}$ by the call of H-TRYEXTENDQUERIEDPATH.

Algorithm 36: H-APPLYANDCHECK($n_{i,u}, w$)

```

1 for  $j \leftarrow 1$  to  $|w|$ ;  $w = x_1 \cdot \dots \cdot x_{|w|}$  do
2   if  $\bar{s}_i u x_j \notin T$  then QUERY( $n_{i,u}, x_j$ )
3   consider  $n_{i,u x_j}$  instead of  $n_{i,u}$ 
4   if an unexpected response is observed then break
5 H-UPDATEDOMAINS( $n_{i,u}$ )
6 H-TRYEXTENDQUERIEDPATH( $n_{i,u}$ )

```

The function H-IDENTIFY that selects which sequence is to be applied next is described in Algorithm 37. If the given node $n_{i,u \cdot x}$ is already consistent with just one state node, then the incoming input x is taken as the next queried sequence (lines 1–2). This choice was discussed at the end of Section 15.2.1. There are two options if the domain $\phi(\bar{s}_i u x)$ is not singleton. The reached state is to be either verified or identified. If the state is known, that is, $\delta_M^*(s_i, u \cdot x)$ is defined, then verification by a separating sequence happens (lines 3–4). Otherwise, the state is to be identified using adaptive separating sequences (lines 5–6). Note that a necessary condition for verification is $|u| > 0$ because the transition function δ is defined by identification of next states of state nodes ($u = \varepsilon$). The choice of adaptive separating sequences was described in Section 15.2.1. The separating sequence w for the purpose of verification is chosen from the distinguishing set $E_{j,\varepsilon}$ of the related state s_j . It separates the most state nodes of states in $\phi(\bar{s}_i u x)$ from $n_{j,\varepsilon}$ and is the shortest among such sequences. It means that w or its prefix is also in $E_{k,\varepsilon}$ for $s_k \in \phi(\bar{s}_i u x)$, the responses from $n_{j,\varepsilon}$ and $n_{k,\varepsilon}$ are different, and this holds for as many states s_k of $\phi(\bar{s}_i u x)$ as possible. After a sequence is selected, H-APPLYANDCHECK is called in order to query the chosen sequence w .

When a selected sequence is applied and domains are reduced accordingly, the algorithm tries to continue with querying from the reached leaf node. There is a function H-TRYEXTENDQUERIEDPATH captured in Algorithm 38 for this purpose. If the given leaf node $n_{i,u}$ has been just recognized as a new state node, then it is extended by the first input symbol of the input alphabet X and the next state is identified (lines 1–6). Otherwise, the previous nodes up to the closest state node $n_{i,\varepsilon}$ are checked whether they can be extended. In fact, the check goes from the closest state node $n_{i,\varepsilon}$ so a suitable node $n_{i,v}$, $u = v \cdot v'$, with the shortest access sequence is extended first. There are three requirements on the node $n_{i,v}$: it is not identified as $|\phi(\bar{s}_i v)| > 1$; it could be

Algorithm 37: H-IDENTIFY($n_{i,ux}$)

```

1 if  $|\phi(\bar{s}_i ux)| = 1$  then
2    $w \leftarrow x$  // attempt to reveal new information
3 else if  $\delta^*(s_i, ux)$  is defined to  $s_j$  then // verification
4    $w \leftarrow$  the shortest sequence that separates  $n_{j,\varepsilon}$  from the most SNs
   that relate to states in  $\phi(\bar{s}_i ux)$ 
5 else // identification
6    $w \leftarrow$  the shortest adaptive separating sequence that has the lowest
   probability of not identifying states in  $\phi(\bar{s}_i ux)$ 
7 H-APPLYANDCHECK( $n_{i,ux}, w$ )

```

an extra state or its next state, that is, $|v| \leq l + 1$; and there is a sequence w starting with v' that separates some states of $\phi(\bar{s}_i v)$ (line 7). The separating sequence w has the fixed prefix v' and the rest is derived by the functions used in H-IDENTIFY to select (adaptive) separating sequences. State nodes of states in $\phi(\bar{s}_i v)$ follow v' and the reached nodes form a domain $\phi'(\bar{s}_i u)$ used in the functions that choose a sequence. Note that these two functions work with nodes and not with states as it was explained in Section 15.2.1. The domain $\phi'(\bar{s}_i u)$ can be smaller than $\phi(\bar{s}_i v)$ because v' does not have to be observed from some state nodes. As H-Algorithm 37 specifies on lines 3–6, the chosen sequence w' is a separating sequence if $\delta_M^*(s_i, u)$ is defined. Otherwise, w is an adaptive separating sequence starting with v' . If there is not such a predecessor $n_{i,v}$, then the given leaf node is extended only if it could be an extra state or its next state. In such a case, the function H-IDENTIFY is called (line 12 of Algorithm 38).

Algorithm 38: H-TRYEXTENDQUERIEDPATH($n_{i,u}$)

```

1 if  $n_{i,u}$  is a state node, that is,  $u = \varepsilon$  then
2   QUERY( $n_{i,\varepsilon}, x$ ), where  $x$  is the alphabetically lowest input symbol
3   if the response reveals a new state then //  $\phi(\bar{s}_i x)$  is empty
4     H-UPDATEDOMAINS( $n_{i,x}$ )
5     H-TRYEXTENDQUERIEDPATH( $n_{i,x}$ )
6   else H-IDENTIFY( $n_{i,x}$ )
7 else if there is  $n_{i,v}$  such that  $u = v \cdot v'$ ,  $|v| \leq l + 1$  and there is a
   separating sequence  $w = v' \cdot w'$  reducing  $\phi(\bar{s}_i v)$  then
8   consider such  $n_{i,v}$  with the shortest  $v$ 
9    $\phi'(\bar{s}_i u) \leftarrow \{n_{j,v'} \mid s_j \in \phi(\bar{s}_i v) \wedge \bar{s}_j v' \in T\}$ 
10   $w'$  is the shortest (adaptive) sequences that separates the most nodes
   in  $\phi'(\bar{s}_i u)$ , it is obtained as in Algorithm 37 lines 3–6
11  H-APPLYANDCHECK( $n_{i,u}, w$ )
12 else if  $|u| \leq l + 1$  then H-IDENTIFY( $n_{i,u}$ )

```

The last piece of the H-learner is the processing of a counterexample w . Each obtained counterexample (CE) is first queried and domains are update

accordingly. However, a CE can contain redundant information, such as unnecessary loops, and so be quite long. Then a new state is not possible to reveal by the standard approach in H-UPDATEDOMAINS because ‘close to leaves’ nodes have spare distinguishing sets. It means that such a node does not have enough outgoing sequences that would separate the node from all state nodes. The inconsistency is captured by the inconsistency II, see Section 15.2.2. That is, there is a node $n_{i,u}$ such that $\delta^*(s_i, u) \notin \phi(\bar{s}_i u)$ if the response of the conjectured model M to the given CE w is still incorrect.

Algorithm 39: H-UPDATEWITHCE(w)

```

1 let  $w'$  be the shortest prefix of  $w$  such that  $\delta^*(s_0, w') \notin \phi(\bar{s}_0 w')$ 
2 for  $j \leftarrow |w'|$  down to 1;  $w = x_1 \cdot \dots \cdot x_{|w|} = u_j \cdot x_j \cdot v_j$  do
3   let  $s_i = \delta^*(s_0, u_j)$  and  $s_k = \delta(s_i, x_j)$ 
4   if  $\bar{s}_k v_j \notin T$  then
5     H-APPLYANDCHECK( $n_{k,\varepsilon}, v_j$ )
6     if a new state is revealed then break
7   if  $\bar{s}_i x_j v_j \notin T$  then
8     H-APPLYANDCHECK( $n_{i,x_j}, v_j$ )
9     if a new state is revealed then break

```

The function H-UPDATEWITHCE in Algorithm 39 describes which sequences are further queried based on the given counterexample. The function follows Theorem 14.1 of the decomposition of a counterexample. There is a suffix v_j that distinguishes the next state of state s_i on x_j from the assumed next state $s_k = \delta_M(s_i, x_j)$. A new state is thus revealed if v_j is queried from both n_{i,x_j} and $n_{k,\varepsilon}$. However, which of suffixes of w is the distinguishing one? All suffixes are queried gradually from the shortest one until the distinguishing one is found and a state is revealed. Moreover, some of the shortest suffixes are already queried so they are not considered. Let $w = w' \cdot w''$ such that $\delta_M^*(s_0, w') \notin \phi(w')$. Then all suffixes of w'' were queried because the function H-UPDATEWITHCE queries v_j from $n_{k,\varepsilon}$ and $x_j \cdot v_j$ from $n_{i,\varepsilon}$ where $x_j \cdot v_j = v_{j-1}$. Therefore, v_j 's are usually queried from the related $n_{k,\varepsilon}$ by the previous loop on j , that is, when j was $j + 1$. As w'' was queried, it distinguished $n_{k,\varepsilon}$ of $s_k = \delta_M^*(s_0, w')$ from node $n_{w'}$. If w' is the shortest prefix of w with the property, then $w'' = v_j$ was not probably queried from related n_{i,x_j} so w'' is the first suffix to query.

The H-learner needs to store the observation tree that is like a testing tree constructed by the H-method in the end of learning. Hence, the worst case space complexity can be estimated to $O(n^3 p^{l+1})$ as was derived in Section 10.1. The time complexity depends on the size of the observation tree because the observed traces are compared repeatedly to identify particular states. The H-learner thus spends most of the time by updating domains in H-UPDATEDOMAINS. Let $|T|$ denote the size of the observation tree so that $|T| \in O(n^3 p^{l+1})$. Assume the worst case. The last queried sequence u consists of the access sequence of length $n - 1$, an extension of length

$l + 1$ and a separating sequence of length in $O(n)$. As l is usually smaller than n , $l \in O(n)$ and so $|u| \in O(n)$. Note that separating sequences are chosen from the observed ones that are the shortest, hence, the length in $O(n)$. There are $O(n + l)$ nodes and n state nodes to check during the processing u . These $O(n + l)$ nodes are compared with at most n SNs on the related suffix of u that has length in $O(n + l)$. Hence, these comparisons run in $O(n(n + l)^2)$ which is $O(n^3)$. Much harder is the update of domains of state nodes. The number of consistent nodes is bounded by $|T|$. Thus, n state nodes, each with $O(|T|)$ consistent nodes to compare on the sequence of length in $O(n)$, give the running time $O(n^2|T|)$ in the worst case. The total time complexity in the worst case is estimated by the assumption that the function H-UPDATEDOMAINS is called for each node of the observation tree (after it is added to the OTree). It means that the H-learner runs in $O(n^2|T|^2)$, that is, $O(n^8p^{2l+2})$, in the worst case.

15.3.2 Running Example

The H-learner was proposed and described in detail in the previous section. This section shows the learning by the algorithm on an example. The black box is represented by a DFA in Figure 15.5. It has binary input alphabet $X = \{a, b\}$ and 4 states A–D such that only D is accepting, that is, $\lambda_N(q, \uparrow) = 1$ if and only if $q = D$. Figure 15.5 shows the state behaviour table (output and transition functions) and the state diagram of the machine. Algorithms 34–39 are followed to learn this black box. Only one extra state is considered, that is, $maxExtraStates = 1$, as it suffices to learn such a small machine without the need of a counterexample. The teacher is able to provide output queries (OQ).

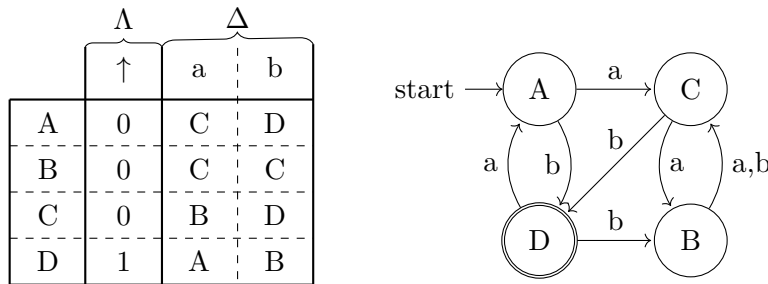


Figure 15.5: Black box represented by a 4-state DFA with the accepting state D

The learning starts with the reset of the black box and querying \uparrow to obtain the output of the initial state; output query 1 is $T(\varepsilon, \uparrow)$. The root $n_{0,\varepsilon}$ of the observation tree (OTree) with the relating initial state s_0 of the conjectured model are created. According to Algorithm 34, *Unchecked* is filled up with $n_{0,\varepsilon}$ and the root is chosen for checking first. Its access sequence (ε) has the length of 0 which equals to the value of l . Therefore, the condition on line 5 of Algorithm 34 is not satisfied and the algorithm goes directly to check the next states of the initial state.

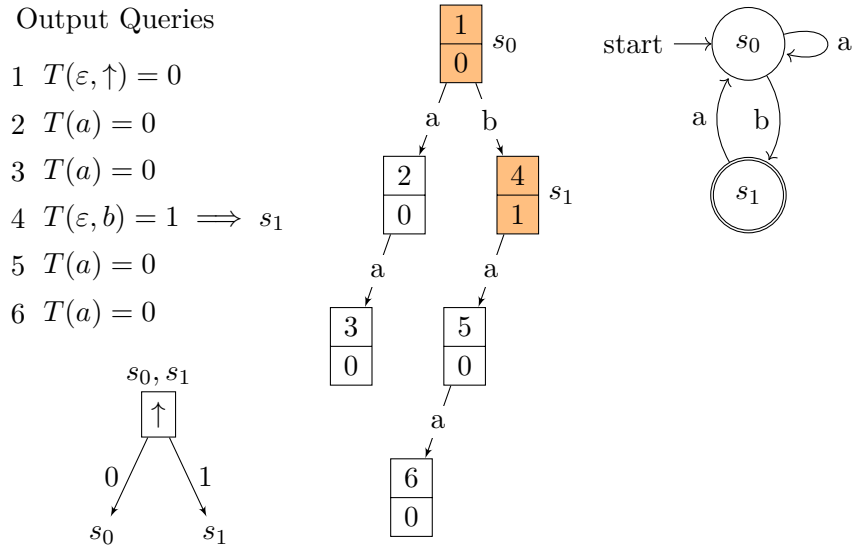


Figure 15.6: The first 6 output queries reveal the state D (s_1)

The first 6 output queries with the constructed observation tree and the conjectured model are depicted in Figure 15.6. Moreover, Figure 15.6 also shows a separating sequence that distinguishes states s_0 and s_1 , or A and D in the black box. Note that a node of OTree is labelled with the number of the related output query and the state output in the lower part. State nodes are then marked with the labels next to the nodes.

The target state on the input ‘a’ is checked first. $T(a)$ replies with 0 so that the reached state cannot be distinguished from the initial state by the state output. The additional feature of the H-learner takes place here. According to the function H-IDENTIFY (Algorithm 37), the next input to query is again ‘a’. The target state could be distinguished from the initial one if they had different output on ‘a’; output query 3 is $T(a)$. Unfortunately, the output is the same so another input is tried from the initial state. The target state on ‘b’ responds with 1 that reveals a new state s_1 . The conjectured model thus contains two states, the rejecting initial s_0 and the accepting s_1 , with a self-loop on ‘a’ from s_0 and a transition on ‘b’ from s_0 to s_1 . The input \uparrow forms the separating sequence of the states; it is depicted in bottom left of Figure 15.6.

The black box is currently in the state corresponding to $n_{1,\varepsilon}$. Therefore, the algorithm queries $T(a)$ to obtain the output of the next state of s_1 on ‘a’. The input ‘a’ was chosen because it is the lowest alphabetically. The observed output is 0 and thus the next state is assumed to be s_0 . A transition on ‘a’ from s_1 to s_0 is added to the conjectured model as Figure 15.6 shows. Similarly to the query 3, $T(a)$ is queried again to try to reveal new information.

The queue *Unchecked* is filled up with all state nodes as a new state has just been revealed. Particularly, it contains $n_{0,\varepsilon}$ and $n_{1,\varepsilon}$ in this order. Both successors of $n_{0,\varepsilon}$ are identified because $|\phi(a)| = |\phi(b)| = 1$. Therefore, the

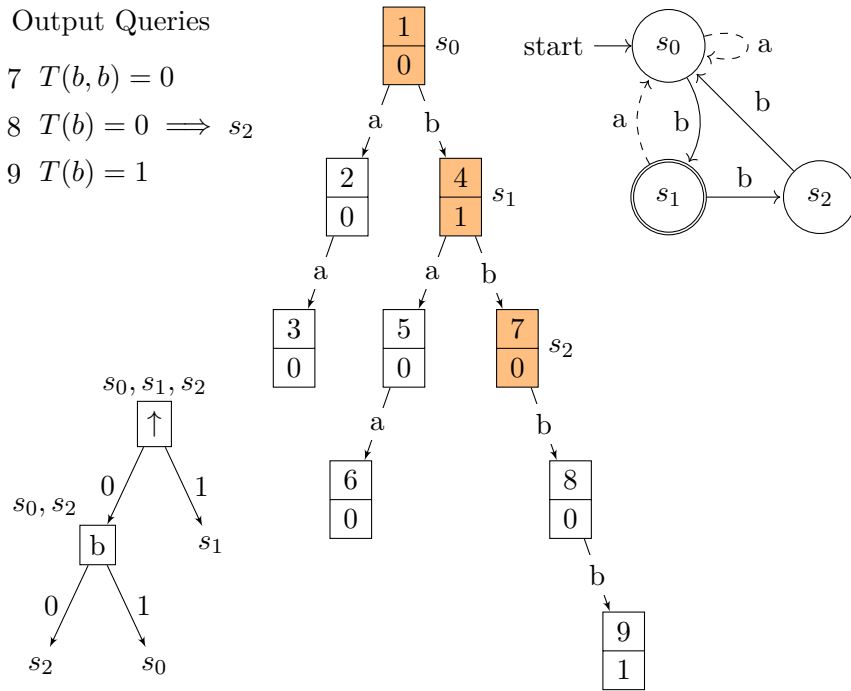


Figure 15.7: The next 3 output queries reveal the state B (s_2)

algorithm checks the next states of s_1 . The first one (on 'a') is identified and the second has not been observed yet. Thus, the output query 7 is $T(b, b)$ that replies with 0. The transition on 'b' from s_1 is assumed to lead to s_0 . Nevertheless, the function H-IDENTIFY asks for another query on 'b' as it is the input by which the current state is reached; output query 8 is $T(b)$. The expected output is $\lambda_M(s_0, b) = 1$, however, the observed output is 0. The node $n_{1,b}$ thus cannot relate to s_0 and a new state s_2 is revealed. The function H-UPDATEDOMAINS (Algorithm 35) eliminates s_0 from the domain $\phi(bb)$ so it becomes empty. The state s_2 is introduced and is added to all domains of nodes consistent with $n_{1,b} = n_{2,\varepsilon}$. The domains $\phi(a)$, $\phi(ba)$ and $\phi(bbb)$ are the most important because they relate to next states. These next states will be needed to identify again. This is reflected in the conjectured model in Figure 15.7 by dashed transition lines. With state s_2 , a new separating sequence 'b' was observed. The updated adaptive separating sequence is shown in bottom left of Figure 15.7. The sequence is immediately used to identify the next state of s_2 ; output query 9 is $T(b)$. It responded as s_0 so that the transition on 'b' from s_2 leads to s_0 as the current conjectured model in Figure 15.7 shows. The query 9 was asked due to the function H-TRYEXTENDQUERIEDPATH (Algorithm 38) that called H-IDENTIFY.

All state nodes fill the queue *Unchecked* again and are processed in the order s_0 , s_1 and s_2 . The successor of the root node on 'a' is not identified as it is consistent with both $n_{0,\varepsilon}$ and $n_{2,\varepsilon}$. Therefore, an (adaptive) separating sequence of these state nodes is chosen and applied in $n_{0,a}$; output query 10 is

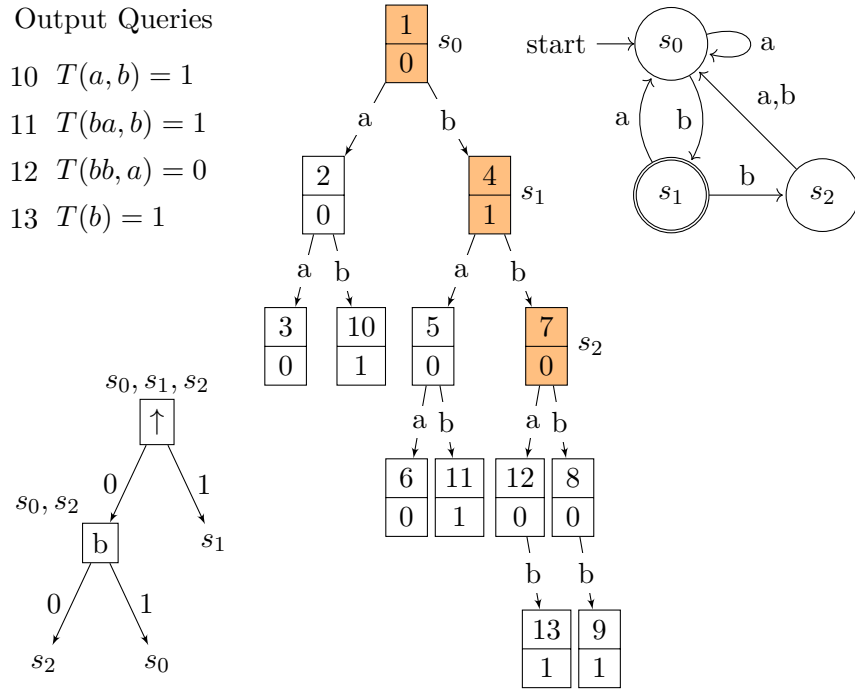


Figure 15.8: The next 4 output queries make the OTree closed for 0 extra states

$T(a, b)$. The output 1 identifies the state relating to $n_{0,a}$ as s_0 . Therefore, the transition on 'a' from s_0 is again confirmed to lead back to s_0 . The transition on 'a' from s_1 is confirmed in the same way; the separating sequence 'b' is applied in the successor $n_{1,a}$. The remaining undefined transition is the one leading from s_2 on 'a'. The state reached by this transition has the same output as states s_0 and s_2 ; $T(bb, a) = 0$. Therefore, the separating sequence 'b' is applied to identify the target state $\delta_M(s_2, a)$. The output 1 distinguishes the state from s_2 so that the target state is assumed to be s_0 . Then, a completely specified conjecture is formed as the observation tree is closed for 0 extra states, see Figure 15.8. A new state was not revealed so that the shortest adaptive separating sequence remains the same (bottom left of Figure 15.8).

Algorithm 34 defines that the number l of considered extra states is incremented by 1 if the OTree is closed for the current l . It means that transitions leading from the next states will be verified next. The queue *Unchecked* is now filled up with $n_{0,a}$, $n_{1,a}$, $n_{2,a}$ and $n_{2,b}$ in this order. Both transitions from $n_{0,a}$ are observed (line 19 in Algorithm 34) but the successor on 'a' is not identified (line 25 in Algorithm 34). Hence, the function H-IDENTIFY chooses the separating sequence 'b' to verify $n_{0,aa}$ to correspond to s_0 ; output query 14 is $T(aa, b)$. H-APPLYANDCHECK (Algorithm 36) then reveals a new state s_3 as the observed output 0 does not match the expected one. Note that H-UPDATEDOMAINS first reduces $\phi(aa)$ to $\{s_2\}$ and then eliminates s_0 from $\phi(a)$ that becomes empty and a new state is observed with a separating

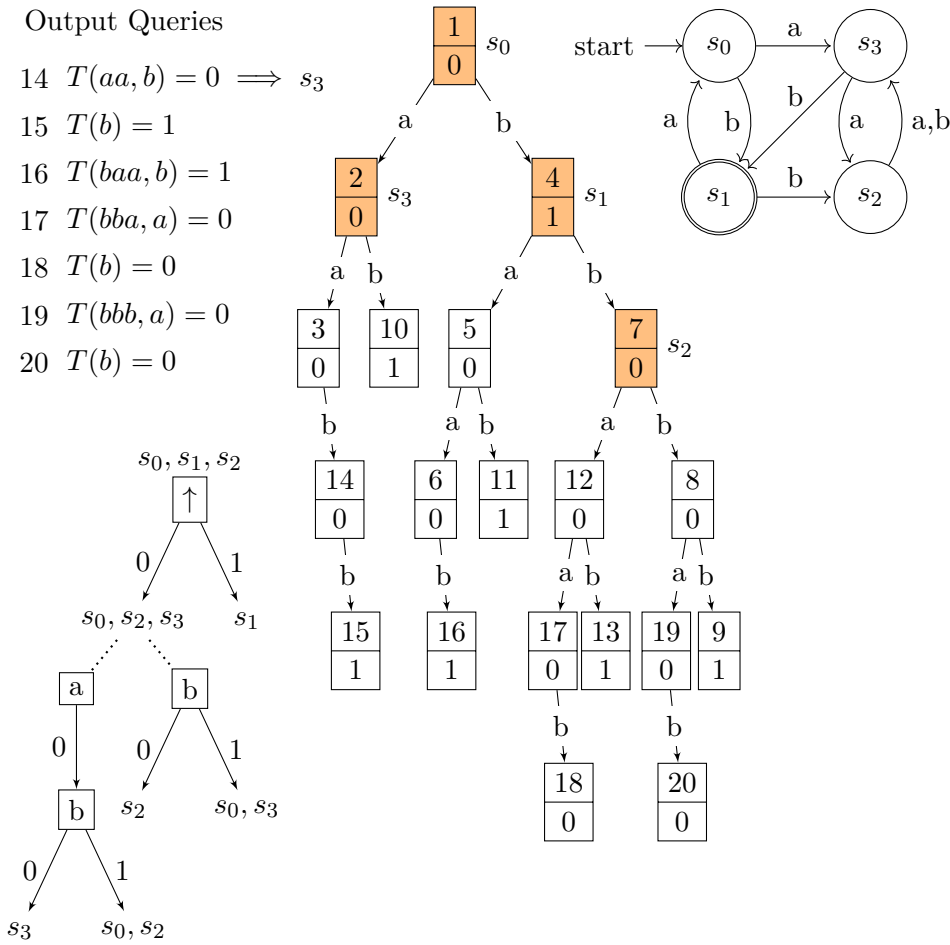


Figure 15.9: The next 7 output queries complete the 4-state DFA

sequence ‘ab’. As ‘ab’ does not separate s_0 and s_2 , there is no observed adaptive distinguishing sequence. Both shortest adaptive separating sequences, ‘b’ and ‘ab’, thus need to be applied to identify a node in the worst case. The sequences merged into one with the common prefix \uparrow are shown bottom left in Figure 15.9. Note that both sequences have the same Distinguishing score of $\frac{1}{4}$ but ‘b’ is shorter and so it is preferable for the identification. The black box is currently in $n_{3,ab}$. Therefore, the function `H-TRYEXTENDQUERIEDPATH` looks at $n_{3,a}$ and $n_{3,ab}$ whether their domains can be reduced by extending the current path in the OTree. The domain $\phi(\bar{s}_3a)$ is $\{s_2\}$ so that it does not need any further separating sequence. However, $\phi(\bar{s}_3ab)$ contains all states but s_1 and thus `H-IDENTIFY` with $n_{3,ab}$ is called. The state relating to the node is not known as the transition $\delta_M(s_2, b)$ is not defined; $\phi(\bar{s}_2b) = \{s_0, s_3\}$. Hence, the preferred adaptive separating sequence is applied; output query 15 is $T(b)$. Notice that the algorithm could be wiser here and apply ‘ab’ separating s_0, s_3 that are the expected states of $n_{3,ab}$. After 1 is observed, the domain $\phi(\bar{s}_3ab)$ is reduced to $\{s_0, s_3\}$. Then, the algorithm makes the

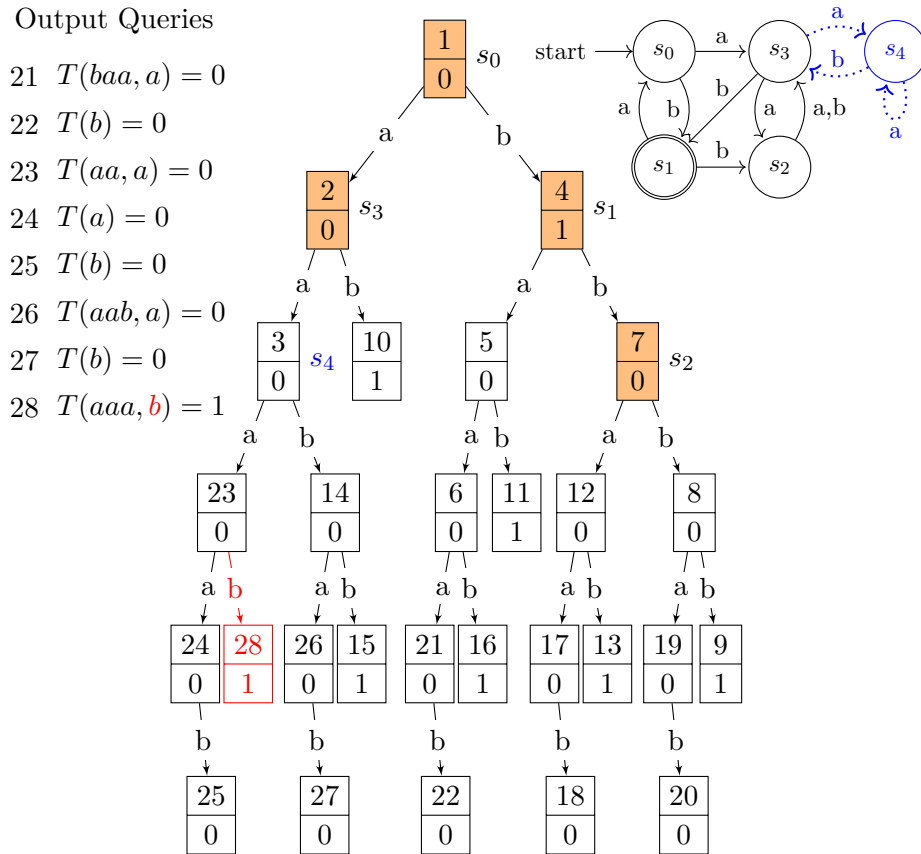


Figure 15.10: Theorem 8.12, condition 2. is used to eliminate possible extra states

OTree closed for no extra state again by applying ‘b’ in $n_{1,aa}$ and ‘ab’ in the other two unidentified successors. Figure 15.9 shows the observation tree with the completely specified conjectured model after 20 output queries. The observation tree is closed for 0 extra states and the conjectured model matches the black box so the learning would stop if an equivalence query was asked.

Next 14 output queries verify that there is not one extra state. Output query 28 is particularly interesting as its purpose is not to reduce the domain of a node. It follows lines 26–28 of Algorithm 34 that describe the second condition of the H-condition (Theorem 8.12). If this condition was not covered, for example, ‘b’ was not queried in $n_{3,aa}$, then there would be a 5-state machine distinguishable from the 4-state conjecture. Figure 15.10 shows both the observation tree after 28 output queries and a possible 5-state model of the black box if output query 28 is missing. The query completes the separating sequence ‘ab’ that distinguishes nodes $n_{3,a}$ and $n_{3,aa}$. Without this sequence, both nodes can relate to the same extra state s_4 although they relate to different states as it is captured by their domains $\phi(\bar{s}_3a) = \{s_2\}$ and $\phi(\bar{s}_3aa) = \{s_3\}$.

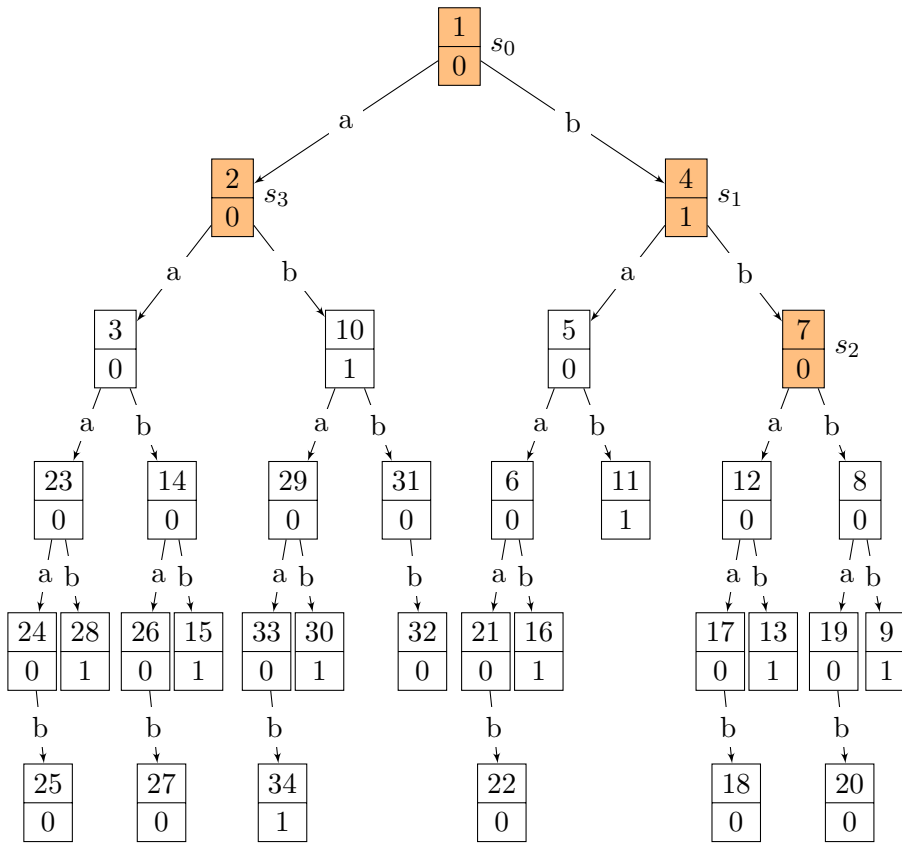


Figure 15.11: Observation tree closed for 1 extra state by the H-learner

The final observation tree that is closed for 1 extra state is shown in Figure 15.11. The last 6 output queries verify target states of both transitions from $n_{3,b}$. The queries can be derived from the OTree in Figure 15.11 as the nodes are labelled with the number of the related output query.

The H-learner asked 34 output queries T , applied 17 resets and queried 68 input symbols in total to learn the model of the black box. It provides a guarantee that the model is correct or the black box has more than 5 states as 1 extra state was considered during the learning. Moreover, the conjectured model was correct after 20 output queries (10 resets and 36 queried symbols) when the completely specified 4-state DFA were constructed.

15.4 SPY-learner

The SPY-learner is another implementation of the observation tree approach. It is inspired by the SPY-method (Section 9.7) and it aims to satisfy the SPY-condition (Theorem 8.13). It means that the convergence of sequences is employed and a fixed set of harmonized state identifiers is used for verification of transitions in the second phase of the observation tree approach.

The use of convergence of sequences enables the learner to derive a completely specified conjectured model faster in terms of the interaction with the black box. The reason is that when a transition is defined by identifying the target state, the last queried sequence u (that identified the target state) can be often extended to verify another transition, such as the one that leads from the target state and starts the separating suffix of u (used to identify the target state). On the other hand, this comes with a price of more difficult resolving of inconsistency as described in Section 15.2.2. However, it is sufficient to make $\Phi([\bar{s}_i x])$ singleton in order to identify the target state of transition (s_i, x) in the first phase of the observation tree approach.

Harmonized state identifiers (HSI) are formed of the shortest separating sequences captured in the observation tree. HSIs are extended when a new state node $n_{k,\varepsilon}$ is revealed. For each state s_i the shortest separating sequence of $n_{i,\varepsilon}$ and $n_{k,\varepsilon}$ is added to the HSIs of s_i and s_k . HSIs are then used in the second phase of the observation tree approach to construct *requested suffixes* that are needed to be queried from particular CNs to verify the related transitions. Requested suffixes depend on the number l of extra states and are constructed like in the SPY-method.

The SPY-learner makes the observation tree closed for the increasing l gradually. When all transitions are verified for l extra states, l is increased by 1 and all sequences are considered to be convergent only with themselves until their convergence is proven for the increased l . Unfortunately, it means that the convergence of sequences is utilized only when a transition was just verified and a particular requested suffix can extend the last queried sequence. All other requested suffixes are queried from the related state nodes as they are reached by the shortest access sequences. Hence, the other convergent sequences are not used. The choice of the state nodes to extend them aims to minimize the number of queried symbols. This is the difference from the testing method that can extend any convergent sequence with zero cost in terms of the length of access sequences.

The following sections describe the implementation of the SPY-learner with an explanation on a running example.

15.4.1 Implementation

The implementation of the SPY-learner is proposed in Algorithms 40–48. The observation tree approach is followed so that l specifies the number of extra states that are currently considered, and *nodeBB* points to the node of OTree that corresponds to the current state of the black box. Besides the

observation tree T and the conjectured model M , the SPY-learner maintains the convergent graph, requested suffixes if $l > 0$ and keeps track of unverified transitions.

Algorithm 40 describes the main learning cycle of the SPY-learner. The cycle is controlled by the boolean variable *learnt* that becomes false if the stop condition is met, that is, the observation tree is closed for the given *maxExtraStates* and there is no counterexample (or EQs are not allowed at all), or the user is satisfied with the conjectured model M . Before the main cycle starts, the convergent graph and domains are initialized in the function SPY-GENERATEREQUESTEDSUFFIXES that is also called (line 25) when l was just increased (line 16) and the learning structures are to be initialized as l did not reach *maxExtraStates*. If there is an inconsistency between the observation tree T and the conjectured model M , then it is resolved by the function RESOLVEINCONSISTENCY defined in Algorithm 33; the function QUERYSEQANDCHECK in RESOLVEINCONSISTENCY is implemented by SPY-QUERYSEQANDCHECK. Otherwise, for each unverified transition, the target state is either identified using SPY-IDENTIFYNEXTSTATE if $l = 0$ (lines 10–11), or verified by a requested suffix (lines 12–14). For each transition (s_j, x) leading to s_k , there are requested suffixes for both s_j and s_k . The requested suffix for the origin state s_j are prioritized (line 13). Note that $n_{i,u}.state$ represents for each node the state of the conjectured model to which the node corresponds, that is, $n_{i,u}.state = \delta_M^*(s_i, u)$. In the first phase of the observation tree approach when $l = 0$, $n_{i,u}.state$ does not have to be set as some transitions are not defined yet. The function SPY-GETNODETOEXTEND called on line 9 checks if the last queried sequence can be extended and in some cases explained later, it can set $n_{i,u}$ to null.

The function SPY-GENERATEREQUESTEDSUFFIXES in Algorithm 41 describes how the requested suffixes are constructed for the given l . For 0 extra states, HSIs are not known, hence, all transitions that are not covered by the state cover \bar{S} are marked ‘unverified’. Moreover, the convergent graph and domains are initialized (lines 2–5). In order to verify a transition (s_i, x) that leads to s_j , a set U of requested suffixes is generated for state s_j as $U = X^{\leq l} \circ HSI_k$ where all sequences start in s_j . A set of requested suffixes for s_i is then $x \cdot U$. As a reminder, the operation \circ is a concatenation that depends on the reached state, for example, U contains HSI_j of state s_j , then it contains $x' \cdot HSI_{k'}$ where $s_{k'} = \delta_M(s_j, x')$, then it can contain $x' \cdot x'' \cdot HSI_{k''}$ where $s_{k''} = \delta_M^*(s_j, x'x'')$ if $l > 1$, and so on. The function SPY-GENERATEREQUESTEDSUFFIXES checks immediately whether all the generated requested suffixes are already queried from the related state nodes. If they are queried, the transition is verified but it is marked ‘confirmed’ because the corresponding convergent classes need to be merged which is done in the function SPY-PROCESSIDENTIFIED. Otherwise, the transition is marked ‘unverified’.

Algorithm 42 describes the function SPY-PROCESSIDENTIFIED that first checks if there is a new state node revealed and if not, it merges convergent nodes that were proven to be convergent. There are two merging functions:

Algorithm 40: SPY-learner

```

input : A teacher
input :  $maxExtraStates$ 
output: A conjectured model  $M$ 

1  $l \leftarrow 0$  // the number of extra states currently considered
2 initialize the OTree  $T$  with the root  $n_{0,\varepsilon}$  (the initial state  $s_0$  of  $M$ )
3 SPY-GENERATEREQUESTEDSUFFIXES( $l$ )
4  $learnt \leftarrow$  false
5 while not  $learnt$  do
6   if  $T$  and  $M$  are inconsistent then
7     RESOLVEINCONSISTENCY( $T, M$ )
8   else if there is an unverified transition then
9      $(n_{i,u}, x) \leftarrow$  SPY-GETNODETOEXTEND( $nodeBB$ )
10    if  $l = 0$  and  $n_{i,u}$  is not null then
11      SPY-IDENTIFYNEXTSTATE( $n_{i,u}, x$ )
12    else if  $n_{i,u}$  is not null then
13       $w \leftarrow$  a requested suffix to verify transition  $(n_{i,u}.state, x)$  such
        that if there is no suffix for  $n_{i,u}.state$ , then take one assigned
        to  $s_k = \delta(n_{i,u}.state, x)$  and consider  $n_{k,\varepsilon}$  instead of  $n_{i,u}$ 
14      SPY-QUERYANDCHECKEACH( $n_{i,u}, w$ )
15    else
16       $l \leftarrow l + 1$ 
17      if  $l > maxExtraStates$  then
18        if EQs are allowed then
19          if there is a counterexample  $w$  obtained on EQ then
20             $l \leftarrow l - 1$ 
21            SPY-QUERYSEQANDCHECK( $n_{0,\varepsilon}, w$ )
22            continue
23           $learnt \leftarrow$  true
24        else
25          SPY-GENERATEREQUESTEDSUFFIXES( $l$ )
26          SPY-PROCESSIDENTIFIED()
27    if user is satisfied with  $M$  then  $learnt \leftarrow$  true
28 return the conjecture  $M$ 

```

Algorithm 41: SPY-GENERATEREQUESTEDSUFFIXES(l)

```

1 foreach transition  $(s_i, x)$  not covered by  $\bar{S}$  do
2   if  $l = 0$  then
3     generate a part of convergent graph covering the subtree of  $n_{i,x}$ 
4     initialize  $\Phi$  by  $\phi$  for each node of the subtree
5     mark  $(s_i, x)$  unverified
6   else
7     requested suffixes to verify  $(s_i, x)$  such that  $s_j = \delta(s_i, x)$  are:
8        $x \cdot X^{\leq l} \circ HSI_k$  for  $s_i$  and  $X^{\leq l} \circ HSI_k$  for  $s_j$ 
9     if all these are queried from  $n_{i,\varepsilon}$  and  $n_{j,\varepsilon}$ , respectively then
10      | mark  $(s_i, x)$  confirmed
11    else
12      | mark  $(s_i, x)$  unverified
13      | remember those requested suffixes that were not queried

```

one for the case when $l = 0$ and one when $l > 0$. If $l = 0$, each identified convergent node, that is, CN with a single RN in its domain Φ , is merged into the corresponding reference node. This merging takes place in the convergent graph that needs to remain deterministic and so the successor CNs on common inputs are merged as well. In the end of the first phase of the observation tree approach, the convergent graph looks like the conjectured model as it contains just the RNs. If a CN $[v]$ is to be merged in an RN $[\bar{s}_k]$, then the following is done.

- (i) The inconsistency IV (wrong merge) is checked by $[v]$, that is, $[\bar{s}_k]$ must be in $\Phi([v])$.
- (ii) Each node $n_{i,u}$ in $[v]$ is assumed to correspond to s_k , that is, $n_{i,u}.state = s_k$.
- (iii) Each CN $[u] \in \Phi([\bar{s}_k])$ that is separated from $[v]$ is removed from $\Phi([\bar{s}_k])$ and $[\bar{s}_k]$ is eliminated from $\Phi([u])$.
- (iv) The inconsistency III (empty CN domain) is checked by each CN considered in (iii).
- (v) If the predecessor of $[v]$ is an RN $[\bar{s}_i]$ such that $v = \bar{s}_i x$, then the transition (s_i, x) is marked as ‘verified’, is defined to lead to s_k and is added to the conjectured model M .

If a CN $[v]$ is to be merged in another CN $[u]$ such that both are not RNs, then the domain of the merged CN is an intersection of $\Phi([v])$ and $\Phi([u])$. The second merging function, when $l > 0$, is similar but the actual merging does not take place. The convergent graph is not changed for $l > 0$. Instead, each node of the observation tree possesses an attribute if it is already proven to be in the related RN or not. If a CN $[u]$ is to be ‘merged’ in an RN

$[\bar{s}_k]$, then all requested suffixes for s_k that are queried from a node of $[u]$ are eliminated. Moreover, if all the requested suffixes related to a transition are queried, the transition is marked ‘confirmed’ in order to be processed by this merging function.

Algorithm 42: SPY-PROCESSIDENTIFIED()

```

1 if there is  $n_{i,u}$  with empty domain  $\phi(\bar{s}_i u)$  then
2   | SPY-MAKESTATENODE( $n_{i,u}$ )
3 else if  $l = 0$  and  $M$  and  $T$  are consistent then
4   | foreach CN  $[\bar{s}_i u]$  such that  $\Phi([\bar{s}_i u]) = \{[\bar{s}_j]\}$  do
5     | merge  $[\bar{s}_i u]$  into  $[\bar{s}_j]$  and then recursively their successors
6     | - during merge: update  $\Phi$ , check for inconsistency, mark ‘verified’
        |   any transition that starts and ends in RNs - add it to  $M$ , and
        |   each  $n_v.state$  gets  $s_k$  if  $[v]$  merged into the RN  $[\bar{s}_k]$ 
7 else if  $M$  and  $T$  are consistent then
8   | foreach confirmed transition  $(s_i, x)$  such that  $s_j = \delta(s_i, x)$  do
9     | mark  $(s_i, x)$  verified
10    | merge  $[\bar{s}_i x]$  into  $[\bar{s}_j]$  and then recursively their successors
11    | - during merge: eliminate requested suffixes queried from a node
        |   just merged to an RN, mark ‘confirmed’ any transition that has
        |   all requested suffixes queried

```

After the inconsistency I for a node $n_{i,u}$ is observed, that is, $\phi(\bar{s}_i u)$ is empty, the learning structures are updated by the function SPY-MAKESTATENODE defined in Algorithm 43. It first checks whether $n_{i,u}$ is a child of a state node. If not, another instance of the inconsistency I is revealed by querying sequences that aim to reduce the domain ϕ of the parent $n_{i,u'}$ of $n_{i,u}$ (lines 2–14). Figure 15.12 sketches how other inconsistency I is found. First, the state s_j corresponding to the parent $n_{i,u'}$ is considered and its state node (SN) $n_{j,\varepsilon}$ is extended with x where $u = u' \cdot x$. If the parent is not identified yet, any state of its domain $\phi(\bar{s}_i u')$ is considered as s_j . The function looks at $n_{j,x}$ and its domain $\phi(\bar{s}_j x)$. As the domain of $n_{i,u}$ is empty, for each SN there is a sequence separating the SN and $n_{i,u}$. When such a separating sequence w is queried from $n_{j,x}$, it eliminates the corresponding state s_k from $\phi(\bar{s}_j x)$ or s_j from $\phi(\bar{s}_i u')$, see Figure 15.12. If the latter case happens, either the domain $\phi(\bar{s}_i u')$ can become empty and thus a new instance of the inconsistency I is found, or $n_{i,u'}$ was identified as s_j (line 13 of Algorithm 43) and thus $\phi(\bar{s}_i u')$ is inconsistent, or another s_j of $\phi(\bar{s}_i u')$ is tried to be eliminated from the domain. Note that $n_{i,u'.state}$ is ‘null’ if the node is not identified. If $n_{j,x}$ responds to all separating sequences w ’s like $n_{i,u}$, then $\phi(\bar{s}_j x)$ becomes empty and a new instance of the inconsistency I is found as well.

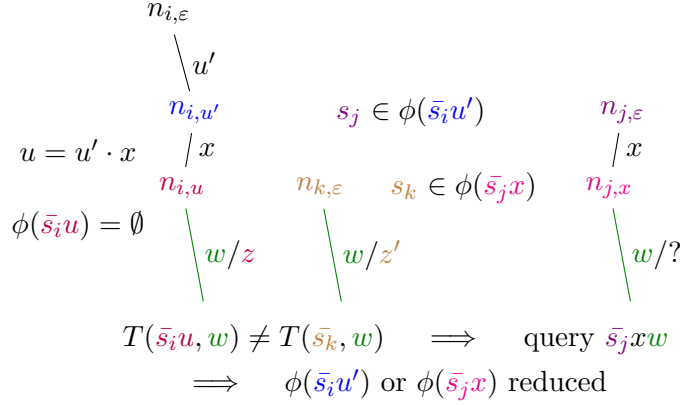


Figure 15.12: Resolving inconsistency I if the parent of inconsistent node $n_{i,u}$ is not a state node

Algorithm 43: SPY-MAKESTATENODE($n_{i,u}$)

```

1 let  $n_{i,u'}$  be the parent of  $n_{i,u}$  and  $u = u' \cdot x$ 
2 if  $n_{i,u'}$  is not an SN, that is,  $u' \neq \epsilon$  then
3   if  $\phi(\bar{s}_i u')$  is empty then return SPY-MAKESTATENODE( $n_{i,u'}$ )
4   while  $\phi(\bar{s}_i u')$  is not empty do
5     let  $s_j$  be  $n_{i,u'}$ .state if it is set and it is in  $\phi(\bar{s}_i u')$ , otherwise let  $s_j$ 
6     be any state in  $\phi(\bar{s}_i u')$ 
7     SPY-QUERYSEQANDCHECK( $n_{j,\epsilon}$ ,  $x$ , false)
8     foreach  $s_k \in \phi(\bar{s}_j x)$  do
9        $w \leftarrow$  separating sequence of  $n_{i,u}$  and  $n_{k,\epsilon}$ 
10      SPY-QUERYSEQANDCHECK( $n_{j,x}$ ,  $w$ , false)
11      if  $s_j \notin \phi(\bar{s}_i u')$  then break
12    if  $\phi(\bar{s}_j x)$  or  $\phi(\bar{s}_i u')$  is empty then
13      return SPY-MAKESTATENODE( $n_{j,x}$  or  $n_{i,u'}$  respectively)
14    else if  $s_j = n_{i,u'}$ .state then
15      return // inconsistent domain  $\phi(\bar{s}_i u')$ 
15 add a new state  $s_k$  to the conjecture  $M$ ;  $n_{i,u} = n_{k,\epsilon}$ 
16 update HSIs with separating sequences queried from  $n_{k,\epsilon}$ 
17 add  $s_k$  to the domains of all nodes that are not separated from  $n_{k,\epsilon}$ 
18  $l \leftarrow 0$ 
19 SPY-GENERATEREQUESTEDSUFFIXES( $l$ )
20 SPY-PROCESSIDENTIFIED()

```

Algorithm 43 continues on line 15 if $n_{i,u}$ is a child of an SN, that is, u is an input x . Then, a new state node $n_{k,\epsilon}$ is created in the place of $n_{i,x}$. It corresponds to the new state s_k of the conjectured model. As was mentioned, separating sequences of $n_{k,\epsilon}$ and other SNs form the harmonized state identifier of s_k and update HSIs of the other states. Domains ϕ are

updated with s_k such that the state is added to the domain of each node $n_{j,v}$ that is consistent with $n_{k,\varepsilon}$, that is, there is no separating sequence of $n_{j,v}$ and $n_{k,\varepsilon}$. Before the convergent graph and domains Φ are initialized by SPY-GENERATEREQUESTEDSUFFIXES and SPY-PROCESSIDENTIFIED, the number l of extra states is reset to 0. Notice that the function SPY-QUERYSEQANDCHECK is called on lines 6 and 9 with the third parameter set to false.

The third parameter of the function SPY-QUERYSEQANDCHECK prevents from a cyclic call of SPY-MAKESTATENODE with the same node $n_{i,u}$. Algorithm 44 describes the function SPY-QUERYSEQANDCHECK. It queries all symbols of the given sequence w one by one such that QUERY (Algorithm 32) is called only on symbols that are not already queried from the related node. After the entire sequence is queried from the given node, $n_{i,u}$ points to the reached leaf node of the observation tree. The observed response is analysed by the function SPY-UPDATEDOMAINS that updates all domains that are affected by the suffix of w which was just queried. If the processing of identified nodes is allowed (which is by default), then SPY-PROCESSIDENTIFIED is called.

Algorithm 44: SPY-QUERYSEQANDCHECK($n_{i,u}, w, fullCheck = true$)

```

1 for  $j \leftarrow 1$  to  $|w|$ ;  $w = x_1 \cdot \dots \cdot x_{|w|}$  do
2   if  $\bar{s}_i u x_j \notin T$  then
3      $\lfloor$  QUERY( $n_{i,u}, x_j$ ) and update  $[\bar{s}_i u x_j], \Phi([\bar{s}_i u x_j])$  accordingly
4      $\rfloor$  consider  $n_{i, u x_j}$  instead of  $n_{i,u}$ 
5 SPY-UPDATEDOMAINS( $n_{i,u}$ )
6 if  $fullCheck$  then SPY-PROCESSIDENTIFIED()

```

There is another way how to query a sequence from a particular node. It is defined by the function SPY-QUERYANDCHECKEACH in Algorithm 45. The difference from SPY-QUERYSEQANDCHECK is that SPY-QUERYANDCHECKEACH analyses the responses after the output to each input symbol is observed. It means that SPY-QUERYANDCHECKEACH is more flexible in terms of the minimization of the interaction with the black box. It stops querying immediately after an inconsistency is observed due to the analysis of the observed traces. It is used when a transition is to be identified or verified and no inconsistency is observed. In contrast, SPY-QUERYSEQANDCHECK is called when an observed inconsistency is to be resolved so that the response to the entire sequence is more desirable than the inconsistencies revealed by individual symbols. If a transition is to be verified, that is, $l > 0$, SPY-QUERYANDCHECKEACH also checks whether some of requested suffixes were queried (line 5 of Algorithm 45).

Both SPY-QUERYSEQANDCHECK and SPY-QUERYANDCHECKEACH call the function SPY-UPDATEDOMAINS to analyse the observed responses. It is defined in Algorithm 46. For the given leaf node n_u , the function checks all nodes n_v along the path u in the order from n_u to the root of the observation

Algorithm 45: SPY-QUERYANDCHECKEACH($n_{i,u}, w$)

```

1 for  $j \leftarrow 1$  to  $|w|$ ;  $w = x_1 \dots x_{|w|}$  do
2   if  $\bar{s}_i u x_j \notin T$  then
3     QUERY( $n_{i,u}, x_j$ ) and update  $[\bar{s}_i u x_j], \Phi([\bar{s}_i u x_j])$  accordingly
4     if  $l > 0$  then
5       check suffixes of  $\bar{s}_i u x_j$  if they are requested from the states
           reached by the related prefixes; mark ‘confirmed’ any
           transition that has all requested suffixes queried
6     SPY-UPDATEDOMAINS( $n_{i,u}$ )
7     SPY-PROCESSIDENTIFIED()
8     if an inconsistency is observed then return
9   consider  $n_{i,u x_j}$  instead of  $n_{i,u}$ 

```

tree. Let $u = v \cdot w$. If n_v is an SN of s_i , then s_i is eliminated from the domains ϕ of all nodes $n_{v'}$ that were consistent with $n_{i,\varepsilon}$ but now are separated by w . Moreover, if $[v']$ is in $\Phi([\bar{s}_i])$, it is removed from the domain and $[\bar{s}_i]$ is eliminated from $\Phi([v'])$. If n_v is not an SN, then the domain $\phi(v)$ is reduced such that every state s_i is removed from $\phi(v)$ if $n_{i,\varepsilon}$ was just separated from n_v by w . The domain $\Phi([v])$ and the domains of RNs are updated in the same way as in the case when n_v is an SN (lines 6–8 of Algorithm 46). Note that there cannot be an RN $[\bar{s}_k]$ in $\Phi([v])$ if s_k is not in $\phi(v')$ of a $v' \in [v]$. After any domain is updated, it is checked for an inconsistency or whether the related CN is identified in the case of Φ and $l = 0$. There are two types of inconsistency with respect to ϕ and one type with respect to Φ , see Section 15.2.2 for their description. A convergent node $[v]$ is identified if $\Phi([v])$ is singleton.

Algorithm 46: SPY-UPDATEDOMAINS(n_u)

```

1 foreach  $n_v$  such that  $u = v \cdot w$  do
2   if  $n_v$  is an SN, that is,  $n_v = n_{i,\varepsilon}$  then
3     remove  $s_i$  from the domains of all nodes that are separated by  $w$ 
4   else
5     remove from  $\phi(v)$  all states which SNs are separated from  $n_v$  by  $w$ 
6   foreach  $[v'] \in \Phi([v])$  do
7     if  $n_v$  and  $[v']$  are separated by  $w$  then
8       remove  $[v']$  from  $\Phi([v])$  and  $[v]$  from  $\Phi([v'])$ 
9 remember any node  $n_v$  with empty or inconsistent domain  $\phi(v)$ , and
   if  $l = 0$ , then any node  $n_v$  with CN domain  $\Phi([v])$  of size less than 2

```

Algorithm 47 describes the function SPY-IDENTIFYNEXTSTATE that is called in the first phase of the observation tree approach to identify the target state of a particular transition. It first makes sure that the transition is cap-

tured in the observation tree (line 1) and then basically follows Section 15.2.1 in the choice of separating sequence w . If the target state is identified immediately by its unique state output, that is, the domain Φ of the reached node $n_{i,ux}$ is singleton, then the input of the transition is repeated (line 3). Otherwise, an adaptive separating sequence would be chosen according to Section 15.2.1. Nevertheless, that procedure is used only if the domain Φ contains more than 2 RNs. It is because a separating sequence for the case of $|\Phi([\bar{s}_i ux])| = 2$ can be chosen much more easily, that is, without calculating the Distinguishing score. The shortest separating sequence is selected. In the end, the chosen sequence w is queried using SPY-QUERYANDCHECKEACH.

Algorithm 47: SPY-IDENTIFYNEXTSTATE($n_{i,u}, x$)

```

1 SPY-QUERYANDCHECKEACH( $n_{i,u}, x$ )
2 if  $n_{i,ux}.state$  is set then
3   |  $w \leftarrow x$  // attempt to reveal new information
4 else if  $\Phi([\bar{s}_i ux]) = \{[\bar{s}_j], [\bar{s}_k]\}$  then
5   |  $w \leftarrow$  the shortest separating sequence of  $n_{j,\varepsilon}$  and  $n_{k,\varepsilon}$ 
6 else
7   |  $w \leftarrow$  the shortest adaptive separating sequence that has the lowest
   |   probability of not identifying RNs in  $\Phi([\bar{s}_i ux])$ 
8 SPY-QUERYANDCHECKEACH( $n_{i,ux}, w$ )

```

The last part of the SPY-learner to describe is how it chooses which node should be extended. It is specified by the function SPY-GETNODETOEXTEND in Algorithm 48. It is called in both phases of the observation tree approach before a separating sequence to identify or verify a transition is chosen, in particular, it is called from the main learning cycle in Algorithm 40. The function receives the leaf node $n_{i,u}$ reached by the last queried sequence as a parameter. It first finds the predecessor $n_{i,u'}$ of $n_{i,u}$ that is the closest to $n_{i,u}$ and is proven to correspond to $n_{i,u'}.state$. If $n_{i,u'}$ is not equal to $n_{i,u}$, then a possible extension is checked with respect of the value of l . If $l = 0$, then there could be an adaptive separating sequence reducing the domain $\Phi([\bar{s}_i u'x])$ and starting with u'' where $u = u'xu''$. After such an adaptive separating sequence is queried (line 7), SPY-GETNODETOEXTEND returns null to notify the main learning cycle in Algorithm 40 that no other sequence is to be appended. It is similar in the case of $l > 0$ but requested suffixes are checked instead of selecting an adaptive separating sequence. Moreover, requested suffixes of the predecessors of $n_{i,u'}$ are checked as well (lines 10–12). A different extension of $n_{i,u}$ happens if the predecessor $n_{i,u'}$ is $n_{i,u}$, that is, $n_{i,u}$ is proven to correspond to a state s_k , and the following conditions hold. There needs to be a transfer sequence v from s_k such that v consists of verified transitions and there is either an unverified transition from $\delta_M^*(s_k, v)$ or a requested suffix for that reached state. In addition, v needs to be shorter than the shortest access sequence of a state that has unverified outgoing transition in order to be more efficient than resetting the black box and

querying another sequence. If all this holds, v is queried and $n_{i,uv}$ returned together with an input x that represents either the unverified transition or the prefix of the requested suffix for $\delta_M^*(s_k, v)$. A separating sequence can be thus chosen and appended to $n_{i,uv}$ (or $n_{i,uvx}$) in the main learning cycle. Note that if an inconsistency is observed in SPY-QUERYSEQANDCHECK and is not resolved by SPY-MAKESTATENODE, this information is propagated back to the main learning cycle in Algorithm 40 immediately. This is also the case when SPY-QUERYSEQANDCHECK is called in RESOLVEINCONSISTENCY. If no extension of $n_{i,u}$ happens in SPY-GETNODETOEXTEND, the function chooses a node and an input symbol x that should be extended next (lines 17–20 of Algorithm 48). If $l = 0$, then the state node $n_{j,\varepsilon}$ is returned such that \bar{s}_j is the shortest access sequence and there is unverified transition from s_j . If $l > 0$, the function calculates for each unverified transition $(s_{i'}, x)$ that leads to a state $s_{j'}$, an estimate of the number of queried symbols spent on the access sequences in order to verify the transition. The estimate is obtained as the number of requested suffixes multiplied by the length of the related access sequence $\bar{s}_{i'}$ or $\bar{s}_{j'}$. The transition $(s_{i'}, x)$ that has the smallest estimate is assumed to be verified with the least effort, therefore, the related state node $n_{i',\varepsilon}$ and x are returned.

The space complexity of the SPY-learner can be estimated by the size of the observation tree T and the size of the convergent graph. The size of T denoted by $|T|$ was derived in Section 10.1 to be in $O(n^3 p^{l+1})$. The size of the convergent graph with the domains Φ is $O(n^4 p^l)$ as was derived in Section 11.1. Hence, the worst case space complexity of the SPY-learner is $O(n^4 p^l + n^3 p^{l+1})$. The SPY-learner spends most of the time by comparing the observed traces in the functions SPY-UPDATEDOMAINS and SPY-PROCESSIDENTIFIED. Although SPY-UPDATEDOMAINS deals with the domains Φ of convergent nodes, its worst case running time is the same as for H-UPDATEDOMAINS, that is, $O(n^2 |T|)$. It is because of the assumption that there is not a common sequence much longer than n that would not be a separating sequence of the compared nodes. It means that either there is a separating sequence of length in $O(n)$ or the corresponding suffix is not queried from the node with which n_v is compared in H-UPDATEDOMAINS. As in the case of H-UPDATEDOMAINS, SPY-UPDATEDOMAINS could be called $|T|$ times. The function SPY-PROCESSIDENTIFIED compares traces when a CN $[v]$ is to be merged in an RN $[\bar{s}_k]$. The size of the domain Φ of an RN is bounded by the size of T and $[v]$ can be compared with $[\bar{s}_k]$ on all successors such that their number is also bounded by $|T|$. Therefore, reducing of $\Phi([\bar{s}_k])$ by $[v]$ is done in $O(|T|^2)$. Note that this estimate cannot be reached as the domain Φ of an RN never contains all CNs, the CNs are not compared on all convergent nodes, and the convergent graph is most of the time smaller than T . In the worst case, each CN representing a single node of the OTree is merged in an RN and this is done n times as the convergent graph is reset to be like the OTree after a new state is revealed. It means that in total the domain Φ of an RN is checked $n \cdot |T|$ times in SPY-PROCESSIDENTIFIED. Thus, the

Algorithm 48: SPY-GETNODETOEXTEND($n_{i,u}$)

```

1 let  $n_{i,u'}$  be the closest predecessor of  $n_{i,u}$  such that  $n_{i,u'}.state$  is set
2 let  $s_j$  be the state with the shortest access sequence  $\bar{s}_j$  such that there is
  an unverified transition from  $s_j$ 
3 if  $u \neq u'$  then
4   if  $l = 0$  then
5     let  $u = u' \cdot x \cdot u''$ 
6     if there is an adaptive separating sequence  $w$  of RNs in  $\Phi([\bar{s}_i u' x])$ 
       such that  $w$  starts with  $u''$  then
7       SPY-QUERYANDCHECKEACH( $n_{i,u'x}, w$ )
8       return (null,  $\uparrow$ )
9   else
10    if there is a requested suffix  $w$  for  $n_v.state$  such that  $v$  is a prefix
      of  $\bar{s}_i u'$ ,  $vv' = \bar{s}_i u$  and  $w$  starts with  $v'$  then
11      SPY-QUERYANDCHECKEACH( $n_v, w$ )
12      return (null,  $\uparrow$ )
13 else // the leaf is identified, let  $s_k$  be  $n_{i,u}.state$ 
14   if there is a transfer sequence  $v$  of verified transitions from  $s_k$  such
     that  $|v| < |\bar{s}_j|$  and there is an unverified transition  $(\delta_M^*(s_k, v), x)$  or
     a requested suffix starting with  $x$  for state  $\delta_M^*(s_k, v)$  then
15     SPY-QUERYSEQANDCHECK( $n_{i,u}, v$ )
16     return ( $n_{i,uv}, x$ )
17 if  $l = 0$  then return ( $n_{j,\varepsilon}, x$ ) where  $x$  is unverified from  $s_j$ 
18 foreach unverified transition  $(s_{i'}, x)$  such that  $s_{j'} = \delta_M(s_{i'}, x)$  do
19   calculate its value as  $|\bar{s}_{i'}| \cdot r_{i'} + |\bar{s}_{j'}| \cdot r_{j'}$  where  $r_{i'}, r_{j'}$  are the numbers
     of requested suffixes for  $s_{i'}, s_{j'}$  to verify the transition  $(s_{i'}, x)$ 
20 return ( $n_{i',\varepsilon}, x$ ) related to  $(s_{i'}, x)$  with the minimal value

```

worst case time complexity of the SPY-learner is $O(n^2|T|^2 + n|T|^3)$ which is $O(n^8 p^{2l+2} + n^{10} p^{3l+3})$.

15.4.2 Running Example

This section describes how the SPY-learner learns the deterministic finite automaton defined in Figure 15.5. The setting is the same as for the H-learner (Section 15.3.2), that is, learning with 1 extra state is considered and the teacher provides output queries (OQ) so that the learner can query symbols of a sequence one by one.

The learning starts with the reset of the black box and obtaining the state output of the initial state; output query 1 is $T(\varepsilon, \uparrow)$. The first symbol of the input alphabet is then queried and as the reached state is immediately identified as the initial state, the symbol repeated; output queries 2 and 3 are $T(a)$. The conjectured model thus contains only the initial state s_0 and

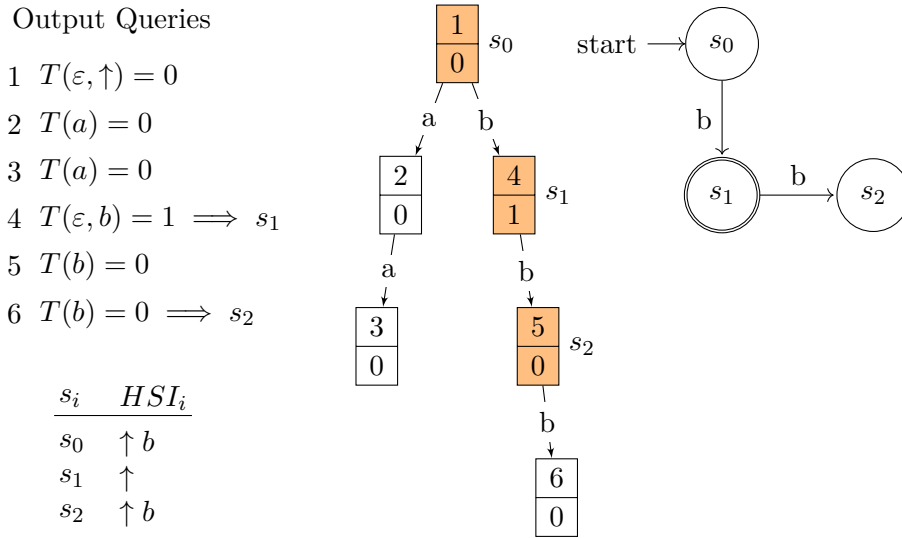


Figure 15.13: The first 6 output queries reveal 3 states of the black box

its self-loop on input ‘a’. All three nodes of the observation tree, $n_{0,\varepsilon}$, $n_{0,a}$ and $n_{0,aa}$, are assumed to be convergent. Hence, the leaf node $n_{0,aa}$ that corresponds to the current state of the black box is considered as its own closest identified predecessor $n_{i,u'}$ in the function `SPY-GETNODETOEXTEND`. The branch on lines 13–16 of Algorithm 48 is thus processed. There is just one unverified transition and that is (s_0, b) . As there is not a transfer sequence v of verified transitions that would be shorter than the access sequence of s_0 which is 0, $(n_{0,\varepsilon}, b)$ is returned by `SPY-GETNODETOEXTEND`. `SPY-IDENTIFYNEXTSTATE` then calls `SPY-QUERYANDCHECKEACH` so that the black box is reset and ‘b’ is queried; output query 3 is $T(\varepsilon, b)$. The output 1 makes the domain $\phi(b)$ empty and so `SPY-MAKESTATENODE` is called from `SPY-PROCESSIDENTIFIED`. As the parent of $n_{0,b}$ is a state node, lines 2–14 of Algorithm 43 are skipped. The new state is added to the conjectured model as s_1 and $n_{0,b}$ becomes the related state node $n_{1,\varepsilon}$. The harmonized state identifiers of both states contain only the separating sequence \uparrow . `SPY-PROCESSIDENTIFIED` then merges $n_{0,a}$ into $n_{0,\varepsilon}$ because $\phi(a)$ contains just s_0 and so $\Phi([a]) = \{n_{0,\varepsilon}\}$. The transition (s_0, a) is thus verified to lead to s_0 . Both transitions from s_1 are the only ones unverified. The learner then returns back to the main cycle in Algorithm 40 (without finishing `SPY-IDENTIFYNEXTSTATE`). Note that this immediate return is done in any call of `SPY-QUERYANDCHECKEACH` if an inconsistency or a new state is revealed during its processing.

`SPY-GETNODETOEXTEND` chooses to extend the current leaf node $n_{1,\varepsilon}$ with the unverified symbol ‘b’; ‘b’ is preferable because it leads to $n_{1,\varepsilon} = n_b$. Hence, `SPY-IDENTIFYNEXTSTATE` calls `SPY-QUERYANDCHECKEACH` and the output 0 identifies $n_{1,b}$ as s_0 . The small optimization feature described in the end of Section 15.2.1 takes place here again as ‘b’ is then chosen (line 3 of Algorithm 47) to be queried once again; output query 6 is $T(b)$. The

output 0 does not match the expected 1 produced by s_0 on ‘b’. It means that the domain $\phi(bb)$ becomes empty because $n_{0,\varepsilon}$ and $n_{1,b}$ are now separated by ‘b’ and so SPY-UPDATEDOMAINS eliminates s_0 from the domain. SPY-MAKESTATENODE makes $n_{1,b}$ a new state node $n_{2,\varepsilon}$, adds s_2 to the conjectured model and extends domains of ‘a’, ‘aa’, ‘bb’ and ‘bbb’ with s_2 . The current observation tree, conjectured model and updated harmonized state identifiers are shown along with the related output queries in Figure 15.13. No successor of a state node is consistent with just one state node, therefore, all four transitions that are not covered by the state cover $\{\uparrow, b, bb\}$ are unverified.

The variable $nodeBB$ points to $n_{2,b}$ when SPY-GETNODETOEXTEND is called from the main learning cycle. As $\phi(bbb) = \{s_0, s_2\}$ and so $n_{2,b}$ is not identified, $n_{2,\varepsilon}$ is considered as the closest identified predecessor in SPY-GETNODETOEXTEND and the branch on lines 4–8 of Algorithm 48 is processed. There is an adaptive separating sequence ‘b’ that can reduce the domain $\Phi([bbb])$ that corresponds to $\phi(bbb)$ as $[bbb]$ contains only the node $n_{2,b}$. Hence, ‘b’ is queried from the leaf node $n_{2,b}$ (without resetting the black box). SPY-GETNODETOEXTEND returns (null, \uparrow) so that no other sequence is queried before the function is called again in the next round of the main learning cycle. The output 1 on ‘b’ identifies $n_{2,b}$ as s_0 , hence, $n_{2,b}$ is merged in $[\varepsilon]$ in SPY-PROCESSIDENTIFIED. As a consequence, the leaf node $n_{2,bb}$ is merged in $[b]$ which means that it is identified as s_1 . There is an unverified transition from the initial state s_0 , therefore, SPY-GETNODETOEXTEND returns $(n_{0,\varepsilon}, a)$. The black box is reset in order to query the separating sequence ‘b’ of s_0 and s_2 that are consistent with the target state of (s_0, a) represented by the node $n_{0,a}$; output query 8 is $T(a, b)$. The transition is identified to lead to s_0 as the response to ‘b’ is 1. The nodes $n_{0,a}$ and $n_{0,aa}$ are merged in $[\varepsilon]$ and the leaf node $n_{0,ab}$ is merged to the reference node of s_1 , that is, $[b]$. In the next round of the main learning cycle, SPY-GETNODETOEXTEND chooses ‘a’ to extend the current leaf node $n_{0,ab}$ because it is an unverified transition from the corresponding state s_1 and the shortest access sequence of a state with an unverified transition is $\bar{s}_1 = b$. It means that it would require more symbols to query before the transition could be tested and this way it can be tested immediately; output query 9 is $T(a)$. The separating sequence ‘b’ is then queried and the output 1 identifies the target state of (s_1, a) as s_0 . It is similar with the last unverified transition (s_2, a) . The access sequence of s_2 is ‘bb’ but s_2 can be reached by the verified transition (s_1, b) from the current leaf node $n_{0,abab}$ that is identified as s_1 . According to line 15 of Algorithm 48, ‘b’ is queried to reach a node that should correspond to s_2 ; output query 11 is $T(b)$. The target state of the unverified transition responds with 0, that is, $T(a) = 0$, so that the separating sequence ‘b’ is appended to identify the state to correspond to s_0 . At this point, the observation tree is closed for 0 extra states so that the conjectured model is completely specified. Both structures are shown in Figure 15.14. The number l of extra states is increased to 1 and the SPY-learner enters the second phase of the observation tree approach. Figure 15.14 also captures the harmonized state identifiers and all requested suffixes that are needed to verify the transitions that are not

covered by the state cover \bar{S} . The requested suffixes are based on the number l of extra states and the HSIs as defines SPY-GENERATEREQUESTEDSUFFIXES in Algorithm 41. For example, the transition (s_2, b) leads to s_0 , therefore, the requested suffixes for s_0 contains HSI_0 which is just ‘b’. As $l = 1$, it also contains ‘a’ and ‘b’ extended with the HSI of the reached state, that is, s_0 leads to itself on ‘a’ so that ‘ab’ is in the related requested suffixes and on ‘b’ it leads to s_1 that is identified by the state output so that just ‘b’ is sufficient. The requested suffixes for the origin state of a transition are the requested suffixes for the target state prepended with the related input, that is, ‘bab’ and ‘bb’ are the requested suffixes for s_2 with respect to the transition (s_2, b) . Note that the stout symbol is not shown in the requested suffixes as the state output is obtained on the input that reaches the particular state. The requested suffixes that are already queried are strikeout in Figure 15.14.

The first requested suffix that is queried is ‘aab’ for s_0 ; output query 14 is $T(aa, b)$. The observed output 0 does not match the expected one that would produce state $\delta_M^*(s_0, aab)$. Therefore, l is set to 0 after a new state node $n_{3,\varepsilon}$ is created from $n_{0,a}$ and the conjectured model, the convergent graph and domains are updated. The node $n_{0,a}$ was identified as s_0 , hence, just HSI_0 is updated with the separating sequence ‘ab’ of states s_0 and s_3 . The updated HSIs are shown in Figure 15.15. The following 9 output queries identify the target states of unverified transitions and allow to construct a completely specified conjectured model with 4 states as also captured in Figure 15.15. First, output queries 15 and 16 defines the transition (s_2, b) as $n_{3,a}$ is identified as s_2 because of its unique response to ‘ $\uparrow b$ ’. Then, the black box is reset and transferred to the state corresponding to s_1 in order to verify the transition on ‘a’ by the separating sequence ‘ab’. Note that the target state of (s_1, a) could correspond to s_0 or s_3 because of the response to ‘b’ (output query 10). The last unverified transition is from s_2 on ‘a’. As s_2 can be reached from the current leaf node $n_{1,aab}$ only by single ‘b’ which is less than the length of its access sequence ‘bb’, the SPY-learner queries ‘b’ and then identifies (s_2, a) using ‘ab’; output queries 20–23. After 23 output queries, 5 resets and 26 symbols queried in total, the conjectured model matches the black box so that the learning would stop if an equivalence query was asked. Nevertheless, the SPY-learner aims to provide the guarantee of 1 extra state. Hence, it generates requested suffixes for each state to verify the transitions that are not covered by the state cover $\{\varepsilon, a, b, bb\}$. The requested suffixes are shown on the bottom left of Figure 15.15; the strikeout ones are already captured in the observation tree.

The second phase of the observation tree approach in the case of the SPY-learner is captured in terms of the final observation tree in Figure 15.16. It starts with SPY-GETNODETOEXTEND that chooses which transition should be verified first. The transitions are compared according to their values that are calculated based on the number of requested suffixes and the length of the access sequences of the related states (lines 18–20 of Algorithm 48). For instance, (s_1, a) has 2 requested suffixes for s_1 that is reach by ‘b’ so that its value is $1 \cdot 2 = 2$. The transition (s_3, b) has the value of 3 because there are 2

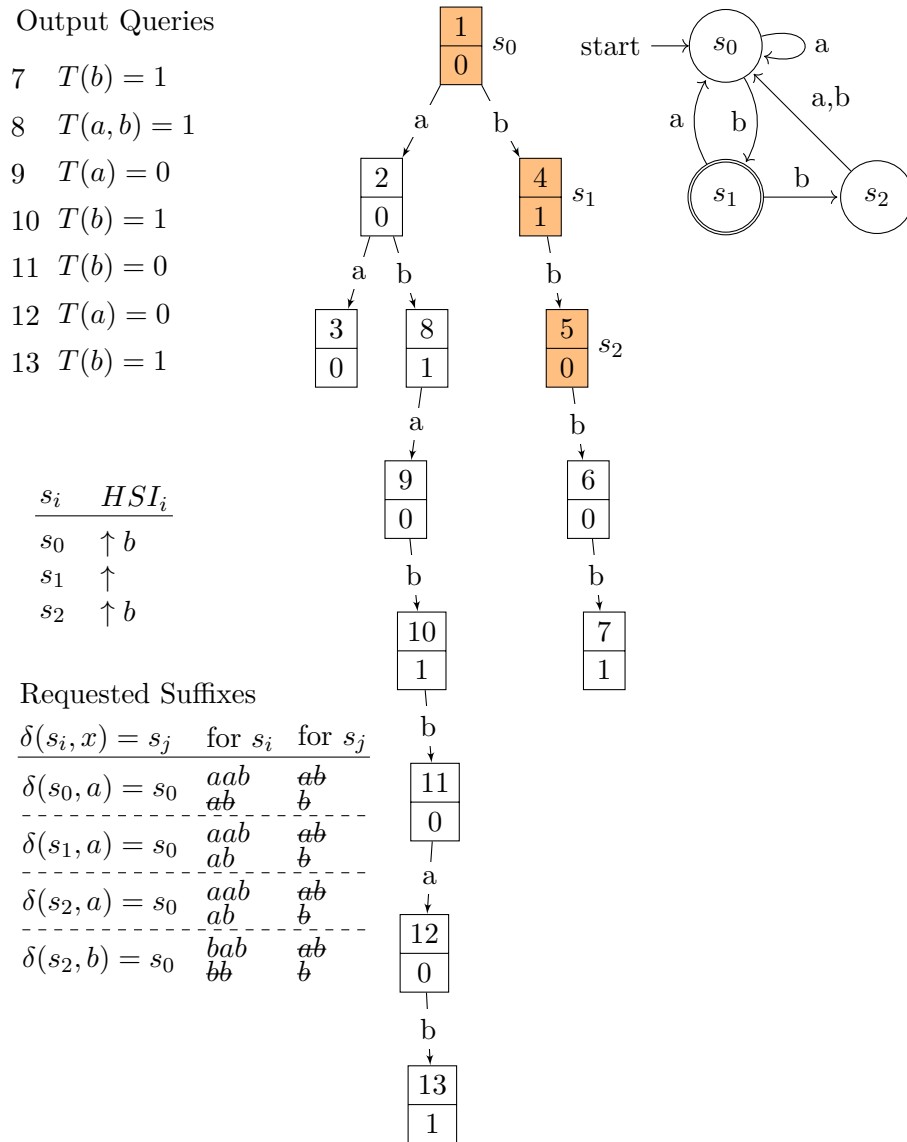


Figure 15.14: The next 7 output queries complete a 3-state conjectured model

requested suffixes for s_3 and one requested suffix for the target state s_1 , that is, $1 \cdot 2 + 1 \cdot 1 = 3$. The lowest value is 2 and the first processed transition with this value is (s_1, a) , hence, it is selected to be verified first. Note that the transition (s_2, b) has also the value of 2 but its value is calculated after the one for (s_1, a) . After both ‘aaab’ and ‘ab’ are queried from $n_{1,\varepsilon}$ (output queries 24–26), the values of unverified transitions are calculated again because some requested queries could be captured in the observation tree after $n_{1,a}$ is merged in $[\varepsilon]$. Unfortunately, this is not the case so that (s_2, b) is chosen to be verified next. It leads to s_3 and thus after (s_2, b) is verified by ‘bab’, the last queried sequence is extended with ‘b’ to eliminate the requested suffix ‘abb’ for s_3 (output queries 27–29). The transition (s_3, b) is the next to be verified

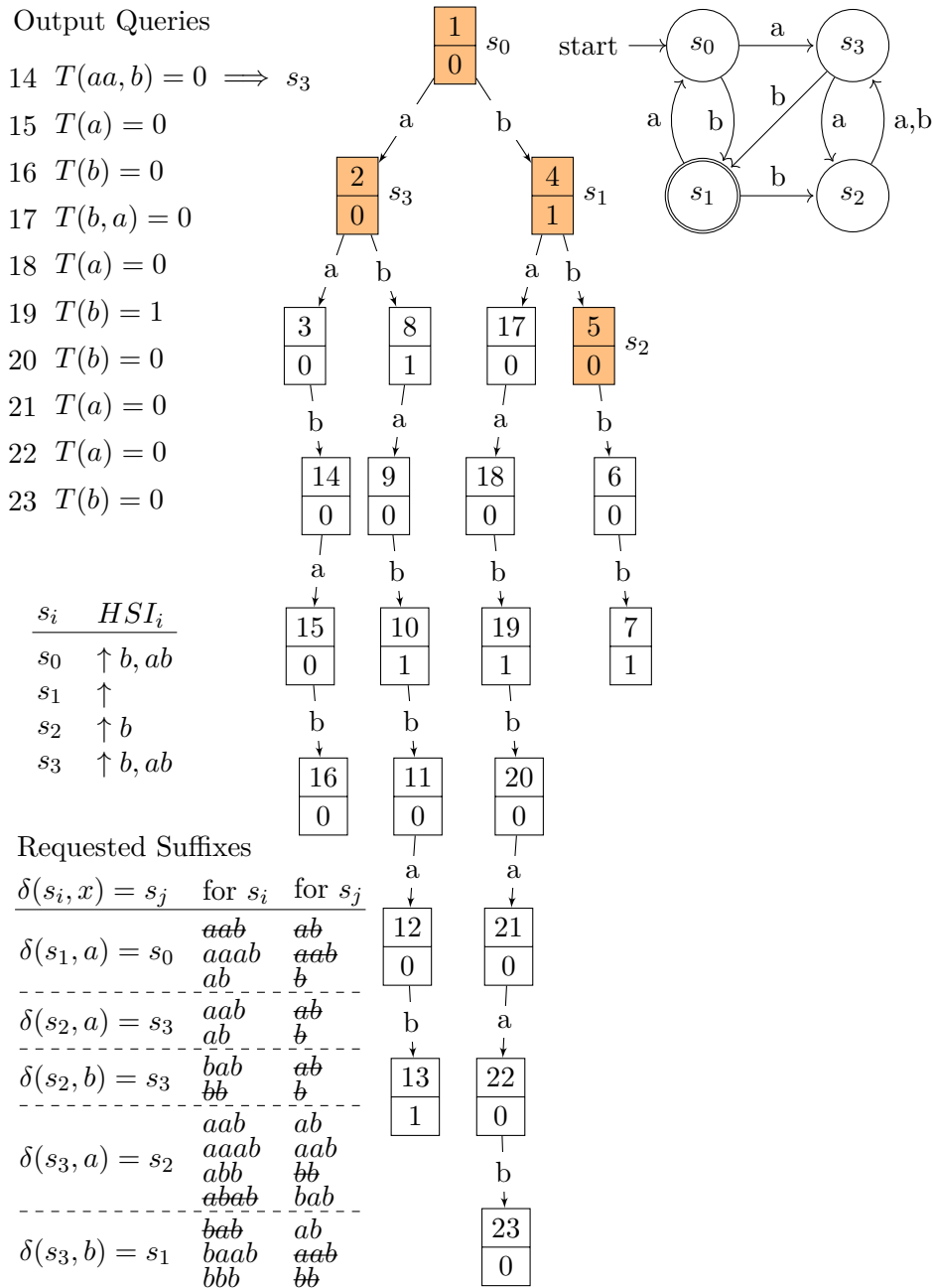


Figure 15.15: The next 10 output queries complete the 4-state DFA

as there are only two related requested suffixes, ‘baab’ and ‘bbb’, for s_3 . After both are queried (output queries 30–33), not just (s_3, b) is verified but also (s_2, a) . It is because the requested suffixes, ‘aab’ and ‘ab’, are queried from $n_{1,aab}$ and $n_{2,babb}$ (nodes 20 and 11 in Figure 15.16) and these nodes are merged to the RN of s_2 after (s_3, b) is verified. As the current leaf node $n_{2,bbb}$ (node 33) is identified as s_3 , the requested suffix ‘aaab’ of the last unverified

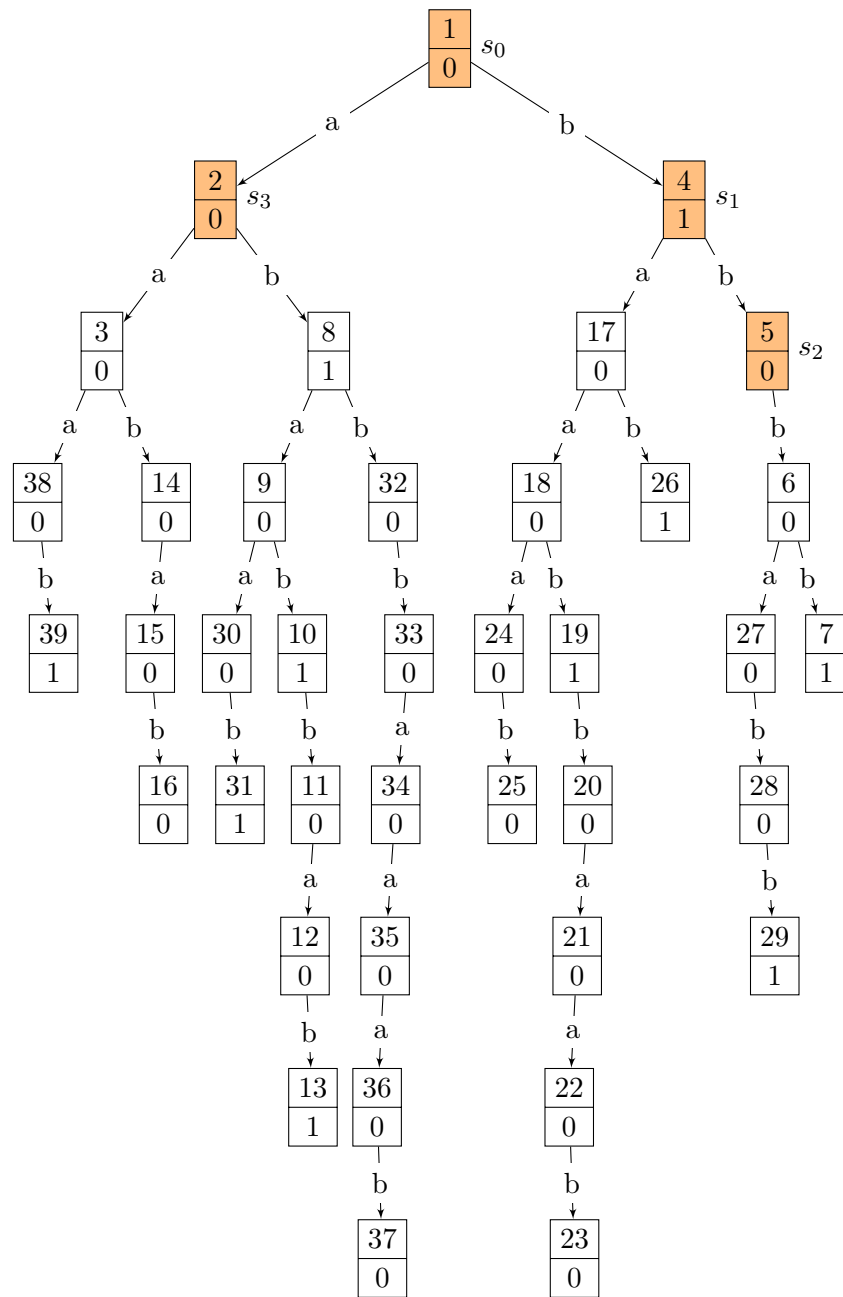


Figure 15.16: Observation tree closed for 1 extra state by the SPY-learner

transition (s_3, a) can be immediately queried. The learning is finished by the last requested suffix 'aab' queried from $n_{2,\varepsilon}$.

The SPY-learner learns the 4-state DFA using 39 output queries, 11 resets of the black box and 57 symbols queried in total and it provides a guarantee that the model is correct or the black box has more than 5 states. The conjectured model was completely specified and with 4 states just after 23 output queries, 5 resets and 26 queried symbols.

15.5 S-learner

A new learning algorithm called the S-learner is proposed in this section including two subsections that describe its implementation and explain how it learns on a running example. The S-learner implements the observation tree approach but instead of an adaptation of testing theory, it uses a testing method directly in the second phase of the approach. The testing method is the S-method (Chapter 11) and thus the S-learner aims to meet the requirements of the S-condition (Theorem 8.14).

The S-learner is very similar to the SPY-learner as it utilizes the convergence of sequences to construct a completely specified conjectured model faster. However, it does not deal with requested suffixes or harmonized state identifiers because the S-method can handle the second phase of the observation tree approach. The S-method can extend the given observation tree T (which is a testing tree in the testing terminology) such that it returns test sequences that are not covered by T but together with T they form an $(n+l)$ -complete test suite for the given machine M with n states and the given number l of extra states. Such additional test sequences are called *requested queries* from the perspective of the S-learner. Note that the S-method can plan and optimize which sequences are to be extended for different reasons and so it saves some symbols and resets compared to the SPY-learner that may extend a single sequence several times.

The way how the number l of extra states is handled is changed compared to both the H- and SPY- learners. When a new state node is revealed, l is reset to 0 which is the same for all the learners. However, after a completely specified conjectured model is constructed, l is set to ES that represents the maximal number of extra state that was considered so far. The reason is that once it was considered ES extra states to reveal a new state node, it is likely that this number will be needed to reveal another state. In addition, a test suite for $l + 1$ extra states usually extends test sequences of a test suite for l extra states, that is, an $(n + l + 1)$ -complete test suite usually includes an $(n + l)$ -complete test suite. Therefore, if test sequences of the $(n + l + 1)$ -complete test suite are queried directly without querying test sequences of the $(n + l)$ -complete test suite before, then several resets of the black box and many symbols are saved. Moreover, the S-method can optimize $(n + l + 1)$ -complete test suite based on a smaller T that does not contain $(n + l)$ -complete test suite, in a different way due to the convergence of sequences. Why not consider the given number $maxExtraStates$ of maximal extra states as l directly? There are two reasons. First, the size of an $(n + l)$ -complete test suite grows exponentially in the number l so that an $(n + l + 1)$ -complete test suite has roughly p -times more sequences than an $(n + l)$ -complete test suite where p is the number of input symbols. Hence, the learner could spend a lot of time querying longer sequences before it would query a sequence that reveals an inconsistency even though this sequence would be short and included in a test suite for a less number of extra states.

Second, the user can provide the number *maxExtraStates* greater than it is needed to reveal all states without an equivalence query.

The use of a testing method may seem like a combination of a standard learning algorithm and a teacher that implements equivalence queries using a testing method. However, the difference from the standard learning algorithms was discussed in Section 15.1 and is summed up as follows. First, the S-method creates a test suite as an extension of the observed traces. Second, the S-learner has the responses to the test sequences that are queried before an inconsistency is observed. Both differences usually do not hold in the setting of a standard learning algorithm and EQ approximated using a testing method.

■ 15.5.1 Implementation

The S-learner is implemented in the same way as the SPY-learner. Its implementation is proposed in Algorithms 49–53 with references to the functions of the SPY-learner that are identical to the one of the S-learner. It maintains the observation tree T , the conjectured model M and the convergent graph. The main learning cycle is a little changed as the S-learner handles the second phase of the observation tree approach differently.

Algorithm 49 describes the main learning cycle that is controlled by the boolean variable *learnt*. Before the learning starts, l , ES and the structures are initialized and all transitions from the initial state s_0 are marked ‘unverified’. The main learning cycle thus starts with verification of transitions by identification of their target states (lines 9–11). After all transitions are verified, l is increased (lines 15–16) and the S-method generates the requested queries (lines 26–27) if l did not reach the given *maxExtraStates*. The requested queries are then gradually queried from the longest one (line 13). After all requested queries are queried, the observation tree is closed for l extra states and l is increased again. If *maxExtraStates* is reached, then the learning stops unless a counterexample is obtained. In such a case, the counterexample is queried and processed (line 22) which results in an inconsistency. Any inconsistency observed other than I means a new state node is processed by the function `RESOLVEINCONSISTENCY` (Algorithm 33). The calls of `QUERYSEQANDCHECK` in `RESOLVEINCONSISTENCY` are implemented by `S-QUERYSEQANDCHECK`. Note that an inconsistency can appear both during the identification (lines 10–11) and during the verification (line 13). In addition, the learning can be stopped by the user (line 28).

The function `S-QUERYSEQANDCHECK` is similar to its version in the SPY-learner. The only difference is that the response to each input is compared with the expected output if the function is called to query a requested query (a call on line 13 of Algorithm 49). If the outputs differ, then the function stops querying. The queried suffix is then analysed using `SPY-UPDATEDOMAINS` defined in Algorithm 46. The function `S-PROCESSIDENTIFIED` is called in the end if the call of `S-QUERYSEQANDCHECK` is not from `S-MAKESTATENODE` that sets the third argument to false; the argument *fullCheck* prevents from an infinite loop by calling `S-MAKESTATENODE` on the same node.

Algorithm 49: S-learner

```

input : A teacher
input :  $maxExtraStates$ 
output: A conjectured model  $M$ 

1  $l \leftarrow 0$  // the number of extra states currently considered
2  $ES \leftarrow 1$  // the maximal number of extra states considered so far
3 initialize the OTree  $T$  with the root  $n_{0,\varepsilon}$  (the initial state  $s_0$  of  $M$ )
4 mark all transitions from  $s_0$  ‘unverified’
5  $learnt \leftarrow false$ 
6 while not  $learnt$  do
7   if  $T$  and  $M$  are inconsistent then
8     | RESOLVEINCONSISTENCY( $T, M$ )
9   else if there is an unverified transition then //  $l = 0$ 
10    |  $(n_{i,u}, x) \leftarrow S\text{-GETNODETOEXTEND}(nodeBB)$ 
11    | if  $n_{i,u}$  is not null then S-IDENTIFYNEXTSTATE( $n_{i,u}, x$ )
12   else if there is a requested query  $w$  then //  $l > 0$ 
13    | S-QUERYSEQANDCHECK( $n_{0,\varepsilon}, w$ ) where  $w$  is the longest one
14   else
15     | if  $l > 0$  then  $ES \leftarrow ES + 1$ 
16     |  $l \leftarrow ES$ 
17     | if  $l > maxExtraStates$  then
18       | if EQs are allowed then
19         | if there is a counterexample  $w$  obtained on EQ then
20           |  $l \leftarrow 0$ 
21           | if  $ES > 1$  then  $ES \leftarrow ES - 1$ 
22           | S-QUERYSEQANDCHECK( $n_{0,\varepsilon}, w$ )
23           | continue
24     |  $learnt \leftarrow true$ 
25     | else
26       |  $T' \leftarrow S\text{-method}(M, l, T)$ 
27       | get requested queries as  $T' \setminus T$ 
28   if user is satisfied with  $M$  then  $learnt \leftarrow true$ 
29 return the conjecture  $M$ 

```

Algorithm 50: S-QUERYSEQANDCHECK($n_{i,u}, w, fullCheck = true$)

```

1 for  $j \leftarrow 1$  to  $|w|$ ;  $w = x_1 \dots x_{|w|}$  do
2   | if  $\bar{s}_i u x_j \notin T$  then
3     | QUERY( $n_{i,u}, x_j$ ) and update  $[\bar{s}_i u x_j], \Phi([\bar{s}_i u x_j])$  accordingly
4     | if  $w$  is a requested query and outputs differ then break
5     | consider  $n_{i,u x_j}$  instead of  $n_{i,u}$ 
6 SPY-UPDATEDOMAINS( $n_{i,u}$ )
7 if  $fullCheck$  then S-PROCESSIDENTIFIED()

```

Algorithm 51 specifies the function S-PROCESSIDENTIFIED that is like its version in the SPY-learner. The only difference is that S-PROCESSIDENTIFIED does not contain the case of $l > 0$ because it is not needed. The convergent graph remains to look like the conjectured model when l is increased. The S-learner does not need to keep track of which sequences are proven to be convergent as it can guarantee that the observation tree is closed for l extra states when all requested queries are queried. Until then, the requested queries can be imagined as possible counterexamples so that if there is an inconsistency, the exploration stops and the inconsistency is resolved.

Algorithm 51: S-PROCESSIDENTIFIED()

```

1 if there is  $n_{i,u}$  with empty domain  $\phi(\bar{s}_i u)$  then
2   | S-MAKESTATENODE( $n_{i,u}$ )
3 else if  $l = 0$  and  $M$  and  $T$  are consistent then
4   | foreach  $n_{i,u}$  such that  $\Phi(\bar{s}_i u) = \{\bar{s}_j\}$  do
5     | merge  $[\bar{s}_i u]$  into  $[\bar{s}_j]$  and then recursively their successors
6     | - during merge: update  $\Phi$ , check for inconsistency, mark ‘verified’
        |   any transition that starts and ends in RNs - add it to  $M$ , and
        |   each  $n_v.state$  gets  $s_k$  if  $[v]$  merged into the RN  $[\bar{s}_k]$ 

```

The function S-MAKESTATENODE is also similar to the one in the SPY-learner but the S- version has less work to do as it does not have to update harmonized state identifiers. At first, a suitable node $n_{i,u}$ is found such that it needs to have empty domain and its parent is a state node (lines 2–14 of Algorithm 52). Then, it is possible to make a new state node from $n_{i,u}$. The corresponding new state s_k is added to the conjectured model (line 15) and also to all domains ϕ of nodes that are consistent with the new state node $n_{k,\varepsilon}$ (line 16). The convergent graph is initialized to copy the observation tree and the transitions not covered by the state cover \bar{S} are marked ‘unverified’. The target states of some of these transitions can already be identified. Hence, S-PROCESSIDENTIFIED is called in the end. Note that the same initialization of the convergent graph and domains Φ is also done in the SPY-learner but inside the function SPY-GENERATEREQUESTEDSUFFIXES.

The function S-IDENTIFYNEXTSTATE is the same as SPY-IDENTIFYNEXTSTATE defined in Algorithm 47 but the calls of SPY-QUERYANDCHECKEACH on lines 1 and 8 are replaced with the S- version. The S- version of SPY-QUERYANDCHECKEACH is also defined by the SPY- version. There are two changes to Algorithm 45 that describes SPY-QUERYANDCHECKEACH. The S- version is not called when $l > 0$, therefore, lines 4 and 5 of Algorithm 45 can be omitted. The call of SPY-PROCESSIDENTIFIED on line 7 is replaced by S-PROCESSIDENTIFIED. It is similar in the case of S-GETNODETOEXTEND that is called only if $l = 0$. It means that Algorithm 53 describes the function S-GETNODETOEXTEND like SPY-GETNODETOEXTEND (Algorithm 48) without the parts relating to the case when $l > 0$. There is a small change. If the given leaf node is identified as a state s_k and there is an unverified transition

Algorithm 52: S-MAKESTATENODE($n_{i,u}$)

```

1 let  $n_{i,u'}$  be the parent of  $n_{i,u}$  and  $u = u' \cdot x$ 
2 if  $n_{i,u'}$  is not an SN, that is,  $u' \neq \varepsilon$  then
3   if  $\phi(\bar{s}_i u')$  is empty then return S-MAKESTATENODE( $n_{i,u'}$ )
4   while  $\phi(\bar{s}_i u')$  is not empty do
5     let  $s_j$  be  $n_{i,u'}.state$  if it is set and it is in  $\phi(\bar{s}_i u')$ , otherwise let  $s_j$ 
6     be any state in  $\phi(\bar{s}_i u')$ 
7     S-QUERYSEQANDCHECK( $n_{j,\varepsilon}, x, false$ )
8     foreach  $s_k \in \phi(\bar{s}_j x)$  do
9        $w \leftarrow$  separating sequence of  $n_{i,u}$  and  $n_{k,\varepsilon}$ 
10      S-QUERYSEQANDCHECK( $n_{j,x}, w, false$ )
11      if  $s_j \notin \phi(\bar{s}_i u')$  then break
12    if  $\phi(\bar{s}_j x)$  or  $\phi(\bar{s}_i u')$  is empty then
13      return S-MAKESTATENODE( $n_{j,x}$  or  $n_{i,u'}$  respectively)
14    else if  $s_j = n_{i,u'}.state$  then
15      return // inconsistent domain  $\phi(\bar{s}_i u')$ 
16
17 add a new state  $s_k$  to the conjecture  $M$ ;  $n_{i,u} = n_{k,\varepsilon}$ 
18 add  $s_k$  to the domains of all nodes that are not separated from  $n_{k,\varepsilon}$ 
19  $l \leftarrow 0$ 
20 foreach transition  $(s_j, x)$  not covered by  $\bar{S}$  do
21   generate a part of convergent graph covering the subtree of  $n_{j,x}$ 
22   initialize  $\Phi$  by  $\phi$  for each node of the subtree
23   mark  $(s_j, x)$  unverified
24 S-PROCESSIDENTIFIED()

```

from this state, then the node is immediately returned along with the input x of the unverified transition (line 8). The other part of the function correspond to the ones in SPY-GETNODETOEXTEND. The function tries to extend the leaf node $n_{i,u}$ with an adaptive separating sequence if $n_{i,u}$ is not identified as a particular state. It tries to extend $n_{i,u}$ with a ‘short’ transfer sequence v leading to a state with an unverified transition if the leaf node is identified. Otherwise, it returns the state node that is reached by the shortest access sequence and has an unverified transition leading from the corresponding state.

There are several places where the S-learner can be optimized in terms of the interaction with the black box. One such optimization is to analyse requested queries and query first those that are more likely to reveal an inconsistency. This could be done based on the conjectured model and the observed traces.

The S-learner is almost the same as the SPY-learner. The worst case space complexity is $O(n^4 p^l + n^3 p^{l+1})$ as derived in Section 15.4.1. Note that both the observation tree and the convergent graph are shared with the S-method so that there is no increase in the space complexity. The functions SPY-

Algorithm 53: S-GETNODETOEXTEND($n_{i,u}$)

```

1 let  $n_{i,u'}$  be the closest predecessor of  $n_{i,u}$  such that  $n_{i,u'}.state$  is set
2 let  $s_j$  be the state with the shortest access sequence  $\bar{s}_j$  such that there is
   an unverified transition from  $s_j$ 
3 if  $u \neq u'$  then
4   | let  $u = u' \cdot x \cdot u''$ 
5   | if there is an adaptive separating sequence  $w$  of RNs in  $\Phi([\bar{s}_i u' x])$ 
6   |   | such that  $w$  starts with  $u''$  then
7   |   |   | SPY-QUERYANDCHECKEACH( $n_{i,u'x}, w$ )
8   |   |   | return (null,  $\uparrow$ )
9 else // the leaf is identified, let  $n_{i,u}.state$  be  $s_k$ 
10  | if there is an unverified transition  $(s_k, x)$  then return ( $n_{i,u}, x$ )
11  | if there is a sequence  $v$  of verified transitions from  $s_k$  such that
12  |   |  $|v| < |\bar{s}_j|$  and there is an unverified transition  $(\delta^*(s_k, v), x)$  then
13  |   |   | S-QUERYSEQANDCHECK( $n_{i,u}, v$ )
14  |   |   | return ( $n_{i,uv}, x$ )
15 return ( $n_{j,\varepsilon}, x$ ) where  $x$  is unverified from  $s_j$ 

```

UPDATEDDOMAINS and S-PROCESSIDENTIFIED take most of the time. Their time complexities were estimated in Section 15.4.1 to $O(n^2|T|)$ and $O(|T|^2)$, respectively. SPY-UPDATEDDOMAINS is called at most $|T|$ times during the learning and S-PROCESSIDENTIFIED at most $n \cdot |T|$ times. A lot of time is also consumed by the S-method that is called at most $n + l$ times, that is, $O(n)$ times. In the worst case, the S-method runs in $O(n^7 p^{2l+2} + n^9 p^{2l+1})$ as was derived in Section 11.1. In total, the worst case time complexity of the S-learner is $O(n^2|T|^2 + n|T|^3 + n^8 p^{2l+2} + n^{10} p^{2l+1})$ which is $O(n^8 p^{2l+2} + n^{10}(p^{3l+3} + p^{2l+1}))$.

15.5.2 Running Example

This section provides an explanation how the S-learner learns on an example. The black box is represented by the 4-state deterministic finite automaton defined in Figure 15.5. The learning setting is the same as in Section 15.3.2 and Section 15.4.2 that describe the running example for the H-learner and the SPY-learner, respectively. It means that at most 1 extra state is considered during the learning and that the teacher is able to answer output queries.

The S-learner starts with obtaining the state output of the initial state after it resets the black box; output query 1 is $T(\varepsilon, \uparrow)$. Then, it tries the first input by calling S-QUERYANDCHECKEACH from S-IDENTIFYNEXTSTATE. As the observed output does not differ from the one of the initial state s_0 and so the reached state is identified as s_0 , the input ‘a’ is queried again due to the small optimization described in the end of Section 15.2.1. The response is also the same, hence, $n_{0,aa}$ is merged to $[\varepsilon]$ that represents the reference node of state s_0 . The function S-GETNODETOEXTEND then chooses

which node to extend by which input. As there is an unverified transition on ‘b’ from s_0 , the function returns the current leaf node $n_{0,aa}$ and ‘b’ as the input that should extend that node. This is due to the condition on line 9 of Algorithm 53 that is one of few parts where the S-learner differs from the SPY-learner. S-IDENTIFYNEXTSTATE is called with $n_{0,aa}$ and ‘b’ as arguments. The response to the output query $T(b)$ is 0 like for the previous queries, therefore, the transition (s_0, b) is defined to lead back to s_0 . The input ‘b’ is then queried again as S-IDENTIFYNEXTSTATE defines and the output does not match the expected one. Therefore, the inconsistency I is revealed, see Section 15.2.2 for the types of inconsistency. S-MAKESTATENODE (Algorithm 52) is thus called such that $n_{0,aabb}$ is the inconsistent node $n_{i,u}$ because $\phi(aabb) = \emptyset$. Unfortunately, the parent of $n_{0,aabb}$ is not a state node and so another instance of the inconsistency I needs to be revealed according to lines 3–14 of Algorithm 52. The parent $n_{0,aab}$ is identified as s_0 , hence, ‘b’ as the input leading from $n_{0,aab}$ to $n_{0,aabb}$ is queried from $n_{0,\varepsilon}$ (line 6 of Algorithm 52). Note that S-QUERYSEQANDCHECK is called with the third argument set to false so that S-PROCESSIDENTIFIED is not called after the input is queried. The response to ‘b’ is 1, hence, the domain $\phi(b)$ becomes empty and S-MAKESTATENODE calls itself on line 12. A new state node $n_{1,\varepsilon}$ is created from $n_{0,b}$. The state s_1 is added to the conjectured model and to the domain $\phi(aabb)$ of node $n_{0,aabb}$ that is the only one consistent with $n_{1,\varepsilon}$. After the convergent graph and domains Φ are initialized, S-PROCESSIDENTIFIED is called and $n_{0,aa}$ is found to be a witness of the inconsistency I as $\phi(aa)$ is empty.

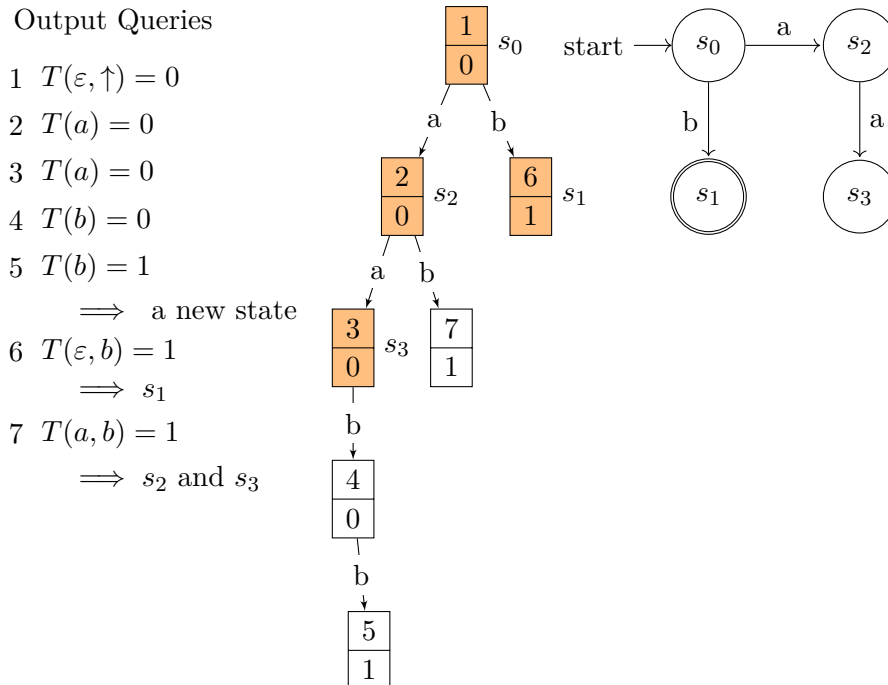


Figure 15.17: The first 7 output queries reveal all 4 states of the black box

S-MAKESTATENODE is called with $n_{0,aa}$ as the parameter. The parent $n_{0,a}$ is not a state node but is identified as s_0 . Hence, ‘a’ is checked that it is queried from $n_{0,\varepsilon}$ (line 6 of Algorithm 52) and the domain $\phi(a)$ is considered in the cycle on lines 7–10. The separating sequence of $n_{0,aa}$ and $n_{0,\varepsilon}$ is ‘b’ and thus it is queried from $n_{0,a}$. The output 1 eliminates s_0 from $\phi(a)$ not because of the sequence ‘b’ but because of the new separating sequence ‘ab’. S-MAKESTATENODE calls itself again (line 12 of Algorithm 52) and the new state s_2 is created. Nevertheless, even the new state node $n_{2,\varepsilon}$ is not consistent with $n_{2,a}$ so that the domain $\phi(aa)$ remains empty and thus S-MAKESTATENODE is called once again from S-PROCESSIDENTIFIED. This time, the parent of the inconsistent node is a state node, hence, the node is transformed into a state node and state s_3 is added to the conjectured model. The first 7 output queries that revealed all 4 states of the black box is shown in Figure 15.17 along with the corresponding observation tree and the current conjectured model. Note that s_2 and s_3 are swapped compared to the learning by the H- and SPY- learners.

The S-learner needs further 14 output queries to construct a completely specified conjectured model; these queries are captured in Figure 15.18. It starts with an extension of the last queried sequence in order to define the transition (s_1, b) ; $n_{2,b}$ is identified as s_1 due to its unique state output 1 and ‘b’ is chosen to be identified first because it is the symbol leading to $n_{2,b}$. The target state is identified as s_3 because of the response to the separating sequence ‘b’; output query 9 is $T(b)$. S-GETNODETOEXTEND then finds the separating sequence ‘ab’ that reduces $\Phi([\bar{s}_3b])$ and queries it (lines 4–7 of Algorithm 53). As there is an unverified transition from s_1 that has the access sequence of length 1, the next output query resets the black box and queries this transition on ‘a’; output query 12 is $T(b, a)$. The separating sequences ‘b’ and ‘ab’ are then queried to identify the target state as s_0 ; output queries 13–15. The S-learner is implemented to call S-GETNODETOEXTEND only after the chosen transition is verified, therefore, $n_{1,ab}$ (node 13 in Figure 15.18) was not extended even though it was identified as s_1 immediately due to its unique state output. The last undefined transition is (s_3, a) and it is shorter to query ‘b’ from the current leaf node $n_{1,aab}$ (node 15) to reach a node related to s_3 than reset the black box and query the access sequence ‘aa’ of s_3 . The next output query is thus $T(b)$ and ‘ab’ follows. As the response to the separating sequence ‘b’ is 1, the target state of (s_3, a) is either s_0 or s_2 . Therefore, the separating sequence ‘ab’ is queried from $n_{3,a}$. Then, all transitions are defined so that the conjectured model is completely specified and the learning would stop if an equivalence query was asked. The S-learner increases the number l of extra states to 1 and calls the S-method. The S-method constructs a splitting tree based on the provided conjectured model and then extends the observation tree to include an $(n + l)$ -complete test suite for the conjectured model with n states. It returns the test sequences that are not captured by the given observation tree. Both the splitting tree and the test sequences (called requested queries) are shown in Figure 15.18.

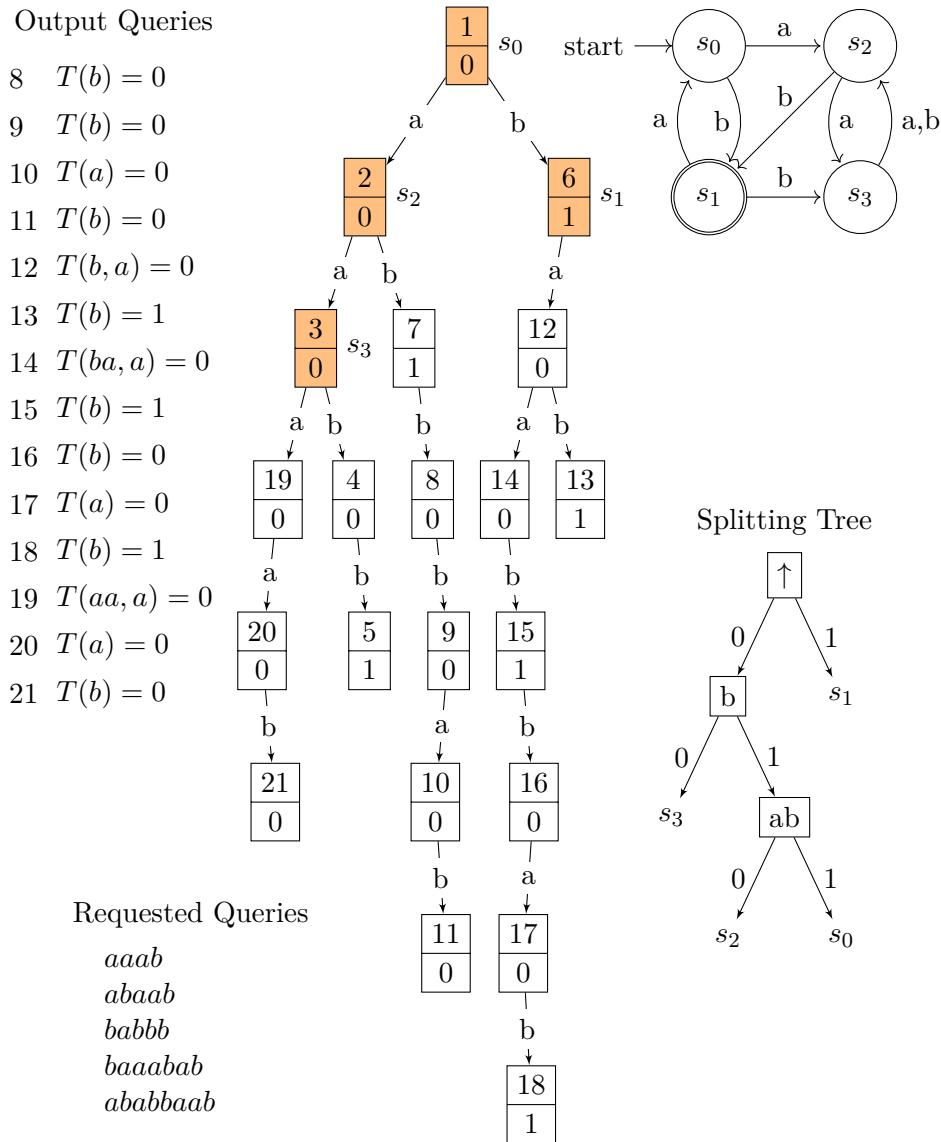


Figure 15.18: The next 14 output queries complete the 4-state DFA

The requested queries are queried gradually from the longest one as it is shown in the final observation tree in Figure 15.19. The S-learner learns the correct model of the black box defined in Figure 15.5 using 36 output queries, 11 resets of the black box and 55 symbols queried in total. Moreover, a guarantee is provided. The model is either output-equivalent to the black box or the black box has more than 5 states as 1 extra state was considered during the learning. Note that the completely specified 4-state DFA was learnt after 21 output queries, 6 resets and 26 queried symbols.

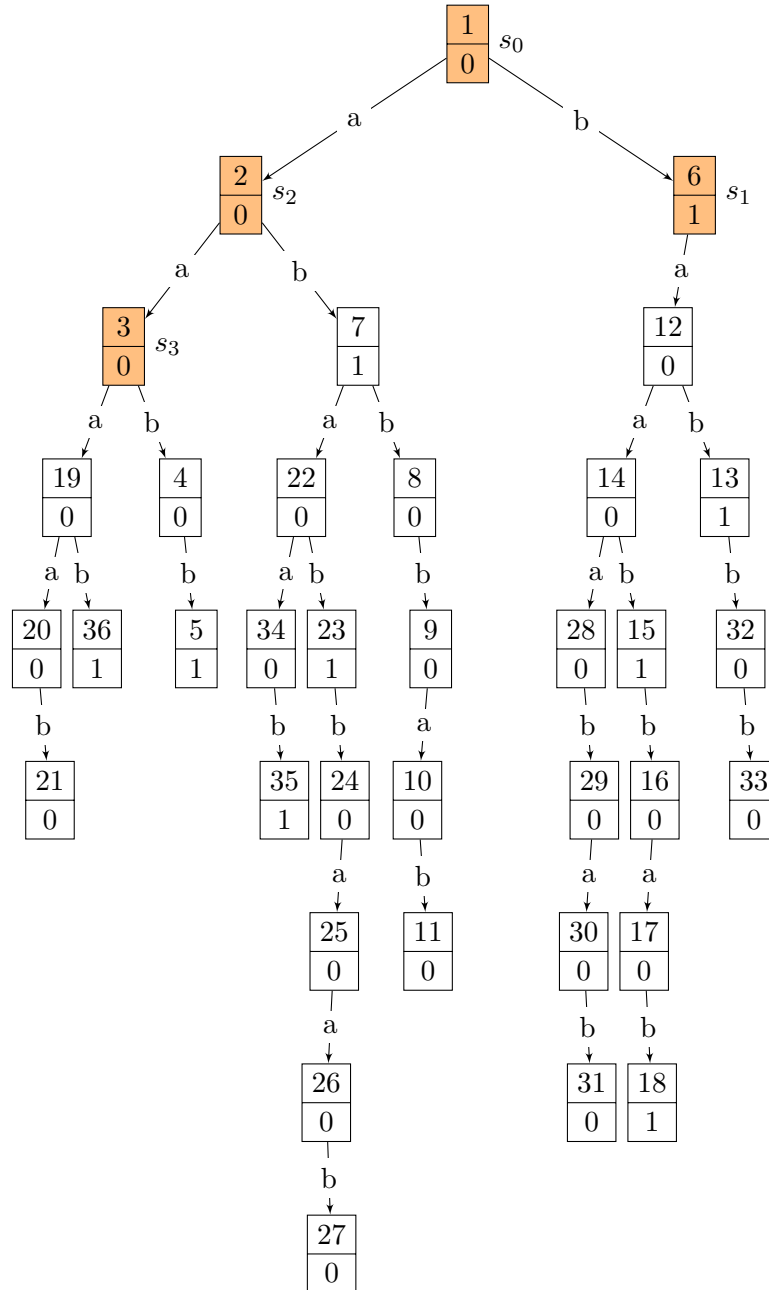


Figure 15.19: Observation tree closed for 1 extra state by the S-learner

15.6 General Learning Framework

The observation pack was proposed in [BDGW96] as a unifying theoretical framework for the standard active-learning algorithms. Nevertheless, it is not general enough to cover the three new learners that were proposed in the previous sections. Hence, this section proposes a *General Learning Framework* (GLF) that unifies and abstracts all active-learning algorithms rather from the implementation point of view.

The General Learning Framework consists of 4 steps called *Choose*, *Identify*, *Update* and *Check*. Algorithm 54 describes the GLF in a pseudocode. There are two loops, a main one and a nested one. Each algorithm needs to have a stop condition, for example, that there is no counterexample obtained on an equivalence query. The condition stops the main loop and is checked in the step *Check*. In addition, the step can update inner variables like the considered number of extra states in the case of the learner based on the observation tree approach. The other three steps are included in the inner loop. The loop identifies the target state of a selected transition. At first, the step *Choose* selects a transition for identification. The step *Identify* identifies the target state and then the learning structures are updated by the step *Update*.

Algorithm 54: General Learning Framework

```

1 repeat
2   while there is an unverified transition do
3      $t \leftarrow$  choose an unverified transition           (Choose)
4     identify the target state of  $t$                        (Identify)
5     check and update unverified transitions             (Update)
6   check the stop criterion and update unconfirmed transitions (Check)
7 until stop criterion is met

```

After the standard active-learning algorithms are introduced in the next chapter, Section 16.8 describes how each learning algorithm specifies the General Learning Framework so that it can be seen its unifying contribution.

Chapter 16

Standard Learning Algorithms

All active-learning algorithms specify the General Learning Framework (Section 15.6) and in fact they implement the observation tree approach but using a different learning structure. They are restricted to the first phase of the observation tree approach so that they do not consider extra states. All the algorithms handle two sets of sequences but they differ in the way how they store these sets and how they extend the sets during the learning. The first set includes *access sequences* that lead to recognized states. Recognition, or identification, is done by *separating sequences* covered in the second set.

This chapter describes active-learning algorithms for resettable machines. The L^* algorithm, the Discrimination tree algorithm, the Observation pack algorithm, the TTT algorithm and the Quotient algorithm are ordered by the time of their proposal and by their efficiency as well. The GoodSplit algorithm is the sole representative of unsupervised learning as it does not require a teacher. On the other hand, the learned model does not have to be a correct representation of the black box. All algorithms are described on the DFA defined in Figure 15.5 and their comparison on this reference machine follows in the next chapter. Note that the stOut symbol \uparrow is not counted among queried symbols. The last section of this chapter summarizes the algorithms and shows how they implement the General Learning Framework.

16.1 L^* algorithm

The L^* algorithm was proposed by Angluin in 1986 [Ang86] and it implements the idea of finite automata identification by Gold [Gol72]. L^* was adjusted to learn Mealy machines in [Nie03] but was also adapted to learn *deterministic finite cover automata* [Ipa12] and indirectly *I/O automata* using *interface automata* [AV10]. The TL^* algorithm is a direct extension of L^* that learns temporal guards on transitions after L^* learns underlying state diagram [LAD⁺11]. L^* was generalized for nondeterministic machines, *residual finite-state automata* in particular, in the NL^* algorithm [BHKL09] and then even for *universal* and *alternating automata* so the UL^* and AL^* algorithms were proposed in [AEF15]. The most promising application of active learning and testing seems to be adaptive model checking (AMC) [GPY02] and grey box checking [EGPQ06] that are based on black box checking [PVY99]. Both

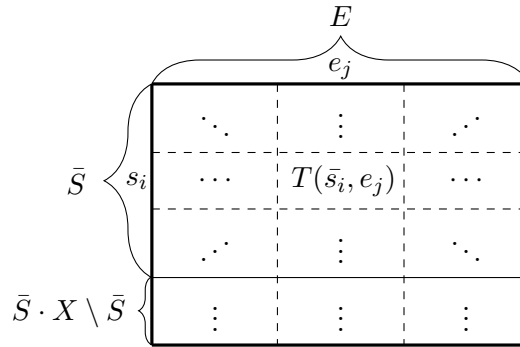


Figure 16.1: An observation table (\bar{S}, E, T)

AMC and grey box checking use testing as a task separated from the learning, hence, it duplicates a lot of queries that the learner already asked. AMC employs L^* to learn a model that is then passed to the model checker. If a discrepancy is found, it is checked against the system and a counterexample is returned to L^* if the discrepancy is not confirmed in the system. On the other hand, if all properties hold in the conjectured model, the W-method is employed to test the model against the system. AMC thus provides a software verification. Grey box checking uses knowledge about parts of the system that are so-called white boxes because the definition of their behaviour is available as source code for example. Another adaptive approach is called Active Continuous Quality Control [WNS⁺13]. It extracts separating sequences from the observation table used by the L^* and the separating sequences are employed when one learns the model of updated system. Experimental evaluation of this so-called adaptive learning was reported in [HMvdP18]. Two systems, ToDoMVC and SSH, were learnt by two versions of the L^* algorithm and the W-method (Section 9.3) was used to approximate equivalence queries. In [AKT⁺14], several implementations of bounded retransmission protocol were learnt by L^* using automatic abstract refinement that allows one to work with infinite value domains.

L^* uses an *observation table* to capture the learning process. An observation table (OT) is a triple (\bar{S}, E, T) , where \bar{S} is a nonempty finite prefix-closed set of access sequences, E is a nonempty finite suffix-closed set of separating sequences and T is a finite function of pairs of input sequences. An OT is commonly visualized as a table with rows labelled with sequences of $(\bar{S} \cup (\bar{S} \cdot X))$, that is, access sequences and their successors, columns are labelled with separating sequences of the set E and T describes the content of cells. Figure 16.1 shows a sketch of OT. T is the output query defined in Section 14.1. It takes two input sequences, the label u of the row and the label v of the column, and maps them to the output of their concatenation; the related cell contains $T(u, v)$. The black box is reset before each output query T .

An observation table is assumed to be *closed* and *consistent* before an EQ is asked. Let $\text{row}(\bar{s})$ denote the row of OT labelled with \bar{s} , that is, $\text{row}(\bar{s}) = f$

where $f(e) = T(\bar{s}, e)$ for all $e \in E$. An OT is *closed* if for each $u \in \bar{S} \cdot X$ there is $\bar{s} \in \bar{S}$ such that $\text{row}(u) = \text{row}(\bar{s})$. In other words, each next state is identified as a state with the access sequence in \bar{S} . An OT is *consistent* if for all $\bar{s}_i, \bar{s}_j \in \bar{S}$ such that $\text{row}(\bar{s}_i) = \text{row}(\bar{s}_j)$ hold that for all $x \in X$ $\text{row}(\bar{s}_i \cdot x) = \text{row}(\bar{s}_j \cdot x)$. In other words, equal states reached by different access sequences need to lead to the same next states.

If an OT is closed and consistent, a conjectured model M based on it can be constructed. \bar{S} is taken as the set of states S and transitions are given by the definition of a closed OT. Such a conjectured model is consistent with the OT, that is, $\lambda_M^*(s, e) = T(\bar{s}, e) \forall s \in S \forall e \in E$, because of the suffix-closedness property of the set E . Moreover, if all $\text{row}(\bar{s})$ of $s \in S$ are unique, then the conjectured model is minimal because of the prefix-closedness property of the set \bar{S} .

The L* algorithm starts with $\bar{S} = \{\varepsilon\}$, that is, the access sequence of the initial state, and E filled with \uparrow if the black box has outputs by states (Moore, DFA and DFSM) and with all input symbols if the black box has outputs on transitions (Mealy, DFSM). In the main learning loop, the OT is made consistent and closed. Then the algorithm creates a conjectured model based on the OT and asks an EQ. If a counterexample is returned, the OT is extended and the cycle is repeated. If the conjectured model is correct, the learning stops.

Making the table consistent and closed follows the definitions of both properties. If the OT is not consistent, then there are \bar{s}_i and \bar{s}_j in \bar{S} and $x \in X$ such that $\text{row}(\bar{s}_i) = \text{row}(\bar{s}_j)$ and $\text{row}(\bar{s}_i \cdot x) \neq \text{row}(\bar{s}_j \cdot x)$. Therefore, there is $e \in E$ such that $T(\bar{s}_i \cdot x, e) \neq T(\bar{s}_j \cdot x, e)$. The inconsistency of \bar{s}_i and \bar{s}_j is solved by adding $x \cdot e$ to E as this sequence distinguishes them. The OT is thus enlarged by a column labelled with a new separating sequence each time the OT is found to be inconsistent. If the OT is not closed, then there is $\bar{s} \in \bar{S}$ and $x \in X$ such that $\text{row}(\bar{s} \cdot x)$ is different from each row with label in \bar{S} . Therefore, $\bar{s} \cdot x$ is added to \bar{S} as the access sequence of a new state. The OT is extended by $|X|$ rows that represent the next states of the new state, that is, for all $x_i \in X$ a row with label of $\bar{s} \cdot x \cdot x_i$ is added.

There are several methods for processing a counterexample (CE) and extending the OT appropriately. Only the original one proposed by Angluin extends the set \bar{S} which is the way how the OT can become inconsistent. Therefore, the L* algorithm does not have to check the consistency of OT after other CE processing methods as they extend only the set E . Nevertheless, some methods add only one sequence to E which could break the suffix-closedness of the set. The conjectured model based on such OT thus does not have to be minimal and its output can be different to the one stored in the OT. This issue can be solved by the *semantic suffix closedness* that adds additional separating sequences to E . A discussion on semantic suffix closedness with a running example follows the description of CE processing methods in the next sections.

It was proven that the L* algorithm learns in terms of membership queries polynomially in the number of states and the length of the longest counterex-

ample [Ang86]. The time complexity is similar because the analysis of the observed responses is a comparison of rows of the OT. The size of OT depends on the CE processing method but can be estimated such that it has at most np rows and n columns where n is the number of states of the black box and p is the number of inputs. The length of labels of rows and columns depend on obtained counterexamples. Let w be the longest counterexample. Then, there are at most $np \cdot n$ output queries of length $n + |w|$ and the number of comparisons of rows can be bounded by $np \cdot n \cdot n \cdot |w|$. Therefore, the worst case time complexity is also polynomial in the number of states and the length of w .

16.1.1 AllPrefixes

The first method for processing a counterexample was proposed by Angluin in the original paper of the L^* algorithm [Ang86]. The method, called here *AllPrefixes*, simply adds the CE and all its prefixes to \bar{S} . This preserves the prefix-closedness of the set \bar{S} but it may break consistency of the OT. Therefore, consistency needs to be checked. As \bar{S} is a set, only sequences that are not in \bar{S} are added. Besides the extension of the set \bar{S} , the OT is also enlarged accordingly; all sequences of $\bar{S} \cup \bar{S} \cdot X$ become labels of their unique rows in the OT.

16.1.2 OneSuffix - binary search

Rivest & Shapire showed in [RS93] that the consistency of the OT does not have to be checked if only the set E is extended by a counterexample. Moreover, they proposed a method for such extension of E . Their method, called here *OneSuffix - binary search*, finds a separating suffix v of the CE (Theorem 14.1) using binary search. The suffix v then extends E and thus reveals a new state. Unfortunately, a new conjectured model based on the extended OT can still have an incorrect output on the CE as the suffix-closedness property of E was broken. Note that this issue does not restrict the L^* algorithm to learn a correct model but it asks more EQs than it is necessary. Semantic suffix closedness (Section 16.1.7) can solve this issue by adding a particular suffix of v .

16.1.3 AllSuffixesAfterLastState

Shahbaz proposed a method that does not break the suffix-closedness of E in [Sha08]. His method, called here *AllSuffixesAfterLastState*, finds the longest prefix u of the given counterexample w such that u is the label of a row of the OT. Then the remaining suffix v of the CE, that is, $w = u \cdot v$, extends the set E with all its suffixes. Therefore, E remains suffix-closed and the conjectured model will be minimal. As E is a set, only sequences that are not in E are added.

16.1.4 Suffix1by1

The method *Suffix1by1* was proposed by Irfan in [Irf10, IOG10]. It chooses the shortest suffix v of the given counterexample that stops the OT from being closed and adds it with all of its suffixes to the set E . In particular, *Suffix1by1* extends the OT with the suffixes of the CE. It starts with the shortest one that is not in E . If the OT is not closed after a suffix extended it, no further (longer) suffix is added. After the OT is made closed again, that is, \bar{S} is enlarged, the given CE is checked again. Further suffixes are added by the same procedure if the sequence is found to be a counterexample for the updated conjecture. The method is an optimized version of the *AllSuffixesAfterLastState* method, especially for long counterexamples generated by a random walk.

16.1.5 SuffixAfterLastState

A combination of ideas behind the *OneSuffix* - binary search method and the *AllSuffixesAfterLastState* method is a new method called *SuffixAfterLastState*. It adds only one suffix v of the given counterexample to the set E . The suffix v is found such that the longest prefix u of the CE t included in $\bar{S} \cup \bar{S} \cdot X$ is determined at first. Then, the longest suffix v is found in the remaining sequence w ($t = u \cdot w$) such that v satisfies Theorem 14.1, that is, the response on v of the black box differs from the output of the conjectured model. As for the *OneSuffix* - binary search method, the suffix-closedness of E can be broken so semantic suffix closedness (Section 16.1.7) should be checked.

16.1.6 Other Counterexample Processing Approaches

There are many other possible ways to split a counterexample and extend the OT. Isberner & Steffen inspired by the *OneSuffix* - binary search method proposed an abstract framework for counterexample analysis in [IS14]. They showed 3 new searches for a separating suffix (Theorem 14.1). Particularly, the exponential search, the partition search and the eager search. However, any search technique can be employed to find such a suffix and thus used as a part of the L* algorithm.

16.1.7 Semantic Suffix Closedness

Steffen et al. proposed the notion of *semantic suffix closedness* [SHM11] as follows.

Definition 16.1. Let M be the conjectured model for the observation table (\bar{S}, E, T) . Then E is called *semantically suffix closed* for M , if for any two states $s_1, s_2 \in S$ and any decomposition $v_1 \cdot v_2 \in E$ of any suffix with $T(\bar{s}_1, v_1 \cdot v_2) \neq T(\bar{s}_2, v_1 \cdot v_2)$ there is also $\text{row}(s'_1) \neq \text{row}(s'_2)$ where s'_1, s'_2 are successors of s_1, s_2 in M on v_1 , that is, $s'_1 = \delta^*(s_1, v_1)$ and $s'_2 = \delta^*(s_2, v_1)$.

They claimed that every conjectured model constructed from an OT with semantically suffix closed E is minimal. An algorithm for making the set E semantically suffix closed was also proposed. It finds a new separating

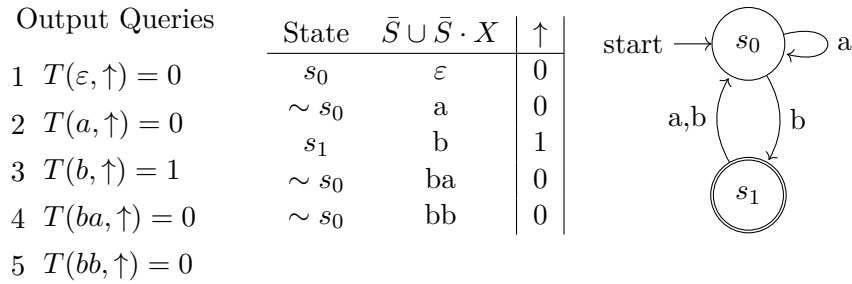


Figure 16.2: Initial closed observation table and the related conjectured model

sequences so the conjectured model M becomes minimal. Particularly, if M is not minimal, there are two undistinguished states s_1, s_2 . Let e be the sequence from E that distinguishes these two states, that is, $T(\bar{s}_1, e) \neq T(\bar{s}_2, e)$. Then, there needs to be a suffix d of e that will reveal a new state after d extends the OT. This comes from the fact that some successors of s_1, s_2 are not distinguished in the conjectured model. The proposed algorithm follows the path on e from both states and when the states collapse in M , the remaining part of e is the wanted suffix d .

Unfortunately, this does not work if there is a cycle among undistinguished states and the separating sequence e follows such a cycle. The following example captures this issue and also shows a conjectured model constructed from the OT with semantically suffix closed set E that is not minimal. Therefore, semantic suffix closedness is not a sufficient condition for the conjectured model to be minimal.

Consider the 4-state DFA shown in Figure 15.5. It operates on the binary alphabet $\{a, b\}$ and only state D is accepting, that is, it responds with the output of 1. At first, the L^* algorithm asks output queries on the empty sequence and inputs ‘a’ and ‘b’. Output query 3, $T(b, \uparrow)$, reveals a new state s_1 (corresponding to state D) as the obtained output differs from the output of the initial state s_0 . Therefore, two more output queries T are asked before the OT becomes closed (and consistent). Figure 16.2 shows the current OT and the conjectured model constructed from it.

Assume that the teacher returns the shortest counterexample, ‘aab’, to the equivalence query asked with respect to the current conjectured model. Note that a sequence in the format ‘(aa)ⁱb’ for any $i > 0$ is a counterexample.

AllPrefixes adds sequences ‘a’, ‘aa’ and ‘aab’ to the set \bar{S} . Then separating sequences ‘b \uparrow ’ and ‘ab \uparrow ’ are formed during the procedure of making the OT consistent. The OT is filled by output queries and is found closed. The second EQ confirms that the conjecture is correct.

AllSuffixesAfterLastState extends the set E with suffixes ‘ab’ and ‘b’. The procedure of making the OT closed then reveals states C and B of the black box so the model is learnt.

Suffix1by1 adds first the suffix ‘b’ to the set E so state B of the black box is revealed by the procedure of making the OT closed. Then, the given CE ‘aab’ is found to be a counterexample even for the 3-state conjectured model,

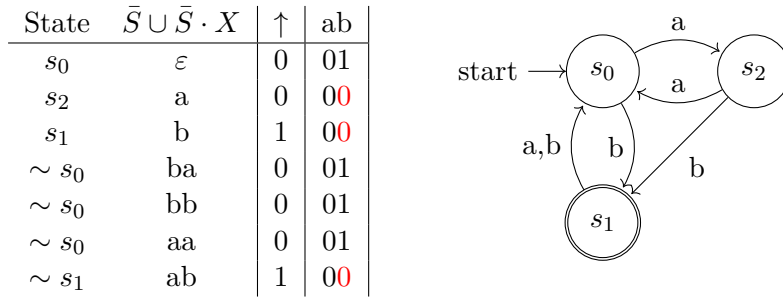


Figure 16.3: Observation table extended with ‘ab’ and the updated model

therefore, the suffix ‘ab’ extends E . State C of the black box is observed and the conjectured model is correct after the OT is made closed again.

Both *OneSuffix - binary search* and *SuffixAfterLastState* identify ‘ab’ as the separating suffix of the CE. Figure 16.3 shows the closed OT after ‘ab’ extended the set E . State s_2 corresponding to state C of the black box is revealed and the conjectured model is updated. The set $E = \{\uparrow, ab\}$ of the OT in Figure 16.3 is semantically suffix closed according to Definition 16.1. Nevertheless, the conjectured model constructed from the OT (Figure 16.3) is not minimal as states s_0 and s_2 are undistinguishable. It also means that the conjectured model does not respond correctly to some sequences whose observed outputs are stored in the OT. These wrong outputs are red in the OT in Figure 16.3.

Another separating suffix is needed to resolve this inconsistency. The procedure proposed in [SHM11] advises to find undistinguishable states and then to follow the path of the separating sequence from E . The separating sequence for states s_0 and s_2 is ‘ab’. However, the states collapse in the conjectured model in the very end of the sequence so there is no nonempty suffix. In this case, one could employ any other method to find a separating suffix of ‘ab’, for example, *Suffix1by1*. The suffix ‘b’ needs to be added to reveal the last state and make the conjectured model consistent with the observation table.

In conclusion, the running example showed that the semantic suffix closedness is not a sufficient condition for consistency of the conjectured model with respect to the observation table. Nevertheless, undistinguishable states in the conjectured model means that additional suffix needs to extend the set E to make the conjectured model consistent with the observed traces. Some output queries can be saved in general if the suffix-closedness of E is not a requirement.

The L^* algorithm learns the reference DFA using 2 equivalence queries. Using *AllPrefixes* for processing a CE, it asks 33 output queries and 105 symbols in total. *AllSuffixesAfterLastState* needs only 28 output queries and 78 symbols to learn. The other three CE processing functions use 29 output queries and 80 symbols (81 symbols in the case of *Suffix1by1*). The number of resets is always the same as the number of output queries.

16.2 Discrimination Tree algorithm

Kearns & Vazirani proposed a completely different approach to learning than L^* has in [KV94]. Their algorithm captures knowledge about the black box in a decision tree. The tree was later called a *discrimination tree* (DT) and so even the algorithm.

Each node of DT contains a sequence. Sequences of inner nodes form the set E of separating sequences. Access sequences of the set \bar{S} label leaves of the tree. Edges are labelled with outputs on the separating sequence of the corresponding parent node. Labels of edges are given by output query $T(\bar{s}, e)$ defined in Section 14.1. A DT is a decision tree and thus the following property holds for each two access sequences \bar{s}_i and \bar{s}_j stored in the leaves of DT. Let n_{ij} be the common ancestor of the leaves relating to \bar{s}_i and \bar{s}_j in the DT that separates them, that is, $T(\bar{s}_i, e_{ij}) \neq T(\bar{s}_j, e_{ij})$ where e_{ij} is the separating sequence stored in n_{ij} . Then both s_i and s_j produce the same output on each separating sequence that labels a predecessor of n_{ij} in the DT. These outputs are labels of the related edges on the path from the root to n_{ij} .

An important procedure of the Discrimination tree algorithm is *sifting*. Sifting is an identification, or classification, of an input sequence using the DT. A sequence u is to be identified with which access sequence (state) is consistent. An output query $T(u, e)$ is asked for each separating sequence e of inner node on the path to the leaf with the consistent state. Particularly, $T(u, e)$ where e is the separating sequence of the root node is queried first. The successor with the incoming edge labelled with the observed output is chosen to be processed next. If it is a leaf, the consistent state is found. If it is an inner node, another output query $T(u, e_i)$ is asked but with the separating sequence e_i of the inner node. This procedure is repeated until a leaf is reached or there is no child under the observed output. If the output does not match any label of edges from the inner node n_i , a new state is revealed and u is its access sequence. A new leaf node labelled with u is appended to n_i and the edge gets a label of the observed output.

The Discrimination tree algorithm starts with the root of the DT labelled with the empty sequence that stands for the access sequence of the initial state. If the black box has outputs by states, that is, Moore, DFA or DFSM, the current root is appended under a new one with label of \uparrow . The output of the initial state labels the edge from the root to the leaf with the empty sequence.

A conjectured model is constructed from the DT such that the set \bar{S} of access sequences represents the set of states S . The transition on $x \in X$ from the state s is formed in the way that the sequence $\bar{s} \cdot x$ is sifted through the DT and the next state is identified. If no new state is revealed and a complete conjectured model is created, then an EQ is asked.

Figure 16.4 shows the first completely specified conjectured model and the related discrimination tree in the case of learning the reference DFA (Figure 15.5). The first output query $T(\varepsilon, \uparrow)$ asks for the state output of

Output Queries

- 1 $T(\varepsilon, \uparrow) = 0$
- 2 $T(a, \uparrow) = 0$ (sift)
- 3 $T(b, \uparrow) = 1$ (sift) $\implies s_1$
- 4 $T(b, \uparrow) = 1$ (state output)
- 5 $T(ba, \uparrow) = 0$ (sift)
- 6 $T(bb, \uparrow) = 0$ (sift)

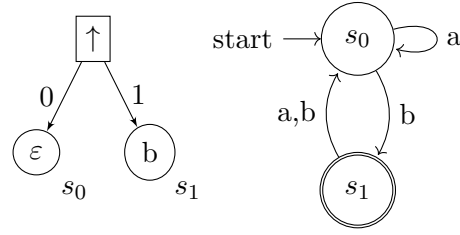


Figure 16.4: Discrimination tree and the related conjectured model after the first 6 output queries

the initial state s_0 . The output is also used to label the edge leading to the leaf of DT relating to s_0 . The next two output queries identify next states of s_0 . State D reached by ‘b’ in the black box is accepting in contrast with the initial state, therefore, the third output query $T(b, \uparrow)$ reveals a new state s_1 during the procedure of sifting. The implementation of the algorithm in FSMlib (Appendix A) then requires an additional query to add a new state to the conjectured model with a correct state output. Output queries 5 and 6 identify next states of s_1 by sifting as Figure 16.4 shows.

The DT algorithm finds in the counterexample obtained in response to an EQ the shortest prefix that is not consistent with the conjectured model. It means that prefixes starting from the shortest one are sifted through the DT and if the identified state does not match the state reached in the conjectured model by following the path of the same prefix, then the prefix is chosen. Let $v = u \cdot x$ be the shortest such prefix and x its last input. As state s_i reached on v in the conjecture is different from state s_j reached on v in the black box, the previous states are also different. Therefore, the DT algorithm finds the separating sequence e_{ij} of s_i and s_j and prepends it with the last input of v so a new separating sequence $e = x \cdot e_{ij}$ is created. The separating sequence e_{ij} labels the common ancestor node in the DT of both leaves related to states s_i and s_j . A new inner node with the separating sequence e replaces the leaf of the DT that relates to the state s_k reached by the prefix u in the conjectured model, that is, the one before the last input of v was applied. Another output query is asked on the access sequence of s_k and e which provides the label to the edge leading to the leaf related to s_k . The second leaf is created with the access sequence u and an incoming edge labelled with $T(u, e)$.

Figure 16.5 shows how the obtained counterexample ‘aab↑’ is processed and how it updates the DT and the conjectured model M . According to M in Figure 16.4, the output is 0 on the first two symbols as the conjectured model stays in the initial state s_0 on ‘a’. It corresponds to output queries 7 and 8 that provide the outputs of the black box. However, the outputs on the next input (‘b’) of the CE are not equal so that the states reached on prefix ‘aa’ in M and the black box need to be different. Output query 9 identifies the reached state as s_0 . It should be s_1 according to the conjectured model.

Output Queries

- 7 $T(a, \uparrow) = 0$ (sift)
- 8 $T(aa, \uparrow) = 0$ (sift)
- 9 $T(aab, \uparrow) = 0$ (sift) $\implies s_2$
- 10 $T(\varepsilon, b \uparrow) = 11$ (update DT)
- 11 $T(aa, b \uparrow) = 00$ (update DT)
- 12 $T(aa, \uparrow) = 0$ (state output)
- 13 $T(a, b \uparrow) = 11$ (update conjecture)
- 14 $T(ba, b \uparrow) = 11$ (update conjecture)
- 15 $T(bb, b \uparrow) = 00$ (update conjecture)
- 16 $T(aaa, \uparrow) = 0$ (sift)
- 17 $T(aaa, b \uparrow) = 11$ (sift)
- 18 $T(aab, \uparrow) = 0$ (sift)
- 19 $T(aab, b \uparrow) = 11$ (sift)

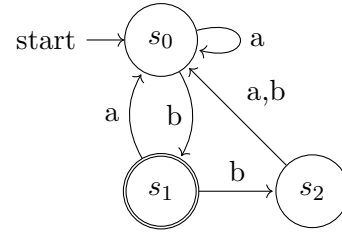
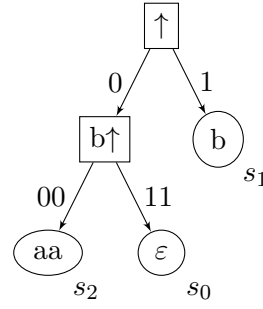


Figure 16.5: Updated model and the DT after processing the first CE $aab \uparrow$

A new separating sequence is created from the last input ‘b’ concatenated with the separating sequence of s_0 and s_1 which is \uparrow . The sequence ‘ $b \uparrow$ ’ labels a new inner node that replaces the leaf related to s_0 because s_0 was reached by the prefix ‘aa’ in M . The prefix ‘aa’ becomes the access sequence of the new state s_2 . Next two output queries obtain the labels of edges from the new inner node. The conjectured model is then updated. State s_2 is added with the output derived by output query 12 and next three queries update transitions that led to s_0 . Note that the conjectured model is created from scratch by the original algorithm, therefore, some output queries are saved by updating only related transitions. Transitions from s_2 are identified by sifting (output queries 16–19).

The DT algorithm asks next equivalence query on the complete 3-state conjecture and the counterexample ‘ $aab \uparrow$ ’ is returned again. The reason can be seen in Figure 16.5. State s_2 relates to the access sequence ‘aa’ stored in the DT, however, this is not reflected in the conjectured model as $\delta_M^*(s_0, aa) = s_0$. In addition, the conjectured model is not consistent with the DT because $\lambda_M^*(\delta_M^*(s_0, aa), b \uparrow) = 11$ that does not match the label 00 of the corresponding edge in the DT.

The separating sequence ‘ $ab \uparrow$ ’ derived from the CE is sufficient to reveal the last state and finish the learning. The Discrimination tree algorithm asks 3 equivalence queries, 32 output queries and 76 symbols in total. The number of resets is equal to the number of output queries.

16.3 Observation Pack algorithm

The Observation pack algorithm combines the Discrimination tree algorithm and the L* algorithm. It was proposed by Howar in [How12]. The name of the algorithm refers to the theoretical notion of *observation pack* (OP) proposed by Balcazar et al. in [BDGW96] but the idea of the algorithm does not follow the idea of observation pack.

The structure for learning is a *discrimination tree* (DT) with component tables in leaves. A *component table* (CT) looks like an observation table (see Section 16.1) but all row labels relate to a single state of the conjectured model. Formally, a component table CT_i is a quadruple (U_i, \bar{s}_i, E_i, T) , where U_i is a finite set of input sequences labelling rows, $\bar{s}_i \in U_i$ is the access sequence of state s_i , E_i is a finite set of separating sequences labelling columns, and output query T defines outputs stored in cells for all $u \in U_i$ and all $e \in E_i$. A component table CT_i is *closed* if the content of all rows are equal to the row labelled with \bar{s}_i , that is, for all $u \in U_i$ and for all $e \in E_i$ holds $T(u, e) = T(\bar{s}_i, e)$. Component tables were originally called components, however, a new name is introduced as component tables do not match the notion of components of the observation pack in [BDGW96]. In contrast to a component of the observation pack that groups sequences with the same prefix, a closed component table groups all observed sequences leading to a consistent state. States are consistent if they respond equally to all separating sequences. The purpose of component tables is to eliminate repeated output queries in the Discrimination tree algorithm and reduce the number of equivalence queries.

The learning starts with the component table CT_0 relating to the initial state. The set E_0 of separating sequences is initialized with all input symbols. Moreover, if the black box has outputs by states, then E_0 also contains the input \uparrow and CT_0 is appended to the root of the discrimination tree labelled with \uparrow . Otherwise, CT_0 is the root of DT. Figure 16.6 shows the initialization for the 4-state DFA defined in Figure 15.5. Transitions in the conjectured model are assumed to lead back to the initial state that is the only state initially.

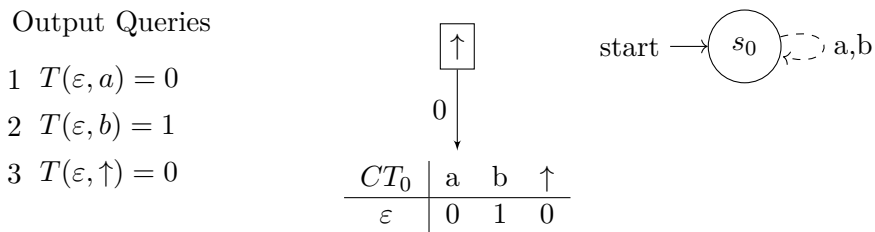


Figure 16.6: The first 3 output queries initialize CT_0 and the DT

A complete conjectured model is constructed if all transitions are defined and all component tables are closed. Then an equivalence query is asked.

Output Queries

- 4 $T(a, \uparrow) = 0$ (sift)
- 5 $T(a, a) = 0$
- 6 $T(a, b) = 1$
- 7 $T(a, \uparrow) = 0$
- 8 $T(b, \uparrow) = 1$ (sift) $\implies s_1$
- 9 $T(b, \uparrow) = 1$ (state output)
- 10 $T(b, a) = 0$
- 11 $T(b, b) = 0$
- 12 $T(b, \uparrow) = 1$

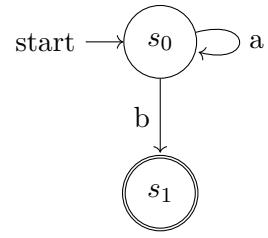
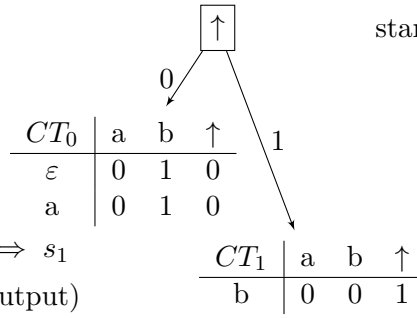


Figure 16.7: Next 9 output queries reveal state s_1

The target state of a transition is determined by sifting through the discrimination tree, see Section 16.2 for the notion of sifting. In the running example, $T(a, \uparrow)$ and $T(b, \uparrow)$ are asked to identify the next states of the initial state on ‘a’ and ‘b’ respectively. When a leaf of the DT is reached so that a related component table CT_i is known, a new row labelled with the sifted sequence is added and filled with corresponding output queries T . Identification of both transitions from the initial state of the reference machine is captured in Figure 16.7. In addition, state s_1 (D in Figure 15.5) is revealed as its state output differs from the output of the initial state. Therefore, a component table CT_1 is created with one row labelled with the access sequence of s_1 . The set E_1 of separating sequences is equal to E_0 . Generally, the initialization of E_i depends on the CE processing approach that is discussed later.

There are two ways to reveal a new state: sifting and unclosed component table. The former one revealed state s_1 as there was not a suitable successor in the discrimination tree. State s_2 is observed by the latter approach. Figure 16.8 shows the identification of transitions from s_1 on both inputs. Both target states of the transitions seem to be consistent with s_0 after sifting (output query 13 and 17). However, when the row of CT_0 is filled in for ‘bb’, CT_0 becomes unclosed. Note that ‘b’ is the access sequence of s_1 so ‘bb’ covers the transition on ‘b’ from s_1 . The difference is observed in the column labelled with ‘b’ (output query 19). It means that a new state s_2 is revealed as the reached state differs from s_0 and so CT_0 is split to become closed again. The DT is updated such that the separating sequence ‘b’ replaces the leaf with CT_0 and both CT_0 and CT_2 are successors of the new inner node. The separated component table CT_2 contains only the row labelled with ‘bb’. The updated DT with the consistent conjectured model is shown in Figure 16.8.

The conjectured model with 3 states is completely specified after 31 output queries and an equivalence query can be asked. The counterexample (CE) ‘aab \uparrow ’ is obtained. The Observation pack algorithm is proposed with three CE

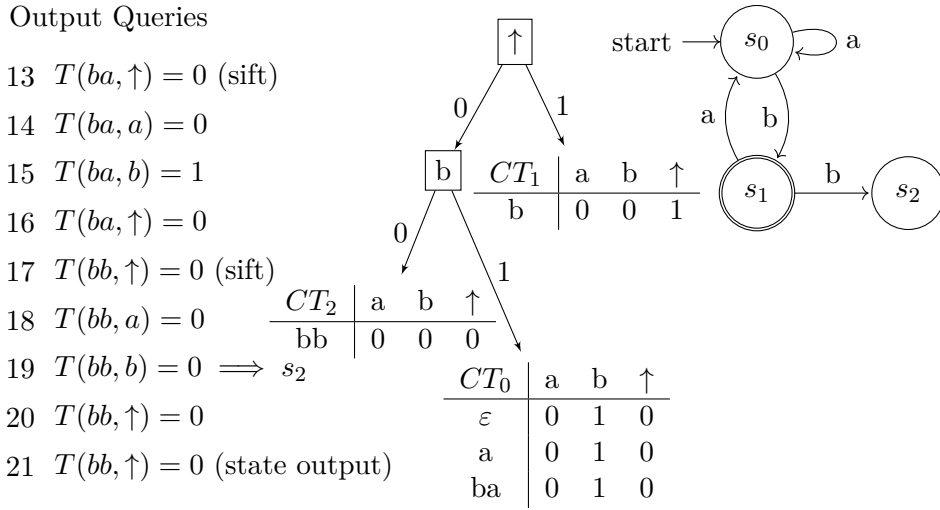


Figure 16.8: Next 9 output queries reveal state s_2

processing functions: *AllGlobally*, *OneGlobally* and *OneLocally*. Depending on the chosen function, component tables are extended with a suffix/suffixes of the given CE. *AllGlobally* adds all suffixes (that are not present) of the CE to all component tables, therefore, it is similar to *AllSuffixesAfterLastState* (Section 16.1.3) of the L^* algorithm. *OneGlobally* and *OneLocally* find the first (longest) separating suffix v of the CE as *SuffixAfterLastState* (Section 16.1.5) does. The difference between the two is the choice of component tables that are extended by the chosen suffix. *OneGlobally* adds the suffix to all component tables which corresponds to *SuffixAfterLastState* in the L^* algorithm, *OneLocally* adds the suffix v only to CT_i where $s_i = \delta_M^*(s_0, u)$ and the obtained CE is $u \cdot v$. Note that *AllGlobally* does not need additional output queries to process a counterexample. In the case of the running example, *OneGlobally* and *OneLocally* ask two output queries T to find the distinguishing suffix ‘ab↑’ of the given CE ‘aab↑’. *OneLocally* adds the suffix to CT_0 as the transition from s_0 on ‘a’ leads back to the initial state.

All component tables extended by separating suffixes are filled in by output queries and some tables become unclosed. Figure 16.9 shows the case of the running example. The state reached by ‘a’ responds to ‘ab↑’ with 000 that differs from the output of the initial state s_0 . Therefore, a new state s_3 is revealed and CT_3 is separated from CT_0 . The separating sequence ‘ab↑’ updates the DT as Figure 16.9 shows.

Initialization of E_i depends on how a new state s_i is revealed and on the CE processing function. If a new state is observed during the procedure of sifting and *OneLocally* is used to process a CE, then E_i is initialized with all input symbols and \uparrow in the case of state outputs. If a component table CT_j is found unclosed and CT_i is separated from CT_j , then E_i contains all sequences of E_j . Otherwise, that is, sifting reveals a new state and *AllGlobally* or *OneGlobally* is used, all sets of separating sequences are the same so E_i gets all sequences of E_0 .

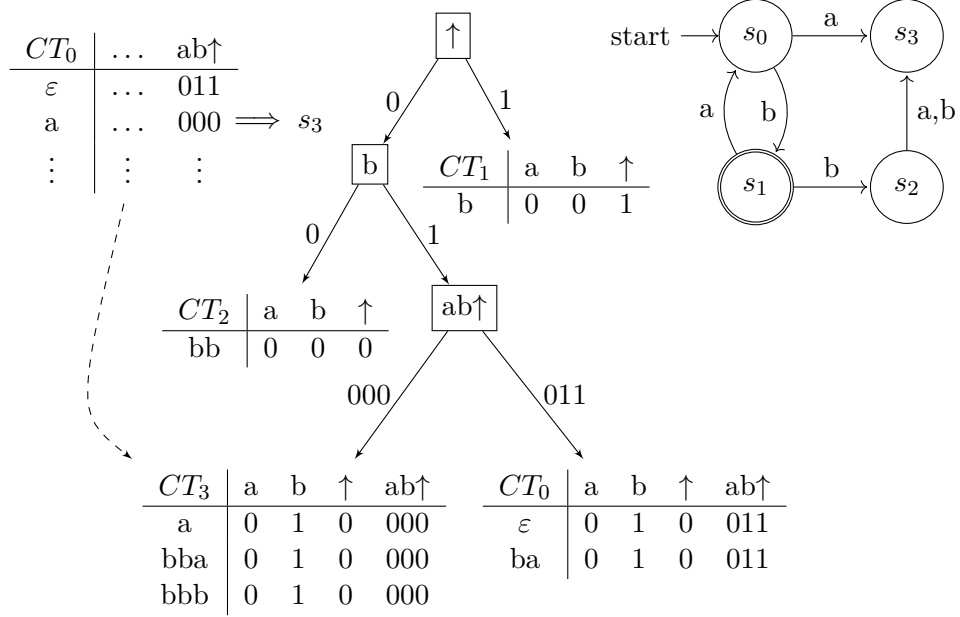


Figure 16.9: State s_3 revealed by the separating suffix ‘ $ab\uparrow$ ’ of CE ‘ $aab\uparrow$ ’

Figure 16.9 shows the DT and the conjectured model after s_3 was revealed. The algorithm asks further output queries to identify target states of transitions from s_3 and then the conjectured model is correct which is proven by an equivalence query. In total, the Observation pack algorithm asks 2 EQs to learn the reference DFA (Figure 15.5). The numbers of output queries T and queried symbols differ for each CE processing function. The number of resets equals to the number of output queries. Using the *AllGlobally* function, 68 OQs are asked and 199 symbols are queried in total. *OneGlobally* asks 52 OQs and 136 symbols. *OneLocally* has the best results, with only 47 OQs and 120 queried symbols. Nevertheless, notice in Figure 16.7 and Figure 16.8 that some output queries are repeated.

16.4 TTT algorithm

The TTT algorithm improves the Discrimination tree algorithm. It was proposed by Isberner et al. in [IHS14]. The name of the algorithm comes from the use of three tree-like structures for learning: a spanning tree, a discrimination tree and a discriminator trie.

A discrimination tree (DT) is described in Section 16.2. On the one hand, the leaves of DT contain access sequences of states of the conjectured model. The spanning tree ensures that the set \bar{S} of these access sequences is prefix-closed. On the other hand, the inner nodes of DT include separating sequences. The discriminator trie helps to make the set E of all separating sequences suffix-closed. If \bar{S} is prefix-closed and E is suffix-closed, then the conjectured

model is consistent with such DT; this also holds for the L^* algorithm, see Section 16.1.

The difference from the DT algorithm is the processing of a counterexample. Therefore, the learning is the same up to the first equivalence query. Initialization and identification of transitions by sifting is described in Section 16.2. In the case of the reference DFA (Figure 15.5), Figure 16.4 shows the first output queries with the related DT and the consistent conjectured model.

An obtained counterexample is split to a prefix u , an input x , and a suffix v according to Theorem 14.1. Let s_u be the state reached by u in the conjectured model M , that is, $s_u = \delta_M^*(s_0, u)$. Then the next state of s_u on x is observed to be a new state as it is distinguished from the current next state $\delta_M(s_u, x)$ by the separating sequence v . The original paper does not propose the way how the decomposition is derived. A function similar to *OneSuffix-binary search* (Section 16.1.2) is thus employed. The access sequence of the new state is formed from the access sequence of s_u appended with x instead of taking $u \cdot x$ directly. This ensures the prefix-closedness of all access sequences that are stored in the leaves of the DT and the spanning tree. The suffix v is used as a temporary discriminator that labels the new inner node of the DT. A discriminator is another name for a separating sequence introduced by the authors of TTT. The new inner node replaces the leaf related to $s_i = \delta_M(s_u, x)$, the leaf is shifted under the new inner node and the second leaf is created for the new state. The conjectured model is then updated accordingly. All transitions leading to s_i are confirmed by sifting through the new subtree, that is, an output query with the separating sequence v is asked for each transition. Sifting then also identifies transitions from the new state.

Figure 16.10 shows processing the counterexample ‘aab’ obtained for the 2-state conjecture depicted in Figure 16.4. The entire CE is first queried to obtain the correct response on it (output query 7). Then the CE is split into halves, that is, a prefix (‘a’) and a suffix (‘ab’) of about the same length. The access sequence of state reached by the prefix is used instead of the prefix in the next output query on the suffix. Output query 8 is thus $T(\varepsilon, ab)$ as $\delta_M(s_0, a) = s_0$ and ε is the access sequence of s_0 . The output differs from the response of the black box on the entire CE. The CE is decomposed into the prefix u (ε), the input symbol x (‘a’) and the suffix v (‘ab’). Note that the processing function would check a longer suffix if the prefix $u \cdot x$ was not only one symbol so it was possible to split it and consider a longer suffix. The new state s_2 separated by ‘ab’ from the initial state s_0 gets ‘a’ as the unique access sequence because it is reached by ‘a’ from s_0 that has the access sequence ε . If the CE was ‘aaaab’ and its decomposition $\langle aa, a, ab \rangle$, then the access sequence of the new state would still be ‘a’ as $\delta_M^*(s_0, aa) = s_0$.

The DT is updated in the described way. The separating sequence ‘ab’ labels the new inner node that has two leaves as successors. One leaf relates to the initial state and the second to the new state s_2 . All transitions leading to s_0 (that was split) are updated in the conjectured model (output queries 10 and 11) and transitions from s_2 are identified by sifting (output queries

Output Queries

- 7 $T(\varepsilon, aab) = 000$
- 8 $T(\varepsilon, ab) = 01 \implies s_2$
- 9 $T(a, \uparrow) = 0$ (state output)
- 10 $T(ba, ab) = 01$ (update model)
- 11 $T(bb, ab) = 01$ (update model)
- 12 $T(aa, \uparrow) = 0$ (sift)
- 13 $T(aa, ab) = 01$ (sift)
- 14 $T(ab, \uparrow) = 1$ (sift)

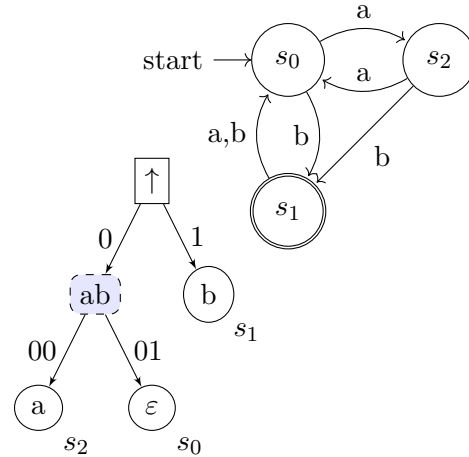


Figure 16.10: Update of the DT and the model after state s_2 is revealed

12–14). The updated discrimination tree and the conjectured model are shown in Figure 16.10.

The main improvements over the DT algorithm are a check of the consistency between the conjectured model and the DT, and a refinement of the DT according to the discriminator trie.

All outputs labelling edges of the DT are checked against the output function of the conjectured model during the consistency check. It means that for each state s_i and each edge e of the DT on the path from the root to the leaf related to s_i such that z labels e and v is the separating sequence of node where e starts, $\lambda_M^*(s_i, v)$ is checked if it equals to z . If not, the conjectured model is not consistent with the DT and the access sequence of such s_i extended with v is used as a new counterexample. This is captured in Figure 16.10 as $\lambda_M^*(s_2, ab) = 01$ but the output should be 00 according to the current DT. Therefore, the counterexample ‘aab’ is employed again. The suffix ‘b’ is identified as a new separating sequence and the last state s_3 is revealed, see Figure 16.11. As the DT is again updated in place of the leaf relating to s_0 , both transitions leading to the initial state are queried with the new separating suffix ‘b’ (output queries 17 and 18). Transitions from s_2 are then identified by sifting through the updated DT (output queries 19–22).

The last three output queries are used in the process of refinement of the Discrimination Tree. Each new separating sequence is added to the DT as a temporary discriminator (blue dashed inner node in the figures). All discriminators need to be as short as possible to reduce the number of queried symbols during sifting. Moreover, for each discriminator $w = x \cdot v$ such that w separates states s_i and s_j there is a sequence v separating the next states $\delta_M(s_i, x)$ and $\delta_M(s_j, x)$. This holds for each complete minimal machine. The algorithm tries to shorten each new discriminator such that the updated discriminator is formed from a used separating sequence prepended with an input symbol. Used separating sequences are stored in the discriminator trie. Initially, it contains only ε in the case of Mealy machine or \uparrow if the black box

Output Queries

- 15 $T(\varepsilon, b) = 1 \implies s_3$
- 16 $T(aa, \uparrow) = 0$ (state output)
- 17 $T(ba, b) = 1$ (update model)
- 18 $T(bb, b) = 0$ (update model)
- 19 $T(aaa, \uparrow) = 0$ (sift)
- 20 $T(aaa, ab) = 00$ (sift)
- 21 $T(aab, \uparrow) = 0$ (sift)
- 22 $T(aab, ab) = 00$ (sift)
- 23 $T(a, b) = 1$ (finalization)
- 24 $T(aaa, b) = 1$ (finalization)
- 25 $T(aab, b) = 1$ (finalization)

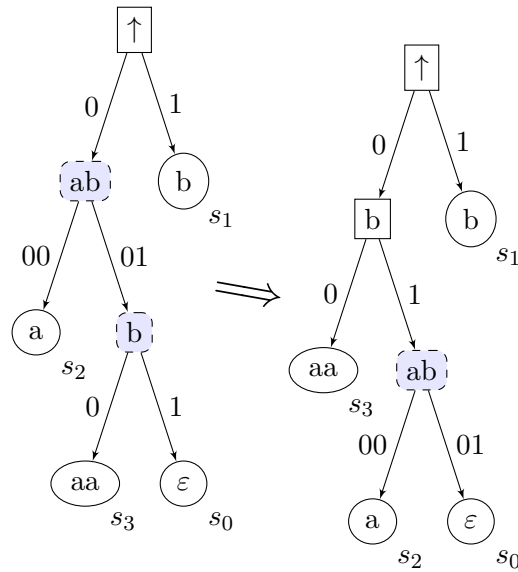


Figure 16.11: The inconsistency reveals state s_3 and the DT is then finalized

has outputs by states. If the algorithm is unable to find a suitable shorter discriminator, the new discriminator remains temporary. The reason is the inconsistency of the conjectured model and the DT. This is the case of the sequence ‘ab’ separating states s_0 and s_2 in the running example. Its inner node of the DT shown in Figure 16.10 remains temporary until the consistency check finds a counterexample and the DT is updated as Figure 16.11 shows.

There can be several temporary discriminators. The algorithm looks gradually at the roots of temporary subtrees, that is, the temporary discriminators whose parents are not temporary. If there is a suitable discriminator that separates some of the states in the temporary subtree, the shortest discriminator replaces the one in the root of the subtree. The chosen discriminator is added to the discriminator trie so it becomes ‘used’. A suitable discriminator means that all used separating sequences remain suffix-closed if the discriminator is added to them. After a shorter discriminator w is chosen to replace the temporary one in the inner node n_i , the subtree is refined according to the outputs of related states on the chosen separating sequence w . If several states in the subtree have the same output on w , then the discriminator of their common ancestor labels a new successor of n_i . This successor becomes the root of a subtree in which all these states are distinguished. As some state-relating leaves change their position in the DT, additional output queries are required. Particularly, Figure 16.11 shows the first transformation of the DT. The algorithm first looks at the first child of the root. States in the subtree are s_2 , s_3 and s_0 . They are distinguished by ‘b’ and as ‘b \uparrow ’ is ‘b’ for DFA so the used separating sequences remain suffix-closed if ‘b’ replaces ‘ab’. State s_2 was not identified by the separating sequence ‘b’, therefore, all transitions leading to it need to be queried with the suffix ‘b’ (output queries 23–25). This verifies transitions according to the updated DT shown

on the right in Figure 16.11. Finally, the temporary discriminator ‘ab’ is found suitable to be added to the discriminator trie. The 4-state conjectured model is consistent with the DT so the second (and the last) equivalence query is asked. The consistency is checked after the refinement stops when no temporary discriminator can be replaced or all discriminators are in the discriminator trie.

The proposal of the TTT algorithm is not clear in some aspects. Therefore, some changes and specifications were done, such as the choice of the CE decomposition function or the update of the conjectured model after a change of the DT. The reference machine (Figure 15.5) is learnt using 2 EQs, 25 output queries, 25 resets and 64 symbols queried in total.

16.5 Quotient algorithm

Petrenko et al. proposed a new inference approach in [PLG⁺14]. The algorithm is called here the Quotient algorithm as it gradually infers partial models, quotients, of the given black box. The structure for learning is an observation tree so that many output queries are saved because no sequence is queried twice or more. The idea of the algorithm was captured in the Direct hypothesis construction (DHC) algorithm that was proposed previously in [SHM11]. The Quotient algorithm inspired the development of the observation tree approach (Chapter 15) and it implements the approach but just the first phase as no extra states are considered during the learning.

The algorithm implements the W-method in the context of active learning. The W-method is the oldest testing method that uses the same characterizing set W to verify all states reached by particular access sequences. In fact, the Quotient algorithm does the same as the L* algorithm but it stores observed traces in the observation tree (OTree) instead of the observation table. Besides the observation tree, the algorithm keeps the set E of separating sequences. All sequences of E are queried from each state that is reached and needs to be identified; the node of the observation tree associated with such a state is ‘extended’ by E . The set E contains initially all symbols of the input alphabet X and then is extended by separating suffixes of obtained counterexamples.

Each node of the observation tree is either a state node or consistent with a state node. All state nodes are distinguished, that is, for each pair of state nodes there is a common sequence leading from both state nodes such that the outputs along the paths are different. Two nodes are consistent if they are not distinguished in the observation tree.

The learning by the Quotient algorithm is described on the reference DFA (Figure 15.5). Figure 16.12 shows the observation tree after the first 3 output queries that identify the initial state. The algorithm starts with the root of observation tree and the conjectured model contains just the initial state s_0 . If the black box has outputs by states, the first output query asks for the output of the initial state. The root of observation tree is extended by E so that a unique identification of the initial state s_0 is obtained. In the example, s_0 is recognized by $\{\uparrow \rightarrow 0, a \rightarrow 0, b \rightarrow 1\}$, see Figure 16.12. Nodes

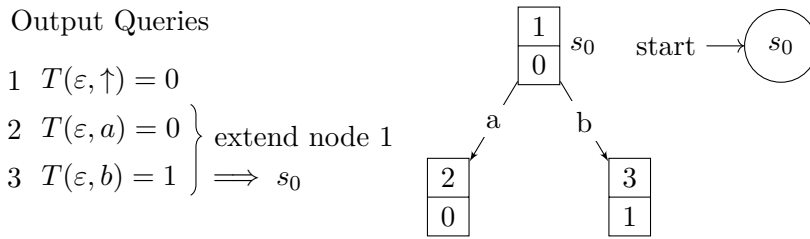


Figure 16.12: The first 3 output queries identify the initial state s_0

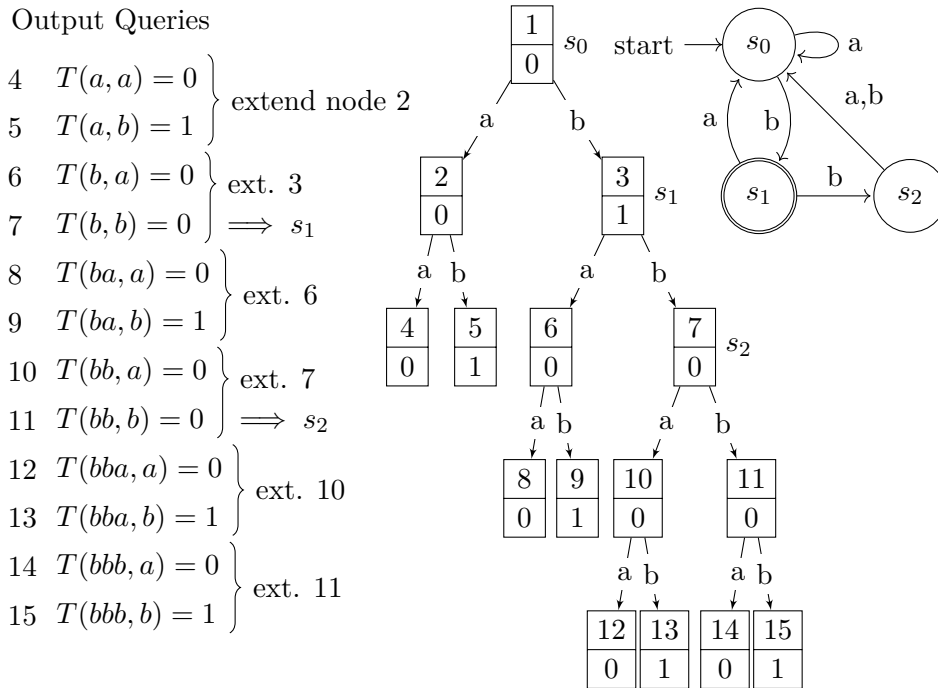


Figure 16.13: The observation tree and the conjecture after 15 output queries

of OTree are numbered according to their creation, that is, by the number of the related output query.

The aim of the algorithm is to make the observation tree closed (for 0 extra states). An observation tree is closed (for 0 extra states) if all successors of state nodes are extended by E and are consistent with state nodes. Therefore, nodes 2 and 3 are extended using output queries 4–7 to identify the next states of s_0 , see Figure 16.13. The subtree of the extended node is then compared with subtrees of state nodes so a consistent one is found. If a node is distinguished from all state nodes, then this node becomes a new state node and the conjectured model is enlarged by one state. This is the case of node 3 in Figure 16.13. The output assigned to the node differs from the state output of s_0 (node 1). Therefore, a new state s_1 is revealed. Transitions from the new state need to be identified. Hence, its successors are extended by E ; output queries 8–11 extend nodes 6 and 7 that are successors of node 3 relating to s_1 . This procedure is repeated until the observation tree is closed.

Output Queries

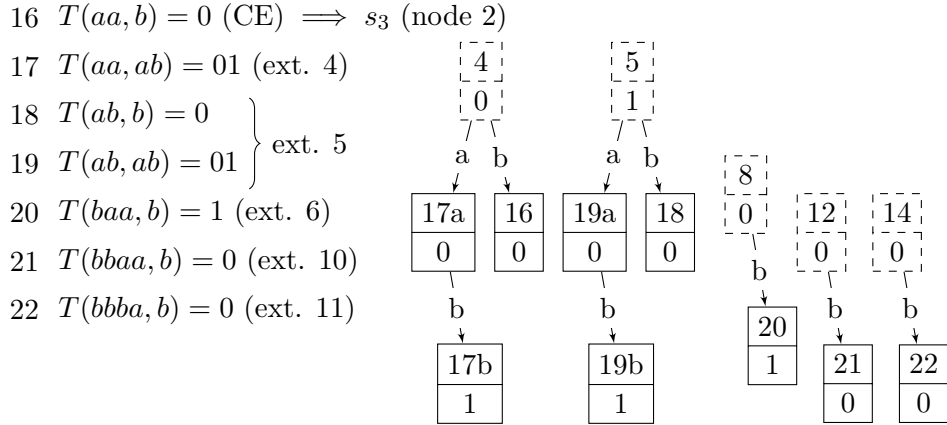


Figure 16.14: Identification of next states using the new separating suffix ‘ab’

The conjectured model is checked for consistency with the closed observation tree. Access sequences of states are prefix-closed because only a successor of a state node can become a new state node. All successors of state nodes are identified by E and consistent with state nodes. Therefore, the conjectured model is well defined. The consistency is thus checked for all children of state nodes’ successors. All these nodes need to be consistent with a state node determined by the transition function δ_M . Formally, a node reached by u from s_0 cannot be distinguished from state node relating to $\delta_M^*(s_0, u)$. For example, node 4 is checked whether it is consistent with s_0 as $\delta_M^*(s_0, aa) = s_0$ according to the conjectured model in Figure 16.13. If an inconsistency is found, there is a sequence w distinguishing the node from the assumed state node. The sequence w extends the set E and thus the observation tree becomes unclosed. The entire procedure of ‘closing’ by applying E in all successors of state nodes is repeated again. This inconsistency would be the type II if domains were used, see Section 15.2.2 for the types of inconsistency between the observation tree and the conjectured model.

An equivalence query is asked after the observation tree is closed and the conjectured model is consistent with it. The obtained counterexample is processed easily. It is added to the observation tree by querying the suffix that is not yet in the tree. Then, the current conjectured model is checked against the updated observation tree so that an inconsistency is found. The inconsistency is again fixed by the extension of E and ‘closing’ the observation tree.

The counterexample ‘aab’ is obtained to the first equivalence query in the running example. Output query 16 then asks for its unobserved suffix and the last state s_3 is revealed by the consistency check. Next 6 output queries make the observation tree closed as Figure 16.14 shows. The learning ends by the second equivalence query.

The Quotient algorithm learns the reference DFA using 2 equivalence queries, 22 resets and output queries and 66 symbols queried in total.

16.6 GoodSplit algorithm

The GoodSplit algorithm was proposed by Eisenstat and Angluin for the ZULU competition in [EA10]. The aim of the competition was to learn a DFA with the restricted number of membership queries and without equivalence queries. Therefore, the GoodSplit algorithm is a representative of unsupervised active-learning algorithms. The number of MQs allowed to ask in the competition was not big enough to learn the model properly or provide a guarantee like the observation tree approach does (extra state coverage). The idea behind was thus to infer a very accurate model as soon as possible.

The algorithm was proposed for learning DFA only, therefore, its adjustment is done to learn DFSMs and work with both types of output queries. Moreover, a CE processing function is added so that all learning algorithms can be compared as the last EQ proves the correctness of the conjectured model.

A set of observed traces was originally used for learning. Nevertheless, the implementation in FSMlib (Appendix A) uses an observation tree instead as it is more compact and several actions can be done easier with this structure, for example, a comparison of states reached by different access sequences.

There are four steps repeated until a stop condition is met. The steps are: 1) find a consistent state node for each next state, 2) identify next states, 3) check the stop condition and update distinguishing sequences, and 4) make random queries. The stop condition was originally the given number of allowed output queries. Nevertheless, the version extended by the author of the thesis receives a parameter *maxDistinguishingLength* according to which the algorithm stops when the length of distinguishing sequences exceeds its value. If equivalence queries are allowed, then an EQ is asked after the length of *maxDistinguishingLength* is reached. The CE processing is described later.

The first step is to find a consistent state node for each successor of state nodes. A state node is a node of the observation tree that represents a particular state of the conjectured model. Successors of state nodes then relate to next states. Two nodes are consistent if they are not distinguished in the observation tree, that is, there is no common sequence leading from both nodes such that outputs along this sequence are different. The step also covers querying the sequences leading to the next states. Figure 16.15 shows the first step in the case of learning the reference DFA (Figure 15.5). After the output of the initial state s_0 is observed, both transitions are queried. The third output query reveals a new state s_1 as its output differs from the one of s_0 . The conjectured model is thus updated and transitions leading from s_1 are queried (output queries 4 and 5). All successors are consistent with a state node (all with s_0) as there are no two states with the same state output. Moreover, all successors are consistent with just one state node and thus they are identified. Hence, the second step does not do anything now and will be explained by its use in the running example.

The algorithm possesses a variable l that determines the current length of distinguishing sequences in use. The set E contains all such sequences. Initially, $l = 1$ so E is filled up with all input symbols. Then it depends on

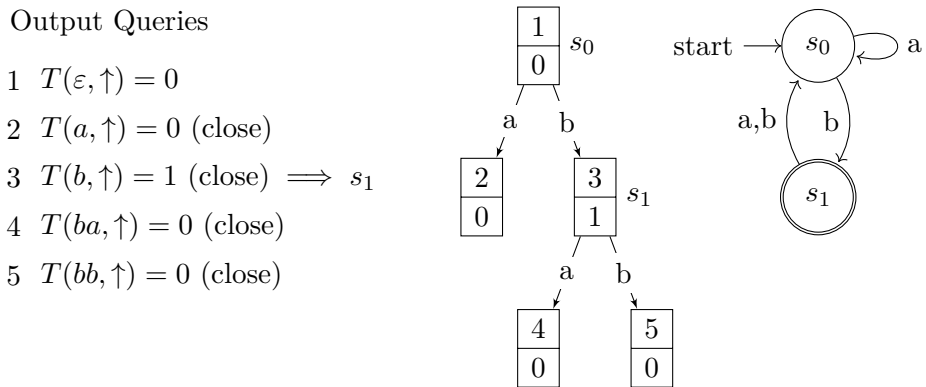


Figure 16.15: The first 5 output queries close the observed traces

the type of output queries T . If only membership queries are allowed, then E contains all sequences of the length up to l , that is, $E = X^{\leq l}$. Otherwise, E includes all sequences of the length l , that is, $E = X^l$; output queries provide an output sequence so the response to the prefixes of queried sequences is obtained as well. Sequences of E are successively queried from the successors of state nodes to distinguish them and to reveal a new state.

The third step of the algorithm checks if l is to be increased and if the stop criterion is met. The variable l is increased by 1 if the fraction of applied extensions and all possible extensions of successors of state nodes (such that the extensions are sequences of E) is greater than 90 %. In the running example, the observation tree in Figure 16.15 has 3 leaves that represent successors of state nodes. Each of these leaves can be extended by both sequences of $E = \{a, b\}$. This gives 6 possible extensions. None of these was applied, that is, no sequence of E was queried from any successor of a state node. All 6 extensions would need to be queried in order to increase l by 1. If l is increased and it becomes greater than the given parameter *maxDistinguishingLength*, then the algorithm either stops or asks an EQ (if it is allowed).

The learning of the reference machine continues with the fourth step because the condition of the third step for increasing l is not met. The purpose of the fourth step is to reveal a new state. A successor n_i of a state node and a sequence e of E such that e is not queried from n_i are chosen at random. The sequence e is then queried from n_i and from the state node n_j that is consistent with n_i if e was not queried from n_j . There is only one such n_j because the second step ensures that the successors of all state nodes are consistent with only one state node. The number of randomly extended successors depends on the number of states in the conjectured model. The fourth step makes $\lceil |S|/2 \rceil$ random choices of successors to extend. An exception to this is if there are not so many available extension or a new state was revealed by the queried sequence. The latter case is an improvement to the original algorithm. It forces the algorithm to update the conjectured model first instead of querying next random sequences.

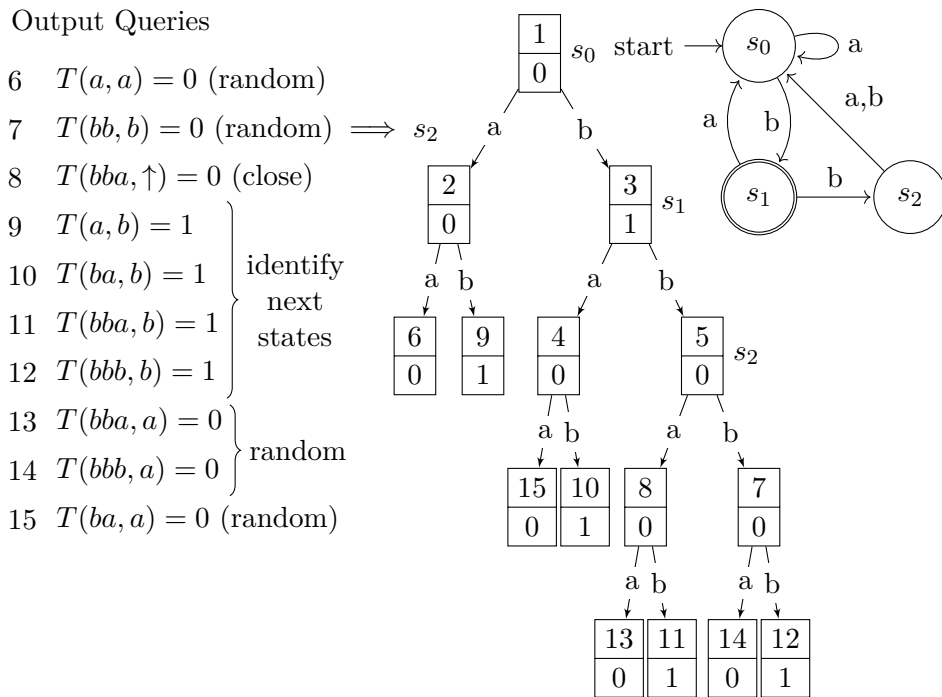


Figure 16.16: All sequences of length 1 were queried after 15 output queries

The conjectured model in Figure 16.15 has 2 states, therefore, just one successors is extended in the fourth step. Output query 6 (Figure 16.16) does not reveal any new information. The next output query is asked again by the fourth step because all next states are still identified and only 1 of 6 possible extensions was applied so no previous step is required. Fortunately, the query reveals a new state s_2 . Therefore, the conjecture is updated and the first step ensures that both transitions from s_2 are queried (output query 8).

The next 4 output queries asked by the second step of the algorithm identify the next states. The identification is needed for successors that are consistent with several state nodes. States s_0 and s_2 have the same state output, therefore, they need to be distinguished by a separating sequence. The sequence is chosen from the set E . Only $b \in E$ separates the states. Therefore, it is applied in all successors of state nodes as they all are consistent with both states nodes. The choice of a separating sequence is described later in general.

The conjectured model has now 3 states and all transitions are identified as it is shown in Figure 16.16. There are 8 possible extension of state nodes' successors and 5 were applied. Hence, l is not increased and the fourth step selects 2 successors to extend ($3 \text{ states} \rightarrow \lceil 3/2 \rceil = 2$). A new state is not revealed so the condition of the third step is checked again. Because 7 of 8 extensions is still lower than 90 %, the algorithm continues with the fourth step. It should ask 2 random queries but there is only one possible extension. Therefore, output query 15 is asked and the algorithm starts the main loop again. There is no change to the conjecture so the first two steps is skipped.

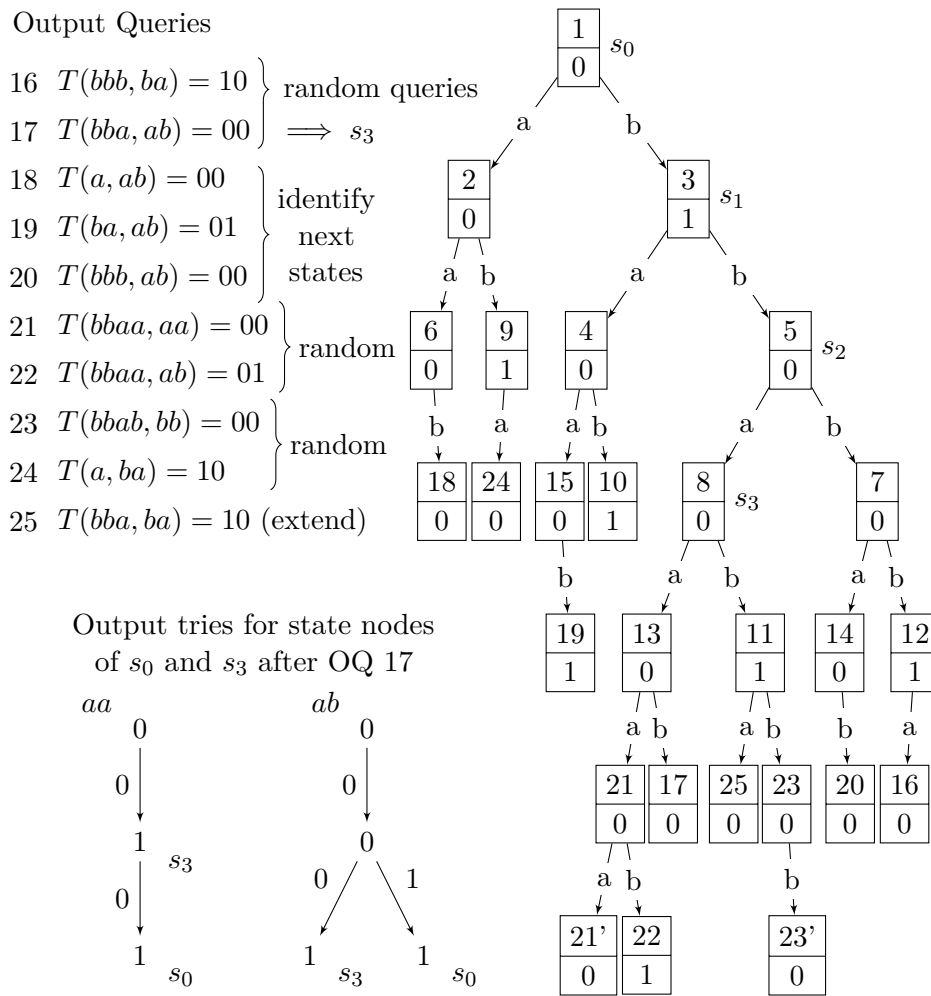


Figure 16.17: Sequence ab reveals the last state

if the entire e was not observed from the node. In fact, $m_{e/z}$ represents the number of state nodes that will not be distinguished if e is applied and z is observed. Therefore, if only a prefix of e was queried and z' was observed, then each $m_{e/z}$ such that z' is a prefix of z needs to include such state node. Moreover, the sequences are compared on $r = c_i - g_e - d_e$, the higher r the better. Therefore, the maximum number g_e needs to be the lowest possible. This implies that all $m_{e/z}$ need to be small in order to e be chosen. It means that it is better if the sequence e was queried from all consistent state nodes. Output tries of sequences 'ba' and 'bb' are similar to the one of 'aa' so their evaluation result is 0 as well; $c_i = 2$, $d_e = 0$ and $g_e = 2$. Therefore, 'ab' is selected as its $r = 2 - 1 - 0 = 1$ is the highest. The described rules for selecting a separating sequence improve the original ones that were proposed for DFA (binary outputs) and MQs only.

Further random extensions are applied after all transitions of the 4-state conjectured model were defined. The algorithm repeats the fourth step until 18 of 20 possible extensions are applied. Then an equivalence query proves

that the conjectured model is correct if *maxDistinguishingLength* is 2 and EQs are allowed. The first step is processed again after the main loop ends to make the conjecture consistent with the observation tree. Notice that output query 25 in Figure 16.17 asks ‘ba’ in the state node of s_3 because the sequence was randomly selected to be queried from node 2 (output query 24) that is identified as s_3 . Each random query needs to be also observed from the state node that correspond to the successor from which the random query is asked.

Counterexamples obtained from EQs are processed similarly to other algorithms. EQs were not allowed originally so that this is an extension of the algorithm. A separating suffix v is first identified by the *SuffixAfterLastState* function (Section 16.1.5). Then *maxDistinguishingLength* and l are updated to *maxDistinguishingLength* = $l = |v|$ and v is queried from the related nodes. The first node is a state node relating to $\delta_M^*(s_0, u \cdot x)$, where $u \cdot x \cdot v$ is a decomposition of the given counterexample. The second node is the successor of state node relating to $\delta_M^*(s_0, u)$ on x . This ensures that a new state is revealed and the separating sequence of the new state is included in E as l was enlarged accordingly.

The GoodSplit algorithm with *maxDistinguishingLength* = 2 learned the reference machine with 35 output queries and 35 resets, 1 EQ and 136 symbols were queried in total. Note that the order of randomly selected extensions can be different and thus the numbers differ as well.

16.7 Other Learning Algorithms

There are many other learners in the literature. Prior to the L* algorithm, Angluin proposed the ID algorithm [Ang81]. It assumes that the access sequences of all black-box states are given. One thus needs to learn only separating sequences. The IDS algorithm [MS10] is an incremental version of the ID algorithm. An incremental learning algorithm produces a sequence of conjectured models that finitely converge to the model of black box. In addition, an incremental learning algorithm reuses knowledge of one conjectured model to create a subsequent version.

Both the ID and IDS algorithms work with predefined set of sequences. It is similar to passive inference where a learner infers a model from given traces. Examples of passive learning algorithms are RPNI [OG92] and RPNI2 [Dup96]. A combination of passive and active inference was described in [DLDL08] such that the proposed Query-driven State Merging (QSM) algorithm learns from given traces but is allowed to pose additional membership queries. The Congruence Generator Extension (CGE) algorithm [Mei10] is similar to the incremental RPNI2 but it uses string rewriting systems representing a finite congruence generator to represent and manipulate automata. Hence, the conjectured model state set is more compact than equivalence relation on input strings used by the other learners.

Theory of testing was incorporated to learning by Zhu in [Zhu96] that interprets the axioms of test adequacy criteria as properties of inductive inference to relate it with software testing. The learning-based testing (LBT)

was then proposed in [Mei04]. A LBT approach is a heuristic method that is based on learning a black-box system using tests as queries. An example of LBT method is the IKL algorithm [MS11] that learns Kripke structures modelling reactive systems. An exhaustive survey of approaches considering both learning and testing is described in [AMM⁺18].

16.8 Summary of the Algorithms

This section compares the active-learning algorithms in two aspects. First, it summarizes the main differences amongst the algorithms. Second, it describes how they implement the General Learning Framework (GLF) proposed in Section 15.6. There are several surveys of learning automata and finite-state machines in the literature. An insight and a comparison of variants of L^* is proposed in [BJLS05]. De la Higuera proposed several overviews: [DIH05] states the use of grammar inference, [DIH10a] includes an extensive seminar on learning and several notions from the seminar are then discussed in [DIH10b]. Similarly, Prajapati summarized learning models and approaches to grammar inference in [Pra11].

All learning algorithms work with two sets, a state cover of access sequences and a set of separating sequences. The algorithms differ in the way how they handle these two sets. The difference is captured for some learners in Table 16.1. All the standard learning algorithms use a fixed state cover and a fixed set of separating sequences. In contrast, the three learners based on the observation tree approach are more flexible as they allow to choose separating sequences or access sequences on the fly. Hence, they can optimize the amount of interaction with the black box.

Algorithm	State cover	Separating sequences
L^* , Quotient	fixed	fixed CSet
DT, TTT	fixed	fixed HSI
H-learner	fixed	chosen on the fly
SPY-learner	chosen on the fly from convergent sequences	fixed HSI
S-learner	chosen on the fly from convergent sequences	chosen on the fly

Table 16.1: Access and separating sequences used by the learning algorithms

Table 16.2 describes implementations of the GLF by each of 7 learning algorithms. All standard algorithms select a transition (s, x) leading from a state $s \in S$ in the step *Choose*. The learner based on the observation tree approach also deals with transitions leading from next states if they assume extra states. The step *Identify* depends on the structure that stores the set E . Separating sequences are chosen from E to identify the reached state. The structures for learning are then updated in the step *Update*. This finishes the inner loop as Algorithm 54 describes. The step *Check* depends on the result of asked equivalence query in most cases. If a counterexample is obtained, it is processed. Moreover, additional procedures like a consistency check precede that in some cases.

Algorithm	Choose	Identify	Update	Check
L*	select row of a next state	fill the row using T	find equal row or extend S	EQ + OT extension according to the CE
DT	select (s, x)	sift $s \cdot x$	update the conjecture or the DT	EQ + DT update according to the CE
OP	select (s, x)	sift $s \cdot x$ + fill the row in the reached CT_i	update the conjecture or the DT and split CT_i	EQ + DT and CT_i 's update according to the CE
TTT	select (s, x)	sift $s \cdot x$	update the conjecture or the DT	EQ + DT update, consistency check and finalization of discriminators according to the CE
Quotient	select (s, x)	apply E		fix inconsistency or add the CE after EQ
GoodSplit	select (s, x)	apply the most distinguishing sequence	reduce the set of successor's consistent nodes	check l and $maxDistinguishingLength$ + random queries
H-learner SPY-learner S-learner	select (s, x) with the shortest s	apply the most informative sequence	reduce the domain of the next state	assume one more extra state

Table 16.2: Interpretation of GLF by the learning algorithms

Chapter 17

Experiments

The three new learners proposed in Chapter 15 are tested and compared against the standard active-learning algorithms (Chapter 16) on three case studies described in Appendix C. At first, the algorithms are experimentally evaluated on randomly generated machines. Then, they learn several models of real systems and the last experiment is to learn a model of the artificial environment GridWorld. This chapter presents the results on these three case studies.

The learners are allowed to ask output queries (OQ) in all three case studies, that is, they obtain a sequence of outputs returned by black box in response to the given input sequence. The teacher in the first two case studies knows the model to be learnt, therefore, the shortest counterexample is always returned in response to an equivalence query if the provided conjectured model differs from the black box. The teacher in the third case study does not know the environment as it is provided by an external tool, hence, equivalence queries are approximated by the SPY-method (Section 9.7). The teachers in the first two and the third case studies are implemented by TeacherDFSM and TeacherBB, respectively (their implementation is described in Appendix A).

There are several measures which the algorithms can be compared on. The numbers of equivalence and output queries are the standard ones. However, the research question RQ III.2 asks for a comparison on the amount of interaction. The interaction with a black box is better captured by the number of resets of the black box and the total number of input symbols that were queried. The total number of queried input symbols is abbreviated to ‘Symbols’ in the results and it excludes the stOut symbol \uparrow . The implementation of learners asks \uparrow after each regular input symbol in the case of machines with state outputs but no additional knowledge is provided when one works with Moore machines or DFA. Learning Mealy machines would seem to be twice faster than learning of the corresponding Moore machines if the stOut symbol counted on. The learning time, or the time that is needed to learn the correct model by a learner, is also one of common measures. A new measure is the *exploration efficiency* (EE). It is developed by the author of the thesis in order to capture the efficiency of the learner’s queries to explore the black box. It is calculated as the number of edges in the observation tree divided by the total number of queried symbols. Hence, it permits one to evaluate how much of

Algorithm	EQs	OQs	Resets	Symbols	EE [%]
L* _{AllPrefixes}	2	33	33	105	26.7
L* _{AllSuffixesAfterLastState}	2	28	28	78	29.5
L* _{Suffix1by1}	2	29	29	81	28.4
L* _{SuffixAfterLastState}	2	29	29	80	28.8
L* _{OneSuffix-binarySearch}	2	29	29	80	28.8
OP _{AllGlobally}	2	68	68	199	16.6
OP _{OneGlobally}	2	52	52	136	16.9
OP _{OneLocally}	2	47	48	120	17.5
DT	3	32	32	76	23.7
TTT	2	25	25	64	31.3
Quotient	2	22	22	66	34.8
GoodSplit: $l = 2$	1	35	35	136	26.5
H-learner: 0 ES	2	20	10	36	52.8
H-learner: 1 ES	1	34	17	68	48.5
SPY-learner: 0 ES	2	23	5	26	84.6
SPY-learner: 1 ES	1	39	11	57	66.7
S-learner: 0 ES	1	21	6	26	76.9
S-learner: 1 ES	1	36	11	55	63.6

Table 17.1: Comparison of the learners on the 4-state DFA (Figure 15.5)

the black box is explored and how much effort was put in it. The greater the value, the better the learner is. Note that if a learning algorithm, such as the L* algorithm, does not use the observation tree (OTree), the queried input sequences can be still put in a prefix tree (which OTree is) and calculate the exploration efficiency.

All the learners were explained on a single machine, the 4-state DFA defined in Figure 15.5. Moreover, the amount of interaction needed for the learning of that machine concluded the running example of each learner. Therefore, Table 17.1 summarizes the first comparison of the described learning algorithms. The last six rows of Table 17.1 presents the result of the three new learners. As the learners use the observation tree for learning and ask output queries for single inputs, the number of output queries equals to the number of nodes of the resulting observation tree. The number of edges of OTree is the number of nodes minus one. Then, it is easy to compute the exploration efficiency (EE) in the last column of Table 17.1 for the new learners. For example, the EE of the S-learner assuming 1 extra state (ES) is $(36 - 1)/55$.

Table 17.1 shows promising results for the new learners that are based on the observation tree approach. The H-, SPY- and S- learners need much less interaction with the black box than any standard learning algorithm if they do not consider extra states ('0 ES' versions). Moreover, if they assume 1 extra state, they still learn with less interaction and they would not need the help of teacher, that is, the learn without any counterexample. Note that

any algorithm asks at least one equivalence query (if it is allowed) in order to confirm that the conjectured model is finally correct, that is, the EQ is the last one just before the learning stops. The GoodSplit algorithm and the S-learner with 0 ES also need only 1 EQ.

17.1 Randomly Generated Machines

The first case study consists of 13 600 randomly generated machines. The machines are described in detail in Appendix C.1. They were generated using the generator in the FSMlib (Appendix B). They can be grouped according to their machine type, the number of states and the number of inputs. There are 4 machine types, each represented equally with 3 400 machines such that a half of them has 5 inputs and the other 1 700 machines have 10 inputs. The number of states ranges from 10 to 1000 and there are 17 state groups of 100 machines. Each machine is learnt by each learner and the results are analysed statistically for each state group, that is, quartiles are calculated over 100 values. All machines and the results are available in the repository FSMmodels v1.3¹.

Figures 17.1–17.3 show the result for deterministic finite state machines (DFSM) with 5 inputs. The boxplots on the right of each figure present the values of machines with 1 000 states such that the whiskers represent the minimum and maximum values. As the teacher provides the shortest counterexample (CE) in response to an EQ, all CE processing functions of the L* algorithm that find a distinguishing suffix perform equally. The AllPrefixes function (Section 16.1.1) is a little worse than the others, hence, it is shown separately. The performance of the other CE processing function is captured by Suffix1by1 (Section 16.1.4). The results of the GoodSplit algorithm (Section 16.6) are not shown at all because it asks much more output queries than the other learners and so it was not possible to learn machines with hundreds of states.

Figure 17.1 captures the standard measures, that is, the numbers of output and equivalence queries. On the one hand, one can see that all the three new learners outperform the standard learning algorithms in terms of the number of output queries when they do not consider extra states, that is, they follow only the first phase of the observation tree approach. On the other hand, the new learners assuming 1 extra states are the only ones that can learn a correct model without any counterexample provided by the teacher.

The numbers of output and equivalence queries depend on each other. This can be seen in Figure 17.1 on the performance of the learning algorithms that use the discrimination tree (DT). The DT and TTT algorithms ask a lot of EQs so that almost all states of the black box are revealed using counterexamples provided by the teacher. This has a positive effect on the number of output queries that is quite low. The trade-off between the numbers of output and equivalence queries is also captured by the versions of the

¹<https://github.com/Soucha/FSMmodels/releases/tag/v1.3>

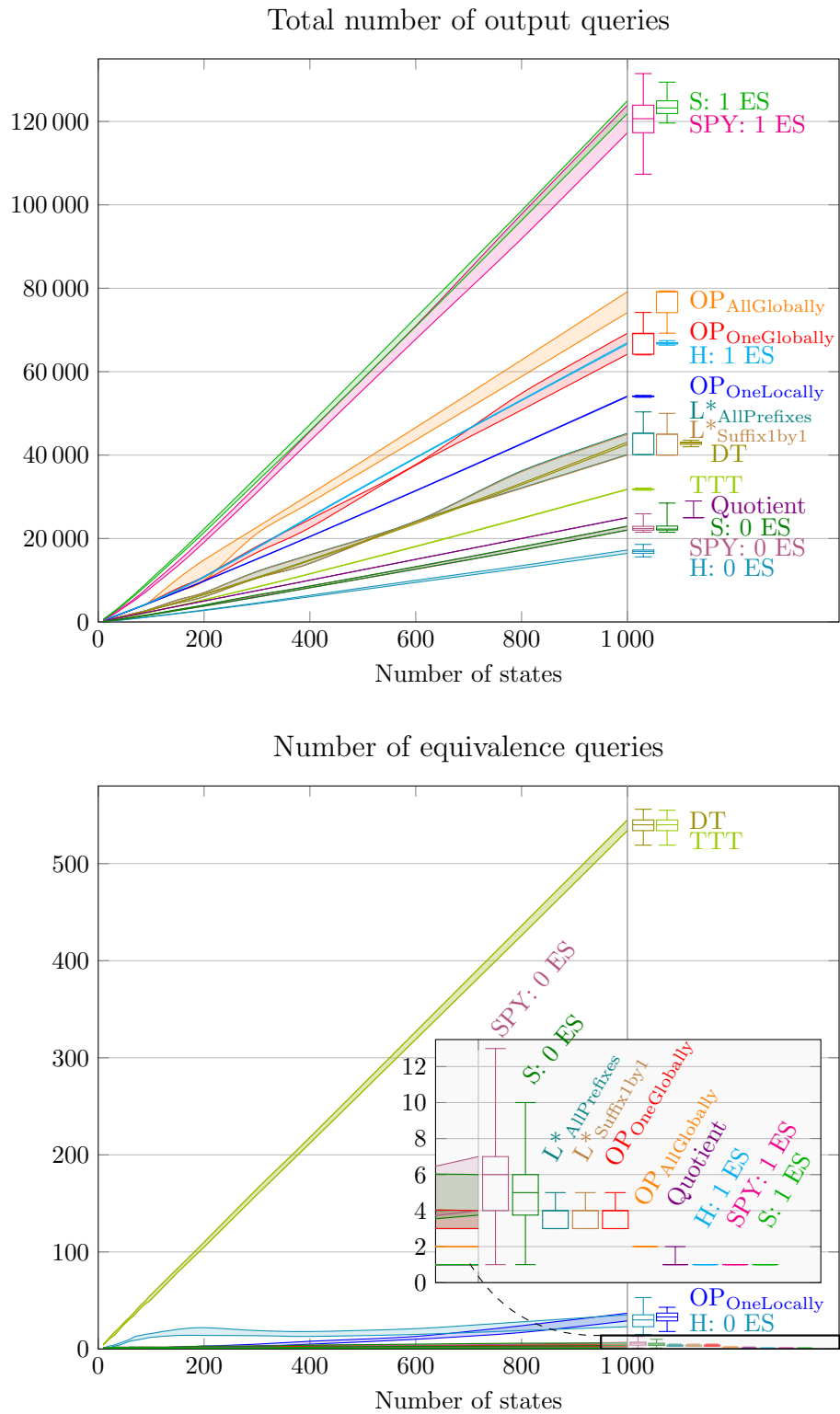


Figure 17.1: DFSMs with 5 inputs: the numbers of output and equivalence queries, 1 and 3 quartile calculated for 100 machines per each state group

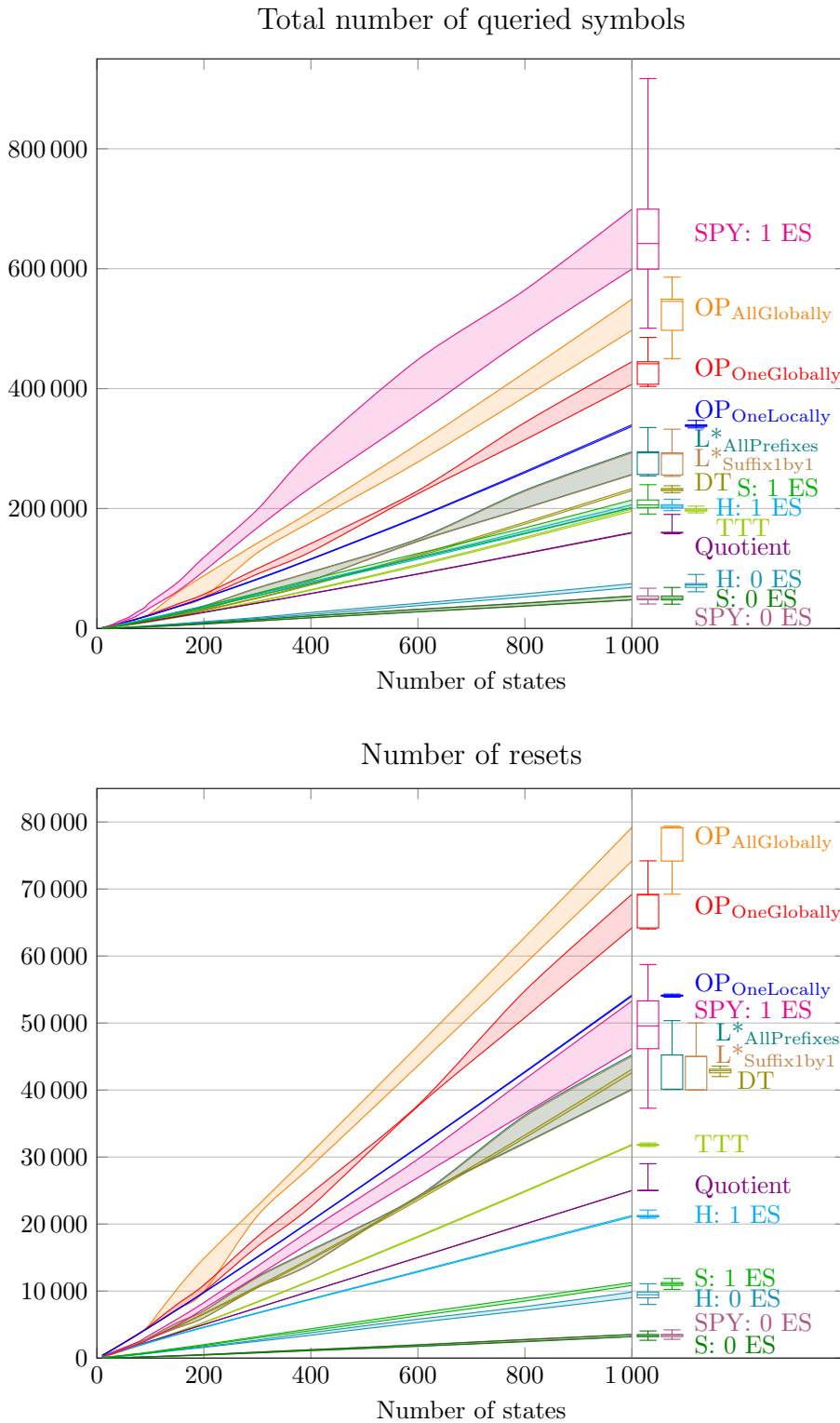


Figure 17.2: DFSMs with 5 inputs: the total number of symbols queried during the learning and the number of resets of the black box, 1 and 3 quartile calculated for 100 machines per each state group

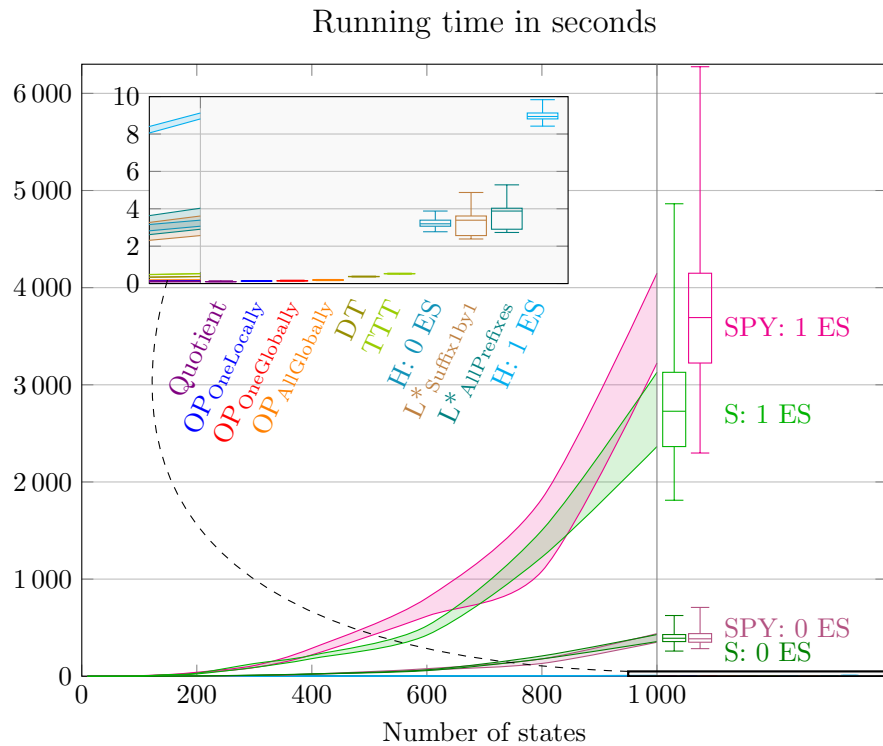
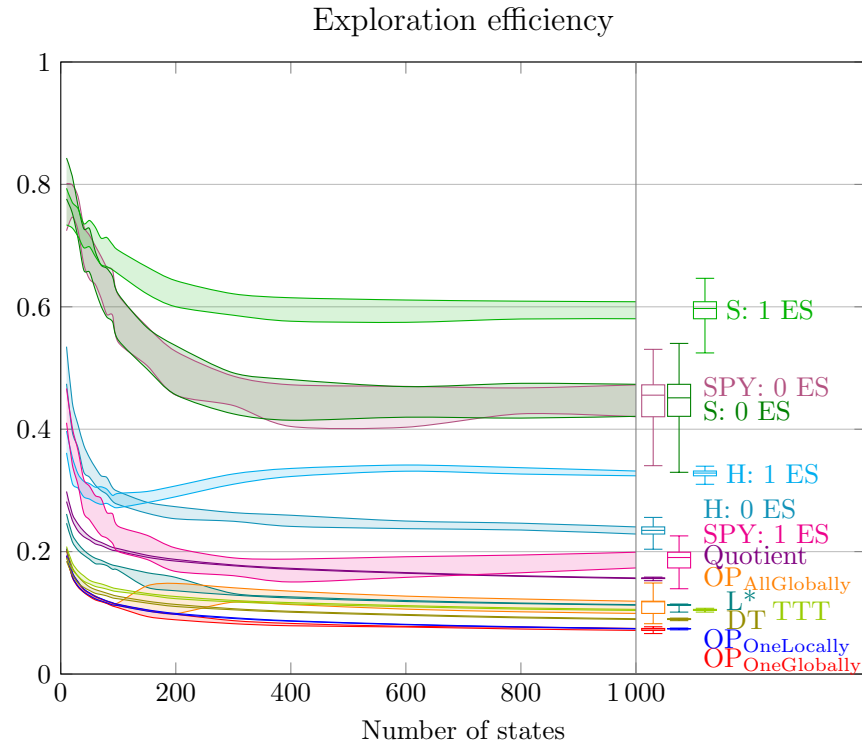


Figure 17.3: DFSMs with 5 inputs: the exploration efficiency (the size of the observation tree divided by the total number of queried symbols) and the learning time, 1 and 3 quartile calculated for 100 machines per each state group

OP algorithm. The AllGlobally version asks the most OQs as it queries the distinguishing suffix from all reached states. Hence, it explores the black box more than the OneGlobally and OneLocally versions and thus it has a higher chance to reveal another state of the black box that is not modelled in the conjecture yet. This is confirmed by the lowest number of equivalence queries amongst the three versions of the OP algorithm.

The interaction with the black box is better captured in Figure 17.2 that shows the total number of input symbols queried during the learning and the number of resets of the black box. There is a significant gap between the performance of the three new learners assuming no extra state and the standard learning algorithms in both measures. Moreover, the H- and S-learners assuming 1 extra state are also better than the standard algorithms (except the TTT and Quotient algorithms in the total number of queried symbols). This is exceptional because they learn using as much interaction with the black box as the best standard learning algorithms but in fact they do not need the teacher and provide the guarantee of 1 extra state.

Figure 17.3 shows the exploration efficiency and the learning time. The exploration efficiency of the three new learners is higher (and thus better) than the standard learning algorithms have and it holds even if the learners assume 1 extra state. The price for the great performance of the new learners is their learning time. It is much higher as the standard algorithms do not analyse the observed traces.

There are other 7 groups of machines for which the results are not visualized here; there are Mealy and Moore machines and deterministic finite automata, each with 5 and 10 inputs, plus DFSMs with 10 inputs. Nevertheless, the observed trends and relative orders of the learners is almost the same for the other 7 combinations of the machine type and the number of inputs. The results can be found in the repository FSMmodels v1.3.

17.2 Models of Real Systems

The second case study consists of three models of real systems. The models were used to evaluate the standard learning algorithms by the research group around the LearnLib as Appendix C.2 describes in detail. The smallest model `peterson2` has 50 states and 18 inputs, the model `sched4` has 97 states and 12 inputs and the largest model `sched5` has 241 states and 15 inputs. All three are deterministic finite automata so that they have only two outputs.

Tables 17.2–17.4 show the results of learning the three models. Similarly to the first case study, the CE processing function `Suffix1by1` of the L* algorithm represents the results of all CE processing functions (except the `AllPrefixes`). It is because the teacher responds to an EQ with the shortest counterexample and the values are almost the same for all CE processing functions. The `GoodSplit` algorithm was able to learn the models with the parameter `maxDistinguishingLength` l set to 2. All learners were allowed to ask equivalence queries. The order of learners in Tables 17.2–17.4 is given by the amount of direct interaction with the black box, that is, the sum of

Learning algorithm	Resets	Symbols	EQs	Seconds	EE [%]
SPY-learner: 0 ES	1 233	9 340	33	2.34	21.0
S-learner: 0 ES	1 235	9 351	38	2.64	20.9
H-learner: 0 ES	1 337	10 057	44	0.32	20.0
TTT	1 811	12 693	49	0.01	10.2
DT	3 773	23 340	49	0.01	5.5
$L^*_{\text{Suffix1by1}}$	16 235	122 906	18	0.06	12.5
SPY-learner: 1 ES	16 534	125 676	1	246.71	13.8
Quotient	17 052	127 251	2	0.07	13.4
H-learner: 1 ES	15 595	131 769	1	4.81	23.9
S-learner: 1 ES	15 455	136 982	1	28.52	12.5
$OP_{\text{OneLocally}}$	18 704	137 296	2	0.05	11.8
$OP_{\text{OneGlobally}}$	19 602	144 835	2	0.05	11.8
$L^*_{\text{AllPrefixes}}$	19 458	147 762	14	0.08	12.5
$OP_{\text{AllGlobally}}$	27 709	244 570	2	0.10	22.3
GoodSplit: $l = 2$	262 752	2 222 603	1	311.57	11.9

Table 17.2: Learning `peterson2`: learners are sorted by the amount of interaction, that is, the number of resets of the black box plus the number of input symbols queried during the learning

the number of resets and the number of input symbols queried during the learning. The number of output queries are not shown as they correspond to the number of resets in the case of the standard learning algorithms; the three new learners ask more output queries than the number of resets they use but this measure does not contribute to the evaluation of real interaction with the black box.

The results on `peterson2` in Table 17.2, on `sched4` in Table 17.3 and on `sched5` in Table 17.4 are very similar in terms of relative comparisons of the learning algorithms. All three models provide a confirmation that the three new learners based on the observation tree approach outperform the standard learning algorithms if they do not consider extra states. Moreover, the assumption of only one extra state is sufficient to learn a correct model without the need of teacher. Note that at least one equivalence query is asked by each learner and it is the last one that confirms that the conjectured model is output-equivalent to the black box. Even with the assumption of 1 extra state, the new learners learn using less interaction with the black box than most standard learning algorithms.

17.3 GridWorld

The third case study is to learn the GridWorld map E that is described in Appendix C.3. It can be modelled by a deterministic finite-state machine with 32 states, 5 inputs, 5 state outputs and 2 transition outputs. The GridWorld is accessed as an external system from the point of view of the

Learning algorithm	Resets	Symbols	EQs	Seconds	EE [%]
SPY-learner: 0 ES	2 007	25 334	68	9.14	14.3
S-learner: 0 ES	2 017	25 438	65	9.09	14.0
H-learner: 0 ES	2 307	28 913	78	0.55	11.0
TTT	3 606	43 757	94	0.03	5.5
DT	11 805	110 183	96	0.05	2.2
S-learner: 1 ES	14 107	178 965	1	181.94	9.8
H-learner: 1 ES	14 254	190 634	1	5.42	14.8
SPY-learner: 1 ES	15 908	203 289	1	350.83	9.0
Quotient	16 741	206 793	4	0.14	8.3
OP _{OneLocally}	18 322	224 021	18	0.10	6.3
L* _{Suffix1by1}	18 655	231 131	15	0.16	7.4
OP _{OneGlobally}	21 736	269 173	4	0.12	6.4
L* _{AllPrefixes}	23 235	283 013	12	0.19	7.6
OP _{AllGlobally}	63 670	1 056 247	4	0.60	18.4
GoodSplit: $l = 2$	149 591	1 944 084	2	42.87	7.8

Table 17.3: Learning `sched4`: learners are sorted by the amount of interaction, that is, the number of resets of the black box plus the number of input symbols queried during the learning

learner implemented in the FSMlib (Appendix A). Therefore, the teacher approximates equivalence queries (EQ) by a testing method, the SPY-method (Section 9.7) in particular. The three new learners are not allowed to ask EQs as they can substitute the approximated EQs by the second phase of the observation tree approach.

Table 17.5 shows the results of learning the GridWorld map E by 6 different learning algorithms. The order of the learners in Table 17.5 is given by the GridWorld simulation steps in the last column. The simulation steps captures precisely the amount of interaction that a learner together with the teacher needs in order to learn the correct model. Note that the number of extra states (ES) assumed by the SPY-method was selected for each standard learning algorithm the lowest possible such that the teacher can still find a counterexample if the conjectured model has not 32 states. Only the TTT, Quotient and L*_{AllPrefixes} are considered from the standard learning algorithms because they can represent the others in the terms of results. The most efficient of the standard learning algorithms is the Quotient algorithm that however needs 4 EQs (implemented by the SPY-method assuming 0 ES). All three new learners can learn the correct model with the assumption of only one extra state. Therefore, when they assume 2 extra states, they also do not need the teacher but they provide a stronger guarantee about the states of the black box. Notice that the learners are comparable with the standard learning algorithms in terms of interaction with the black box even if they assume 2 extra states. It is not mentioned in Table 17.5 but the S-learner assuming 1 ES learns the map E only in 7 894 simulation steps and in the next 23 228 steps the learner verifies the absence of another state. The

Learning algorithm	Resets	Symbols	EQs	Seconds	EE [%]
S-learner: 0 ES	7 726	125 168	67	502.87	10.6
SPY-learner: 0 ES	7 991	129 662	48	368.31	11.0
H-learner: 0 ES	8 578	138 719	164	10.72	8.4
TTT	12 496	201 131	235	0.21	4.4
DT	47 162	555 475	240	0.32	1.6
S-learner: 1 ES	57 725	957 847	1	5 178.73	7.4
H-learner: 1 ES	55 969	964 841	1	121.70	11.6
Quotient	65 941	1 069 076	5	0.88	6.3
OP _{OneLocally}	69 127	1 113 735	50	0.64	4.9
SPY-learner: 1 ES	67 485	1 125 628	1	24 704.79	6.9
L* _{Suffix1by1}	72 339	1 177 436	19	1.54	5.8
L* _{AllPrefixes}	82 099	1 309 732	13	1.62	5.9
OP _{OneGlobally}	83 321	1 356 484	5	0.78	5.0
OP _{AllGlobally}	271 345	5 690 809	5	3.99	10.7
GoodSplit: $l = 2$	725 301	12 120 266	1	42.87	6.1

Table 17.4: Learning `sched5`: learners are sorted by the amount of interaction, that is, the number of resets of the black box plus the number of input symbols queried during the learning

S-learner with 1 ES has one of the highest learning time in the previous two case studies as it analyses the observed traces a lot. In the case of GridWorld where each simulation step takes some time, one can see in Table 17.5 that the learning time of the S-learner assuming 1 ES is one of the lowest.

Learning algorithm	Resets	Symbols	EQs	Seconds	Steps
S-learner: 1 ES	486	9 784	0	620	31 122
H-learner: 1 ES	1 026	10 028	0	829	41 434
Quotient	1 110	7 487	4	615	48 652
+ SPY-method: 0 ES	377	4 835			
SPY-learner: 1 ES	1 801	17 415	0	1 345	74 651
S-learner: 2 ES	2 005	51 300	0	3 443	156 357
H-learner: 2 ES	4 185	44 325	0	3 314	186 274
TTT	1 363	7 870	11	4 145	378 793
+ SPY-method: 2 ES	6 864	131 212			
SPY-learner: 2 ES	9 630	96 493	0	8 134	432 450
$L^*_{\text{AllPrefixes}}$	3 444	28 062	8	4 664	445 285
+ SPY-method: 2 ES	5 641	115 749			

Table 17.5: Learning GridWorld map E: learners are sorted by the number of simulation steps (last column) that corresponds to the amount of interaction, that is, the number of resets of the black box plus the number of symbols queried during the learning by both the learner and the teacher. The teacher approximates equivalence queries by the SPY-method.

Chapter 18

Conclusion

The third part of the thesis is about active learning of deterministic finite-state machines that are completely specified, initially connected and resettable. Chapter 15 proposed the observation tree approach that utilizes the testing theory (discussed in Part II) in the learning. The H-learner (Section 15.3), the SPY-learner (Section 15.4) and the S-learner (Section 15.5) are new active-learning algorithms that implement the observation tree approach. The research question (RQ) III.1 is addressed as the observation tree approach is more general than the framework of Observation Pack (OP). The OP framework does not allow one to choose neither different separating sequences for state identification (used by the H- and S- learners) nor different access sequences (used by the SPY- and S- learners). Therefore, the new learners based on the observation tree approach do not implement the OP framework.

The standard learning algorithms are described in Chapter 16 and explained on a running example. They were extended by the author to work with Definition 1.1 of deterministic finite-state machine and so work for deterministic finite automata or Mealy or Moore machines as well. Semantic suffix closedness (Section 16.1.7) was shown not to be a sufficient property of observation table in order to make a minimal conjectured model.

Section 15.6 proposed the General Learning Framework (GLF) that unifies all active-learning algorithms from the implementation point of view. The implementation of GLF by each learner is described in Section 16.8 that also summarizes the basic differences of the learning algorithms.

The experimental evaluation of all the active-learning algorithms was conducted on three case studies. The results described in Chapter 17 confirmed the improvement in the learning performance by the new learners based on the observation tree approach. If the three new learners follow only the first phase of the observation tree approach and thus do not assume extra states, then they can be directly compared with the standard learning algorithms. The new learners outperformed the standard ones in the interaction with the black box, usually with a significant gap. If the learners based on the observation tree approach assume at least one extra state, they provide a guarantee about the number of states of the black box with respect to the number of states of the conjectured model and the assumed number of extra states. It is thus harder to compare their performance with the standard learning algorithms.

Nevertheless, the results showed that even though the new learners assume one or two extra states, they interact with the black box comparably to the standard algorithms. Moreover, they can learn a correct model without any counterexample provided by the teacher. Hence, RQ III.2 is addressed as well in positive manner. The only drawback seems to be the learning time that is much higher than that of the standard learning algorithms because the new learners analyse observed traces in depth. Nevertheless, the third case study showed that the time spent on the communication with the black box can be much higher than the construction time of output queries to be asked. Therefore, the effort in the analysis of observed traces and querying minimal symbols can result in the lowest learning time.



Conclusions and Future Work

Chapter 19

Conclusions

This thesis deals with three different fields of research that were shown to closely relate to each other. Each research field is explored in a separate part of the thesis. Part I describes state identification sequences and how to construct them. Part II deals with testing of finite-state machines and active learning of finite-state machines is described in Part III. Each part introduces the corresponding research field, the standard methods are described and a contribution is made.

The contribution in each part is a new algorithm. The ST-IADS algorithm constructs a splitting tree for any minimal deterministic finite-state machine and the separating sequences constructed easily from the splitting tree can separate any subset of states with a very few sequences (if not just one). The S-method is a new testing method that employs the ST-IADS algorithm, state domains, and a new sufficient condition for an m -complete test suite (the S-condition). As the S-method combines these novelties, it can become the state-of-the-art testing method. Moreover, it is the first testing method that can extend a given test suite efficiently to become m -complete. The observation tree approach provides a way to utilize the testing theory in active learning. Three new learners, the H-, SPY- and S- learners, were developed based on the observation tree approach. Their improvement to the learning performance was demonstrated in experiments.

All the proposed algorithms were experimentally evaluated on a great suite of machines, see Appendix C for the description of case studies. The results show that the improvements by the contributions are promising but in some cases further evaluation should be conducted as the next chapter discusses. It was shown that the sequences from the splitting tree ST-IADS form harmonized state identifiers (HSI) of less but longer sequences than in the case of HSIs formed of the shortest separating sequences. The S-method constructs the smallest m -complete test suites for randomly generated machines and performs very well in the case of the models of real systems when no extra state is considered. Nevertheless, HSIs formed of the sequences from ST-IADS improve the performance of the SPY-method to the extent that it is the best testing method assuming one or two extra states in the case of the models of real systems. The S-learner employs the S-method when it assumes 1 extra state and so the interaction with the black box is greatly reduced

which was captured on randomly generated machines and the learning of the GridWorld map. The models of real systems as well as randomly generated machines confirmed that all the three new learners significantly outperform the standard learning algorithms if they do not assume any extra state.

The aim of the research was to show that the performance of an active-learning algorithm and a testing method can be improved. It was accomplished as now there is a learner that learns efficiently without teacher and a testing method that constructs small m -complete test suites.



Chapter 20

Future Work

The thesis covers three different research fields and proposes several new ideas in each part. Therefore, some ideas were not possible to implement or to test properly due to time requirements. This last chapter describes what was postponed for future work.

The ST-IADS construction algorithm in Part I could be improved in two ways. Valid inputs could be compared according to their separating ability before an input is chosen to separate a given subset of states. Different scoring functions could be introduced and compared which one leads to better results. The new approach constructing harmonized state identifiers (HSI) and incomplete adaptive distinguishing sequences (IADS) from the splitting tree ST-IADS should be compared against the method proposed in [HT15].

Part II proposes several new ideas to testing of finite-state machines. The new S-condition provides a way to reduce the test suite size, however, it needs more research in order to be used to the full extent by a testing method. Particularly, it is not known when to prove the convergence of two sequences and when it is not necessary; the S-method introduced just one possible way to decide it. The S-method uses only separating sequences constructed from the splitting tree ST-IADS but an improvement in the size of resulting test suite could be achieved if it also considers the sequences that are already in the test suite. The performance of the S-method should be evaluated more on real systems as the SPY-method with HSIs constructed from ST-IADS performed better on the three models of real systems.

The observation tree approach proposed in Part III is very general so that new learners based on it can be easily developed. There are two improvements that the author had in mind. In the first phase of the approach, the construction of adaptive separating sequences could consider the (partially specified) conjectured model besides the observed traces. The empty intersection of convergent nodes' domains as described by Theorem 8.11 could be also employed in order to reduce the amount of interaction with the black box needed to learn a completely specified model. Unfortunately, both possible improvements would lead to even more complex learners. The new learners were evaluated on a lot of machines but further experiments could test the dependence on the number of inputs and/or outputs. The length of provided counterexamples could also be investigated. Very long counterexamples

obtained for example by random walk or system simulation are harder to analyse than the shortest optimal ones. Learning more models of real systems would give the new learners more confidence that they really outperform the standard learning algorithms. Such an experiment could be the inference of SIP and other protocols using the network simulator ns-3¹.

The next step could be to transfer acquired knowledge to testing and learning of machines without reset and extended finite-state machines.

¹<https://www.nsnam.org/>



Appendices

Appendix A

FSMlib

The implementation of all proposed algorithms, the standard active-learning algorithms and the testing methods is included in the *FSMlib*. The *FSMlib* is a C++ library for handling finite-state machines, their testing and learning. It was developed by the author of the thesis and it is available as open source under GNU GPLv3 on GitHub. The version v3.3¹ refers to the implementations used for the comparisons and experiments described in this thesis. There are other tools for handling FSMs, especially their learning. Such tools are *LearnLib* [RSB05, RSBM09] and *libalf* [BKK⁺10]. *LearnLib* is a JAVA framework with GUI for experimenting with the learning process. In contrast, *libalf* is a C++ library with dispatcher for a remote run and a JAVA native interface. After the ZULU competition, *LearnLib* was upgraded to *Next Generation LearnLib* (NGLL) that provides even an interface for the integration of *libalf* [MSHM11]. NGLL was supported by European project CONNECT [GIB⁺12] aiming to provide interoperability interface among various systems. Nevertheless, both tools focus only a particular types of model and do not allow to generalize the algorithms to work with other types of DFSMs easily. Moreover, they do not consider testing or the construction of separating sequences as an important part of the learning. Hence, the *FSMlib* was created.

Besides the algorithms, the *FSMlib* also includes an implementation of the abstract concept of teacher for active-learning algorithms. There are three teachers, *TeacherDFSM*, *TeacherRL* and *TeacherBB*. *TeacherRL* differs from the other two in the type of provided output queries and *TeacherBB* implements equivalence queries differently than the other two teachers. Teachers contain counters of resets, output queries, queried symbols and equivalence queries. The values of counters are then the results of experiments. On request, they can remember all queried sequences and then provide the size of the observation tree which is used for the calculation of exploration efficiency.

TeacherDFSM and *TeacherBB* allow output queries (OQ) in contrast to *TeacherRL* that is restricted to membership queries (MQ) as it focuses on the learning of regular languages. In fact, both *TeacherDFSM* and *TeacherRL* are the same as they hold a model of the black box but they differ in the response to an output query T . *TeacherDFSM* replies with an output sequence

¹<https://github.com/Soucha/FSMlib/releases/tag/v3.3>

observed on the given input sequence and TeacherRL responds only with the last observed output symbol. TeacherBB does not possess a model of the black box and can only interact with it through output queries (OQ) that are also available to the learner.

Answering an equivalence query is easy for TeacherDFSM and TeacherRL as they know the black box but it needs to be approximated in the case of TeacherBB. TeacherBB is initialized with a testing method and the number l of extra states such that the testing method builds an $(n + l)$ -complete test suite for the given conjectured model with n states and the test sequences are then queried. If the observed response to a test sequence differs from the output defined by the conjectured model, the test sequence is returned as a counterexample. The entire test suite is thus queried only if the conjectured model is correct or the black box has more than $n + l$ states. In the case of TeacherDFSM and TeacherRL, the product machine of the conjectured model and the black box is considered to answer an EQ. Its transitions are checked in the breadth-first search (BFS) order from the initial state so the first observed inconsistency reveals the shortest counterexample. Particularly, both models follow an input sequence u and if different outputs are observed, u is returned as a counterexample. Sequences to check are sorted by their length from the shortest, that is, the stOut input symbol is checked first if the state outputs differ. Sequence of the same length are checked in the alphabetical order. If a sequence u reaches s in the conjectured model and q in the black box and there is a shorter sequence u' that lead to the same states, then no sequence starting with the prefix u is checked thereafter. This is equivalent to keeping track of visited states in the product machine to find out when to stop checking. Note that this approach does not require to have any model reduced compared to the case of TeacherBB that minimize the conjectured model first in order to provide it to the testing method.

Appendix B

Generator of Random Finite-State Machines

The FSMlib includes a generator of random deterministic finite-state machines. An initially-connected machine of the given type is generated for the given n, p, q where the type can be DFSM, DFA, Mealy or Moore machine, and n, p, q are the numbers of states, inputs and outputs, respectively. If the generated machine does not meet the given specific property, such as to be minimized, strongly connected or to have an adaptive distinguishing sequence, the generation of a new machine is repeated until the property holds for the machine. If several machines with the same properties are requested, then it is checked that they are different.

For each type of DFSM, all transitions are generated randomly at first, that is, the C function RAND provides an integer that modulo n represents the target state of a transition. States, inputs and outputs are numbered from 0 to $n - 1$, $p - 1$ and $q - 1$, respectively. Some transitions are then changed in order to create an initially connected machine with n states. The change of transitions is done in the following way. Let $R \subseteq S$ be a set of states reachable from the initial state s_0 ; $s_0 \in R$. If there is a state s that is not reachable from s_0 , that is $s \notin R$, a transition connecting two states in R is changed to join s to R . Let e be a transition from a state $s_i \in R$ to a state $s_j \in R$ such that if it is erased from the state diagram, each state in R is still reachable from s_0 . The edge e is changed to lead from s_i to s so that s becomes reachable from s_0 . Changing of the transition function is repeated until $R = S$.

All q outputs should be captured by the output function of the generated machine. Therefore, there are limits for q . A DFA has always at most 2 different outputs, a Moore machine at most n different outputs, a Mealy machine at most $n \cdot p$ different outputs and a DFSM can have up to $n + n \cdot p$ different outputs. If the given q is greater than the limit, then the maximal number is considered instead for q . The output function is first initialized randomly such that the output is obtained as RAND modulo q for each state and/or transition. If an output is not present in the output function, then a state or a transition with the output that occurs more than once in the output function is found and its output is changed to the one with no occurrence.

This way, it is ensured that each output is captured by the output function at least once.

Appendix C

Description of Case Studies

There are three case studies which are used to compare the proposed algorithms with the standard ones. All algorithms are compared on the randomly generated machines first. Then, experiments are conducted on the models of real systems. The third case study compares just the active-learning algorithms as it is about the learning an artificial environment provided by an external tool. The case studies are described in the following three sections. The models of the first two case studies along with the results of all experiments can be found in the GitHub repository FSMmodels v1.3¹.

The first two case studies ran on the High Performance Computing cluster Iceberg of the University of Sheffield with Intel Xeon E5-2650v2 @ 2.6 GHz and 6 TB RAM (used max 128 GB). The third case study ran on a laptop with Intel Core i7-2620M @ 2.7 GHz and 6 GB RAM.

C.1 Randomly Generated Machines

The first case study consists of 13 600 machines that were created by the generator described in Appendix B. There are two groups of 1 700 machines for each of the four types of DFSA; the machine types are DFSA, Mealy and Moore machines, and DFA, and each type is represented by 3 400 machines. One group contains machines with 5 inputs and the machines in the other group have 10 inputs; p is 5 or 10. All machines except DFA have 5 outputs; q is 5 (or 2 for DFA). All machines are also strongly connected and minimal. For each machine type and each p , there are 17 ‘state groups’ of 100 different machines with the same numbers of states. The state groups are referred by the related number of states. There are 10 state groups with n as multiples of 10 in the range from 10 to 100 states. Then, there are 7 state groups with n up to 1000, in particular, these state groups have 150, 200, 300, 400, 600, 800 and 1000 states. Only few machines have an adaptive distinguishing sequence (ADS); they usually have lower number of states (up to 100) and higher number of inputs ($p = 10$).

¹<https://github.com/Soucha/FSMmodels/releases/tag/v1.3>

C.2 Models of Real Systems

The second case study includes three models of real systems that were used to evaluate active-learning algorithms in [BJLS05, How12, IHS14]. The models are examples of Calculus of Communicating Systems (CCS) and their definition can be found in the Edinburgh Concurrency Workbench². `peterson2` is the smallest one and it models the Peterson’s mutual exclusion protocol³. The other two models are schedulers⁴, `sched4` and `sched5` in particular. All three models are deterministic finite automata (DFA) so that they are completely specified and have 2 outputs. The numbers of states and inputs are shown in Table C.1.

Name	States	Inputs
<code>peterson2</code>	50	18
<code>sched4</code>	97	12
<code>sched5</code>	241	15

Table C.1: DFA models of real systems

The DFA models are included in the LearnLib. The model `peterson2` was used for an experimental evaluation of the L* algorithm (Section 16.1) in [BJLS05]. The OP algorithm (Section 16.3) with its 3 counterexample processing functions was tested on all three models in [How12]. The experiments in [IHS14] then employed `peterson2` and `sched4` to evaluate the TTT algorithm (Section 16.4).

C.3 GridWorld in the Brain Simulator

The third case study is to learn an artificial environment such that its model is not available to the teacher. Hence, the teacher is implemented by TeacherBB that interacts with the environment only through output queries. The environment is the map E shown at the top of Figure C.1 and it is a part of the GridWorld scenario of the Brain Simulator⁵ developed by GoodAI.

The partial model at the bottom of Figure C.1 captures a part of the map E shown above it such that the map is formed of the grid of tiles and every tile is modelled by a state. The model is a deterministic finite-state machine so that output symbols are both by states and on transitions. The agent that is directed by the user can move into four directions and thus to explore the map. If the adjacent tile in a particular direction is not accessible, that is, it is outside the map or there is a wall, then the agent stays on the same tile and the output C is received as the response to the input corresponding to the direction. Otherwise, the agent moves in the chosen direction and the

²<http://homepages.inf.ed.ac.uk/perdita/cwb/>

³<http://homepages.inf.ed.ac.uk/perdita/cwb/Examples/ccs/peterson-2.cwb>

⁴<http://homepages.inf.ed.ac.uk/perdita/cwb/Examples/ccs/sched.cwb>

⁵<https://www.goodai.com/brain-simulator>

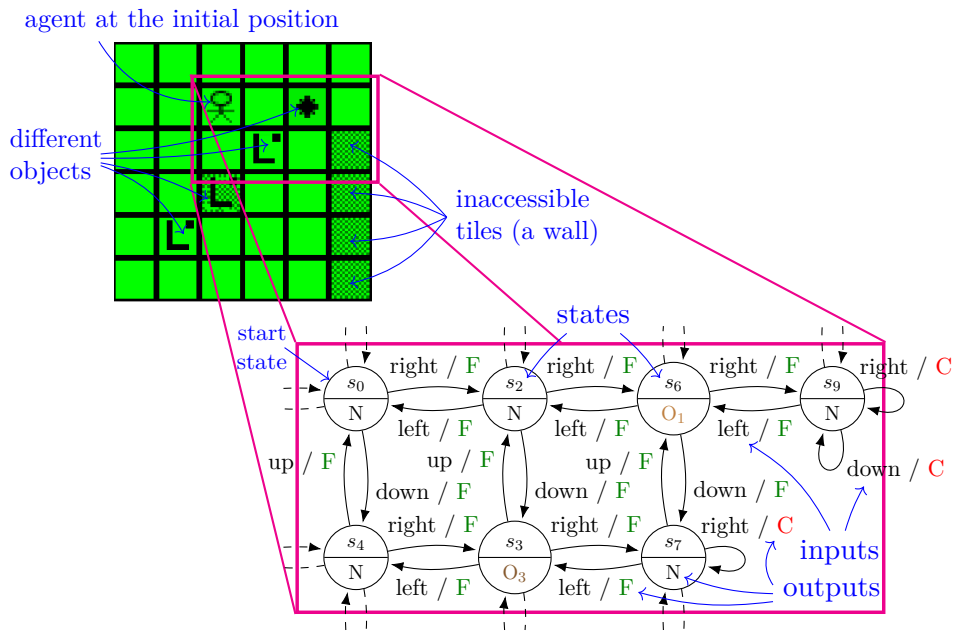


Figure C.1: GridWorld map E and its part modelled by a DFSM

output F is obtained as the response. Each tile can contain an object. The agent asks the input \uparrow in order to detect which object is on the tile where the agent stands; there is usually nothing (the output N). In total, there are 5 inputs that the agent can query (4 directions and 1 to wait on the same tile), 2 transition outputs (C and F) and 5 state outputs (N and 4 different objects).

The learnt model of the GridWorld map E is visualized in Figure C.2 using the FSMvis that is a part of the FSMlib (Appendix A). The DFSM model has 32 states that correspond to each tile of the map E in Figure C.1.

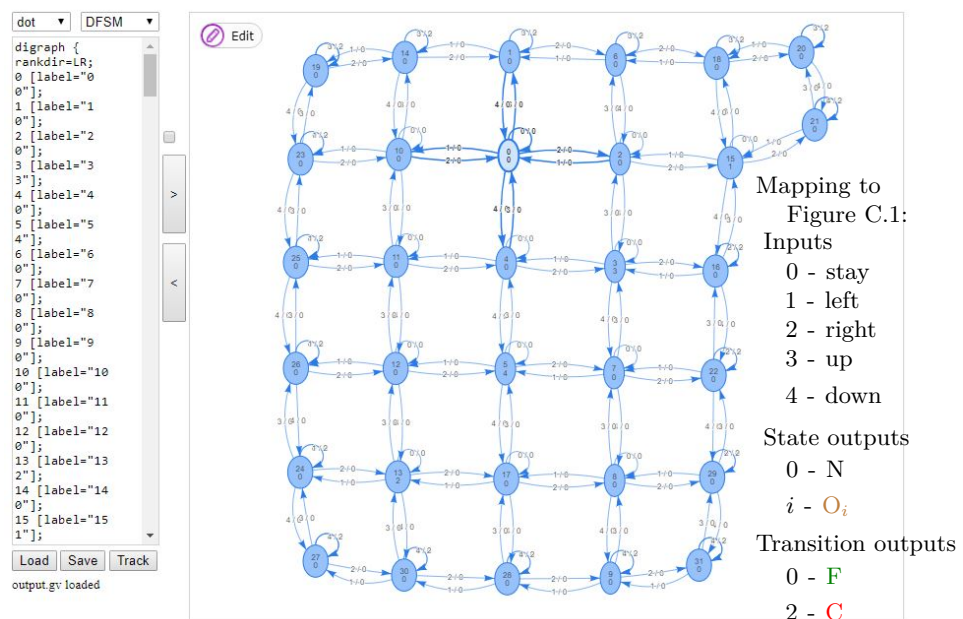


Figure C.2: Learnt model of the GridWorld map E visualized using the FSMvis. States, inputs and outputs are numbered from 0

Appendix D

Flaws in the Related Work

This appendix describes two flaws detected in the related work.

D.1 IADS construction algorithm

Section 3.3.4 sketched the greedy algorithm proposed in [HT15, Algorithm 1]. Unfortunately, the proposed algorithm is not correct and this section points out the inconsistencies. Algorithm 55 describes the greedy algorithm as it was proposed originally except different notation and a typo on line 8 fixed. A node r_i of IADS is defined by the values of functions (I, C, i, o) where I, C are the initial and current sets and i, o are input and output strings labelling the path from the root to r_i ; the root is defined as $(S, S, \varepsilon, \varepsilon)$.

The main issue of the algorithm is that it does not have to stop as a pair of states (s_i, s_j) in Q cannot be separated for two reasons. Either the provided number l is smaller than the length of the shortest separating sequence of s_i and s_j , or any separating sequence of s_i and s_j cannot be created by the algorithm due to the definition of Θ and \mathcal{N} .

The former case depends on the condition on line 8 of Algorithm 55 that stops the extension of the current IADS if its height (of the corresponding tree) reaches the given number l . As the condition is inside the main cycle (line 3) on undistinguished pairs of states, the same IADS is created over and over if a pair of states can be distinguished only by a sequence longer than l .

The latter case depends on unclear conditions on lines 17 and 23. \mathcal{N} is defined as a set of sets of nodes (a set of N_x for all $x \in X$) whereas F is a set of current sets. Hence, the condition $\mathcal{N} \subseteq F$ on line 17 is not formally correct as the sets do not contain elements of the same type. The condition is commented in [HT15] as follows. “If the current sets of all possible children of the current node are in set F , there is no point in investigating this node any more”. The condition thus should be $\forall x \in X \forall r_j \in N_x : C(r_j) \in F$. The condition on line 23 is also misleading. On the one hand, if one follows only Algorithm 55, then the variable *index* remains -1 and so the condition on line 23 holds only if the condition on line 17 holds. As F is initialized to empty set and \mathcal{N} is non-empty for each node r_i , the condition on line 17 is not true unless F is enlarged. Hence, no current set would be added to F following this approach and so this pruning heuristic does not work.

Algorithm 55: Greedy algorithm constructing IADSs

```

input : FSM  $M$ , natural number  $l$ 
output: A set of trees (IADSs)  $\mathbf{T}$ 
1  $Q \leftarrow \{(s_i, s_j) \in S \times S \mid i < j\}$ 
2  $\mathbf{T} \leftarrow \emptyset$  and  $F \leftarrow \emptyset$ 
3 while  $Q \neq \emptyset$  do
4    $\mathcal{N} \leftarrow \emptyset, a \leftarrow 0, h \leftarrow 0$ 
5    $r_i \leftarrow (S, S, \varepsilon, \varepsilon)$  as the root of a new tree  $T_k$ 
6   while  $r_i$  is defined do
7      $max \leftarrow 0, index \leftarrow -1$ 
8     if  $|i(r_i)| < l$  then
9       foreach  $x \in X$  do
10        foreach  $y \in \lambda(C(r_i), x)$  do
11          generate a new node  $r_j$ 
12           $I(r_j) \leftarrow \{s \in I(r_i) \mid \lambda(\delta^*(s, i(r_i)), x) = y\}$ 
13           $i(r_j) \leftarrow i(r_i) \cdot x$ 
14           $o(r_j) \leftarrow o(r_i) \cdot y$ 
15           $C(r_j) \leftarrow \{\delta^*(s, i(r_j)) \mid s \in I(r_j)\}$ 
16          add  $r_j$  to  $N_x$ 
17        if  $\mathcal{N} \subseteq F$  then
18          | add  $C(r_i)$  to  $F$ 
19        else if  $\forall x \in X : \Phi_x(Q, N_x) = 0$  then
20          |  $index \leftarrow \operatorname{argmin}_{x \in X} \Theta_x(F, N_x)$ 
21        else
22          |  $index \leftarrow \operatorname{argmax}_{x \in X} \Phi_x(Q, N_x)$ 
23        if  $index = -1$  then
24          | add  $C(r_i)$  to  $F$ 
25        else
26          foreach  $r_j \in N_{index}$  do
27            if no proper ancestor of  $r_i$  have a current set  $C(r_j)$ 
28              then
29                | add  $r_j$  and the edge  $(r_i, r_j)$  to  $T_k$ 
29           $r_i \leftarrow$  next unprocessed leaf  $r_j$  of  $T_k$ 
30   for all pair of leaf nodes  $r_i, r_j \in T_k, r_i \neq r_j$  do
31     if  $s \in I(r_i), s' \in I(r_j)$  then
32       | pop  $(s, s')$  from  $Q$ 
33       | push  $T_k$  to  $\mathbf{T}$  once
34 return  $\mathbf{T}$ 

```

Moreover, Θ would always return 0 so that it would be useless. On the other hand, if the description of the algorithm in [HT15] is taken into account, the condition on line 23 should represent that the node r_i cannot be refined. According to [HT15], input x refines a node r_i if the states of $C(r_i)$ do not produce the same output to x . It would thus be impossible to construct separating sequences containing a so-called transferring input that transfers states to a different set of states but do not distinguish them. For example, the separating sequences ‘baa’ or ‘cb’ captured in Figure 3.1 could not be created by following the described approach. Neither of approaches lead to a correct algorithm. In addition, the use of Φ and Θ for input selection does not have to be sufficient. The authors stated that if for all inputs Φ is 0 and there are several inputs with minimal value Θ , then the lexicographic order can be used. Nevertheless, it may mean that a separating sequence is not constructed as the needed input is not selected. An example is in Figure 3.1 where ‘a’ is chosen for node r_9 and then the construction stops by the condition on line 28 (cycle check), however, if ‘b’ or ‘c’ were selected, then some states could be separated.

A minor note is that the algorithm contains unused variables a, h and max . The set of possible successors \mathcal{N} is initialized on line 4, however, it should be reset for each node, hence, it should be initialized with empty set on line 7 instead.

D.2 Convergence of Test Sequences

Section 8.1 defined the convergence of test sequences and pointed out that [SPY12, Lemma 3] contains a flaw. This section describes the flaw on an example. At first, the original lemma is defined and the wrong assumption is in bold.

Lemma D.1. ([SPY12, Lemma 3]) *Given a test suite T and an FSM M , let u and v be tests in T which are M -convergent in state s and $N \in F_T$ be an FSM, such that u and v are N -divergent. Let also w be a shortest input sequence, such that M and N are $\{uw, vw\}$ -distinguishable. Then, for each proper prefix w' of w such that $\{u, v\} \cdot \text{pref}(w')$ is F_T -divergence-preserving, the tests in $\{u, v\} \cdot \text{pref}(w')$ reach at least $|w'| + |\delta_M^*(s, \text{pref}(w'))| + 1$ distinct states in N .*

The revised version defined in Lemma 8.7 assumes that w **separates states reached by u and v in N** instead of a weaker assumption of that M and N are $\{uw, vw\}$ -distinguishable.

Figure D.1 sketches a particular example of four test sequences, $uabb$, ubb , $vabb$ and vbb , on two DFSSMs M and N such that u, v are sequences and ‘a’, ‘b’ are input symbols. According to Lemma 8.7, u, v are M -convergent in state s and N -divergent, that is, $\delta_M^*(s_0, u) = \delta_M^*(s_0, v) = s$ and $\delta_N^*(q_0, u) = q \neq q' = \delta_N^*(q_0, v)$. If w is the shortest sequence such that uw or vw distinguishes M and N , then w is ‘bb’ in this example. Consider $w' = \text{‘b’}$ as a proper prefix of w and assume that $\{u, v\} \cdot \text{pref}(w')$ is F_T -divergence-preserving. Then the

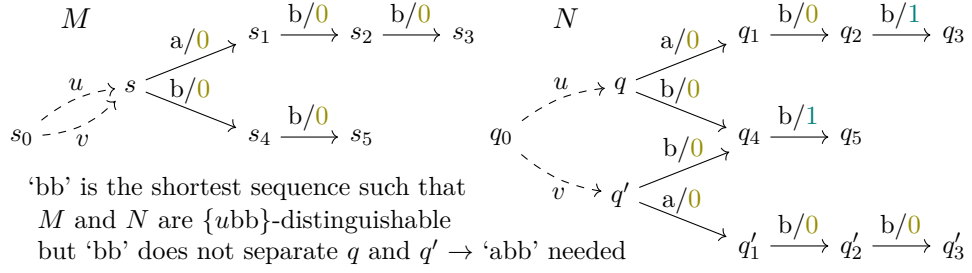


Figure D.1: Convergence of sequences – a flaw in [SPY12, Lemma 3]

tests in $\{u, v\} \cdot \text{pref}(w')$, that is, u , v , ub and vb , should reach at least 4 distinct states in N according to the original [SPY12, Lemma 3]; $|w'| = 1$ and $\delta_M^*(s, \{\varepsilon, b\}) = \{s, s_4\}$ so that the bound is $|w'| + |\delta_M^*(s, \text{pref}(w'))| + 1 = 4$. However, the four tests reach just three distinct states, q , q' and q_4 , in N . Figure D.1 thus captures a counterexample that the original requirement on w is not sufficient. If the sequence ‘abb’ is considered instead of ‘bb’ as it is the shortest one that separates q and q' , then the bound on the number of distinct states holds. Note that if w separates q and q' , then it also distinguishes M and N because there are two different responses to w in N but just one in M . Hence, the original statement holds as well but does not have to be stated explicitly in the revised lemma.

The proofs of [SPY12, Lemma 3] and its revised version in Lemma 8.7 differ only in the *Case 2*, that is, when state s_{k+1} reached by w_{k+1} from s is in the set of states $\delta_M^*(s, \text{pref}(w_k))$. In the original proof of [SPY12, Lemma 3], it was sufficient to show that if uw distinguishes M and N , then uw_{k+1} needs to be N -divergent with all $\{u, v\} \cdot \text{pref}(w_k)$ because of the F_T -divergence-preserving set covering them and the following fact. If uw_{k+1} was N -convergent with a uw_j or vw_j such that $j \leq k$, then w would not be the shortest extension of u that distinguishes M and N because $w_j \cdot w'_{k+1}$ could be used instead. Unfortunately, this reasoning is not sufficient for Lemma 8.7 as there can be a shorter sequence than w that is the extension of u or v and distinguishes M and N but it does not separate states q and q' reached by u, v in N . An example is captured in Figure D.1 where $w'_1 = \text{‘bb’}$ of $w = \text{‘abb’}$ can be used straight from q to distinguish M and N but the response to ‘bb’ is the same as from q' so ‘bb’ does not separate them.



Bibliography

- [AEF15] Dana Angluin, Sarah Eisenstat, and Dana Fisman, *Learning regular languages via alternating automata*, Proceedings of the 24th International Conference on Artificial Intelligence, AAAI Press, 2015, pp. 3308–3314.
- [AKT⁺14] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer, *Improving active mealy machine learning for protocol conformance testing*, Machine learning **96** (2014), no. 1-2, 189–224.
- [AMM⁺18] Bernhard K Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad, *Model learning and model-based testing*, Machine Learning for Dynamic Software Analysis: Potentials and Limits, Springer, 2018, pp. 74–100.
- [Ang81] Dana Angluin, *A note on the number of queries needed to identify regular languages*, Information and control **51** (1981), no. 1, 76–87.
- [Ang86] ———, *Learning regular sets from queries and counterexamples*, Tech. report, Yale University, 1986.
- [AV10] Fides Aarts and Frits Vaandrager, *Learning i/o automata*, CONCUR 2010-Concurrency Theory, Springer, 2010, pp. 71–85.
- [BDGW94a] José L Balcázar, Josep Díaz, Ricard Gavaldà, and Osamu Watanabe, *An optimal parallel algorithm for learning DFA*, Proceedings of the seventh annual conference on Computational learning theory, ACM, 1994, pp. 208–217.
- [BDGW94b] José L Balcázar, Josep Diaz, Ricard Gavaldà, and Osamu Watanabe, *The query complexity of learning DFA*, New Generation Computing **12** (1994), no. 4, 337–358.
- [BDGW96] José L Balcázar, Josep Díaz, Ricard Gavaldà, and Osamu Watanabe, *Algorithms for learning finite automata from queries: A unified view*.

- [BGJ⁺05] Therese Berg, Olga Grinchtein, Bengt Jonsson, Martin Leucker, Harald Raffelt, and Bernhard Steffen, *On the correspondence between conformance testing and regular inference*, Fundamental Approaches to Software Engineering (Maura Cerioli, ed.), Springer, Edinburgh, UK, April 2005, pp. 175–189.
- [BHKL09] Benedikt Bollig, Peter Habermehl, Carsten Kern, and Martin Leucker, *Angluin-style learning of nfa.*, IJCAI, vol. 9, 2009, pp. 1004–1009.
- [BJLS05] Therese Berg, Bengt Jonsson, Martin Leucker, and Mayank Saksena, *Insights to Angluin's learning*, Electronic Notes in Theoretical Computer Science **118** (2005), 3–18.
- [BKK⁺10] Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R Piegdon, *libalf: The automata learning framework*, Computer Aided Verification (Berlin), Springer, July 2010, pp. 360–364.
- [Cho78] Tsun S. Chow, *Testing software design modeled by finite-state machines*, Software Engineering, IEEE Transactions on **4** (1978), no. 3, 178–187.
- [CVO89] Wendy YL Chan, CT Vuong, and MR Otp, *An improved protocol test generation procedure based on uios*, ACM SIGCOMM Computer Communication Review **19** (1989), no. 4, 283–294.
- [DEFM⁺05] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R Cavalli, and Nina Yevtushenko, *Experimental evaluation of FSM-based testing methods*, Software Engineering and Formal Methods, 2005. SEFM 2005. Third IEEE International Conference on, IEEE, 2005, pp. 23–32.
- [DEFY05] Rita Dorofeeva, Khaled El-Fakih, and Nina Yevtushenko, *An improved conformance testing method*, Formal Techniques for Networked and Distributed Systems-FORTE 2005 (Farn Wang, ed.), Springer, Taipei, Taiwan, October 2005, pp. 204–218.
- [DLDL08] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde, *The QSM algorithm and its application to software behavior model induction*, Applied Artificial Intelligence **22** (2008), no. 1-2, 77–115.
- [DIH05] Colin De la Higuera, *A bibliographical study of grammatical inference*, Pattern recognition **38** (2005), no. 9, 1332–1348.
- [DIH10a] ———, *Grammatical inference*, vol. 96, Cambridge University Press Cambridge, 2010.
- [DIH10b] ———, *Learning finite state machines*, Finite-State Methods and Natural Language Processing, Springer, 2010, pp. 1–10.

- [Dup96] Pierre Dupont, *Incremental regular inference*, International Colloquium on Grammatical Inference, Springer, 1996, pp. 222–237.
- [EA10] Sarah Eisenstat and Dana Angluin, *Learning random DFAs with membership queries: the GoodSplit algorithm*, ZULU workshop organised during ICGI (Valencia, Spain), Springer, September 2010, p. 12.
- [EGPQ06] Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu, *Grey-box checking*, International Conference on Formal Techniques for Networked and Distributed Systems (Berlin), Springer, September 2006, pp. 420–435.
- [ES13] Andre Takeshi Endo and Adenilso Simao, *Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods*, Information and Software Technology **55** (2013), no. 6, 1045–1062.
- [FKA⁺91] Susumu Fujiwara, F Khendek, M Amalou, A Ghedamsi, et al., *Test selection based on finite state models*, Software Engineering, IEEE Transactions on **17** (1991), no. 6, 591–603.
- [GIB⁺12] Emmanuelle Grousset, Valérie Issarny, Amel Bennaceur, Antonia Bertolino, Daniela Mulas, Illaria Matteucci, Paul Grace, Gordon Blair, Youssouf Mhoma, Paola Inverardi, et al., *Project final report final publishable summary report*.
- [Gil62] Arthur Gill, *Introduction to the theory of finite-state machines*, McGraw-Hill Book Company, 1962.
- [Gol72] E Mark Gold, *System identification via state characterization*, Automatica **8** (1972), no. 5, 621–636.
- [GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis, *Adaptive model checking*, Tools and Algorithms for the Construction and Analysis of Systems, Springer, Berlin, April 2002, pp. 357–370.
- [HJUY09] Robert M Hierons, G-V Jourdan, Hasan Ural, and Husnu Yenigun, *Checking sequence construction using adaptive and pre-set distinguishing sequences*, Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on, IEEE, 2009, pp. 157–166.
- [HMvdP18] David Huistra, Jeroen Meijer, and Jaco van de Pol, *Adaptive learning for learn-based regression testing*, International Workshop on Formal Methods for Industrial Critical Systems (Cham), Springer, September 2018.
- [Hop71] John Hopcroft, *An $n \log n$ algorithm for minimizing states in a finite automaton*.

- [How12] Falk M Howar, *Active learning of interface programs*, Ph.D. thesis, Technical University of Dortmund, Dortmund, Germany, 2012.
- [HT15] Robert M Hierons and Uraz Cengiz Türker, *Incomplete distinguishing sequences for finite state machines*, *The Computer Journal* **58** (2015), no. 11, 3089–3113.
- [HT16] ———, *Parallel algorithms for generating harmonised state identifiers and characterising sets*, *IEEE Transactions on Computers* **65** (2016), no. 11, 3370–3383.
- [IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen, *The TTT algorithm: a redundancy-free approach to active automata learning*, *Runtime Verification (Toronto, Canada)*, Springer, September 2014, pp. 307–322.
- [IOG10] Muhammad Naeem Irfan, Catherine Oriat, and Roland Groz, *Angluin style finite state machine inference with non-optimal counterexamples*, *Proceedings of the First International Workshop on Model Inference In Testing*, ACM, 2010, pp. 11–19.
- [Ipa12] Florentin Ipate, *Learning finite cover automata from queries*, *Journal of Computer and System Sciences* **78** (2012), no. 1, 221–244.
- [Irf10] Muhammad Naeem Irfan, *State machine inference in testing context with long counterexamples*, *Software Testing, Verification and Validation (ICST)*, 2010 Third International Conference on (Paris, France), IEEE Computer Society, April 2010, pp. 508–511.
- [IS14] Malte Isberner and Bernhard Steffen, *An abstract framework for counterexample analysis in active automata learning*, *International Conference on Grammatical Inference (Kyoto, Japan)*, vol. 34, PMLR, September 2014, pp. 79–93.
- [Kar72] Richard M Karp, *Reducibility among combinatorial problems*, *Complexity of computer computations*, Springer, 1972, pp. 85–103.
- [KK12] Monika Kapus-Kolar, *New state-recognition patterns for conformance testing of finite state machine implementations*, *Computer Standards & Interfaces* **34** (2012), no. 4, 390–395.
- [KK15] ———, *Incremental construction of complete test suites for finite state machine implementations - principles behind the current methods and beyond*, Tech. Report 11855, Jožef Stefan Institute, Department of Communication Systems, Ljubljana, Slovenia, 2015.

- [KV94] Michael Kearns and Umesh Virkumar Vazirani, *An introduction to computational learning theory*, The MIT Press, Cambridge, MA, 1994.
- [LAD⁺11] Shang-Wei Lin, Étienne André, Jin Song Dong, Jun Sun, and Yang Liu, *An efficient algorithm for learning event-recording automata*, Automated Technology for Verification and Analysis, Springer, 2011, pp. 463–472.
- [LPB95] Gang Luo, Alexandre Petrenko, and Gregor v Bochmann, *Selecting test sequences for partially-specified nondeterministic finite state machines*, Protocol Test Systems, Springer, 1995, pp. 95–110.
- [LY94] David Lee and Mihalis Yannakakis, *Testing finite-state machines: State identification and verification*, Computers, IEEE Transactions on **43** (1994), no. 3, 306–320.
- [Mei04] Karl Meinke, *Automated black-box testing of functional correctness using function approximation*, ACM SIGSOFT Software Engineering Notes **29** (2004), no. 4, 143–153.
- [Mei10] Karl Meinke, *Cge: A sequential learning algorithm for mealy automata*, International Colloquium on Grammatical Inference (Berlin), Sp, September 2010.
- [Moo56] Edward F Moore, *Gedanken-experiments on sequential machines*, Automata studies **34** (1956), 129–153.
- [MS10] Karl Meinke and Muddassar Sindhu, *Correctness and performance of an incremental learning algorithm for finite automata*, KTH Royal Institute of Technology, 2010.
- [MS11] Karl Meinke and Muddassar A. Sindhu, *Incremental learning-based testing for reactive systems*, International Conference on Tests and Proofs (Berlin), Springer, June 2011.
- [MSHM11] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria, *Next generation learnlib*, Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2011, pp. 220–223.
- [Nie03] Oliver Niese, *An integrated approach to testing complex systems*, Ph.D. thesis, Technical University of Dortmund, Dortmund, Germany, 2003.
- [OG92] José Oncina and Pedro Garcia, *Inferring regular languages in polynomial updated time*, Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium, World Scientific, 1992, pp. 49–61.

- [PBY96] Alexandre Petrenko, Gv Bochmann, and M Yao, *On fault coverage of tests for finite state specifications*, Computer Networks and ISDN Systems **29** (1996), no. 1, 81–106.
- [Pet91] Alexandre Petrenko, *Checking experiments with protocol machines*, Proceedings of the IFIP TC6/WG6. 1 Fourth International Workshop on Protocol Test Systems IV (Amsterdam, The Netherlands), North-Holland Publishing Co., October 1991, pp. 83–94.
- [PLG⁺14] Alexandre Petrenko, Keqin Li, Roland Groz, Karim Hossen, and Catherine Oriat, *Inferring approximated models for systems engineering*, High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on (Miami Beach, FL), IEEE, January 2014, pp. 249–253.
- [Pra11] Gend Lal Prajapati, *Advances in learning formal languages*, Proceedings of the International MultiConference of Engineers and Computer Scientists, vol. 1, 2011.
- [PVY99] Doron Peled, Moshe Y Vardi, and Mihalis Yannakakis, *Black box checking*, Formal Methods for Protocol Engineering and Distributed Systems, Springer, Boston, MA, 1999, pp. 225–240.
- [PY92] Alexandre Petrenko and Nina Yevtushenko, *Test suite generation from a fsm with a given type of implementation errors*, Proceedings of the IFIP TC6/WG6. 1 Twelfth International Symposium on Protocol Specification, Testing and Verification XII, North-Holland Publishing Co., 1992, pp. 229–243.
- [PY05] ———, *Testing from partial deterministic FSM specifications*, Computers, IEEE Transactions on **54** (2005), no. 9, 1154–1165.
- [RS93] Ronald L Rivest and Robert E Schapire, *Inference of finite automata using homing sequences*, Information and Computation **103** (1993), no. 2, 299–347.
- [RSB05] Harald Raffelt, Bernhard Steffen, and Therese Berg, *Learnlib: A library for automata learning and experimentation*, Proceedings of the 10th international workshop on Formal methods for industrial critical systems, ACM, 2005, pp. 62–71.
- [RSBM09] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria, *Learnlib: a framework for extrapolating behavioral models*, International journal on software tools for technology transfer **11** (2009), no. 5, 393–407.
- [SB18] Michal Soucha and Kirill Bogdanov, *SPYH-method: An improvement in testing of finite-state machines*, 2018 IEEE International Conference on Software Testing, Verification and

- Validation Workshops (ICSTW) (Vasteras, Sweden), IEEE, April 2018, pp. 194–203.
- [SB19] ———, *Observation tree approach: Active learning relying on testing*, The Computer Journal (2019).
- [Sha08] Muzammil Shahbaz, *Reverse engineering enhanced state models of black box components to support integration testing*, Ph.D. thesis, PhD thesis, Grenoble Institute of Technology, 2008.
- [SHM11] Bernhard Steffen, Falk Howar, and Maik Merten, *Introduction to active automata learning from a practical perspective*, Formal Methods for Eternal Networked Software Systems, Springer, 2011, pp. 256–296.
- [SL89] Deepinder Sidhu and T-K Leung, *Formal methods for protocol testing: A detailed study*, Software Engineering, IEEE Transactions on **15** (1989), no. 4, 413–426.
- [SMJ16] Rick Smetsers, Joshua Moerman, and David N Jansen, *Minimal separating sequences for all pairs of states*, International Conference on Language and Automata Theory and Applications, Springer, 2016, pp. 181–193.
- [Sou14] Michal Soucha, *Finite-state machine state identification sequences*, Bachelor’s thesis, 2014.
- [Sou15] ———, *Checking experiment design methods*, Master’s thesis, Czech Technical University in Prague, 2015.
- [SP09a] Adenilso Simão and Alexandre Petrenko, *Checking sequence generation using state distinguishing subsequences*, Software Testing, Verification and Validation Workshops, 2009. ICSTW’09. International Conference on, IEEE, 2009, pp. 48–56.
- [SP09b] ———, *Fault coverage-driven incremental test generation*, The Computer Journal (2009), bxp073.
- [SPY12] Adenilso Simão, Alexandre Petrenko, and Nina Yevtushenko, *On reducing test length for FSMs with extra states*, Software Testing, Verification and Reliability **22** (2012), no. 6, 435–454.
- [Ura92] Hasan Ural, *Formal methods for test sequence generation*, Computer communications **15** (1992), no. 5, 311–325.
- [Vas73] MP Vasilevskii, *Failure diagnosis of automata*, Cybernetics and Systems Analysis **9** (1973), no. 4, 653–665.
- [vHSS17] Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva, *CALF: Categorical automata learning framework*, arXiv preprint arXiv:1704.05676 (2017).

- [WNS⁺13] Stephan Windmüller, Johannes Neubauer, Bernhard Steffen, Falk Howar, and Oliver Bauer, *Active continuous quality control*, Proceedings of the 16th International ACM Sigsoft symposium on Component-based software engineering, ACM, 2013, pp. 111–120.

- [Zhu96] Hong Zhu, *A formal interpretation of software testing as inductive inference*, Software Testing, Verification and Reliability **6** (1996), no. 1, 3–31.