

University of Sheffield

Mealy Machine Induction: A Pattern-Driven Reverse Engineering Approach



Arwa Alfitni

Supervisor 1: Dr Kirill E Bogdanov

Supervisor 2: Dr Neil Walkinshaw

A report submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computer Science

in the

Department of Computer Science

September 2, 2026

Declaration

All sentences or passages quoted in this document from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure.

Name: Arwa Alfitni

Date: September 2, 2026

Abstract

Passive model inference automatically reverse-engineers behavioural models from software execution traces, providing a leading mechanism for system verification and documentation. While the Evidence-Driven State-Merger (EDSM) heuristic is highly effective given dense data, it becomes severely error-prone in data-sparse environments. Hybrid frameworks like DFASAT mitigate this by coupling preliminary merges with exact Boolean satisfiability (SAT) solvers. However, DFASAT relies on a rigid local blocking heuristic that frequently bypasses valid merge candidates, causing an explosion in the SAT search space and compounding execution latencies.

To overcome these limitations, this thesis introduces a pattern-based passive model inference paradigm. The proposed framework mines lightweight sequential pairwise patterns from Mealy machine traces, utilising these patterns as global constraints to block incorrect merges. This approach was integrated into multiple algorithmic configurations, including PAT-EDSM and PAT-DFASAT. The approaches are evaluated across industrial protocol implementations (Bluetooth, OpenSSL, TCP, ABP Sender) and random automata.

The empirical results demonstrate that PAT-DFASAT represents the most robust framework under extreme data sparsity, delivering superior structure and language acceptance ($F1$) scores. Furthermore, a correlation was identified between execution runtime variance and dataset scale. Intermediate data sizes exhibit pronounced variance due to structural ambiguity, whereas early merge errors severely hinder downstream SAT convergence. Conversely, when the target model exhibits behavioural similarity across its states, PAT-EDSM outperforms PAT-DFASAT. This work successfully balances error-prone heuristics against rigid local constraints, advancing the scalability of passive automata induction.

Acknowledgements

All praise and gratitude are due to Allah, God Almighty, for His infinite blessings, for sustaining my family and me, and for granting me the strength and perseverance to complete this journey.

I owe my heartfelt thanks to my mother, Zareenah, for her unwavering love, prayers, and support throughout my studies. I am deeply grateful to my children, Retal, Wateen, and Ghassan, who have been my constant source of joy and motivation. I dedicate this achievement to the soul of my dear father, who was always proud of my academic pursuits and eagerly anticipated this milestone. My profound gratitude also goes to my brothers and sisters (Raid, Areej, Sami, Abdulelah, and Sara) who stood firmly by me every step of the way.

I would like to express my sincere gratitude to my supervisors, Dr Kirill E Bogdanov and Dr Neil Walkinshaw, for their invaluable guidance, immense support, and insightful advice throughout my PhD journey. I have been privileged to learn from them, both academically and personally.

I am equally thankful to the past and present researchers and academics within the AI $\times$ Software Engineering and Testing (ASET) Lab at the University of Sheffield for sharing their knowledge and fostering a collaborative, welcoming research environment. I also extend my warmest thanks to my friends outside the university (Anood, Maha, and Haneen) who generously offered their companionship and essential moral support. Special thanks are also due to my life coach, Bayan, for her boundless support and for providing a safe space to navigate life's difficulties and overcome the most challenging days and situations.

Finally, I gratefully acknowledge the generous financial sponsorship and support provided by the Ministry of Education of the Kingdom of Saudi Arabia and Umm Al-Qura University, which made this doctoral research possible.

Contents

Contents	iv
List of Figures	vii
List of Tables	xii
1 Introduction	1
1.1 Introduction	1
1.2 Research Objectives	3
1.3 Thesis Contributions	4
1.4 Research Outputs	4
1.5 Thesis Overview	4
2 Background	6
2.1 Software Models and their Inference	6
2.1.1 Software as Behaviour Models:	7
2.1.2 Model Inference	10
2.1.3 Challenges in Passive Inference	12
2.2 Passive Model Inference	13
2.2.1 Prefix Tree Acceptor (PTA)	15
2.2.2 State Merging	16
2.2.3 Evidence-Driven State Merging	17
2.2.4 SAT Solver in Passive Inference	24
2.2.5 Patterns in Passive Inference	33

3	Mined Patterns within EDSM	36
3.1	Motivation	36
3.2	Methodological Overview	38
3.3	Pattern Mining	39
3.4	PAT-EDSM	41
3.5	PAT-DFASAT	47
3.6	Integrating Mined Patterns into the SAT Solver	52
3.7	Integrating Mined Patterns into the DFASAT Framework (Pattern-guided DFASAT)	53
3.8	Integrating Mined Patterns into the SAT solver of DFASAT (Pattern-guided DFASAT+PAT-SAT)	54
3.9	Implementation	54
3.10	Summary	55
4	Experimental Setup	56
4.1	Use Cases	56
4.2	Training Data	57
4.3	Evaluation Methodology	61
4.3.1	Language Evaluation	63
4.3.2	Structural Evaluation	66
4.3.3	Pairwise Evaluation	69
4.3.4	Proportion of Correct Merges	72
4.3.5	Execution Time	72
4.4	Summary	73
5	Results and Discussion	74
5.1	Results	74
5.1.1	Real-world Use Cases	77
5.1.2	Randomly Generated Use Cases	99
5.1.3	Results Summary	122
5.1.4	Discussion	124

5.2	Threats to Validity	126
5.2.1	External Threats to Validity	126
5.2.2	Internal Threats to Validity	127
6	Related work	128
6.1	The K-tail Algorithm	128
6.2	The L-sharp algorithm:	130
6.3	Improvements to EDSM	131
6.3.1	Biased EDSM score using Markov Model	131
6.3.2	Shared Evidence-Driven State Merging	132
6.3.3	The ED-BEAM Algorithm	133
6.3.4	Genetic Algorithm and EDSM	133
6.4	Pattern-based Automata Learning	134
6.4.1	Markov Model in automata learning	136
7	Conclusions and Future Work	138
7.1	Conclusion	138
7.2	Future Work	140
	Bibliography	142
	Appendices	146
A	Generating Random Automata	147
B	APTA to CNF Formula	151
C	Results	155

List of Figures

2.1	Text Editor Automaton	8
2.2	Example of a non-deterministic Mealy machine	9
2.3	The difference between the Passive and the Active learning approaches	11
2.4	Prefix Tree Acceptor(PTA)	15
2.5	Target Model	15
2.6	Merge States 1 and 2 raise a nondeterminism	16
2.7	Resolve the nondeterminism by merging states 7 and 12	16
2.8	Merge State 4 and 5 raise a nondeterminism	17
2.9	Conflict cannot be resolved	17
2.10	EDSM Example: PTA - States with the same border line correspond to the same state in the target model (same PTA as in Figure 2.4).	20
2.11	EDSM Example: Merge states 1 and 2 - Tables are numbered according to their implementation order. Blank cells in the tables indicate that the blue state was not evaluated against the red states because an earlier blue state was promoted to red.	20
2.12	EDSM Example: More Comparisons	21
2.13	EDSM Example: more merges and the final model	22
2.14	Transform a PTA into a consistency graph.	25
2.15	PTA (same PAT as in Figure 2.4)	28
2.16	Finding the best pair	29
2.17	Merge states 3 and 10	29
2.18	Final Model inferred by DFASAT	30
2.19	The inferred model by standard EDSM - copy of Figure 2.13b	30
3.1	Methodology overview	38

3.2	Text Editor Example (same as Figure 2.1)	40
3.3	A PTA for the Text Editor System - (same PTA in Figure 2.4)	40
3.4	The result configuration of merging states 1 and 2 - (same as Figure 2.7)	42
3.5	PTA of the Text Editor (same PTA as in Figure 2.4) - States with the same border line correspond to the same state in the target model. Tables are numbered according to their implementation order. Blank cells in the tables indicate that the blue state was not evaluated against the red states because an earlier blue state was promoted to red.	44
3.6	Performing the first Merge between state 2 and state 3	44
3.7	Performing the second merge between state 1 and state4	45
3.8	The final model inferred by PAT-EDSM	45
3.9	The Final Model inferred by EDSM - copy of Figure 2.13b	46
3.10	Text Editor - target model (same as Figure 2.1)	47
3.11	Text Editor Target Model (same as Figure 2.5)	47
3.12	Example of adding the reject state to the inferred model	49
4.1	The Target Model $\mathcal{M}$ (same as Figure 2.1)	61
4.2	Examples of three different learned models	62
4.3	The target and the inferred models	67
4.4	Visualising the Structural Difference between $\mathcal{M}$ and $\mathcal{M}_3$ shown in Figure 4.3	68
4.5	A PTA and its target model - (copy of Figure 2.5)	72
5.1	Bluetooth use case - Reference Automata (the initial state is s0)	77
5.2	Results for the Bluetooth use case with different data sizes	78
5.3	Bluetooth - Runtime results: comparing the learners' performance	79
5.4	Bluetooth - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	80
5.5	OpenSSL use case - Reference Automata (the initial state is s0)	81
5.6	Results for the OpenSSL use case with different data sizes	82
5.7	OpenSSL - Runtime results: comparing the learners' performance	83
5.8	OpenSSL - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	84
5.9	Coffee Machine use case - Reference Automata (the initial state is s0)	85

5.10 Results for the Coffee Machine use case with different data sizes	86
5.11 Coffee Machine - Runtime results: comparing the learners' performance	87
5.12 Coffee Machine - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	88
5.14 Comparing two inferred models using PAT-EDSM and PAT-DFASAT for the Coffee Machine use case at size (Data $\times$ 32)	89
5.15 TCP use case - Reference Automata (the initial state is s0)	90
5.16 Results for the TCP use case with different data sizes	91
5.17 TCP - Runtime results: comparing the learners' performance	92
5.18 TCP - A12 results: comparing the learners' performance	93
5.19 ABP Sender use case - Reference Automata (the initial state is (5))	94
5.20 Results for the ABP Sender use case with different data sizes	95
5.21 ABP Sender - Runtime results: comparing the learners' performance	96
5.22 ABP Sender - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	97
5.23 Random1 use case - Reference Automata (the initial state is S1)	98
5.24 Results for the Random1 use case with different data sizes	99
5.25 Random1 - Runtime results	100
5.26 Random1 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	101
5.27 Random2 use case - Reference Automata (the initial state is V0)	102
5.28 Results for the Random2 use case with different data sizes	103
5.29 Random2 - Runtime results	104
5.30 Random2 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	105
5.31 Inferred model by PAT-DFASAT for the Random2 use case	106
5.32 Random3 use case - Reference Automata (the initial state is V0)	107
5.33 Results for the Random3 use case with different data sizes	108
5.34 Random3 - Runtime results	109
5.35 Random3 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	110
5.36 Random4 use case - Reference Automata (the initial state is V0)	111

5.37	Results for the Random4 use case with different data sizes	112
5.38	Random4 - Runtime results	113
5.39	Random4 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	114
5.40	Random5 use case - Reference Automata (the initial state is V0)	115
5.41	Results for the Random5 use case with different data sizes	115
5.42	Random5 - Runtime results	116
5.43	Random5 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	117
5.44	Random6 use case - Reference Automata (the initial state is V0)	118
5.45	Results for the Random6 use case with different data sizes	119
5.46	Random6 - Runtime results	120
5.47	Random6 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)	121
6.1	Merging states using k-tail strategy	130
6.2	Resource ownership pattern	136
A.1	Adding a new node with the Forest-Fire algorithm - dashed lines represent new edges and the solid line represents the addition of a new state	147
A.2	Example of Generating a 4-state random automaton using the forest fire algorithm .	148
A.4	Generating a 4-state random automaton using the modified forest-fire algorithm . .	149
B.1	PTA	151
C.1	Coffee Machine: PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints) - Data size: $ Q $ between 130 and 1300	156
C.2	OpenSSL - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints) - Data size: $ Q $ between 1200 and 4100	156
C.3	Random1 - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints) - $ Q $ between 150 and 1500	157
C.4	TCP - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints) - $ Q $ between 2500 and 9900	157
C.5	Bluetooth - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)	158
C.6	Coffeemachine - BiasedSiccoSAT vs BiasedSiccoSATPAT - Data size: $ Q $ between 130 and 1300	158

C.7	OpenSSL - BiasedSiccoSAT vs BiasedSiccoSATPAT Data size: $ Q $ between 1200 and 4100	159
C.8	Random1 - BiasedSiccoSAT vs BiasedSiccoSATPAT - $ Q $ between 150 and 1500 . . .	160
C.9	TCP - BiasedSiccoSAT vs BiasedSiccoSATPAT - $ Q $ between 2500 and 9900	160
C.10	Coffee Machine - Pattern-guided DFASAT vs. Standard DFASAT - Data size: $ Q $ between 130 and 1300	161
C.11	OpenSSL - Pattern-guided DFASAT vs. Standard DFASAT - Data size: $ Q $ between 1200 and 4100	161
C.12	Random1 - Pattern-guided DFASAT vs. Standard DFASAT - $ Q $ between 150 and 1500	162
C.13	TCP - Pattern-guided DFASAT vs. Standard DFASAT - $ Q $ between 2500 and 9900	162

List of Tables

2.1	Encoding Mealy machines identification into SAT	27
3.1	Configurations for the proposed learners	39
3.2	Mined Patterns for the Text Editor Example	41
3.3	Encoding Mealy machine identification into SAT.	48
4.1	Use cases configuration used for the evaluation	57
4.2	Summary of the experimental setup	61
4.3	The Confusion Matrix	66
4.4	Interpretation of the A_{12} value	70
5.1	The Initial Training Data for all use cases	75
5.2	The Maximum size for Training Data for all use cases	75
5.3	Detailed configuration for all use cases - copy of Table 4.1	76

Chapter 1

Introduction

1.1 Introduction

Software has become a foundational component of modern society. Over the past decade, the scope and complexity of software engineering have expanded exponentially to include banking systems, digital government services, mobile applications, industrial machines, and much more. Within this widespread landscape, deploying low-quality software that fails to meet end-user requirements can result in severe financial losses, occasionally amounting to billions of dollars globally. Software testing and formal verification mitigate these operational risks. However, their efficacy is significantly dependent on the availability of correct and precise system specifications. In practice, these specifications are defined as models.

Software models provide a standardised graphical language and notation for formally describing the internal architecture and operational dynamics of a system. These models can characterise a system either at the structural source-code level or at the behavioural specification level. At the code level, architectural components represent distinct classes or functions and their relationships. Conversely, behavioural models describe the system specification, where components represent discrete operations, inputs, outputs, and the sequential flow of events.

While many paradigms exist to represent behavioural models, such as state machines, dataflow networks, declarative temporal logics, and stochastic agent models, they differ fundamentally in their underlying abstractions and analytical focus. Non-automata (such as dataflow) approaches excel at tracking continuous physical processes or high-level distributed interaction traces; however, automata (or state machine) formalisms remain the standard for specifying control logic, execution protocols, and deterministic state transitions. Understanding the distinct boundaries of these various paradigms ensures precise architectural specification and formal verification. This thesis focuses exclusively on behavioural models in the form of state machines.

State machines are an effective tool for system comprehension, offering a visual map that simplifies analysis of legacy software or undocumented network protocols. Beyond basic inspection, state machines enable several critical verification and validation tasks throughout the software engineering life cycle. During system design, they provide an explicit visual structure that establishes a transparent communication channel between developers and stakeholders. During the testing phase, these models enable automated test suite generation via model-based testing [32, 56, 40]. Furthermore, formal models facilitate model checking [14], allowing developers to evaluate a system against its temporal logic properties, spot bugs early in development, and support refactoring embedded controllers. [40, 39]. When these specifications are complete and synchronised with the actual implementation, they serve as a source of truth, guaranteeing software correctness and

structural integrity.

Ideally, models are constructed by designers and developers during the initial development phase and are continuously updated as the system evolves. In practice, however, manual model construction is highly time-consuming. Developers frequently neglect to update initial design models or avoid creating them entirely, relying instead on informal, abstract sketches. This omission results in a lack of detailed, formal specifications essential for software testing and maintainability.

To address this challenge, various model inference techniques have been developed to automate specification generation [27, 31, 52, 3, 44, 26, 16]. Model inference (also known as machine induction or automata learning) attempts to infer an automaton that accurately encapsulates the behaviour of the target system, bridging the gap between the documentation and the existing implementation to support subsequent testing or formal verification tasks.

Software model inference is conventionally divided into two primary paradigms: passive and active learning. Active learning frameworks [4] facilitate a direct interaction between the inference engine (i.e., the learning algorithm) and the actual system through an oracle interface. An active learner proactively explores the system’s state space by generating and executing queries. These queries fall into two distinct categories: equivalence queries, which either confirm the validity of a hypothesised model or return a counterexample, and membership queries, which verify whether specific sequences of events are accepted by the system. This interactive query mechanism allows the algorithm to dynamically collect missing operational data, providing an effective way to resolve structural ambiguities during model construction.

In contrast, the passive learning paradigm operates solely on a static dataset reflecting the system’s runtime behaviour, such as execution traces. By processing these static datasets, the passive learner induces a candidate model that maintains strict compatibility with the observed execution sequences [2]. Passive inference bypasses the operational overhead of live interaction, making it particularly advantageous when (for example) live querying poses security risks, or execution logs are already abundantly available. This thesis focuses on passive learning.

Passive model learning relies on the availability of a complete sample of the behaviour. Although the definition of a complete sample varies considerably across different model-inference techniques, constructing an accurate target model strictly requires datasets that are both structurally complete and representative of the underlying system [20, 54]. To satisfy these conditions, the execution traces must exhibit sufficient diversity to fulfil two fundamental criteria:

- Every valid state transition within the target system must be traversed.
- All non-equivalent states must be structurally distinguished.

The practical challenge is that such characteristic samples are typically massive and scale exponentially with system complexity [54]. In real-world software engineering contexts, developers rarely have access to such exhaustive data; instead, inference proceeds from sparse, incomplete execution logs. Consequently, a central objective of passive learning research is to infer highly accurate models despite severe dataset sparsity.

Fundamentally, the primary objective of model inference is the identification of the minimal model that remains consistent with a provided set of execution traces [2]. Model complexity of a model is typically quantified by state-space cardinality; thus, a minimal model is the representation with the fewest possible states that correctly classifies a given set of samples. Identifying the minimal model consistent with a given sample set is a known NP-hard optimisation problem [26, 24].

More broadly, passive model inference presents severe computational and structural challenges. In practice, passive algorithms must rely exclusively on pre-collected, static trace data without an interactive oracle, making them acutely vulnerable to data sparsity, unobserved operational paths, and stochastic label noise. This complexity is further exacerbated by large alphabet cardinalities, where the sample size required for accurate model reconstruction grows linearly with the event space. Furthermore, a deficit of rejected traces frequently induces overgeneralisation, causing the inferred model to accept a language far broader than the target system’s actual operational scope and preventing the learner from accurately delineating valid state boundaries.

Several existing inference algorithms employ greedy heuristics [27, 31] to mitigate data limitations and learn the minimal model. However, they remain highly error-prone when operating under extreme data sparsity. Other approaches have employed strategies to identify behavioural patterns from the available trace data [36, 43, 44]. Yet, these strategies are often computationally expensive and rely heavily on predefined, domain-specific pattern libraries, which are rarely available in practice. Consequently, a pronounced research gap persists in developing a robust methodology to induce accurate models from highly sparse data by dynamically extracting and leveraging domain knowledge.

To address this limitation, this thesis introduces a novel model-inference framework that mines patterns from sparse execution traces to guide the inference process. By leveraging the available trace data to formulate high-level behavioural rules, this approach uncovers underlying patterns that would typically require an immense amount of training traces to detect [52]. Thereby effectively mitigating the challenges associated with data sparsity.

We integrate this pattern-based approach into two standard state-machine inference algorithms, using the mined patterns as a secondary source of information. Whenever the learner (i.e., the inference algorithm) proposes an assumption about the target machine, the framework checks the resulting configuration against these patterns. If a violation is detected, the proposed assumption is rejected. This constraint mechanism ensures that the inferred model remains entirely consistent with the traces while adhering to the mined patterns.

Our technique extends two foundational model inference frameworks:

- Evidence-Driven State Merging (EDSM): Representing a baseline in model inference [31], this greedy heuristic is based on the local density of matching trace data.
- DFASAT: An advanced framework [27] that applies preliminary rigid heuristics to EDSM before encoding a Boolean Satisfiability (SAT) solver to find a model with the smallest set of states.

1.2 Research Objectives

This thesis addresses the following specific Research Objectives:

- RO1** Design and implement an automated method for mining behavioural patterns from execution traces and integrating them into existing passive model inference algorithms.
- RO2** Measure the effectiveness of integrating the mined-patterns (from **RO1**) into existing passive model inference algorithms.
- RO3** Critically evaluate the performance and scalability of the proposed pattern-based algorithms (from **RO2**) across a diverse range of software systems and Random automata.

1.3 Thesis Contributions

Our proposed approach introduces the following primary contributions to the field of passive model inference:

- C1 Lightweight Pattern-mining Methodology:** To achieve the goals of **RO1**, we developed a novel, computationally lightweight method that extracts patterns from training traces.
- C2 Integrating the mined patterns into existing algorithms** We designed and implemented a pipeline that leverages the mined patterns (from **C1**) to guide state selection within the EDSM heuristic. Furthermore, we adapted the DFASAT framework to design a hybrid, pattern-based SAT encoding scheme.
- C3 Comprehensive Empirical Evaluation:** To achieve the goals of **RO2** and **RO3**, we systematically evaluated our implementations using real-world reactive protocols. Specifically, the Bluetooth protocol, OpenSSL, ABP Sender protocol, and TCP state machines. In addition to randomly generated automata. The use cases are validated using efficiency and accuracy metrics to meet the goals of **RO2**. The training data are extremely sparse, so the baseline, DFASAT, failed to infer a correct model due to a lack of traces. The evaluation also includes the runtime for each approach across different data sizes to meet the goals of **RO3**.

1.4 Research Outputs

1. Conference paper (under peer review): Arwa Alfitni, Kirill Bogdanov, and Neil Walkinshaw "Automated Pattern-based Model Inference", submitted to **ICGI 2026, 17th International Conference on Grammatical Inference** <https://icgi2026.tudelft.nl/cfp.html>
2. Open-source tool: a tool in Python implementing the mentioned algorithms in this thesis is publicly available on GitHub at <https://github.com/arwaalfitni/Pattern-based-passive-automata-Learning>
3. Data: all generated random automata used in this thesis are publicly available on GitHub at https://github.com/arwaalfitni/Pattern-based-passive-automata-Learning/tree/main/reference_automata
4. Inferred Models: all inferred models produced by our algorithms and the baseline DFASAT with their evaluation results are publicly available on GitHub at <https://github.com/arwaalfitni/Pattern-based-passive-automata-Learning/tree/main/Experimental%20Results>

1.5 Thesis Overview

The remainder of this thesis is structured as follows:

- Chapter 2 (Background): Establishes the theoretical foundations of model inference. It introduces the formal definitions and operational mechanics of the baseline EDSM and DFASAT learning algorithms used throughout this work.
- Chapter 3 (Mining Patterns within EDSM): Details the primary methodology and algorithmic contributions of this thesis. It presents our novel lightweight pattern-mining method

and details how these mined patterns are integrated as constraints within the state-merging pipeline.

- Chapter 4 (Experimental Setup): Outlines the empirical framework established to validate the proposed algorithms. This includes a detailed description of the industrial case studies, trace generation parameters, and evaluation metrics.
- Chapter 5 (Results and Discussion): Presents a comprehensive analysis of the empirical findings. It critically evaluates the performance of our pattern-based frameworks against the baseline in terms of accuracy, computational efficiency, and runtime. Finally, it discusses the research limitations.
- Chapter 6 (Related Work): Reviews the existing literature in the field of passive model inference. It critically examines prior improvements to the EDSM algorithm and provides a comprehensive overview of how temporal and sequential patterns have traditionally been used to guide model inference.
- Chapter 7 (Conclusions and Future Work): Summarises the key insights and core contributions documented in this thesis, and outlines prospective directions for future investigations in passive model inference.

Chapter 2

Background

Effective software design, development, and maintenance depend heavily on accurate program specifications. To be beneficial, these specifications must be precise, up to date, and accessible. Models provide formal, visual descriptions that are unambiguous. They facilitate the identification of bugs and the understanding of the system’s structure [35]. They are also significant for model-based testing [47, 32]. Furthermore, these models foster a collaborative environment, enabling developers and stakeholders to align their understanding of system requirements [52]. This shared comprehension enhances communication and streamlines the development process.

Despite their importance, formal models are often missing, incomplete, or provided only informally [52]. The high time and financial costs associated with manual generation and maintenance often lead developers to omit formal specifications [52, 35]. To address this, one can extract specifications either statically from source code [57, 42] or dynamically by observing a program’s execution [1, 36, 27, 52]. The process of deriving an automaton-based model is known as model inference (also known as automata learning or model induction). This chapter establishes the conceptual framework for representing software behaviour visually and examines the existing research in model inference that serves as the foundation for our proposed approach.

The chapter is structured as follows:

- Section 2.1 (Software Models and their Inference): Provides a formal definition of Mealy machines and explores various frameworks for inferring models from data.
- Section 2.2 (Passive Model Inference): Focuses on the techniques most prevalent within the passive inference domain, explores the existing challenges, and introduces the basic knowledge for the proposed approach.
- Section 3.1 (Motivation): Concludes by identifying the critical gaps in existing literature that necessitate the contributions of this thesis.

2.1 Software Models and their Inference

This section establishes the foundational definition of software models (specifically Mealy machines) and examines how developers model the behavioural characteristics of real-world systems. A software model, such as a state machine, provides a formal representation of a system’s behaviour, illustrating the temporal logic that governs event sequences and explicitly defining which actions

are permitted or prohibited at any given state. Finally, we contrast the two primary paradigms in model inference: passive and active.

2.1.1 Software as Behaviour Models:

Representing the behaviour of software systems through models significantly enhances comprehensibility [52]. Modern software structure typically comprises numerous interconnected components and complicated interdependencies; models (automata) mitigate this complexity by providing a rigorous abstraction of event sequences and control flows. Such models facilitate a systematic understanding of system transitions and state-space explorations.

Walkinshaw *et al.* [54] conducted a comparative analysis of software characteristics. The study involved 20 existing state-machine models, primarily sourced from the research literature and industrial protocols. The results highlight that, in software models, each state is reachable from the starting state, each state has a unique label for each transition to adjacent states, and models describe the behaviour of the software with the smallest number of states. Furthermore, models do satisfy specific architectural benchmarks, including decoupling the alphabet size from global structural features, maintaining a linear or near-linear relationship between depth (the longest “shortest walk” from the starting state to any other state) and state count, and ensuring that any state is reachable from the root state.

This thesis focuses on model inference of real software systems, derived from the analysis of empirical event sequences. In this thesis, the term ‘automata’ will refer to the behavioural models (specifically, state machines).

Mealy Machine

Mealy machines are automata in which the transition label is defined as an input-output pair [51]. The input is an action that triggers the system to enter a state, while the output is the operational response. While finite automata (automata with a finite set of states) are traditionally employed as recognisers for formal languages, Mealy machines extend this abstraction to model the execution traces of interactive systems. This distinction is critical when analysing modern systems, which function through a continuous exchange of information with their environment. In this context, the Mealy machine serves as a formal transducer, transforming sequences of input events into output responses. Consequently, the Mealy architecture serves as a pivotal model for accurately characterising the state-transition behaviour of dynamic systems.

For instance, the operational logic of a simplified text editor can be formally demonstrated via the model illustrated in Figure 2.1. The system starts in State *A*. From this baseline configuration, the operation *open/1* triggers a transition to State *B*; that is, from state *A*, if an input *open* is chosen, the system will be in State *B* with output 1 as a result of the transition. In this example, output 1 indicates a proceeding transition, while output 0 indicates a termination transition. For instance, from State *A*, the inputs *edit* or *exit* trigger a transition to State *C* with output 0.

A deterministic Mealy machine generates a specific output based on the current state and the received input symbol. In the aforementioned text editor example, the system starts in State *A*. Upon receiving the input *open*, the machine executes a transition to State *B*. In addition to this state change, the machine produces the output 1, signifying the successful execution of the command. Conversely, undefined output (such as output 1 for input *edit* from State *A*) is explicitly prohibited by the model. In such instances, the software remains unresponsive, as these input/output pairs do not correspond to valid state transitions within the established grammar of the system. In a deterministic Mealy machine, every state-input mapping provides a deterministic path to both a subsequent state and a unique output.

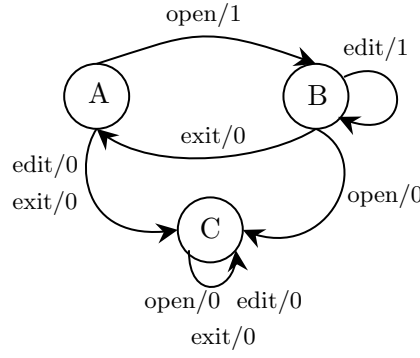


Figure 2.1: Text Editor Automaton

Definition 1 (Deterministic Mealy Machine) A Deterministic Mealy machine is defined as a five-tuple: $\mathcal{M} = (Q, q_0, I, O, \mathcal{T})$, where:

Q is a finite, non-empty set of states.

q_0 is an initial state, an element of Q .

I and O are the input and output alphabet (finite, non-empty, disjoint sets of symbols).

$\mathcal{T}$ is the transition function: $\mathcal{T} \subseteq Q \times I \times O \times Q$ Where a transition is a 4-tuple (q, i, o, q')

$\mathcal{M}$ is deterministic if and only if for each pair $(q, i) \in Q \times I$, there exist at most one transition $(q, i, o, q') \in \mathcal{T}$, for some $(o, q') \in O \times Q$

According to Definition 1, the transition (q, i, o, q') means state q has an outgoing transition with the input i that goes to state q' and produces output o .

Figure 2.1 illustrates a representative example of such a deterministic Mealy machine. The automaton is defined by the following components:

- The set of states: $Q = \{A, B, C\}$.
- The initial state: $q_0 = A$.
- The input alphabet: $I = \{\text{open}, \text{edit}, \text{exit}\}$.
- The output alphabet: $O = \{0, 1\}$.
- The transition function: $\mathcal{T}$, representing the set of all defined state transitions.

On the other hand, a **Non-Deterministic Mealy machine** is one with at least one state which has multiple outgoing transitions with the same input, leading to more than one final state. Definition 2 introduces the formal definition of a nondeterministic Mealy machine.

Definition 2 (Non-Deterministic Mealy machine) A Nondeterministic Mealy machine is defined as an five-tuple: $\mathcal{M} = (Q, q_0, I, O, \mathcal{T})$, where:

Q, q_0, I, O , and $\mathcal{T}$ have the same definition of the Mealy machine (Definition 1)

$\mathcal{M}$ is nondeterministic if for some pair $(q, i) \in Q \times I$, there exist at least two transitions $(q, i, o_1, q_1), (q, i, o_2, q_2) \in \mathcal{T}$, such that $o_1 \neq o_2$ or $q_1 \neq q_2$

Figure 2.2 shows an example of a nondeterministic Mealy machine. Notice that state B has two outgoing transitions with input *edit*; a self-loop and another leading to state C . According to Definition 2 the pair $(B, \text{edit}) = \{(B, \text{edit}, 1, B), (B, \text{edit}, 0, C)\}$, which means that state B is a non-deterministic state. Thus, the automaton is a nondeterministic Mealy machine.

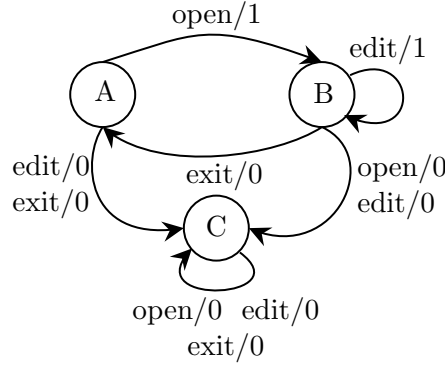


Figure 2.2: Example of a non-deterministic Mealy machine

Our primary objective is to learn a deterministic Mealy machine. During the iterative learning process, the intermediate hypothesis automata may be temporarily transformed into a Non-deterministic Mealy Machine. To maintain the integrity of the target specification, we employ a resolution strategy to eliminate non-determinism (as discussed in Section 2.2.2). Consequently, the final model produced by our learner is guaranteed to be a fully deterministic Mealy machine.

Input-Enabled Automata

A deterministic Mealy machine is defined as input-enabled when every state possesses **exactly one** outgoing transition for every symbol in the input alphabet, $i \in I$. This property ensures a comprehensive definition of the system’s response to any conceivable input sequence. In practical software implementations, not every input necessitates a state transition or a distinct operational action. Frequently, systems are designed to ignore inputs that are meaningless in the current operational context. There are two primary mechanisms employed to handle these “out-of-context” or unexpected inputs:

1. **Self-loops:** The system may implement a self-loop in the current state, often accompanied by a null or informative output. This signifies that the input has been recognised but does not warrant a change in the underlying state. For instance, in a text editor, invoking the *save* command when the document has no unsaved changes typically results in a self-loop. The system remains in the “Saved” state without performing redundant disk I/O.
2. **Sink States:** Alternatively, the system may take a transition to a sink state (also known as a “dead” or “trap” state). A sink state is characterised by the absence of transitions to any other state, essentially trapping the automaton in a terminal configuration. For example, consider a security-critical authentication module. If a user exceeds the maximum number of permitted login attempts, the system may transition from an “Authenticating” state to a “Locked” state. Once in the “Locked” state, all further inputs (such as password entries) lead back to the same state, preventing further access until a manual administrative override occurs. In this context, the “Locked” state serves as a sink state, representing a terminal condition for that specific execution trace.

This thesis focuses on inferring an input-enabled Mealy machine that uses the sink-state structure to handle unexpected inputs.

2.1.2 Model Inference

The primary objective of model inference is to synthesise a high-fidelity representation that accurately reflects the system’s operational dynamics, thereby mitigating the impact of missing documentation [52, 36, 35]. Moreover, model inference aims to find the smallest “minimal” model (the smallest number of states), assuming the target model is minimal and applying Occam’s razor: smaller is simpler [60].

Inferred models are an effective tool for system comprehension and offer a transparent communication channel between developers and stakeholders. Beyond basic inspection, models enable several critical verification and validation tasks for the software. These models enable automated test suite generation and facilitate model-based testing [32, 56, 40] and model checking [14, 34], allowing developers to evaluate a system against its temporal logic properties, spot bugs early in development, and support refactoring embedded controllers. [40, 39].

Integrating inferred hypothesis models into black-box model checking and model-based testing allows effective verification of complex systems without accessible internal designs [40, 39]. Black-box checking is highly effective when full system models are intractable or impossible to learn completely due to state-space explosion [39]. Testing frameworks can detect safety violations early in the verification cycle by performing model checking directly on intermediate hypothesis models during the learning process, rather than waiting for full model convergence. In [40], counterexamples flagged by the model checker are executed against the target (actual) software; if confirmed, a genuine failure is exposed, whereas a divergence provides feedback to refine the inferred model. This dynamic feedback loop between learning, conformance testing, and optional runtime monitoring dramatically increases test generation efficiency and bug detection depth, enabling fast identification of deep faults even when learning a complete system model is intractable [40, 39].

Weyuker [56] proposes a method that uses the inferred model to assess test adequacy (whether a software has been tested thoroughly enough to stop testing). Software testing and model inference as inverse processes [56]: while testing attempts to expose errors in a given program relative to a specification, inference attempts to reconstruct the simplest underlying model from execution data. A set of tests is enough to stop testing if and only if the model inferred from that test set is mathematically equivalent to both the target software being tested and its formal specification.

Walkinshaw [52] noted that evaluating a black-box system in the absence of explicit requirements is particularly challenging, as it is impossible to determine the adequacy of a given test suite without a reference model. Software model inference has subsequently gained significant traction within the testing industry as a means to generate relevant test cases and reduce the effort of manual verification [35]. Numerous research studies have integrated the induction of state machines with testing frameworks [25]. For instance, Walkinshaw [52] utilised the concept of inductive inference to derive formal models directly from test sets to improve overall system coverage. An interesting work by [13] employed model inference to construct a behavioural model of Android apps during testing. This model is then used to generate targeted user inputs that drive the application into unexplored states, with the resulting execution data recursively fed back into the system to refine the model.

Active and Passive Inference

The inference of software models is generally categorised into two paradigms: passive and active learning. In the passive learning paradigm, the learner operates exclusively on a fixed corpus of information regarding the system’s behaviour. In the software engineering domain, this data may manifest as execution traces, system logs, or existing test suites. By processing these datasets, the learner induces a hypothesis model that remains strictly compatible with the observed execution traces [2]. The objective of this process is to generate a formal structure that simulates the actual model of the system (known as System Under Learning (SUL), or target model) for subsequent verification or testing.

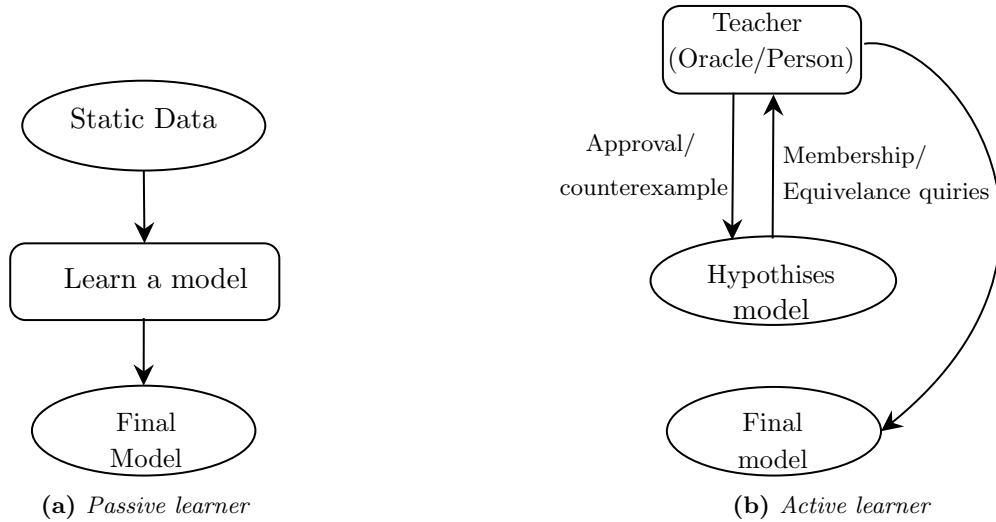


Figure 2.3: The difference between the Passive and the Active learning approaches

A significant limitation of passive learning is its inherent dependency on the volume and coverage of the provided traces. In many practical scenarios, static datasets are insufficient to fully capture the system’s logic, particularly as the complexity and state space of the software increase [15]. Consequently, passive models may fail to accurately represent the system’s behaviour if the underlying training data is sparse.

Active inference [4] addresses these shortcomings by facilitating direct interaction with the target system. Rather than relying on a static archive, the active learner proactively explores the system by posing structured queries to an oracle. This iterative process, outlined in Figure 2.3b, is designed to resolve ambiguities and systematically map the state space.

The queries typically comprise two primary mechanisms:

1. **Membership Queries:** The learner interrogates the oracle to determine if a specific sequence of events (a trace) is permissible within the system’s language.
2. **Equivalence Queries:** The learner proposes a hypothesis model to the oracle to verify its correctness. If the model is deficient, the oracle provides a counter-example (a trace that the hypothesis incorrectly classifies), which the learner then uses to refine the model.

Passive learning prioritises computational efficiency by decoupling the inference engine from the target system. However, it suffers from an inherent sensitivity to the quality of the training data. While active learning addresses this limitation by facilitating guided interaction to ensure model convergence, it introduces a substantial infrastructural complexity and relies on the strict assumption of an accessible, omniscient oracle with complete knowledge of the hidden model.

Ideally, the choice between these paradigms thus involves a critical trade-off between the ease of offline processing and the high-fidelity structural guarantees afforded by interactive querying.

In many real-world software engineering and reverse-engineering contexts, active querying is fundamentally infeasible due to the absence of such an oracle. Examples include performing post-mortem malware analysis, evaluating retired legacy software, or analysing network traffic collected from remote network hardware. When an interactive teacher or a reliable simulator is unavailable, active learning frameworks become inapplicable.

Furthermore, interacting with a live production environment can cause operational disruptions, data corruption, or physical damage (e.g., active testing of industrial control systems, automotive electronics, or medical devices). When active queries carry financial risk, safety hazards, or extreme time delays (such as rate-limited web APIs), passively relying on recorded logs is far safer and cheaper.

From an architectural standpoint, active learning relies heavily on resetting the system to its initial state between queries to test different paths. Many real-world systems (e.g., legacy software, live physical machinery, databases without snapshot capabilities) lack an automated, fast, or reliable reset mechanism. Building an active harness for such systems is often technically impossible or prohibitively expensive.

Under these operational constraints, passive inference becomes uniquely significant. It serves as the only viable mechanism for reconstructing system behaviour, leveraging historical log files, execution traces, and pre-existing telemetry data to approximate the underlying model without requiring live system interaction.

This research focuses primarily on passive model inference. Throughout the remainder of this work, the term “model inference” refers implicitly to passive techniques, except where active inference frameworks are explicitly discussed.

2.1.3 Challenges in Passive Inference

Inferring a deterministic finite automaton of minimal size from a finite set of samples is NP-complete [24]. In practice, the efficiency and accuracy of passive inference frameworks are heavily dictated by the characteristics and quality of the available execution traces. Several interconnected factors exacerbate this computational difficulty:

- **Data Sparsity:** Data sparsity represents a significant impediment, particularly within the context of passive learning, where the model inference relies exclusively on pre-collected, static training data. Established passive learning algorithms, such as EDSM, often exhibit degraded performance with sparse datasets or target automata (i.e. the model that the algorithm is trying to infer) with high state counts [1].
- **Alphabet Cardinality:** The size of the alphabet is a critical determinant of inference difficulty. The alphabet is a set of all events recognised by the target model. A linear correlation exists between alphabet cardinality and the sample size required for accurate model reconstruction [38]. In systems characterised by large alphabets, learning from restricted samples is inherently insufficient to capture the full operational logic of the software.
- **Imperfect set of Traces:** Contemporary learning techniques often presuppose that the set of execution traces is comprehensive and precisely represents system properties. The Imperfect set of traces does not fully represent the system’s features, where parts of the

system are not covered by the traces set. Consequently, these methods frequently struggle in scenarios where only an imperfect set of traces is available, as they typically require high-fidelity data for successful inference [59]. In industrial environments, however, achieving such “perfect” trace acquisition is often infeasible.

- **Stochastic Noise:** The presence of noisy or erroneous training data introduces further complexity. Conventional learning paradigms generally operate under the assumption of trace integrity. Significant challenges arise when labels are incorrectly assigned. For instance, invalid behaviours are classified as valid. While some research has addressed this through robust learning strategies, noise identification, and adaptive thresholds [29, 58, 5], mitigating label noise remains a persistent obstacle in the field.
- **Overgeneralisation:** A deficit of rejected traces frequently results in overgeneralisation, a phenomenon where the inferred model accepts a language significantly broader than the target system’s actual behavioural scope [52]. Without sufficient negative constraints, the learner cannot accurately delineate the boundaries of the valid state space. Notably, this does not apply to Mealy machines, in which the learner avoids some of the overgeneralisation because the output function λ has to remain deterministic.

The challenges previously identified apply to all state machine structures. This thesis focuses on inferring Mealy machines, as they provide a more authentic representation of real-world systems by incorporating both inputs and outputs on transition labels. However, this increased expressiveness broadens the target machine’s characteristics, specifically with respect to alphabet cardinality. In a Mealy machine, the effective alphabet size constitutes the Cartesian product of the input and output sets ($I \times O$). Moreover, Mealy machines in software engineering are often input-enabled, which introduces further difficulties:

- **Expanded Transition Density:** Completeness increases the total number of transitions, significantly broadening the observed behaviour.
- **Semantic Ambiguity:** In an input-enabled automaton, transitions intended to represent unexpected inputs (such as self-loops or transitions to a “sink-state”) may appear as valid, positive behaviours, whereas semantically they are not permitted transitions from the current state.
- **Class Confusion:** Structurally, every transition leads to a state that is a valid state, even if the transition semantically denotes an invalid sequence. This overlap complicates the distinction between valid system operations and error-handling paths. In our evaluation (Section 4.3), we describe the methodology used to resolve this semantic confusion, ensuring a fair and rigorous assessment of our learner.

2.2 Passive Model Inference

In model inference, passive learning algorithms construct a finite-state representation by observing a fixed sample of the target model. The samples are usually execution traces. A trace is an ordered sequence of events representing a specific execution or operational function of a system. A trace typically originates from the initial state q_0 , progresses through a series of transitions triggered by inputs. A trace is formally defined in Definition 3

Definition 3 (Trace) A trace t consists of a sequence of input/output pairs

$$\langle (i_1, o_1), \dots, (i_i, o_i), \dots, (i_n, o_n) \rangle$$

with $i_i \in I$ (I is the Input alphabet), $o_i \in O$ (O is the output alphabet), and $n \in \mathbb{N}$ defining the sequence length.

Within the framework of formal languages, traces are classified as either positive or negative. A positive trace constitutes a valid sequence of events permitted by the system's specification, thereby belonging to the system's language. In a positive trace, there is a transition for each pair of input-output (i_i, o_i) from the current state to another state in $\mathcal{M}$. The following are examples of positive traces for the Text Editor system illustrated in Figure 2.1:

$\langle \text{open}/1, \text{edit}/1, \text{exit}/0, \text{open}/1, \text{open}/0 \rangle$
 $\langle \text{open}/1, \text{exit}/0, \text{edit}/0 \rangle$

Note that the first trace represents a path that starts from the initial state (state A) in Figure 2.1, where each following pair is a valid transition label to another state in the model. Starting from state A, the pair $\text{open}/1$ means the system can traverse from state A to state B with the input *open* and produce the output 1. Once in state B, the system remains in the same state upon receiving input *edit* and produces the output 1. Next, the system can perform the transitions (B, exit, 0, A), (A, open, 1, B), (B, open, 0, C) sequentially.

Conversely, a negative trace is one that the system explicitly disallows. In deterministic, input-enabled Mealy machines, where each state has an outgoing transition for each $i \in I$, the negative trace in this context has at least one transition that produces an incorrect output upon receiving an input from the current state. The following are examples of negative traces for the Text Editor system (Figure 2.1):

$\langle \text{open}/1, \text{edit}/1, \text{edit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/0 \rangle$

The first trace can traverse between states with the first two pairs ($\text{open}/1$ and $\text{edit}/1$), represented by the transitions (A, open, 1, B) and (B, edit, 1, B). For the last pair, the system does not have a defined transition for $\text{edit}/0$ from state B (the current state). The same applies to the second trace: from the initial state (state A), the system can traverse to state B with input *open* and output 1, but cannot perform any transition step from state B upon receiving the input *edit* while producing the output 0.

For the purpose of model inference, it is necessary to distinguish between positive and negative traces. This information is essential for the learner (the inference algorithm), as it defines the system's constraints and establishes the boundaries of the language. Without negative traces, the inference process lacks the necessary “pressure” to avoid over-generalisation [52], which would otherwise result in a model that accepts a wider range of behaviours than the actual system allows.

However, with Mealy machines, the over-generalisation issue can be partially avoided by utilising the determinism property applied by the transition function $\mathcal{T}$ (Definition 1). Two states are distinct if, upon receiving the same input, they result in two different outputs via a one-step transition relation (explained in detail later in this section). For all the work in this thesis, we require only positive traces.

2.2.1 Prefix Tree Acceptor (PTA)

The construction of the Prefix Tree Acceptor (PTA) constitutes the initial phase of the hypothesis-building process of model inference. This structure serves as the primary representation from which the final model is induced. In this work, A PTA is a tree-based representation constructed exclusively from a set of positive traces, with each event (input/output pair) serving as a label for a transition between two states. As illustrated in Figure 2.4, the construction of this tree is for the following are examples of positive traces for the Text Editor system illustrated in Figure 2.1:

$\langle \text{open}/1, \text{edit}/1, \text{exit}/0, \text{open}/1, \text{open}/0 \rangle$
 $\langle \text{open}/1, \text{exit}/0, \text{edit}/0 \rangle$
 $\langle \text{open}/1, \text{exit}/0, \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{open}/0, \text{open}/0 \rangle$
 $\langle \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{open}/0, \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{edit}/1 \rangle$

To maintain determinism, sequences with the same prefix labels enter the same state, demonstrating the overlapping event sequences that are consolidated into a prefix-shared structure. Note that each state in the PTA of Figure 2.4 has the same shape (borderline) as the corresponding states of the target model in Figure 2.5. For example, the root of the PTA corresponds to the initial state of the target model (state A); the root of the PTA will have the same borderline as state A. According to Guillaumier [26], a model can be induced either directly from its initial tree-representation or through the formal encoding of its transition matrix (a matrix used to describe the probabilities of transitioning between states).

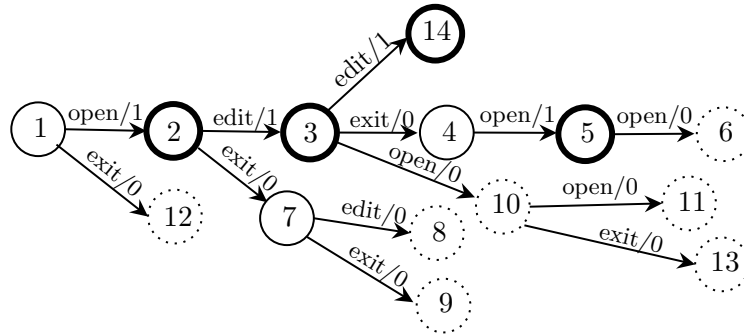


Figure 2.4: *Prefix Tree Acceptor (PTA)*

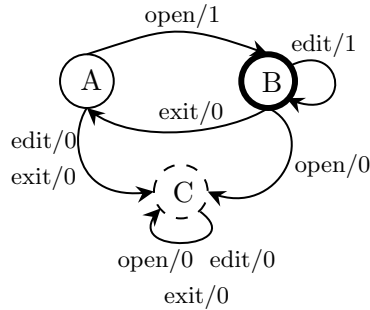


Figure 2.5: *Target Model*

2.2.2 State Merging

State merging is the most prevalent technique employed in passive automata learning. The foundational objective of this approach is to systematically collapse the states of a PTA to recover a compact automaton that closely mirrors the original target system. Merging two states consolidates all of their respective incoming and outgoing transitions into a newly formed state. Formally, if state s is merged with state s' , the resulting state s'' inherits all incoming transitions originally directed to s and s' , alongside all outgoing transitions originating from both states.

A fundamental constraint on this process is that state merging must strictly preserve the determinism of the automaton. In a deterministic Mealy machine (Definition 1), this property is governed by the transition function $\mathcal{T}$, which serves as the primary criterion for evaluating state consistency. Two states are considered inconsistent if they share a transition with the same input symbol but produce different outputs. Two states are considered inconsistent if they share a transition with the same input symbol but produce different outputs.

When a merge creates two distinct transitions with the same input-output pair targeting different destination states, the target states themselves must be merged. This cascading process of folding states continues recursively until all local nondeterminism is fully resolved. For example, in Figure 2.4, merging state 1 and state 2 yields a consolidated state 1 featuring a self-loop labelled `open/1` (see Figure 2.6), an outgoing transition to state 3 labelled `edit/1`, and a non-deterministic split under the label `exit/0` targeting both state 7 and state 12. Because this split violates the determinism property (i.e., $(1, \text{exit}, 0, 7), (1, \text{exit}, 0, 10) \in \mathcal{T}$, where state $7 \neq \text{state } 12$), the conflict is dynamically resolved by cascading the merge to combine states 7 and 12, as illustrated in Figure 2.7.

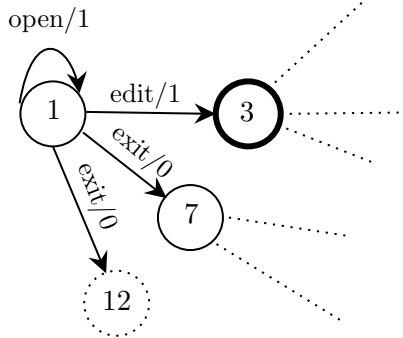


Figure 2.6: Merge States 1 and 2 raise a nondeterminism

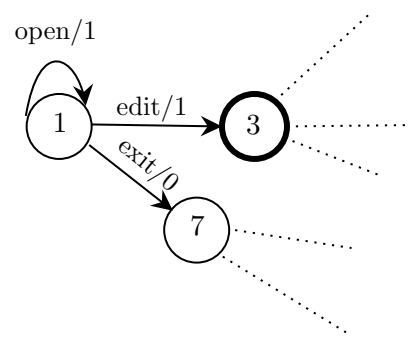


Figure 2.7: Resolve the nondeterminism by merging states 7 and 12

However, not all non-determinism can be successfully resolved. If a cascading merge leads to an explicit structural contradiction (output non-determinism), the initial merge operation is blocked and rejected. The merge with the output non-determinism is the only case where the non-determinism can't be solved, and the merge is rejected. For instance, attempting to merge state 4 and state 5 introduces non-determinism (Figure 2.8) by producing a self-loop around state 4 labelled $\text{open}/1$ alongside a transition to state 6 labelled $\text{open}/0$. Resolving this requires a secondary merge between states 4 and 6 (Figure 2.9), which ultimately leaves state 4 with two self-loops under the same input symbol but with conflicting outputs ($\text{open}/1$ and $\text{open}/0$). This directly violates the output determinism criteria (i.e., $(4, \text{open}, 0, 4), (4, \text{open}, 1, 4) \in \mathcal{T}$, where $\text{output } 0 \neq \text{output } 1$). Because this contradiction cannot be resolved, the initial merge between states 4 and 5 is deemed invalid and is blocked.

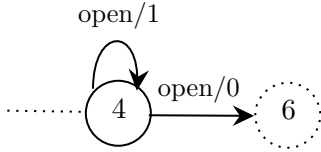


Figure 2.8: Merge State 4 and 5 raise a nondeterminism

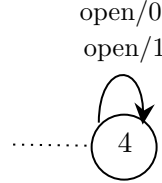


Figure 2.9: Conflict cannot be resolved

The primary challenge of state-merging lies in correctly folding a tree-structured PTA back into the system's original structure. Strategies for selecting candidate state pairs have been extensively documented in the literature [10, 31, 16, 26]. These strategies select states that are most likely to refer to the same state (of the actual target model). Among a wide range of different strategies, the Evidence-Driven State Merging (EDSM) algorithm [31] has been widely established as a reliable and practical algorithm.

2.2.3 Evidence-Driven State Merging

Evidence-Driven State Merging (EDSM) [31] is a passive learning method and is considered a reliable and practical technique in state merging [27]. It was the winning algorithm for the Abbadingo One DFA Learning competition [31] and remains one of the few approaches capable of managing large target Deterministic Finite Automata (DFA) [60, 31]. The core principle governing EDSM is to defer merging until substantial empirical evidence indicates that a specific merge is likely to be correct. Whenever a valid merge is identified, a heuristic score is calculated, and the candidate state pair with the highest score is selected for merging. Structurally, this score reflects the total number of transitions consolidated during the recursive resolution of local nondeterminism.

The algorithm begins by constructing a PTA from the complete set of training traces. EDSM then evaluates every possible pairwise combination of states (q, p) . A valid merge between a candidate pair of states is one in which nondeterminism (if any) can be resolved, and a merging score $S_{\text{EDSM}}(q, p)$ is computed. If a conflict arises, the merge is classified as invalid, and its score is penalised with a negative value.

For valid merges, the pair is scored based on the degree of overlapping evidence. Specifically, the number of shared labels and isomorphic sub-trees rooted at the two target states. This metric can be calculated by explicitly enumerating the overlapping transitions (Equation 2.1), which is the number of merged transitions between the considered states (either directly or indirectly during solving the non-determinism). In other words, two states p and q have an overlapping transition if each state has a transition with the same input/output pair label; when merging p and q , the new state will be non-deterministic, and the next states for both transitions must be merged to resolve the non-determinism.

Formally, according to Definition 1, two states p and q of a model $\mathcal{M}$ have an overlapping transition if there are two transitions (p, i, o, p') and $(q, i, o, q') \in \mathcal{T}$. The total number of overlapping transitions between p and q is the sum of overlapping transitions between any two states merged as a result of merging the two sub-trees rooted at p and q . For example, if the next states p' and q' have overlapping transitions, the total number of transitions of p and q is $\text{overlapping_transitions}(p, q) + \text{overlapping_transitions}(p', q')$.

$$S_{\text{EDSM}}(p, q) = |\text{overlapping_transitions}(p, q)| \quad (2.1)$$

The EDSM score can also be calculated by computing the net reduction in the total state count resulting from the merge (minus one), as formalised in Equations 2.2:

$$S_{\text{EDSM}}(q, p) = (|Q_{\mathcal{M}}| - |Q_{\mathcal{M}'}|) - 1 \quad (2.2)$$

where $q, p \in Q_{\mathcal{M}}$ represent the set of states prior to the merge, and $Q_{\mathcal{M}'}$ denotes the set of states in the resulting collapsed automaton. For instance, $S_{\text{EDSM}}(1, 2) = 1$ in the example of Figure 2.4. Within the standard EDSM framework, a score of zero implies an absence of overlapping evidence to justify the state collapse; however, such an operation may still constitute a valid merge. Consequently, baseline configurations typically leave it to the developer’s discretion to either permit the merge or treat the zero-score pair as unmergeable. In our implementation (Algorithm 1), we enforce a strict evidence-driven constraint, permitting a merge operation to proceed if and only if there is sufficient structural justification (i.e., $S_{\text{EDSM}}(q, p) > 0$).

Our primary experiments showed that EDSM is more likely to make mistakes and perform incorrect merges (merge two states that refer to different states in the target (hidden) model) when it considers a merge with a zero-score. Although a score of zero means that the merge is valid, it lacks evidence that the two states to be merged actually refer to the same state (in the target model). When we implemented the EDSM to consider the zero score, the result showed more incorrect merges during the first few merges. Performing incorrect merges in the early phase affects the following selection decisions, which means a risk of a cascade of incorrect merges. Conversely, when a merge is primed only when there is evidence (i.e., $S_{\text{EDSM}}(q, p) > 0$), it mitigates the risk of early incorrect merges.

A candidate pair receives a default negative score ($S_{\text{EDSM}}(q, p) = -1$) if the merge is structurally invalid. In a Mealy framework, an invalid merge occurs when it maps identical input sequences to conflicting output symbols. For instance, $S_{\text{EDSM}}(4, 5) = -1$ in the example of Figure 2.9.

EDSM then evaluates all candidate state combinations in the current iteration and merges the pair with the highest score S_{EDSM} . This evaluation and execution loop repeats iteratively until no valid, positive-scoring pairs remain, concluding the learning process.

To enhance the efficacy of the EDSM algorithm, various search procedures have been developed; see, e.g., [30, 12]. Among these, the Blue-fringe framework [31] is recognised as a leading methodology. By prioritising state-pair candidates, this approach effectively constrains the search landscape.

The Blue-Fringe Strategy

The Blue-Fringe strategy, often referred to as the Red-Blue algorithm [31], is an efficient search heuristic that manages the search space during state merging. Unlike exhaustive search methods that evaluate all possible state pairs, the Blue-Fringe approach focuses on a specific “frontier” of candidate merges, significantly reducing computational overhead.

The strategy partitions the states of the PTA into three disjoint categories using a colouring heuristic:

- **Red States (*RED*):** These represent the “kernel” or the core of the induced hypothesis automaton. Red states are assumed to be mutually distinct and incompatible, serving as the basis for the final model’s state space.
- **Blue States (*BLUE*):** These constitute the “fringe” or search frontier. A blue state is defined as any immediate non-red successor of a red state.
- **White States (*WHITE*):** These represent the remaining, unvisited nodes within the tree structure that lie beyond the current frontier.

The inference process proceeds through an iterative cycle of evaluation:

1. **Merge Evaluation:** The algorithm attempts to merge a selected blue state $b \in BLUE$ into an existing red state $r \in RED$. The merge is speculatively executed and evaluated. It is considered valid only if the two states are structurally compatible and do not violate determinism.
2. **Promotion:** If a blue state is found to be incompatible with all existing red states in *RED*, it is “promoted” to the red set ($RED \leftarrow RED \cup \{b\}$). This promotion signifies that the state encapsulates a fundamentally novel behaviour that cannot be merged with the current hypothesis cluster.
3. **Expansion:** Once a target blue state is either successfully merged or promoted, the search fringe is updated. If the state is promoted to red, its immediate successors (previously classified as white) are coloured blue ($BLUE \leftarrow BLUE \cup \text{successor}(b)$), effectively advancing the frontier deeper into the graph.

This systematic frontier expansion continues recursively until the blue state set is exhausted ($BLUE = \emptyset$). As demonstrated by Lang *et al.* [31], the Blue-Fringe strategy guarantees that the learning algorithm evaluates the most “evidence-rich” regions of the automaton first, typically yielding a more stable and structurally accurate model than naive merging approaches.

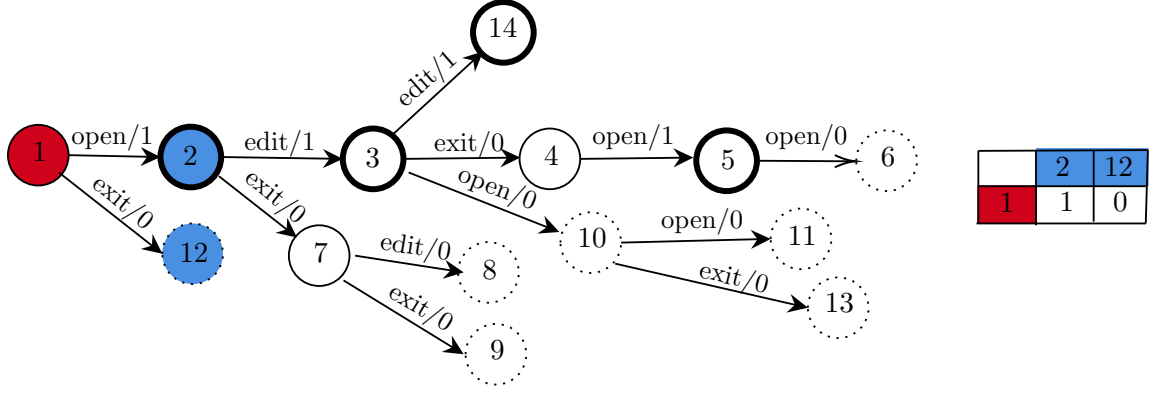


Figure 2.10: EDSM Example: PTA - States with the same border line correspond to the same state in the target model (same PTA as in Figure 2.4).

Example

This example demonstrates the execution workflow of the EDSM algorithm utilising the Blue-fringe framework. Starting with the baseline PTA (Figure 2.10), the root state (state 1) is coloured red under the assumption that it forms the initial kernel of the target hypothesis, while its immediate successors (states 2 and 12) are designated blue. The algorithm evaluates the candidate pairs, yielding scores of $S_{\text{EDSM}}(1, 2) = 1$ and $S_{\text{EDSM}}(1, 12) = 0$. Consequently, the highest-scoring merge is executed between states 1 and 2. As illustrated in Figure 2.11, this operation cascades to combine states 7 and 12 to resolve local non-determinism. Next, the algorithm evaluates the remaining frontier, computing a score of $S_{\text{EDSM}}(1, 3) = -1$. Because state 3 is incompatible with the lone existing red state, it is promoted to red, and its immediate successors (states 14, 4, and 10) enter the blue frontier.

The subsequent iteration involves evaluating all pairwise combinations between the red set $R = \{1, 3\}$ and the blue fringe $B = \{7, 14, 4, 10\}$. State 7 induces a determinism conflict with both existing red states. Specifically, merging state 7 with state 1 violates the output uniqueness constraints yielding two transitions $(1, \text{edit}, 1, 3), (1, \text{edit}, 0, 8) \in \mathcal{T}$. The same applied when merging merging state 7 with state 3 which yield two transitions $(3, \text{edit}, 1, 4), (3, \text{edit}, 0, 8) \in \mathcal{T}$.

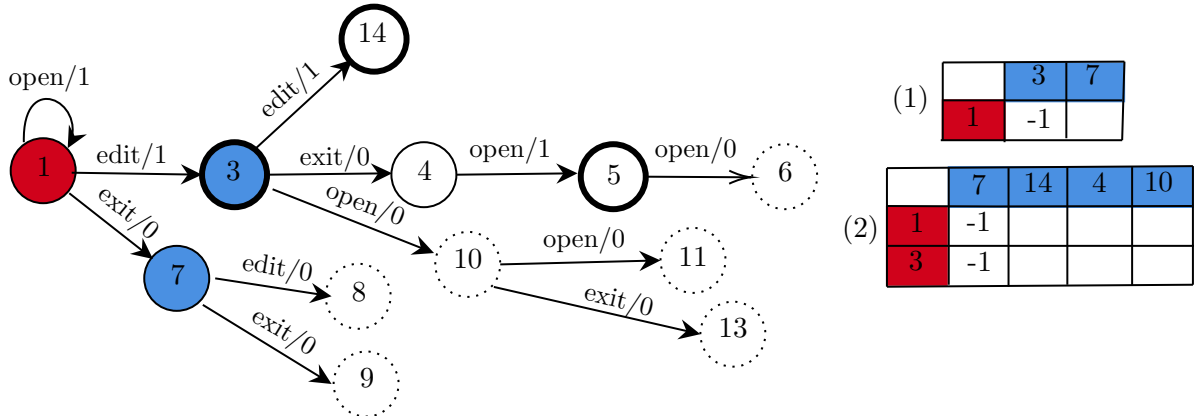


Figure 2.11: EDSM Example: Merge states 1 and 2 - Tables are numbered according to their implementation order. Blank cells in the tables indicate that the blue state was not evaluated against the red states because an earlier blue state was promoted to red.

Consequently, state 7 is promoted to red, and its successors (states 8 and 9) are coloured blue. The next element evaluated in the blue frontier is the leaf state 14. Crucially, the score of any candidate pair containing a leaf node is bound to 0. As established previously, our strict evidence-driven implementation rejects merge operations where $S_{\text{EDSM}} = 0$; hence, state 14 is promoted to red. State 4 is evaluated next; it violates determinism constraints when paired with states 1 and 3,

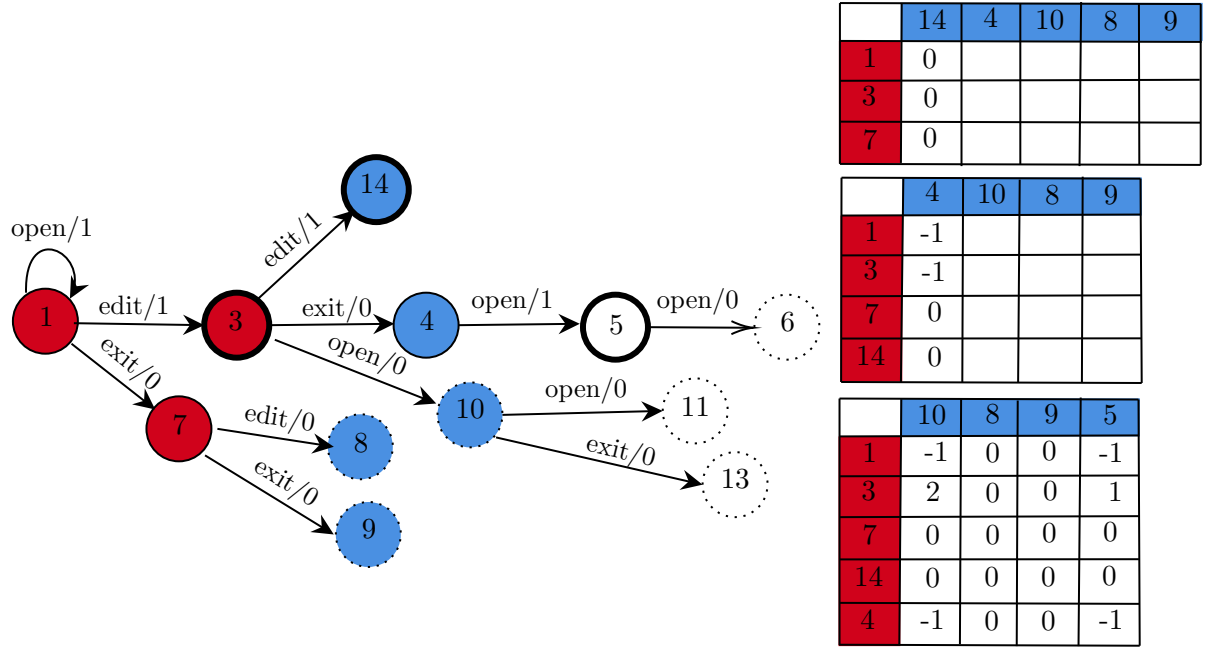
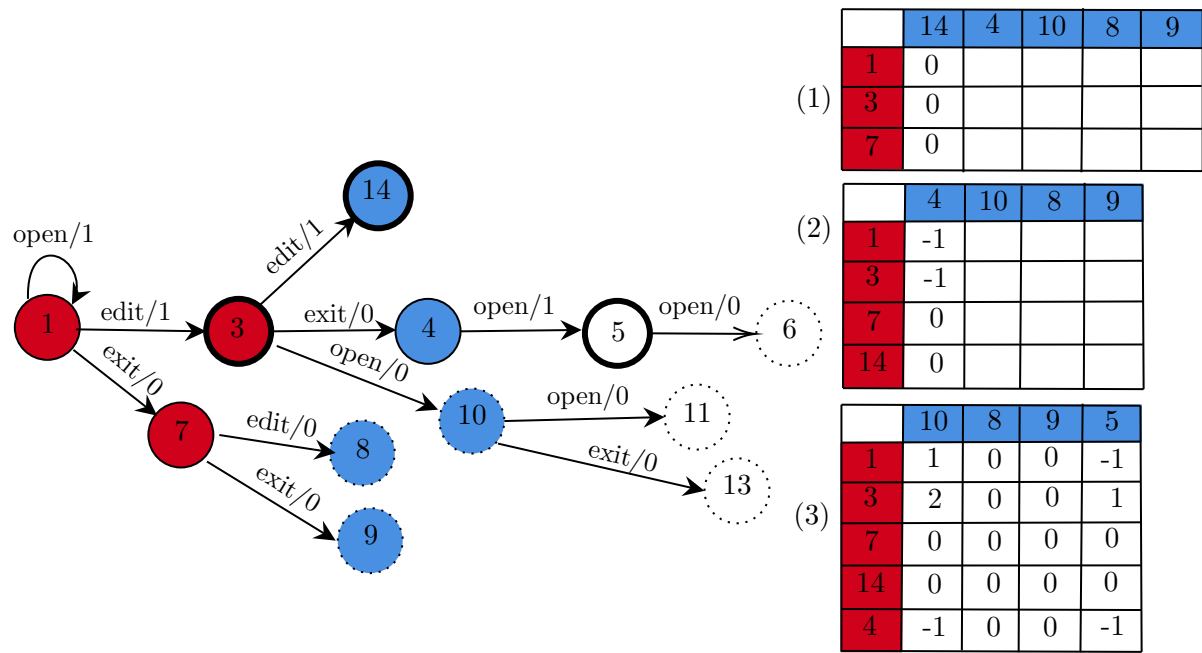


Figure 2.12: EDSM Example: More Comparisons

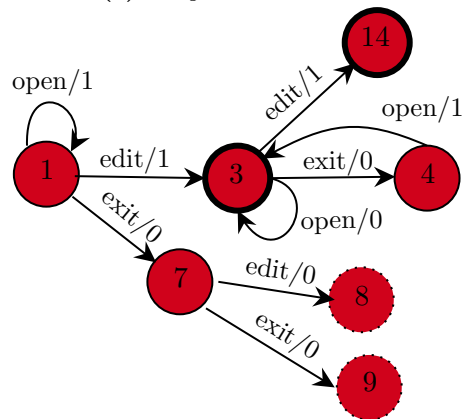
and yields a score of 0 with states 7 and 14. As a result, state 4 is also promoted to red, pushing its successor, state 5, onto the blue fringe.

The next iteration scales to encompass five red states and four blue states, producing twenty candidate pairs (Figure 2.12). Among these, the pair (3, 10) maximizes the evaluation heuristic with a peak score of $S_{\text{EDSM}}(3, 10) = 2$. Executing this merge triggers a cascade of non-determinism resolution, forcing state 11 to fold into state 3 and state 13 to fold into state 4.

Figure 2.13a illustrates the structural topology of the automaton following this consolidation. In the subsequent steps, the leaf states 8 and 9 are sequentially promoted to red. Finally, a merge is executed between states 3 and 5 because $S_{\text{EDSM}}(3, 5) = 1$ is the highest score among all remaining candidate pairs, as shown in Figure 2.13a. This iterative process culminates in the final inferred model shown in Figure 2.13b.



(a) Merge state 3 and 10



(b) Merge state 3 and 5 - Final Model

Figure 2.13: EDSM Example: more merges and the final model

Algorithm 1 shows the implementation of the EDSM with Blue-Fringe optimisation. Note that we made a small modification to the original EDSM algorithm: if the first blue state in the *BLUE_States* set is inconsistent with all red states, it will be prompted to red. However, if the first blue has at least a $S_{EDSM} > 0$ with any red, the algorithm will evaluate all red-blue pairs and pick the highest score (Line 15). In the original algorithm, if a blue is not consistent with all reds, it prompts for a red directly, and the process restarts.

We are prioritising merging states and reducing the state space. When a valid merge, with a score of 1 or higher, is found between the first blue state of the BLUE set and any of the red states, we explore the state space for a pair with a higher score. Our intention is to search for the pair with the highest score as early as possible. If a following pair in the set is marked as inconsistent, we continue to evaluate other pairs as there is at least one valid merge that can be performed. This strategy allows us to compare multiple pairs and then pick the one with the highest score.

Algorithm 1: Evidence-Driven State Merging (EDSM)

Input: A set of traces T
Output: A Mealy machine $\mathcal{M}$ consistent with T

```

1  $\mathcal{M} \leftarrow \text{Build\_PTA}(T)$  // Construct the Prefix Tree Acceptor
2  $RED\_States \leftarrow \{q_0\}$  // colour the initial state red
3  $BLUE\_States \leftarrow \text{Get\_Successor}(q_0)$  // Colour successors blue
4 while  $BLUE\_States \neq \emptyset$  do
5    $Best\_Pair \leftarrow \emptyset$ 
6    $Highest\_Score \leftarrow 0$ 
7   foreach  $b \in BLUE\_States$  do
8      $Not\_consistent\_with\_any\_red = \text{True}$ 
9     foreach  $r \in RED\_States$  do
10      if  $Is\_Consistent(\mathcal{M}, r, b)$  then
11         $Not\_consistent\_with\_any\_red = \text{False}$ 
12        if  $Merging\_Score(\mathcal{M}, r, b) > Highest\_Score$  then
13           $Highest\_Score \leftarrow Merging\_Score(\mathcal{M}, r, b)$ 
14           $Best\_Pair \leftarrow (r, b)$ 
15      if  $Not\_consistent\_with\_any\_red \wedge Best\_Pair = \emptyset$  then
16         $RED\_States \leftarrow RED\_States \cup \{b\}$  // Make b red
17        // Make (b)'s successors blue
18         $BLUE\_States \leftarrow BLUE\_States \setminus \{b\} \cup \text{Get\_Successor}(b)$ 
19        break and go to line 4
20   if  $Best\_Pair \neq \emptyset$  then
21      $\mathcal{M} \leftarrow \text{Merge}(\mathcal{M}, Best\_Pair.r, Best\_Pair.b)$ 
22     // Update the Blue_States set
23      $BLUE\_States \leftarrow \{b \in \text{Get\_Successor}(r) \mid r \in RED\_States\} \setminus RED\_States$ 
24 return  $\mathcal{M}$ 

```

Algorithm 1 starts by constructing a PTA from the training traces (line 1). Next, it marks the root of the PTA (q_0) as red (line 2) and colours its successors blue. The function $\text{Get_Successor}(q_0)$ in line 3 takes a state and returns a list of its children (direct successors). Every blue state in BLUE is evaluated against every red state in RED. The function $Is_Consistent(\mathcal{M}, r, b)$ takes the current version of the model $\mathcal{M}$, the current red state r , and the current blue state b and tries to merge the two states. The function returns true if the merge is valid and there is no output conflict between r and b nor between any two states that will be merged during solving the non-determinism. The function returns false otherwise. If it returns true, the S_{EDSM} score is calculated as explained earlier in this section. The $Best_Pair$ variable saves the pair that has the highest score for the current iteration.

Once a blue state is evaluated against all red states, the algorithm checks if the current blue state can be merged to any red state. If the current blue state is inconsistent with all red states

and there is no previous blue state that can be merged with any red state ($Best_Pair = \emptyset$), the algorithm prompts the current blue state to red and makes its successors blue. Then, it repeats the process.

If the current blue state is consistent with any red state, the algorithm will explore the state space for a better option (a pair with a higher score). Once all the current state space (all possible red-blue pairs) are evaluated (line 19), the algorithm picks the one with the highest score and performs the merge.

The primary limitation of EDSM is its susceptibility to errors, as a high merging score does not strictly guarantee the correctness of a merge. Although EDSM infers a model that is consistent with the training traces set, it fails to capture the correct structure or language of the target model when data are sparse. Its greedy heuristic becomes fragile when the training data is sparse or when the execution traces are long yet uninformative. For example, in the previous example of Figure 2.10, the very first merge was between state 1 and state 2 because this pair has the highest score ($S_{EDSM}(1,2) = 1$) among the current state space. Although the states are compatible and the resulting model after the merge is consistent with the traces, it is an incorrect merge because states 1 and 2 refer to different states with respect to the target model.

Every merge step effectively partitions the remaining training sequences among a smaller set of merged states. In data-sparse environments, this overlapping evidence becomes too thin to reliably distinguish between genuinely valid merges and purely coincidental, localised similarities [27]. In extreme scenarios characterised by large alphabet sizes and low system observation rates, EDSM’s test accuracy can degrade significantly [27] (from 99% in smaller scenarios), dropping to near-random levels of approximately 52% [27]. Furthermore, EDSM is highly sensitive to its early merging decisions. Because it lacks a backtracking mechanism, a single incorrect merge committed early on due to data sparsity can permanently distort the hypothesis topology, making it mathematically impossible to converge on the true target model [26].

While longer execution traces intuitively seem to offer more behavioural insight, they present distinct structural challenges for EDSM if they lack state or transition coverage. Systems with large alphabets naturally generate longer sequence paths. However, if the number of distinct traces does not scale proportionally, the dataset remains sparse relative to the state-space complexity of the underlying target model. This imbalance ultimately results in poor generalisation and weak test performance. Additionally, long traces that exclusively traverse a small, localised portion of the state space (such as repeating the same loops) fail to provide the necessary structural evidence for states deeper within the prefix tree. This induces a severe reduction in identification capability, leaving the EDSM heuristic too weak to confidently execute valid state merges [27].

2.2.4 SAT Solver in Passive Inference

Over the past decade, SAT solvers have undergone a significant evolution in computational efficiency, transitioning from theoretical constructs to high-performance reasoning engines. By leveraging the robust heuristics of modern SAT architectures, diverse combinatorial challenges can be addressed by systematically encoding domain-specific constraints into propositional formulas [27]. This approach has demonstrated remarkable performance across various domains, such as model checking [7] and automata learning [27].

The SAT solver takes a Boolean expression as input and returns a truth value for each variable, which makes the whole expression true. The input expression must be in the form of Conjunctive Normal Form (CNF), which is a conjunction ($\wedge$) of multiple clauses; each clause is a disjunction ($\vee$) of literals. Given the following CNF formula:

$$(a \vee \neg b) \wedge (b \vee c) \wedge (a \vee b \vee \neg c)$$

The SAT solver returns a satisfying truth assignment, for example, $a = \text{True}$, $b = \text{False}$, and $c = \text{True}$. Note that while a SAT solver returns a valid assignment, this solution is not necessarily unique.

The Boolean Satisfiability Problem (SAT) is efficiently utilised for automata identification [60, 27, 46]. The first integration of the SAT solver into automata learning was by Heule *et al.* [27]. To utilise a SAT solver for learning an automaton, the process involves three primary steps: constructing a consistency graph from the PTA, translating that graph into a CNF formula, and finally solving the formula to find a valid state mapping.

Graph Colouring

The construction of a consistency graph relies on graph colouring techniques. The objective is to produce a graph that explicitly defines whether each pair of states is consistent. The process begins by creating a graph in which every state in the PTA is represented as a node, with a distinct colour initially assigned to each. Two nodes are connected by an edge if the corresponding states are inconsistent, meaning they cannot be merged. Figure 2.14 illustrates the transformation of a PTA into a consistency graph. For instance, State 0 cannot be merged with State 2 because it will raise a nondeterminism due to the output conflict of input *open*. The same for merging states 0 and 3. Consequently, state 0 is linked to both states 2 and 3 in the consistency graph, Figure 2.14b.

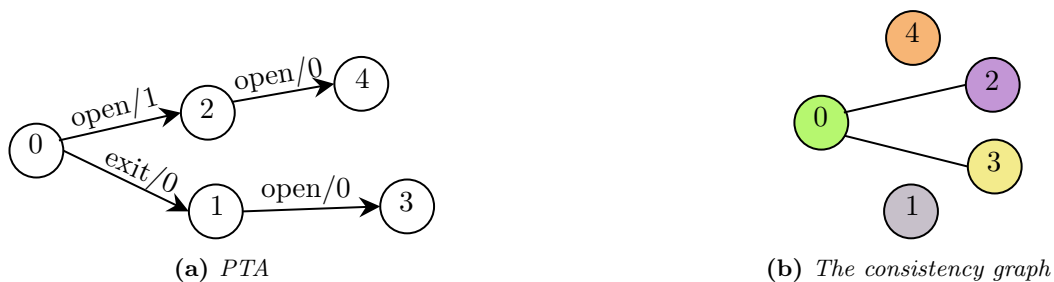


Figure 2.14: Transform a PTA into a consistency graph.

Once the consistency graph is constructed, it can be translated into a CNF formula. A widely adopted method for translating graph colouring problems into a format compatible with SAT solvers is known as “direct encoding” [27, 55, 46]. This approach defines specific clauses for the CNF formula to ensure a logically precise translation of the graph constraints. The resulting formula must ensure, for example, that each state is assigned exactly one colour.

Learning automata with SAT is an exact algorithm that identifies a minimal deterministic automaton (the smallest possible model consistent with the data). This pursuit of minimality is rooted in the principle of Occam’s Razor, which posits that the simplest model (the one with the fewest states) is most likely to be the correct generalisation [60]. To find this minimum, the algorithm first identifies the smallest subset of states “clique” in the consistency graph such that every state is inconsistent with every other state in the set. This clique provides a lower bound for the number of colours, $|C| = k$. The SAT solver looks for a solution with the k colours. The algorithm initially assumes that all remaining states will eventually be merged into one of these initial “seed” states. The process follows an iterative search:

1. A CNF formula is generated to encode the PTA constraints based on the current number of colours, k .
2. The SAT solver attempts to find a satisfying assignment.

3. If a solution is found, the minimal automaton has been identified.
4. If the formula is unsatisfiable, k is incremented, a new formula is generated, and the process repeats until a solution is reached [27].

CNF Formula Construction

This method utilises Boolean auxiliary variables to represent the properties and structure of the potential automaton [27, 46]. Two primary types of auxiliary variables are required:

1. Colouring variables: These represent the assignment of a specific colour (representing a state in the target model) to each state in the PTA.
2. Parenting (Transition) variables: These identify the transitions and relationships between the coloured states.

Colour Variables

Given a consistency graph $G = (Q, E)$ and a set of colours $C = \{1, \dots, k\}$, direct encoding defines the colouring variable $x_{q,c}$, where $q \in Q$ and $c \in C$. The variable $x_{q,c}$ is true if, and only if, the vertex q is assigned the colour c .

Parent Relation Variables

For Mealy machines (Definition 1), Ulyantsev and Tsarev [46] defined the parenting variable as $y_{a,b,i,j}$, where $a \in I$, $b \in O$, and $i, j \in C$. In this context, $y_{a,b,i,j}$ is true if there is an edge from the parent state of colour i to the child state of colour j that produces output b upon receiving input a .

The logical restrictions and clauses governing these variables (such as ensuring determinism and consistency with the original traces) are detailed in Table 2.1. For example, the constraint which ensures each state is assigned exactly one colour is enforced through two distinct sets of clauses:

- At-Least-One colour: This ensures that every state is assigned at least one colour from the available k colours.
- At-Most-One colour: This ensures that no state is assigned more than one colour simultaneously. This is typically achieved by adding clauses that forbid assigning any pair of colours to the same state.

A full transforming example from PTA to CNF can be found in Appendix B.

Symmetry Breaking

In model inference, symmetry breaking is used to eliminate redundant search paths by preventing the SAT solver from exploring multiple versions of the same model that differ only in state labelling. Since a state machine with k states can be represented in $k!$ (k -factorial) identical ways through simple relabelling, a standard solver would waste significant time proving each permutation invalid [27]. By adding Symmetry Breaking Predicates (SBPs), researchers force the solver to only consider a single “canonical” version of each model, effectively collapsing the search space.

The most common implementation involves enforcing a Breadth-First Search (BFS) ordering on the states [45]. This is done by adding constraints that require state $i + 1$ to be reached only after state i has been explored. This structural discipline prevents the solver from getting trapped in symmetric “dead ends”, leading to much faster convergence when searching for the minimal automaton that fits the provided observations.

Table 2.1: *Encoding Mealy machines identification into SAT*

Clauses	Range	Meaning
$(x_{p,1} \vee x_{p,2} \vee \dots \vee x_{p,k})$	$p \in Q$	Every state has at least one colour
$(\neg x_{p,i} \vee \neg x_{p,j})$	$p \in Q; i, j \in C; i < j$	Every state has at most one colour
$(y_{a,b,i,j} \vee \neg x_{p,i} \vee \neg x_{q,j})$	$p, q \in Q; i, j \in C;$ p has a transition a/b to q	A parent relation is set when a state and its parent are coloured
$(\neg y_{a,b,i,j} \vee \neg x_{p,i} \vee x_{q,j})$	$p, q \in Q; i, j \in C;$ p has a transition a/b to q	A parent relation forces a state once the parent is coloured
$(y_{a,b,i,1} \vee y_{a,b,i,2} \vee \dots \vee y_{a,b,i,k})$	$a \in I; b \in O; i \in C$	Each parent relation must target at least one colour
$(\neg y_{a,b,i,h} \vee \neg y_{a,b,i,j})$	$a \in I; b \in O;$ $i, j, h \in C; h < j$	Each parent relation can target at most one colour
$(\neg x_{p,i} \vee \neg x_{q,i})$	$i \in C; p, q \in E$	All determinisation conflicts explicitly added as clauses
C = set of colours, I = set of input, O = set of output, Q = set of states, E = set of conflict edges		

DFASAT

Despite the high efficiency of modern translation methods, the resulting CNF formulas can become prohibitively large for contemporary SAT solvers. For instance, certain highly challenging problems in the Abbadingo competition dataset [31] require hundreds of colours. Such cases can generate nearly 100,000,000 clauses. A figure that far exceeds the practical limits of state-of-the-art solvers (see [27]), which typically perform reliably only up to approximately 5,000,000 clauses.

A significant advantage of direct encoding is its flexibility: it can be applied to a partially identified (partially merged) Deterministic Finite Automaton (DFA) rather than only to a raw PTA. Recognising this, Heule *et al.* [27] proposed a hybrid methodology, called **DFASAT**. This approach involves pre-processing the data using several stages of EDSM (Section 2.2.3) to reduce the initial state space, and then encoding the remaining problem as a SAT formula. This combination leverages the speed of heuristic state merging to bring the problem within the size constraints of exact SAT solvers. DFASAT [27] switches from the EDSM phase to the SAT solver when the number of states of the inferred model is reduced to a 1000 states. Crucially, Heule *et al.* [27] provided empirical evidence demonstrating that a threshold of 1000 states yields an optimal trade-off between greedy heuristic execution speed and total solver capacity.

Although EDSM (Algorithm 1) is considered the baseline in state merging, it is susceptible to errors. A high merging score, which reflects a high degree of similarity between states, does not strictly guarantee the correctness of a merge with respect to the target model. In a massive PTA, two states may appear similar but actually correspond to two distinct states in the target system. For example, in Figure 2.10 (Section 2.2.3), EDSM performed the very first merge between state 1 and state 2 because this pair has the highest score ($S_{EDSM}(1, 2) = 1$) compared to the other pair ($S_{EDSM}(1, 12) = 0$).

Furthermore, a critical limitation of the EDSM is that it does not support backtracking; once a merge is performed, it cannot be undone. In the context of DFASAT, if these initial EDSM merges are incorrect, the SAT solver’s efficiency is compromised. The solver is forced to attempt to “fold” the remaining states into an incorrect partial automaton, often leading to a dead end or an inaccurate model.

To mitigate this, Heule *et al.* [27] introduced a more restrictive heuristic to ensure that EDSM

utilises more robust evidence before committing to a merge. This heuristic blocks merges that introduce “new information” (transitions) to the existing red states. Specifically, consider a red state r and a blue state b ($r, b \in Q$). If r and b are compatible, but b possesses a transition label that r does not, the merge is deemed invalid and assigned a score of $S_{DFASAT}(r, b) = -1$. This constraint only considers the red and blue states, but is not applied to any merge used to resolve the nondeterminism. A valid merge in DFASAT is when $S_{DFASAT}(r, b) \geq 0$. By preventing the blue state from adding new labels to the red state during the heuristic phase, the algorithm ensures that the initial partial model remains a conservative and highly reliable foundation for the subsequent SAT-based search. The algorithm of DFASAT follows these steps:

1. Construct a PTA from traces.
2. Perform some merges with EDSM using the new heuristic until the PTA is reduced to a graph of at most 1000 states.
3. Construct a CNF formula for the partial model.
4. Pass the CNF to SAT.
5. If the SAT solver found a solution, construct a model from the SAT result.
6. If a solution is not found, increase k and iterate until a solution is found.

Example

In this example, we demonstrate the workflow of integrating the DFASAT heuristic in the EDSM algorithm. Consider the baseline PTA (Figure 2.15). DFASAT evaluates the root state (state 1) against state 2 and computes a score of $S_{DFASAT}(1, 2) = -1$, as collapsing this pair would introduce a new “edit” label to a transition originating from red state 1. DFASAT merges pairs with zero score, but in our implementation, we restrict to positive scores only (as illustrated in this example). State 12 is subsequently promoted to red due to a lack of shared evidence with the existing kernel.

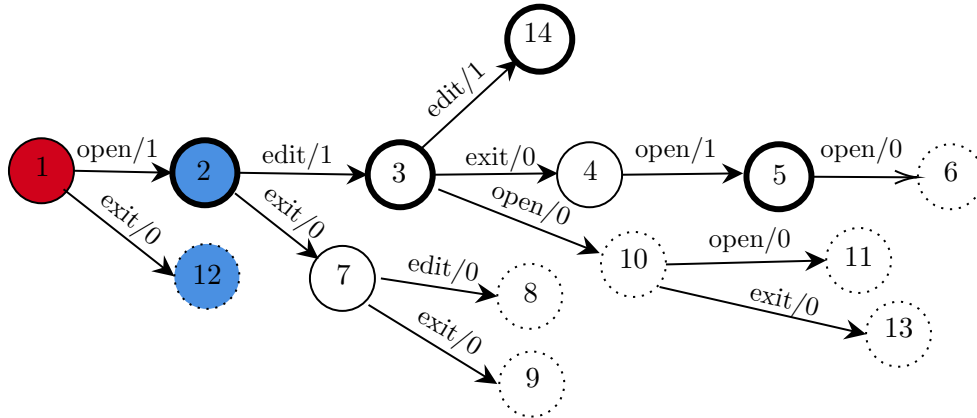


Figure 2.15: PTA (same PAT as in Figure 2.4)

In subsequent iterations, as comprehensively mapped in Figure 2.16, the algorithm explores the frontier of the optimal state pair. If a target blue state proves incompatible with all existing red states under the blocking heuristic, it is automatically promoted to red. Figure 2.16 outlines each evaluation step in a tabular format, indexed by the iteration number on the left. Notably, at iteration 6, the algorithm identifies its first viable merge candidate, yielding $S_{\text{DFASAT}}(1,4) = 1$. DFASAT exhaustively evaluates all remaining active pairs in this iteration to select the pair with the highest score, which emerges as $(3,10)$ with the best score of $S_{\text{DFASAT}}(3,10) = 2$.

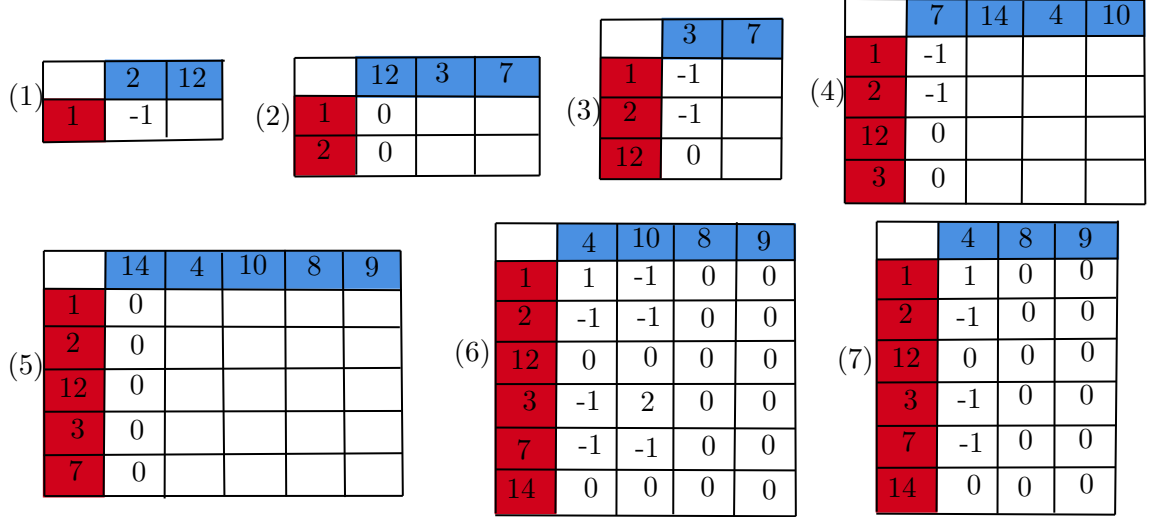


Figure 2.16: *Finding the best pair*

Figure 2.17 illustrates the structure of the automaton following this first merge between states 3 and 10. In the following iteration (Iteration 7), DFASAT executes its second merge, selecting the pair $(1,4)$ with a score of $S_{\text{DFASAT}}(1,4) = 1$. During the final three iterations, states 8, 9, and 6 are sequentially promoted to red, one at a time, due to an information deficit that prevents valid unification. This iterative process concludes with the final model shown in Figure 2.18.

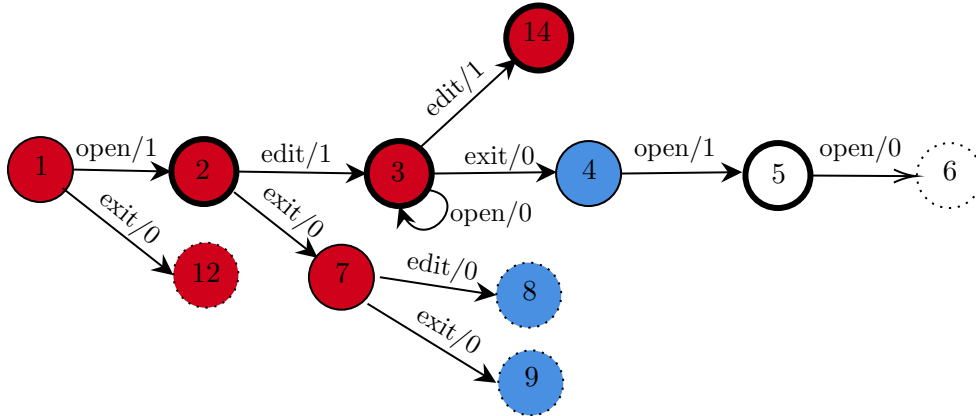


Figure 2.17: *Merge states 3 and 10*

Figure 2.19 illustrates the inferred model using the standard EDSM (Section 2.2.3). In the following, we compare the inferred model by the DFASAT heuristic (Figure 2.18) to the one inferred by the standard EDSM (Figure 2.19), which are both inferred using the same set of training traces. The final model inferred by the DFASAT has 9 states, while the final model inferred by EDSM has 7 states. While EDSM inferred a smaller model, it made more mistakes (merged states that do not refer to the same state in the target model). EDSM performed an incorrect merge between states 1 and 2, as explained earlier (in Section 2.2.3). This merge affects the following state selections.

On the other hand, DFASAT postponed the merging decision until it found stronger evidence.

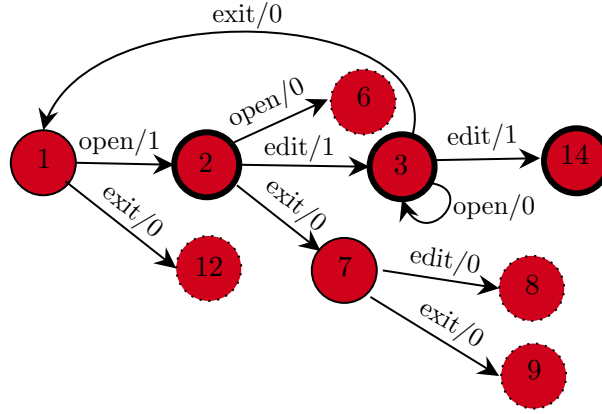


Figure 2.18: *Final Model inferred by DFASAT*

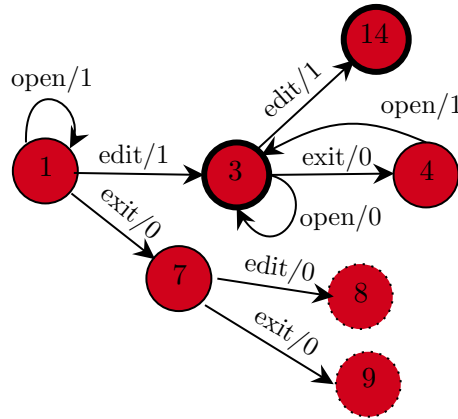


Figure 2.19: *The inferred model by standard EDSM - copy of Figure 2.13b*

It rejected any merge that introduced a new input label to the red state. This strategy avoided merging state 1 to state 2, but also bypassed merging state 2 to state 3. Prioritising not merging increases the red state cluster, which means increasing the number of red-blue pairs. Moreover, this rigid heuristic introduces a risk of bypassing correct merges (states refer to the same state in the target model), which affects the following merges.

Algorithm 2 illustrates the DFASAT algorithm. Within this algorithm, Lines 2 to 21 encapsulate the standard EDSM procedure augmented by the specialised DFASAT blocking heuristic, while Lines 22 to 30 define the subsequent SAT solver integration layer. The functions `Get_Successor`, `Is_Consistent`, and `Merging_Score` are explained earlier in Section 2.2.3. The function `Add_New_Information` takes a pair of states `red` and `blue` and returns `true` if the `blue` state has an outgoing transition with an input that is new to the `red` state; it returns `false` otherwise.

Algorithm 2: DFASAT

```

Input: A set of traces  $T$ 
Output: A Mealy machine  $\mathcal{M}$  consistent with  $T$ 
1  $\mathcal{M} \leftarrow \text{Build\_PTA}(T)$  // Construct the Prefix Tree Acceptor
2  $RED\_States \leftarrow \{q_0\}$  // colour the initial state red
3  $BLUE\_States \leftarrow \text{Get\_Successor}(q_0)$  // Colour successors blue
4 while  $BLUE\_States \neq \emptyset \vee |\mathcal{M}| \geq 1000$  do
5    $Best\_Pair \leftarrow \emptyset$ 
6    $Highest\_Score \leftarrow 0$ 
7   foreach  $b \in BLUE\_States$  do
8      $Not\_consistent\_with\_any\_red = \text{True}$ 
9     foreach  $r \in RED\_States$  do
10      if  $Is\_Consistent(\mathcal{M}, r, b) \wedge \neg \text{Add\_New\_Information}(r, b)$  then
11         $Not\_consistent\_with\_any\_red = \text{False}$ 
12        if  $Merging\_Score(\mathcal{M}, r, b) > Highest\_Score$  then
13           $Highest\_Score \leftarrow Merging\_Score(\mathcal{M}, r, b)$ 
14           $Best\_Pair \leftarrow (r, b)$ 
15      if  $Not\_consistent\_with\_any\_red \wedge Best\_Pair = \emptyset$  then
16         $RED\_States \leftarrow RED\_States \cup \{b\}$  // Make b red
17        // Make (b)'s successors blue
18         $BLUE\_States \leftarrow BLUE\_States \setminus \{b\} \cup \text{Get\_Successor}(b)$ 
19        break and go to line 4
20   if  $Best\_Pair \neq \emptyset$  then
21      $\mathcal{M} \leftarrow \text{Merge}(\mathcal{M}, Best\_Pair.r, Best\_Pair.b)$ 
22     // Update the Blue_States set
23      $BLUE\_States \leftarrow \{b \in \text{Get\_Successor}(r) \mid r \in RED\_States\} \setminus RED\_States$ 
24  $Colours \leftarrow |RED\_States|$  // Define the clique size
25 while  $True$  do
26    $Variables \leftarrow \text{Define\_Variables}(Colours, \mathcal{M})$ 
27    $CNF \leftarrow \text{Build\_CNF}(Variables, \mathcal{M})$ 
28    $SAT\_Result \leftarrow \text{Pass\_to\_SAT}(CNF)$  // Send the CNF to the solver
29   if  $SAT\_Result = SAT$  then
30      $\mathcal{M} \leftarrow \text{ReConstructModel}(Variables, SAT\_Result)$ 
31     break
32    $Colours \leftarrow Colours + 1$  // If UNSAT, add a colour & try again
33 return  $\mathcal{M}$ 

```

The primary differences between classical EDSM (Algorithm 1) and the modified variant adapted for DFASAT are located at Lines 4 and 10 (with specific structural modifications highlighted in yellow). Line 4 in the original algorithm ensures that merging continues exhaustively until no further valid modifications can be executed. Conversely, the DFASAT variant halts the merging phase immediately once the size of the collapsed automaton reaches a threshold of 1000 states or fewer. Furthermore, Line 10 tightens the validity criteria for candidate state pairs. While baseline EDSM deems a merge valid unless it triggers an explicit conflict, DFASAT enforces a significantly more restrictive, conservative constraint: a candidate merge is structurally disallowed if it would introduce a novel label to a pre-existing red state.

Following this heuristic reduction phase, the integration of the SAT solver (Line 22) operates

under the assumption that the minimal bound for the final target automaton corresponds to the cardinality of the red state set ($|RED_States|$). Note that the red states are not necessarily incompatible; DFASAT prompts states to red when its heuristic is violated. However, it considers red states to be unmergeable and forms a clique of these verified, distinct states. They also use this as a symmetry-breaking predicate by assigning a distinct colour for each red state [27].

Initialising the SAT solver with at least $|RED_States|$ colours introduces a trade-off, as it risks omitting the smallest possible model consistent with the training dataset. Although the primary objective of passive model inference is to identify the most minimal state machine that describes the observed data, DFASAT sacrifices state minimality to optimise computational performance. Specifically, DFASAT reduces SAT solver searching time by setting its lower bound to a state threshold below which no valid model is expected to exist. Their empirical results [27] demonstrate that enforcing this colour-bounding heuristic yields significant performance gains compared to the baseline EDSM algorithm.

At Line 24, the algorithm instantiates the necessary Boolean variables, constructs the corresponding CNF formula mapping the structural constraints (Line 25), and passes the resulting encoding to the SAT solver (Line 26). The solver evaluates the instance and returns one of two outcomes:

- *SAT*: The solver identifies a satisfying truth assignment for all variables that renders the complete CNF formula true, indicating that a valid, consistent model exists at the current state bound.
- *UNSAT*: The solver exhaustively traverses the search space and determines that no truth assignment can satisfy the constraints, proving that no valid model can be realised within the specified size bounds.

If the SAT solver yields a satisfiable result, the algorithm decodes the underlying truth assignments to reconstruct and output the final minimal model (Line 28). Conversely, if the instance is deemed unsatisfiable, it implies that an automaton of the current size cannot soundly represent the training traces under the required consistency invariants. Under these conditions, the algorithm increments the targeted state bound. This effectively expands the structural search space of the hypothesis (Line 30) and dynamically regenerates the CNF formula. This iterative search loop repeats until the solver converges on a satisfiable assignment, guaranteeing the discovery of the minimal possible model $\mathcal{M}$ (such that $|Q_{\mathcal{M}}| \geq |RED_States|$) that remains perfectly consistent with the provided trace data.

While DFASAT reduces the risk of making early, irreversible bad merges, it makes mistakes. Its rigid approach often causes it to reject valid state pairs, leading to poor decisions later on. For example, in the previous scenario, DFASAT’s very first choice is incorrect: it executes the merge $S_{DFASAT}(3, 10) = 2$. Merging states 3 and 10 does not violate the DFASAT heuristic, but the states refer to two different states with respect to the target model. DFASAT blocked valid merges in the early iterations, and the following pairs will differ if they are performed. It completely missed immediate opportunities to restore the correct structure, leading to incorrect later decisions.

The main issue is that DFASAT uses a local blocking heuristic, looking only at a state’s direct outgoing transitions. When data is sparse and the input alphabet I is large, states rarely share the same outgoing labels, causing DFASAT to mistake valid merges for errors. Although the SAT solver should theoretically fix these issues later, these early mistakes cause a domino effect of bad merges and expand the search space, making the process much slower. Furthermore, while DFASAT stops new labels from being added to the two states being merged, it loosely allows them to be added to downstream states during non-determinism resolution (like in the last merge of the previous

example). This highlights the need for a more balanced approach, one that avoids both the loose rule of EDSM and the overly strict one of DFASAT.

In the following section, we explore the utilisation of patterns in passive learning. Patterns are another approach to approximating the system behaviour, which can be used directly or as an additional source of information to improve other algorithms and the inference process.

2.2.5 Patterns in Passive Inference

We can describe software automata as a collection of “mini-behaviours”, where each behaviour represents a specific feature or sub-feature of the target system. In everyday terms, software is often viewed as a set of rules (generic or specific) that dictate what a system should and should not do. For example, a text editor might follow these rules:

- A file can only be saved if it contains unsaved changes.
- Once a file is closed, the system prohibits all other operations until the file is reopened.

This type of information is highly valuable and reveals significant details about the target system. In the context of automata learning, we refer to these behaviours as patterns. Patterns provide essential guidance to the learner, particularly in passive learning scenarios where the learner relies exclusively on static data such as traces, logs, or test cases. By incorporating these patterns, the learner gains additional depth and perspectives that the raw training data might not explicitly contain. This allows the inference process to “fill in the gaps” of the system behaviour that are missing from the recorded traces. Ultimately, system models consist of these mini-patterns. Whether researchers define them manually or extract them from data, learners use them in the inference process. Furthermore, the learner can combine multiple patterns to construct the final target model.

A significant challenge in automata learning is the need for a vast number of traces to reach a reliable target automaton, particularly for complex systems [52]. In practice, deducing underlying system behaviour often requires extensive data. Patterns address this by capturing permitted and prohibited behaviours without the need for exhaustive trace sets. Properties generally fall into two categories:

- Safety properties: These specify that a certain “bad” state must never occur. For example, a transition labelled “close” should never be followed by a transition labelled “save”.
- Liveness properties: These specify that a certain “good” state must always eventually be reachable. For example, after a transition labelled “open”, a transition labelled “close” will eventually be found.

This distinction is vital because the type of property determines which model-checking technique is required for verification [52]. Researchers use safety properties to block incorrect merges [36] or to generate additional negative traces [52]. In contrast, liveness properties assist in building models by finding the combination of all required system behaviours [44, 43, 23].

Patterns originate from two primary sources: they are either defined by the user or mined from existing data. User-defined patterns are inherently more reliable and provide a direct, authoritative guide for the learning process. However, in practice, these patterns are often unavailable or difficult to obtain, as formalising expert knowledge requires significant time and effort. Conversely,

mined patterns are more readily accessible but are generally less reliable, as their accuracy depends entirely on the quality and completeness of the training data. To mitigate this, specific validation techniques, such as confidence thresholds [36] or statistical filtering, must be applied to ensure that mined patterns represent true system invariants rather than incidental noise. Therefore, in black-box learning scenarios where the learner only has access to execution traces, mining remains the most effective approach, provided it is supported by robust reliability checks.

In the following, we examine the broader landscape of how other researchers have integrated patterns into the model inference process.

Patterns to Block Merges

Lo *et al.* [36] integrated temporal patterns with state merging to improve the accuracy of automata deduced from execution traces. Their framework begins by analysing traces to derive statistically significant temporal features that capture relationships between discontinuous, potentially distant events. These mined patterns subsequently guide the K -tail [10] state-merging algorithm: two states are deemed mergeable if they share identical future event sequences and do not violate any established pattern, thereby reducing the likelihood of over-generalisation. Within their framework, a pattern is only mined if it satisfies a 100% confidence threshold, meaning the matching post-condition occurs every time the corresponding pre-condition is met [36]. While our proposed methodology shares a similar structural pipeline, it diverges from their work in three fundamental ways: Lo *et al.* extract complex, long-sequence temporal properties.

Patterns Composition:

Decomposing a system into multiple components (such as classes, modules, and services) that represent specific liveness properties ensures high modularity and a clear separation of concerns [44]. When the overall system structure is unknown, but the individual components are available, researchers can construct a system model from these components. Several studies adapted this approach [44, 43, 23]. The general approach involves extracting “mini-patterns” from the provided data and combining them to construct the final automaton.

Reger *et al.* [43] propose a methodology that begins with a library of simple patterns, which are iteratively combined into increasingly complex structures. These composite structures are validated against both positive and negative traces. Their final stage involves consolidating all successful positive patterns into a single automaton that accepts all positive behaviours and rejects all negative ones.

Similar to [43], a study by Shin *et al.* [44] employs a divide-and-conquer strategy for the inference process. The authors use system logs to first infer individual models for each system component. They then combine these separate models by accounting for the event flows reflected in the logs.

Damasceno *et al.* [18] describe an alternative approach to pattern combination. They infer and merge mini-automata to construct the complete system behaviour. Each of which describes a specific system functionality. Using a model comparison method [53], they identify similar states between two configuration models. They then integrate necessary transitions or states to include the behaviour of both models while avoiding redundancy. The system iteratively updates this compact model by comparing it to one configuration at a time until it integrates all models.

Methods of [43, 44], and [18] rely on predefined pattern libraries, whereas Gabel *et al.* [23] introduced Javert, a tool that infers models for Java programs without a library. They observed that

as automata grow, the computational cost of operations such as finite-state expansion, intersection, and minimisation increases significantly. To address this, they construct complex patterns from small, generic templates, such as alternating sequences $(ab)^*$ and resource ownership $(ab^*c)^*$.

Javert can automatically learn simple, generic patterns and compose them using two inference rules: branching and sequencing. The branching rule considers patterns with the same ending event, while the sequencing rule considers patterns with compatible ending events. Both rules are general and could be applied to all micro-patterns. Javert keeps applying the two rules to the provided micro-patterns until the rules are no longer applicable.

The aforementioned frameworks [43, 44, 23] maintain and query a diverse library of patterns in stages. There is a need to reduce the computational overhead of the mining process and simplify the overall architectural complexity. Moreover, these approaches rely entirely on the patterns to build the final automaton. In sparse data scenarios, where mined patterns may be incomplete or incorrect, using patterns as a secondary constraint rather than the primary building block would improve the learner’s accuracy in inferring models that approximate the target model.

Patterns to Add New Information

Linear Temporal Logic (LTL) patterns have been utilised by Walkinshaw *et al.* [52] to significantly reduce the reliance on massive execution trace sets. In their framework, the target machine must satisfy pre-defined LTL constraints alongside standard trace data. They employ a model checker to verify that intermediate hypothesis machines do not violate these established properties. If a violation is detected, the model checker generates a counterexample, which is appended to the initial trace set before restarting the entire inference process.

This approach is applicable to both passive and active learning techniques. In a passive learning context, the framework learns a model from the given data and evaluates the resulting hypothesis against the user-defined LTL constraints during every state-merge operation. If a violation occurs, the system treats the counterexample as a negative instance, adds it to the training set, and restarts the learning process with this expanded dataset. Otherwise, the merge is accepted, and the learner proceeds to the next pair of candidate states.

Chapter 3

Mined Patterns within EDSM

The accuracy of passive automaton learning relies heavily on training data density and quality, as traditional state-merging heuristics perform poorly when inferring target topologies from sparse execution logs [52]. Although invariant- and pattern-based frameworks can guide learners through data-sparse environments [36, 52], existing solutions depend on unavailable user-defined templates or computationally expensive mining algorithms. To overcome these limitations where baseline algorithms like DFASAT fail, this chapter presents a hybrid learning framework that combines state-merging heuristics with a flexible, lightweight pattern-mining strategy. By systematically analysing a Prefix Tree Acceptor (PTA) to identify the absence of specific pairwise transitions, the proposed learner extracts global event constraints to prevent invalid state merges, ensuring the inferred automaton strictly preserves underlying system properties even under severe data sparsity.

3.1 Motivation

EDSM (Section 2.2.3) has long been established as the baseline methodology in passive automata learning. However, while EDSM solves many complex problems, it frequently falters on the toughest instances. Specifically, with exceptionally large target automata and highly sparse training data. Recognising these limitations, several researchers have attempted to refine EDSM (e.g., [1, 27, 26]). Among these attempts, DFASAT (Section 2.2.4) stands out as one of the most sophisticated hybrid frameworks available. Yet, even DFASAT struggles and makes critical errors when data is extremely scarce.

To advance the baseline, we address several fundamental limitations in the existing algorithms:

- **The Trap of Sparse Training Data:** Existing algorithms struggle to reliably learn an accurate automaton from highly sparse data. Without enough trace information to distinguish unique states, algorithms either over-merge (collapsing distinct behaviours into a single state) or under-merge (leaving the state tree fragmented and unreduced).
- **Flawed Selection Criteria:** The core challenge of state merging is identifying exactly which state pairs to combine. Current criteria rely too heavily on immediate, localised evidence, which can easily mislead the learner in the early stages.
- **The Inability to Backtrack:** The vast majority of passive inference algorithms operate without a safety net. They cannot undo a bad merge. Because implementing a full backtracking mechanism is computationally expensive, any early mistake permanently derails the topology of the final model.

- **The High Stakes of Initial Merges:** In state-merging, the very first merges dictate the shape of the emerging hypothesis. Making a mistake at the beginning of the process triggers a compounding cascade of errors across all subsequent iterations.
- **The Complexity of Mealy Machine Inference:** Standard passive inference techniques lose their edge when applied to Mealy machines. Deterministic Finite Automata (DFA) induction only handles simple binary state acceptance (accept/reject). Accept states indicate the terminal state for positive traces, while reject states indicate the terminal state for negative traces. In DFA induction, a pair of states is valid for merging if both states have the same acceptance status (both accept or both reject). Mealy machine inference requires mapping intricate input-output transformations onto every single transition. This added dimension exponentially expands the search space and increases structural complexity.

A promising avenue to combat data sparsity is integrating behavioural patterns directly into the inference process. Unfortunately, existing approaches typically rely on user-defined, manually hardcoded patterns [52] or specialised mining techniques that require domain expertise [44]. In practical settings, these flawless, high-fidelity structural blueprints are rarely available.

A massive methodological gap persists among the rigid, static constraints of the DFASAT algorithm, the unconstrained permissiveness of classical EDSM, and the valuable, latent knowledge locked in the training data. To bridge this gap, we need a novel algorithm that balances these constraints to reliably infer Mealy machines. There is a clear, compelling opportunity to extract the domain knowledge already embedded in the training traces and equip the EDSM framework with a flexible, lightweight heuristic to guide the learning process toward learning the correct target model.

In this thesis, we propose a novel passive learning algorithm that extracts and leverages structural patterns within sparse training data (Contribution **C1**) to guide state-merging decisions during Mealy machine inference (Contribution **C2**). To evaluate the efficacy and scope of our approach (Contribution **C3**), we address the following foundational research questions:

Research Questions

- RQ1** How does incorporating mined patterns within EDSM and DFASAT algorithms improve the accuracy of the final model compared to DFASAT?
- RQ2** How does incorporating mined patterns within EDSM and DFASAT algorithms assist learners in making superior merging decisions?
- RQ3** How does incorporating mined patterns within EDSM and DFASAT algorithms affect the learning process?

3.2 Methodological Overview

Figure 3.1 illustrates an overview of the operational pipeline of our proposed frameworks. The primary objective of our architecture is to reliably reverse-engineer a Mealy machine from execution traces via passive learning. As charted in the pipeline, the general process begins with positive training traces that represent the permissible behaviour of the system under learning. We construct a baseline PTA from these traces and apply our pattern-mining method to the tree structure (this addresses the first contribution **C1**). Next, these mined patterns, along with the PTA, are passed to the learner to infer a model consistent with the training data.

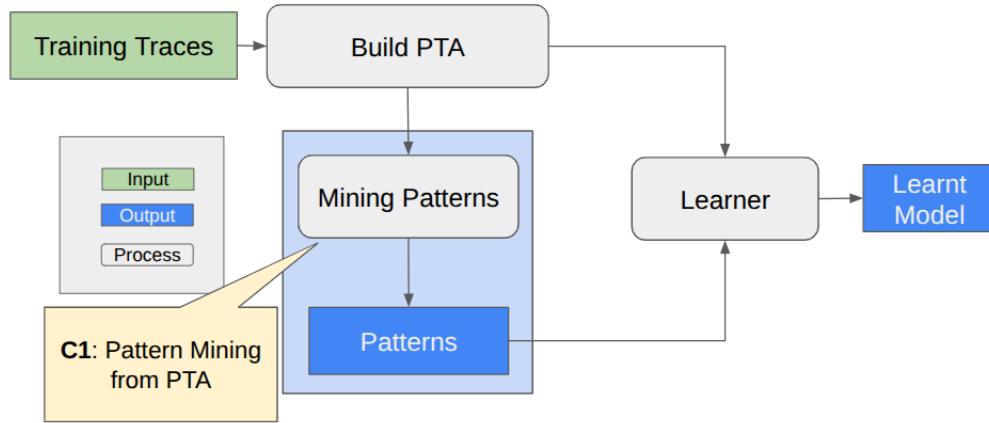


Figure 3.1: Methodology overview

To implement the previous pipeline, we designed multiple learners with different configurations. Specifically, this configuration introduces interconnected sub-contributions corresponding to the second contribution **C2** as follows:

- C2-1 Integrating the mined patterns into EDSM (PAT-EDSM):** The design of a state-merging pipeline that leverages these mined patterns to dynamically guide the EDSM state selection.
- C2-2 Tailored SAT for the Input-Enabled property:** The derivation of exact propositional clauses tailored for a SAT solver to guarantee the inference of an input-enabled Mealy machine.
- C2-3 Pattern-based DFASAT Adaptation (PAT-DFASAT):** The structural adaptation of the DFASAT framework to design a hybrid approach that performs initial merges using EDSM with mined patterns to guide state merges (instead of the DFASAT heuristic) prior to executing the SAT solver.
- C2-4 Pattern-to-CNF Propositional Encoding:** The development of a systematic encoding scheme that translates extracted sequential patterns into a Conjunctive Normal Form (CNF) formula. This contribution is an extension of contribution **C2-2**, where we translate the mined patterns into CNF for an Input-enabled Mealy machine.
- C2-5 Integrating the mined patterns into standard DFASAT (Pattern-guided DFASAT):** The design of a pipeline to integrate the mined patterns in addition to the standard DFASAT heuristic for state selection.
- C2-6 Integrating patterns into both EDSM and SAT (PAT-DFASAT+PAT-SAT):** The design of a pipeline to integrate the mined patterns into EDSM for state selection and then integrate the patterns into the SAT solver.

C2-7 Integrating patterns into both DFASAT heuristic and SAT

(Pattern-guided DFASAT+PAT-SAT): The design of a pipeline to integrate the mined patterns in addition to the standard DFASAT heuristic and then integrate the patterns into the SAT solver.

Table 3.1 maps the sub-contributions for **C2** across the learners and clarifies the different configurations for each learner.

Table 3.1: *Configurations for the proposed learners*

	(C2-2)			
	EDSM with DFASAT heuristic	EDSM with patterns constrains	SAT for Mealy with Input-enabled property	SAT for Mealy with Input-enabled property and patterns constrains (C2-4)
PAT-EDSM (C2-1)		X		
DFASAT	X		X	
PAT-DFASAT (C2-3)		X	X	
Pattern-guided DFASAT (C2-5)	X	X	X	
PAT-DFASAT + PAT-SAT (C2-6)		X		X
Pattern-guided DFASAT + PAT-SAT (C2-7)	X	X		X

3.3 Pattern Mining

In this section, we introduce a novel, lightweight extraction method that extracts behavioural patterns from the structural topology of a PTA (Contribution **C1**). Consider the following training trace for the Text Editor example in Figure 3.2:

$$\langle open/1, edit/1, exit/0, open/1, open/0 \rangle$$

The positive training traces represent only the system’s permissible behaviour. Implicitly, this formulation assumes a closed-world perspective: any behaviour not observed in the initial data is considered invalid under the target specification. For instance, in the given trace, the label *open/1*

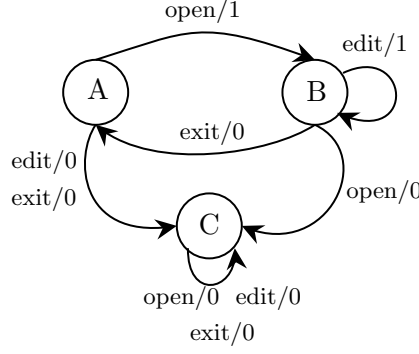


Figure 3.2: Text Editor Example (same as Figure 2.1)

is followed by *edit/1* and *open/0*, but it is never followed directly by *exit/0*. Consequently, we can assume that the final model should not permit the sequential behaviour $\langle \dots, \text{open/1}, \text{exit/0}, \dots \rangle$. Thereby, these unobserved behaviours should be excluded from the final inferred model. Our strategy systematically harvests the PTA to detect these unobserved behaviours, thereby alerting the learner whenever a candidate state merge violates an implicit behavioural negative constraint.

At a primitive level, a sequential behaviour can be represented by a pair of contiguous events. For example, within a given training dataset, a specific event *b* might never be immediately observed after an event *a*. To generalise this, for every transition label, we evaluate which subsequent labels never follow it anywhere across the PTA.

Formally, let $\mathcal{M}$ be a Mealy machine as defined in Definition 1, and $P(i_1/o_1, i_2/o_2)$ be a pattern identification function, where $i_1, i_2 \in I$ and $o_1, o_2 \in O$. The function P identifies a strict behavioural pattern where a transition labelled i_1/o_1 is followed by other transitions within the PTA, but never by i_2/o_2 .

For instance, in the baseline PTA illustrated in Figure 3.3, $P(\text{open/1}, \text{edit/0})$ constitutes a valid pattern because the PTA does not have any state with an incoming transition labelled *open/1* and an outgoing transition labelled *edit/0*. In other words, we never observe the subsequence $\langle \text{open/1}, \text{edit/0} \rangle$ in the PTA.

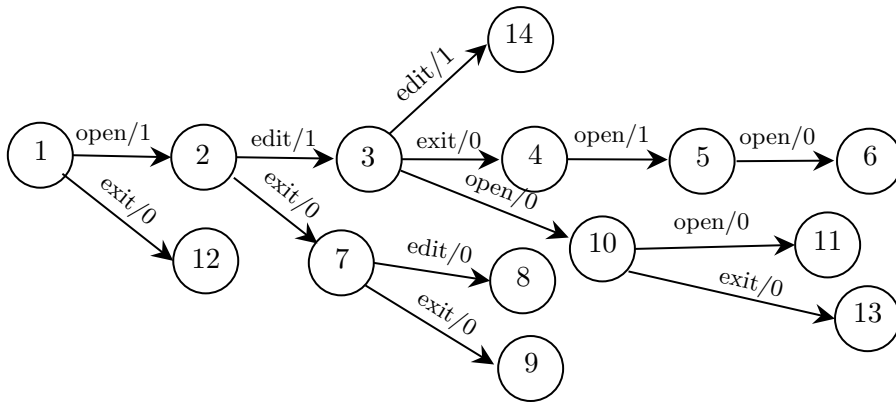


Figure 3.3: A PTA for the Text Editor System - (same PTA in Figure 2.4)

A comprehensive list of these extracted patterns is detailed in Table 3.2.

Note that the label *edit/0* is never followed by any subsequent transition within the observed traces. Consequently, based on the available data, we explicitly exclude *edit/0* from serving as a valid precondition for any behavioural pattern.

$p_1 = P(open/1, open/1)$	$p_5 = P(exit/0, open/0)$
$p_2 = P(open/1, edit/0)$	$p_6 = P(exit/0, edit/1),$
$p_3 = P(edit/1, open/1)$	$p_7 = P(open/0, open/1)$
$p_4 = P(edit/1, edit/0)$	$p_8 = P(open/0, edit/1)$
	$p_9 = P(open/0, edit/0)$

Table 3.2: *Mined Patterns for the Text Editor Example*

To implement the pairwise tracking, for every input/output pair, we retrieve all transitions sharing the same current label (i.e., input/output). Formally, a transition is defined as a 4-tuple (s, i, o, s') , comprising a source state s , an input symbol i , an output symbol o , and a target state s' . For each transition, matched input i and output o , we evaluate the subsequent outgoing transitions originating directly from the destination target state s' . Finally, we isolate the non-following label pairs; any combination of a current label and an unobserved subsequent label constructs a pattern constraint.

Our mined patterns are strictly of length two, where a specific input/output pair is never followed by another pair. We utilise this minimal pattern length to maintain a lightweight computation during both pattern extraction and constraint-verification phases. Extending constraint patterns to length k (where a terminal pair is absent) introduces significant scalability challenges, particularly during iterative learning. As state-merging progresses, combining states frequently introduces loops and cycles into the candidate Mealy machine. Checking whether a merge violates a pattern of length k requires exploring paths up to length k through these newly formed cyclic structures. This verification step quickly becomes computationally expensive, as cycles lead to a combinatorial explosion of potential paths to trace. In contrast, verifying constraints of length 2 involves evaluating only immediate forward transitions from the merged states, keeping constraint validation fast and manageable.

Furthermore, longer sequence patterns are highly sensitive to data sparsity. In finite training datasets, specific sequences of length k may be absent simply because they were not exercised during trace collection, rather than being fundamentally invalid. Evaluating candidate patterns of length k from a static dataset also scales exponentially. Restricting constraint evaluation to length 2 mitigates the risk of overfitting to arbitrary trace lengths, allowing the inference process to extract robust structural invariants while maintaining an efficient search space throughout state merging. Consequently, our primary objective is to leverage concise, low-overhead patterns, thereby avoiding unnecessary algorithmic complexity within the overall model inference pipeline.

3.4 PAT-EDSM

Our first proposed approach to optimise the EDSM algorithm (Algorithm 1) is PAT-EDSM (Contribution **C2-1**). PAT-EDSM algorithm is an extension of the standard EDSM framework. It integrates a global view of the mined patterns (as demonstrated in the previous section) with EDSM’s local heuristic, which facilitates highly accurate merging decisions. The core mechanism evaluates all possible behaviours between event pairs and blocks merges that introduce a violation of the mined patterns (of Section 3.3), either during the initial state collapse or downstream during non-determinism resolution.

Whenever a valid state merge is proposed, the resulting automaton configuration is evaluated against the mined patterns (extracted as explained previously). Specifically, for every pair of states selected for merging (and during subsequent merges required to resolve nondeterminism), the structure of the merged state is validated against its incoming and outgoing transitions. If any

resulting state exhibits a combination of incoming and outgoing transition labels that matches a mined pattern constraint, the merge is immediately blocked and penalised with a negative score.

For example, consider the merge between states 1 and 2 in Figure 3.3. Executing this operation triggers a merge between states 7 and 12 to resolve nondeterminism. The resulting configuration is illustrated in Figure 3.4. Although this merge is valid, the resulting state introduces a self-loop on the label *open/1*, which directly violates the pattern constraint $p_1 \in P$ (Table 3.2). Consequently, the PAT-EDSM learner rejects the merge and assigns the pair a negative score.

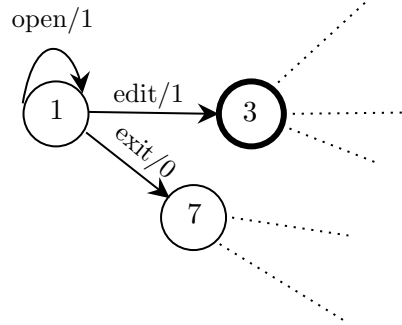


Figure 3.4: The result configuration of merging states 1 and 2 - (same as Figure 2.7)

As outlined in Algorithm 3, PAT-EDSM modifies the standard EDSM framework in two key phases: first, a pattern mining routine is executed immediately following the construction of the PTA (Line 2); second, a pattern validation is embedded within the consistency check (Line 11), ensuring state pairs are only scored if they satisfy all mined properties. The boolean function `Violate_P` takes four parameters: the list of mined patterns P , the current inferred model $\mathcal{M}$, the red state r , and the blue state b . This function merges the blue state into the red state, and then checks if the resulting model violates any of the patterns in P . It returns `True` if there is any violation and `False` otherwise. The algorithm only considers a pair of red-blue states to be valid if the resulting model is deterministic (checked by `is_Consistent`, demonstrated in Section 2.2.3) and does not violate any mined patterns (checked by $\neg \text{Violate_P}$).

Algorithm 3: PAT-EDSM

Input: A set of traces T
Output: A Mealy machine $\mathcal{M}$ consistent with T

```

1  $\mathcal{M} \leftarrow \text{Build\_PTA}(T)$  // Construct the Prefix Tree Acceptor
2  $P \leftarrow \text{Mine\_Patterns}(\mathcal{M})$  // Extract patterns from PTA
3  $RED\_States \leftarrow \{q_0\}$  // colour the initial state red
4  $BLUE\_States \leftarrow \text{Get\_Successor}(q_0)$  // Colour successors blue
5 while  $BLUE\_States \neq \emptyset$  do
6    $Best\_Pair \leftarrow \emptyset$ 
7    $Highest\_Score \leftarrow 0$ 
8   foreach  $b \in BLUE\_States$  do
9      $Not\_consistent\_with\_any\_red = \text{True}$ 
10    foreach  $r \in RED\_States$  do
11      // If consistent with  $T$  &  $P$ 
12      if  $Is\_Consistent(\mathcal{M}, r, b) \wedge \neg Violate\_P(P, \mathcal{M}, r, b)$  then
13         $Not\_consistent\_with\_any\_red = \text{False}$ 
14        if  $Merging\_Score(\mathcal{M}, r, b) > Highest\_Score$  then
15           $Highest\_Score \leftarrow Merging\_Score(\mathcal{M}, r, b)$ 
16           $Best\_Pair \leftarrow (r, b)$ 
17    if  $Not\_consistent\_with\_any\_red \wedge Best\_Pair = \emptyset$  then
18       $RED\_States \leftarrow RED\_States \cup \{b\}$  // Make b red
19      // Make (b)'s successors blue
20       $BLUE\_States \leftarrow BLUE\_States \setminus \{b\} \cup \text{Get\_Successor}(b)$ 
21      break and go to line 5
22  if  $Best\_Pair \neq \emptyset$  then
23     $\mathcal{M} \leftarrow \text{Merge}(\mathcal{M}, Best\_Pair.r, Best\_Pair.b)$ 
24    // Update the Blue_States set
25     $BLUE\_States \leftarrow \{b \in \text{Get\_Successor}(r) \mid r \in RED\_States\} \setminus RED\_States$ 
26 return  $\mathcal{M}$ 

```

Example:

This example traces the execution of the PAT-EDSM algorithm on the Text Editor example, starting from the baseline PTA shown in Figure 3.5 (previously used to demonstrate the EDSM and DFASAT algorithms). Let $\mathcal{M}'$ denote the initial PTA, P be the set of mined patterns (Table 3.2), and R and B denote the sets of red and blue states, respectively. The algorithm initialises with $R = \{1\}$ and $B = \{2, 12\}$. PAT-EDSM blocks the merge of states 1 and 2 due to a violation of pattern $p_1 \in P$; in contrast, DFASAT blocks this same pair based on localised transition mismatches, whereas EDSM incorrectly executes the merge. In the subsequent iteration, where $R = \{1, 2\}$ and $B = \{12, 3, 7\}$, state 12 is promoted to red due to a lack of overlapping evidence. Across the resulting six candidate pairs (where $R = \{1, 2, 12\}$ and $B = \{3, 7\}$), PAT-EDSM successfully identifies the correct pair (2, 3). Crucially, the pairs (1, 3) and (2, 7) are blocked by a determinism conflict (different outputs for the same input) on the input *open* and *edit*, respectively.

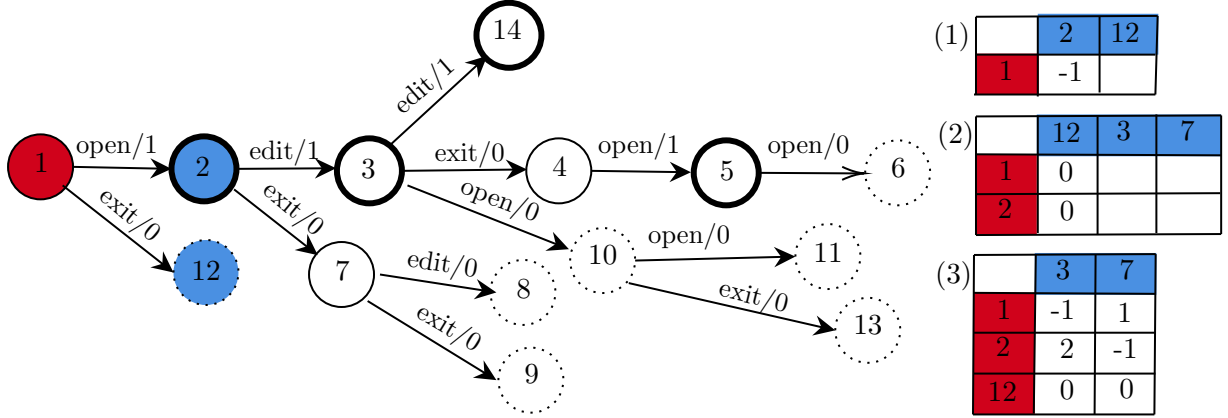


Figure 3.5: PTA of the Text Editor (same PTA as in Figure 2.4) - States with the same border line correspond to the same state in the target model. Tables are numbered according to their implementation order. Blank cells in the tables indicate that the blue state was not evaluated against the red states because an earlier blue state was promoted to red.

Figure 3.6 illustrates $\mathcal{M}'$ following this first merge between states 2 and 3, which cascades to fold state 14 into state 2, and state 7 into state 4 during non-determinism resolution. Next, the updated red and blue sets are $R = \{1, 2, 12\}$ and $B = \{4, 10\}$. The algorithm correctly identifies the best pair $S_{PAT-EDSM}(1, 4) = 2$ as the one with the highest score among all other valid pairs.

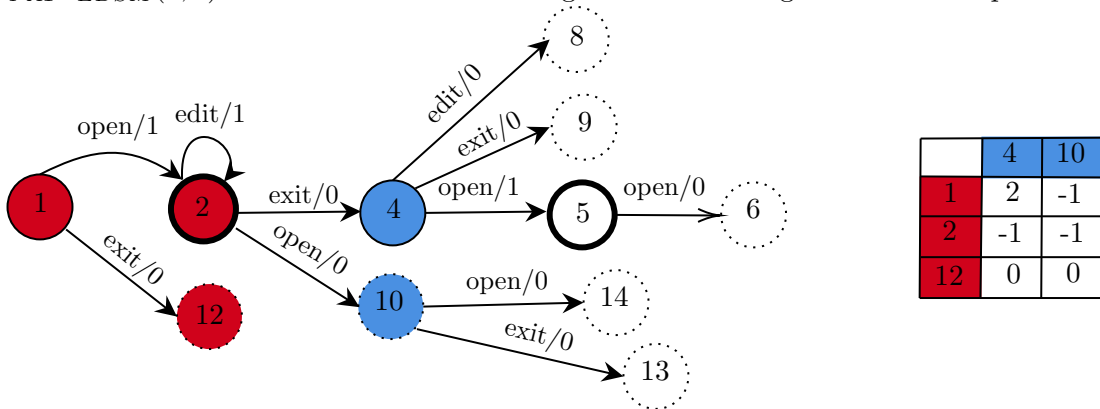


Figure 3.6: Performing the first Merge between state 2 and state 3

The resulting model, presented in Figure 3.7, shows the model after the second merge between states 1 and 4. The following evaluated state (state 10) is promoted to red because the candidate pair (1, 10) introduces a non-deterministic conflict, the pair (2, 10) violates pattern $p_8 \in P$, and the pair (12, 10) yields a score of zero. The remaining states (8, 14, and 13) are coloured red because they do not share labels with any existing red states.

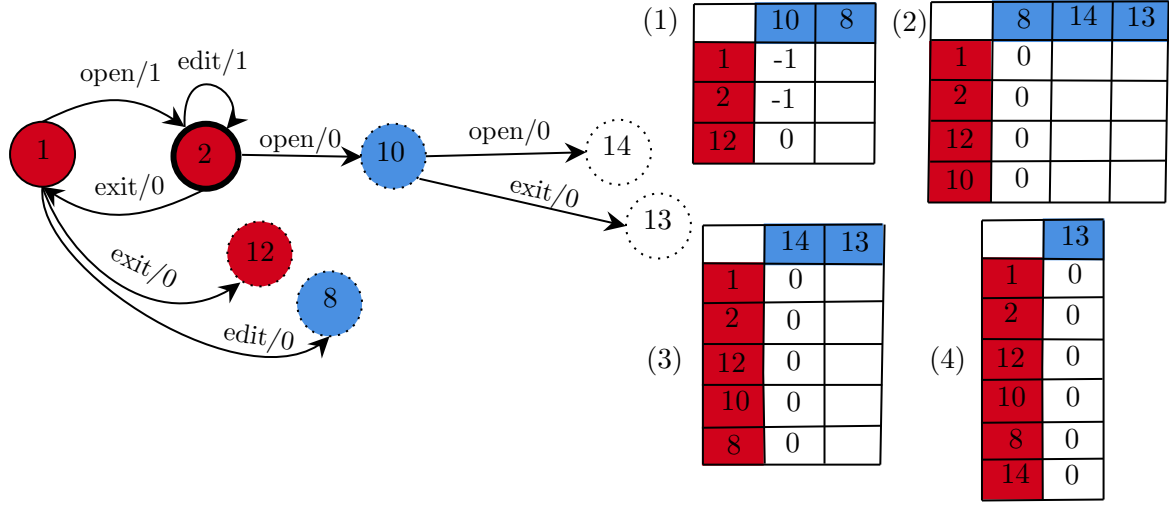


Figure 3.7: Performing the second merge between state 1 and state 4

This progression culminates in the final model shown in Figure 3.8.

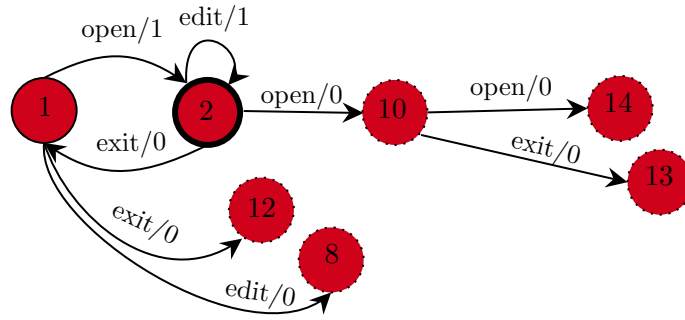


Figure 3.8: The final model inferred by PAT-EDSM

Figure 3.9 illustrates the final model inferred by the standard EDSM algorithm (Algorithm 1). The target model that both the EDSM and PAT-EDSM algorithms aim to infer is shown in Figure 3.10. Evaluating the structural topological similarity between the inferred models and the target model yields the following key observations:

- **Total Number of States:** Both the EDSM and PAT-EDSM algorithms inferred a 7-state model, whereas the target model contains only 3 states.
 - **Identification of the Initial State (state A):** The EDSM model (Figure 3.10) erroneously includes a self-loop for `open/1` at the initial state and lacks an incoming transition for the label `exit/0`. Conversely, PAT-EDSM correctly identifies the initial state (state 1 in Figure 3.8). It accurately resolves all outgoing transitions and correctly connects the transition `open/1` to state 2 in the PAT-EDSM model (corresponding to state B in the target model). However, PAT-EDSM fails to fully resolve state C of the target model, resulting in two distinct target states for the transitions `exit/0` and `edit/0`. Regarding incoming transitions, the PAT-EDSM model correctly preserves an incoming transition for `exit/0` originating from state 2 (which corresponds to State B in the target model).
 - **Identification of State B:** In the EDSM model (Figure 3.10), identifying the state corresponding to State B is ambiguous. In the target model, executing `open/1` from the initial state transitions to state B; tracking this same path in the EDSM model leaves the model in the initial state. While state 3 contains an outgoing transition `exit/0` to state 4 (whereas state B returns to the initial state upon executing `exit/0`), it also has an incoming transition `open/1` from state 4. Furthermore, state 3 includes a self-loop for `open/0` that does not exist on state B.
- In contrast, state 2 in the PAT-EDSM model (Figure 3.8) exhibits the closest structural alignment with state B. Executing `open/1` from the initial state transitions directly to state 2. Furthermore, state 2 includes a self-loop for `edit/1`, matching state B in the target model. Regarding outgoing transitions, state 2 has two exits: `exit/1` returns to the initial state (aligning with state B), and `open/0` transitions to state 10.
- **Identification of the Sink State (State C):** Neither algorithm successfully identify the sink state. However, PAT-EDSM merely under-merges the set of states corresponding to the sink state (specifically States 10, 12, 8, 14, and 13) due to a lack of supporting evidence; merging these under-merged states produces the exact target model. In contrast, EDSM executed structurally incorrect state merges (merging states that do not refer to the same states in the target model), which fundamentally corrupts the topology of the final inferred model.

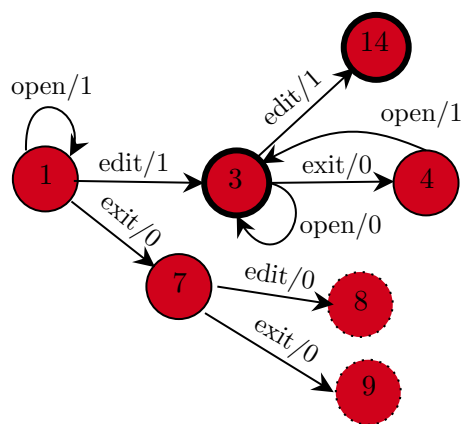


Figure 3.9: The Final Model inferred by EDSM - copy of Figure 2.13b

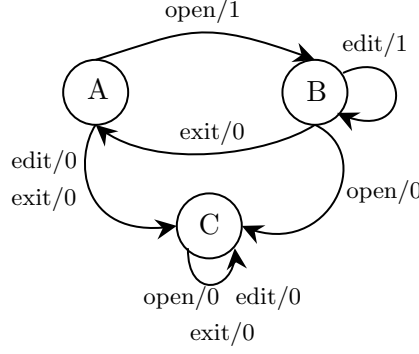


Figure 3.10: *Text Editor - target model (same as Figure 2.1)*

3.5 PAT-DFASAT

In the previous example 3.8, PAT-EDSM was able to identify (compared to the target model in Figure 3.11) states *A* and *B* correctly, but failed to merge states corresponding to state *C* due to lack of evidence. To address such cases, we introduce our second approach for optimising the EDSM algorithm (Section 2.2.3), PAT-DFASAT (Contribution **C2-3**). This approach adapts the DFASAT framework [27] (see Section 2.2.4). PAT-DFASAT performs some initial merges using PAT-EDSM (Algorithm 3), and then passes the intermediate model to the SAT solver to resolve any remaining unmerged transitions. To enhance solving efficiency, we constrain the iterative merging process to run until the state space is reduced by half ($|\mathcal{M}'| = \frac{|Q_{PTA}|}{2}$). Collapsing highly similar state pairs early on drastically minimises the remaining search space. Consequently, this prevents the SAT solver from committing over-merges and allows it to efficiently isolate and resolve regions of the automaton characterised by low evidence.

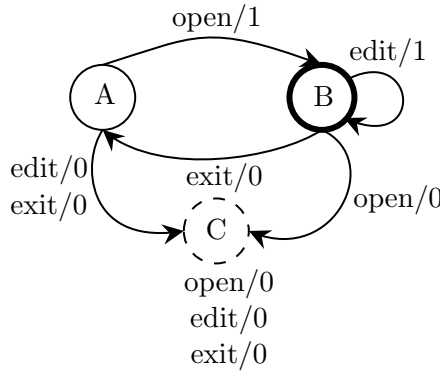


Figure 3.11: *Text Editor Target Model (same as Figure 2.5)*

Employing a fixed threshold for the heuristic-to-SAT transition (as seen in standard DFASAT, detailed in Section 2.2.4) is inherently suboptimal, because target use cases vary significantly in structural scale and behavioural complexity. Selecting a relatively low threshold forces risky state merges when evidence is weak (i.e., low EDSM scores). Conversely, an excessively high threshold leaves small PTA under-merged, passing a massive state space to the SAT solver and subsequently triggering overgeneralisation issues.

Accordingly, we dynamically adjust our implementation for each case study, using the initial size of the PTA to determine the optimal switching point. Executing extra merges increases the probability of introducing irreversible, erroneous state merges based on fragile evidence. Conversely, executing too few heuristic merges leaves pairs with robust evidence unmerged. Our proposed heuristic bound ($|\mathcal{M}'| = \frac{|Q_{PTA}|}{2}$) successfully balance between these two operational extremes.

As discussed in Section 2.2.4, the SAT solver utilises two types of Boolean auxiliary variables to maintain Mealy properties [27, 46]: colour variables ($x_{v,c}$) to label distinct states, and parenting variables ($y_{i,o,p,c}$) to identify transitions, where i for input, o for output, p for the parent state (source), and c for the child state (target). To ensure determinism and consistency, the solver enforces a set of logical clauses (see Table 2.1).

Our framework targets input-enabled machines, where every state $q \in Q$ must possess exactly one outgoing transition for each input symbol $i \in I$. In a deterministic Mealy machine, each parent relation must be linked with only one label of the input/output pair. To enforce this structural constraint in the final model (Contribution **C2-2**), our implementation adapts two foundational clause sets from [46] as follows (A list of our implementation clauses is detailed in Table 3.3):

1. The state-colouring constraints, which dictate that each state maps to exactly one colour via “*At Least One Colour*” and “*At Most One Colour*” clauses.
2. The transition consistency constraints, which ensure consistency with the set of traces T by “*A Parent Relation is set when a State and its Parent are Coloured*” and “*A Parent Relation forces labelling a State once the Parent is Coloured*” clauses.

Table 3.3: Encoding Mealy machine identification into SAT.

Clauses	Range	Meaning
$(x_{p,1} \vee x_{p,2} \vee \dots \vee x_{p,k})$	$p \in Q; k \in C; k \neq \mathbf{r}$	At least 1 colour per state.
$(\neg x_{p,i} \vee \neg x_{p,j})$	$p \in Q; i, j \in C;$ $i < j \ i, j \neq \mathbf{r}$	At most 1 colour per state.
$(y_{a,b,i,j} \vee \neg x_{p,i} \vee \neg x_{q,j})$	$p, q \in Q; i, j \in C;$ p parent for q with transition a/b	A parent relation is set when a state and its parent are coloured.
$(\neg y_{a,b,i,j} \vee \neg x_{p,i} \vee x_{q,j})$	$p, q \in Q; i, j \in C;$ p parent for q with transition a/b	A parent relation forces labelling a state once the parent is coloured.
$(y_{a,1,i,j} \vee y_{a,2,i,j} \vee \dots \vee y_{a,b,i,j})$	$a \in I; b \in O; i, j, R \in C;$ $i \neq R$	At least 1 pair per parent relation.
$(\neg y_{a,b,i,j} \vee y_{a,c,i,\mathbf{r}})$	$a \in I; b, c \in O; b \neq c;$ $i, j, R \in C; i, j \neq \mathbf{r}$	Once a transition to an accept-state is defined, all other options must go to the reject-state.
$(\neg y_{a,b,i,j} \vee \neg y_{a,c,i,k})$	$a \in I; b, c \in O; b \neq c;$ $i, j, \mathbf{r} \in C; j \neq k, \mathbf{r}$	At most 1 pair per parent relation.
$(\neg y_{a,b,\mathbf{r},i})$	$a \in I; b \in O;$ $i, \mathbf{r} \in C$	The reject-state has no outgoing transitions.

C = set of colours, I = set of Inputs, O = set of outputs, Q = set of states, $\mathbf{r} \in C$ is the label of the reject-state

We modify the determinism conflict clauses to explicitly support the input-enabled property. Because the PTA is not typically input-enabled, merging states often fails to yield an input-enabled structure, which would otherwise generate unsatisfiable constraints. To address this data gap, the approach is based on the assumption that any transition not explicitly defined in the training data is treated as invalid. For example, in Figure 3.3, state 1 does not have a transition defined for input *edit*. To add these missing transitions, we introduce the notion of a reject-state. Hence, SAT will add a transition from state 1 to a special reject-state for the input *edit*.

The reject-state is a special state added to the final model to collect all missing transitions while maintaining consistency. The solver routes all undefined inputs to a reject-state. While the missing inputs are easily identified, the corresponding output is unknown; since they lead to a fake state, the solver randomly assigns an output to satisfy the formal requirements. We ensure that the reject-state remains isolated from the functional logic of the model by adding a “*Reject-state has no outgoing transitions*” clause. To reduce the solver’s search space, we add Symmetry Breaking Predicates (see Section 2.2.4) by predefining a specific label for the reject-state. Furthermore, we ensure that the reject-state will never be merged with any other state by the SAT. To maintain this, we reserve a colour $\mathbf{r} \in C$ for the reject state. When building the clause, we make sure that the states which belong to the partial model are not assigned to the colour $\mathbf{r}$ (do not have the same colour as the sink state, see Table 3.3).

The primary purpose of incorporating a reject state, as previously discussed, is to preserve the input-enabled property of a Mealy machine during execution. However, although the reject state is introduced during the SAT-solver phase, it does not represent an actual structural entity in the underlying target system that the learner aims to infer. Consequently, once the solver identifies a satisfiable solution, the reject state and its incoming transitions are pruned. This post-processing step ensures that the final inferred model consists strictly of states derived from valid merges within the initial PTA.

Figure 3.12 illustrates incorporating the reject state $\mathbf{r}$ into the inferred model during the SAT phase. The SAT solver adds the missing transitions (denoted by dashed arrows) from each state to the reject state. These missing transitions are identified by exploring the existing transitions. From each state, if there is an outgoing transition (with an input/output pair) to any already existing state, then a new transition targets the reject state from this state with the same input but a different output. For Example, consider the Mealy machine $\mathcal{M}$ at Figure 3.12 defined by Definition 1 where: $Q = \{1, 2, 3\}$, $q_0 = 1$, $I = \{\text{open}, \text{edit}, \text{exit}\}$, $O = \{0, 1\}$, and $\mathcal{T}$ is all the transitions between all the states in Q (before adding the reject-state $\mathbf{r}$).

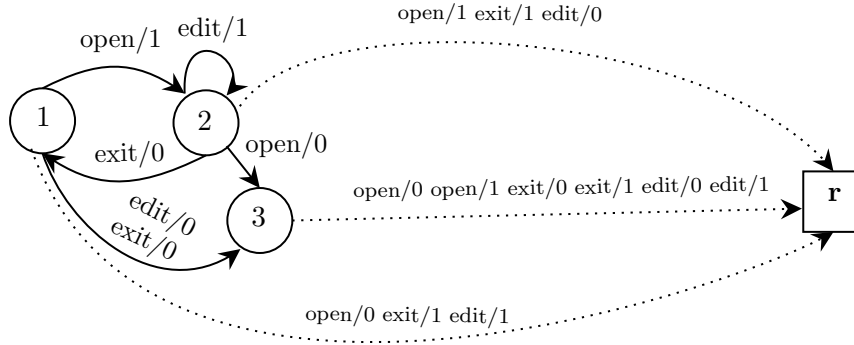


Figure 3.12: Example of adding the reject state to the inferred model

We ensure the minimum number of transitions between two states with the clause “*At least one Pair per Parent Relation*”. To prevent this from inducing illegal non-determinism (i.e., multiple transitions for the same input with differing outputs), we restrict the solver such that only a single transition can target a valid accept-state, forcing any alternative pairs to enter the reject-state. For example, in Figure 3.12, state 1 has outgoing transitions for the pairs: open/1, edit/0, and exit/0, which enter states 2, 3, and 3, respectively. State 1 is a deterministic and input-enabled state, since it has a transition for each $i \in I$. However, we require all other input/output pairs to enter the reject state. In the previous example, the pairs: open/0, edit/1, and exit/1 go to the reject state $\mathbf{r}$.

This condition is maintained via “Once a transition to an accept-state is defined, all other options must go to the reject-state” and “At most one Pair per Parent Relation” clauses.

The final SAT encoding includes:

1. The main types of auxiliary variables are: colour variables ($x_{v,c}$) and parenting variables ($y_{i,o,p,c}$) [46].
2. Ensuring each state must have exactly one colour: “At least a colour”, and “At most a colour” clauses [46].
3. Maintaining consistency with T : “A parent relation is set when a state and its parent are coloured” and “A parent relation forces labelling a state once the parent is coloured” clauses [46].

Our primary contributions to the SAT formulation for the input-enabled property:

4. Symmetry Breaking clause: Predefining a fixed label for the reject-state to optimise solver performance.
5. Implementing the “Rejected state has no outgoing transitions” constraint.
6. Maintaining the input-enabled property via “At least one Pair per Parent Relation”.
7. Maintaining determinism via “Once a transition to an accept-state is defined, all other options must go to the reject-state” and “At most one Pair per Parent Relation” rules.

SAT Solver Configurations

In addition to the aforementioned clause generation rules, the SAT-based optimisation phase incorporates specific execution parameters configured as follows:

- **Solver Time Limits and Termination:** Proving unsatisfiability (UNSAT) is computationally intensive because the solver must exhaustively traverse the variable assignment space. When handling formulas comprised of millions of clauses and variables, this complete verification can take hours or days. Our empirical observations indicate that if a satisfying assignment (SAT) exists, the solver typically discovers it within five minutes, even when processing formulas containing tens of millions of clauses. For instance, a valid solution for the Bluetooth use case (which generated in excess of 86 million clauses) was successfully resolved within this five-minute window, demonstrating that clause density does not inherently bottleneck the satisfiability search if a solution exists. To optimise computational resources, we enforced a strict five-minute timeout per solver instance; if exceeded, the formula is treated as unsatisfiable, and the search space is expanded with an additional colour. Our empirical study, which extended this timeout to 15 minutes, one hour, or no limit, found no variance in the final evaluation metrics across different case studies of various sizes and complexities.
- **Maximum Colour Upper Bounds:** The core objective of passive automata induction is to find an optimal, minimal solution that accurately approximates the target model. If the size of the induced model has no upper bound, the learner fails to provide a meaningful generalisation of the target behaviour. While the SAT solver natively increments the state cardinality until a satisfiable solution is found, we impose a strict upper bound on the maximum number of states to bound computational complexity. Specifically, if the solver cannot identify a satisfying assignment within a limit of twice the target state cardinality ($|Q_{target}| \times 2$), further search space exploration is deemed uninformative, and the algorithm terminates, reporting that no valid model was found.

- Minimum Colour Lower Bounds:** In the standard DFASAT framework, the lower bound for the number of colours is specified by the designated cluster of “red” states, as they represent distinct states of the target system. Our implementation explicitly accounts for potential under-merging anomalies during the heuristic phase. Recognising that the baseline DFASAT heuristic may prematurely bypass valid state merges due to over-restrictive constraints or uninformative zero-evidence scores, we adjust our solver’s colour initialisation boundaries downward to preserve search-space flexibility. Consequently, our pipeline initialises the search space at an absolute minimum of 2 states: one designated accept-state and one reject-state (our specialised reject-state).
- Linear search vs binary search:** To reduce the time required for the SAT solver to converge on a valid solution, we can optimise the current linear search strategy using binary search. Presently, the state-budget search sets a lower bound of 2 states and an upper bound of $|Q_{\text{target}}| \times 2$, incrementing the target state count (or colour budget) sequentially by one each time an UNSAT result is returned. Under a binary search scheme, the solver would evaluate the midpoint between the lower and upper bounds; if the formula is SAT, this midpoint becomes the new upper bound; if UNSAT, it marks the new lower bound. Adopting this logarithmic (log) exploration strategy would significantly reduce the number of required solver invocations. However, integrating binary search is reserved for future work.

PAT-DFASAT algorithm is shown in Algorithm 4. The first phase (Lines 4 to 23) performs the initial merges using the PAT-EDSM algorithm (see Algorithm 3). The second phase starts when $(|\mathcal{M}'| = \frac{|Q_{PTA}|}{2})$, Line 6. In the second phase, the algorithm starts with the minimum number of colours and gradually increases it until a solution is found. For each set of colours, the algorithm defines the variables, builds the CNF formula, and sends it to the solver. Once a solution is found, the final model is constructed from the final result.

Algorithm 4: PAT-DFASAT

Input: A set of traces T
Output: A Mealy machine $\mathcal{M}$ consistent with T

```

1  $\mathcal{M} \leftarrow \text{Build\_PTA}(T)$  // Construct the PTA
2  $P \leftarrow \text{Mine\_Patterns}(\mathcal{M})$  // Extract patterns from PTA
3  $PTA\_size \leftarrow |\text{States}(\mathcal{M})|$ 
4  $RED\_States \leftarrow \{q_0\}$  // colour the initial state red
5  $BLUE\_States \leftarrow \text{Get\_Successor}(q_0)$  // Colour successors blue
6 while  $|\mathcal{M}| > (PTA\_size/2) \wedge BLUE\_States \neq \emptyset$  // Do some merges
7 do
8    $Best\_Pair \leftarrow \emptyset$ 
9    $Highest\_Score \leftarrow 0$ 
10  foreach  $b \in BLUE\_States$  do
11    foreach  $r \in RED\_States$  do
12      if  $Is\_Consistent(\mathcal{M}, r, b) \wedge \neg Violate\_P(P, \mathcal{M}, u, v)$  // If consistent with  $T \& P$ 
13      then
14        if  $Merging\_Score(\mathcal{M}, r, b) > Highest\_Score$  then
15           $Highest\_Score \leftarrow Merging\_Score(\mathcal{M}, r, b)$ 
16           $Best\_Pair \leftarrow (r, b)$ 
17      if  $Not\_consistent\_with\_any\_red \wedge Best\_Pair = \emptyset$  then
18         $RED\_States \leftarrow RED\_States \cup \{b\}$  // Make b red
19         $BLUE\_States \leftarrow BLUE\_States \setminus \{b\} \cup \text{Get\_Successor}(b)$  // Make (b)'s successors
20        blue
21        break and go to line 6
22  if  $Best\_Pair \neq \emptyset$  then
23     $\mathcal{M} \leftarrow \text{Merge}(\mathcal{M}, Best\_Pair.r, Best\_Pair.b)$ 
24    // Update the Blue_States set
25     $BLUE\_States \leftarrow \{b \in \text{Get\_Successor}(r) \mid r \in RED\_States\} \setminus RED\_States$ 
26   $Colours \leftarrow 2$  // label 1:reject-state,  $\geq 2$ :accept-state
27  while  $Colours < |Q_{target}| \times 2$  do
28     $Variables = \text{Define\_Variables}(Colours, \mathcal{M})$ 
29     $CNF \leftarrow \text{Build\_CNF}(Variables, \mathcal{M})$ 
30     $SAT\_Result \leftarrow \text{Pass\_to\_SAT}(CNF)$  // Send the formula to the solver
31    if  $SAT\_Result = SAT$  then
32       $\mathcal{M} \leftarrow \text{ReConstructModel}(Variables, SAT\_Result)$ 
33      break
34     $Colours = Colours + 1$  // If UNSAT, add a colour & try again
35 return  $\mathcal{M}$ 

```

3.6 Integrating Mined Patterns into the SAT Solver

The intermediate automaton generated by the PAT-EDSM algorithm (Algorithm 3) remains consistent with both the provided training traces and the mined behavioural patterns. Because our mined patterns explicitly capture the absence of specific behaviours within the initial prefix tree, they can serve as a vital defensive layer against downstream optimisation errors (overgeneralisation). While the SAT solver theoretically guarantees a minimal model, it is still prone to errors

(such as over-merging states) when operating under severe data constraints. To mitigate this risk, we directly inject the mined patterns into the SAT solver as structural invariants (Contribution **C2-6**: PAT-DFASAT+PAT-SAT). This constraint-satisfaction approach prevents the solver from executing state merges that would fundamentally violate the underlying pattern logic.

The propositional logic for this encoding (Contribution **C2-4**) is as follows: for every sequential pattern $p(i_1/o_1, i_2/o_2)$ evaluated over candidate state-colors $a, b, r \in C$, $\mathbf{r}$ is the reject-state colour, the constraint dictates that if a transition from colour a to colour b exists with the input-output label i_1/o_1 , then a transition with the label i_2/o_2 must be from colour b to the reject-state with colour $\mathbf{r}$. Consequently, the blocking constraint is expressed as the following clause:

$$\neg Y_{i_1, o_1, a, b} \vee Y_{i_2, o_2, b, r} \quad (3.1)$$

Despite the theoretical soundness of this integration, empirical evaluation revealed no statistically significant improvement in the final inferred model compared to PAT-DFASAT. The following are critical bottlenecks and performance limitations:

- **Combinatorial Clause Explosion:** Injecting these patterns introduced a massive number of additional clauses to the Boolean formula, causing a combinatorial explosion that added unnecessary computational complexity and significantly degraded solver performance.
- **Sensitivity to Early Heuristic Errors:** Under conditions of extreme data sparsity, the upstream PAT-EDSM heuristic remains prone to initial merging errors. In these scenarios, enforcing pattern constraints within the SAT solver proves ineffective, as the underlying partial automata are already corrupted.

Detailed figures for all use cases' evaluation can be found in Appendix C. Our evaluation framework encompassed a wide range of trace generation strategies, including completely random sequences, random traces optimised for state coverage, and random traces optimised for full transition coverage.

3.7 Integrating Mined Patterns into the DFASAT Framework (Pattern-guided DFASAT)

The baseline DFASAT algorithm remains prone to errors due to the rigidity of its static blocking rule, which unconditionally rejects any merge that introduces a novel input/output label to a pre-existing red state. To address this limitation, we leverage data-driven domain knowledge of our patterns. Under this augmented paradigm, a candidate merge is blocked if and only if it introduces a novel behavioural sequence that is completely absent within the initial trace data. Combining the conservative structural constraints of DFASAT with these empirical behavioural patterns into a unified framework may mitigate the risk of premature, irreversible merging errors (Contribution **C2-5**). In this hybrid approach, whenever a structurally consistent merge candidate is identified, the algorithm performs a dual-layer verification process: the candidate merge is evaluated to ensure that it neither introduces a novel label into an existing red state nor violates any of the mined sequential patterns.

However, empirical results (Detailed figures in Appendix C) indicate that this integration does not yield a statistically significant improvement over the baseline DFASAT algorithm. In practice, the standard DFASAT heuristic operates as a strict superset of our pattern-based constraints. While DFASAT absolutely prohibits adding any novel transition label to the current red state space,

our pattern-driven constraints block a transition only if that specific label sequence is completely absent from the initial PTA. Consequently, the DFASAT rule is inherently more restrictive: any state merge blocked by the pattern constraints is also fundamentally rejected by DFASAT, whereas the reverse does not hold.

3.8 Integrating Mined Patterns into the SAT solver of DFASAT (Pattern-guided DFASAT+PAT-SAT)

The intermediate graph generated by integrating the mined patterns into the DFASAT framework is consistent with them. Consequently, we can add the patterns as constraints into the SAT solver to optimise the result (Contribution **C2-6**), as illustrated in Section 3.6. This approach performs initial merges using DFASAT’s heuristic and mined patterns, then translates the intermediate model into CNF, adding additional clauses to ensure consistency with the mined patterns.

The experimental results demonstrate that this proposed integration yields no quantifiable improvement over the baseline DFASAT algorithm (Detailed figures in Appendix C). Under extreme data sparsity, both the DFASAT and PAT-EDSM heuristics are prone to initial state-merging errors that ultimately yield structurally flawed intermediate models. Consequently, enforcing pattern-based constraints within the solver in these scenarios proves ineffective, as the underlying structural foundation is already incorrect. Furthermore, these mined patterns introduce a massive number of additional clauses into the Boolean formula, significantly increasing constraint-satisfaction complexity and overall computational time.

3.9 Implementation

The proposed framework is implemented in Python 3 (an open-source implementation is available at <https://github.com/arwaalfitni/Pattern-based-passive-automata-Learning>). Python was selected because it enables rapid development and offers a rich ecosystem of libraries for efficient data preparation and model inference. Additionally, it serves as an effective integration layer that connects our learning engine to external programs. Through the subprocesses, the Python framework formats and passes behavioural constraints directly to the SAT solver.

For the SAT-based approaches, we integrated the Kissat SAT solver [8]. Kissat is a high-performance solver that regularly wins the international SAT Competitions. It achieves high execution efficiency through optimised memory locality and an aggressive clause management system. This system uses advanced in-processing techniques (such as variable and blocked clause elimination) to periodically clean out redundant learned clauses, preventing memory exhaustion [9]. Furthermore, when a formula is unsatisfiable (UNSAT), Kissat generates verifiable certificates of unsatisfiability using proof logging [17], providing a mathematically sound chain of evidence that can be validated by external checkers.

The correctness of the implementation is verified through a combination of manual evaluation and automated test suites that cover diverse operational scenarios. While automated evaluation focuses on unit testing, manual evaluation verifies overall logic across scenarios that vary in size and internal structure. These tests ensure the reliability of both the underlying model inference and the integrated SAT solver pipeline under various constraint loads. To support reproducibility and open science, the complete codebase along with its testing framework is publicly available on GitHub at <https://github.com/arwaalfitni/Pattern-based-passive-automata-Learning>.

3.10 Summary

The quality of a model inferred through passive learning directly depends on the quality of the available data. Learning the hidden structure of a system from its execution traces typically requires a massive amount of data. To maximise data utility, several techniques have been proposed, such as merging states and behavioural patterns. Consequently, there is an opportunity to combine state-merging approaches with a lightweight pattern-mining algorithm to guide the learner toward robust model identification, even when using a limited amount of data.

In this chapter, we illustrated our proposed approaches to bridge this gap. First, we developed a method to analyse the provided data globally and identify the absence of sequential pairwise labels (Contribution **C1**). Then we designed and implemented learners that used these mined patterns in the existing passive learning algorithms (Contribution **C2**). Those include refining the selection criteria in the EDSM algorithm (Contribution **C2-1**), SAT encoding explicitly satisfies the input-enabled property of a Mealy machine (Contribution **C2-2**), and adapting the DFASAT algorithm into a more flexible heuristic framework (Contribution **C2-3**) to perform initial merges that respect our mined patterns, then execute the SAT solver on the remaining unmerged transitions. We also integrate pattern-based constraints into the CNF formula, enabling the SAT solver to find a solution consistent with the execution traces, the intermediate model, and the mined system properties (Contribution **C2-4**).

Furthermore, we outline alternative hybrid configurations that were implemented and evaluated but yielded no statistically significant improvement over the baseline DFASAT framework. These include integrating the DFASAT state-merging heuristic with our mined behavioural patterns for state merging (Contribution **C2-5**), as well as embedding these patterns directly as propositional constraints within the downstream SAT solver layer of the original DFASAT pipeline (Contribution **C2-7**). Finally, we applied the mined patterns to the EDSM state merging algorithm for initial merges and then applied them again to the SAT solver (Contribution **C2-6**).

Chapter 4

Experimental Setup

Our experimental setup was designed to address the research questions (see Section 3.1) by comparing three distinct learners: DFASAT (the baseline), PAT-EDSM, and PAT-DFASAT. Our third contribution (Contribution **C3**) involves evaluating our algorithms: PAT-EDSM - Algorithm 3, and PAT-DFASAT - Algorithm 4 against DFASAT (Algorithm 2). The evaluation validates the efficiency of the algorithm in terms of the runtime required to infer a model from the traces and the impact of using the mined patterns to perform more correct merges where states in the PTA refer to the same state in the target (original) model. Moreover, the evaluation validates the accuracy of the inferred model in terms of accepting the same language of the target system and the internal structure of the inferred model compared to the target model. In this chapter, we discuss the experimental setup and the evaluation methodology in detail.

Research Questions

- RQ1** How does incorporating mined patterns within EDSM and DFASAT algorithms improve the accuracy of the final model compared to DFASAT?
- RQ2** How does incorporating mined patterns within EDSM and DFASAT algorithms assist learners in making superior merging decisions?
- RQ3** How does incorporating mined patterns within EDSM and DFASAT algorithms affect the learning process?

4.1 Use Cases

Our experiments involved real-world and randomly generated automata to capture how learners perform across a variety of Mealy machines. The selected case studies comprise input-enabled, deterministic Mealy machines. These automata span various sizes and application domains. Real-world case studies from public automata learning libraries, such as AALpy [37] and LearnLib [28], are frequently used in the established automata learning literature. In contrast, the randomly generated automata are produced by StateChum [11], an open-source tool that uses the Forest-Fire algorithm [54]. A detailed description of random automata generation via the forest-fire algorithm can be found in Appendix A. Table 4.1 details all use cases with their state counts ($|Q|$), the input and output alphabet size ($|I|$ and $|O|$) and their number of transitions ($|\mathcal{T}|$). Note that $|\mathcal{T}| = |Q| \times |I|$ because all uses cases are deterministic input-enabled Mealy machines.

Table 4.1: *Use cases configuration used for the evaluation*

System Name	Library	$ Q $	$ I $	$ O $	$ \mathcal{T} = Q \times I $
Coffee Machine [6]	LearnLib	6	4	3	24
Bluetooth [41]	AALpy	15	7	11	105
TCP [22]	AALpy	14	10	11	140
OpenSSL [19]	LearnLib	6	7	7	42
ABP Sender [48]	LearnLib	19	9	10	171
Random 1	Generated By StateChum [11]	7	3	2	21
Random 2		12	4	3	48
Random 3		10	4	2	40
Random 4		9	4	2	36
Random 5		10	4	2	40
Random 6		11	4	3	44

4.2 Training Data

Passive learning depends on randomly generated walks (traces) for the learning (training) process. With entirely random walks, if the training data did not include a transition (incomplete data), then the passive learner cannot predict its presence. In practice, the completeness of the training set (where all transitions are explored) significantly impacts the inference outcome. A dense training set, one that covers a high percentage of the target machine’s transitions, allows the learner to identify states and consolidate the model more efficiently. Conversely, incomplete data leads to “islands” of information, making it difficult for the algorithm to distinguish between truly distinct states and those that are simply under-represented in the traces. Furthermore, relying on entirely random walks to achieve full transition coverage requires generating an excessively large number of traces or risks failing to visit deep, unrepresented transitions altogether when the dataset size or trace count is strictly controlled.

To overcome these limitations while preserving stochastic variation, our methodology employs a semi-smart, ϵ -greedy generation strategy that guarantees complete transition coverage within a minimal, high-value dataset. This approach maintains randomness when selecting inputs from the current state or choosing among unvisited transitions, ensuring the algorithm is not forced to follow a deterministic path during trace generation. By enforcing an initial transition-cover baseline before incrementally scaling the dataset with additional random traces, we establish a controlled framework that eliminates missing structural information while systematically evaluating learner performance across increasing data scales.

Our training traces are randomly generated from the use-case models and are excluded from the test-set traces used for evaluation (details later in Section 4.3.1). To address the lack of coverage in the training data, we generate the datasets in two steps:

- Step 1: We ensure that the trace-generation algorithm (Algorithm 5) covers all transitions during the initial exploration.
- Step 2: Once data completeness is reached, we incorporate additional random traces (see Algorithm 6) to provide the learner with additional information.

The “times” ($\times$) indicated alongside the use-case names in the results figures (in Chapter 5) represent the identified PTA sizes, where “Data” represents the minimum number of traces with transition cover. For instance, *Data* $\times$ 16 means that the size of the dataset after step 2 was 16

times that at step 1. The last entry in the results (the largest dataset) was chosen such that both DFASAT and PAT-DFASAT produce isomorphic automata.

We adopt the PTA size as a primary indicator for data scale because it directly reflects the structural complexity of the underlying trace set. This aligns with established literature in passive automata learning, such as the STAMINA competition benchmark [54] and the original DFASAT evaluation [27], which routinely measure instance size by state count. Nevertheless, we explicitly detail exact trace counts at the beginning of Chapter 5 for comprehensive comparison.

Generating the minimal dataset with transition cover

Algorithm 5 is essentially a semi-smart mapping strategy designed to explore every transition in a Mealy machine, without getting bogged down in endless loops. Instead of wandering around completely blindly, it uses a popular machine learning tactic called an ϵ -greedy strategy to balance focused execution with random exploration.

The algorithm keeps generating traces until every single transition in the input Mealy machine $\mathcal{M}$ has been checked off its master list (Line 4). A set of all transitions in $\mathcal{M}$ is retrieved by “Transitions($\mathcal{M}$)”. A transition is defined as a 4-tuple (source-state, input, output, target-state). In its current state, the algorithm retrieves the outgoing transitions from the current state by “Get_outgoing_transitions($\mathcal{M}$, current state)” and checks for the outgoing transitions it hasn’t visited yet (Line 11). If it finds any, it deliberately chooses one to explore the uncharted transitions (Line 12); the algorithm picks a random uncovered transition via “Random(uncovered)”. To keep from getting stuck in a local loop, it occasionally (when a random number $> \epsilon$) picks a completely random action from its available alphabet (Line 15).

If all outgoing transitions from the current state are already visited, the algorithm picks a random input (via Random(*Input_set*)) and retrieves the corresponding transition (via Get_Transition(current_state, input)).

Algorithm 5: Generating Guided Random Traces for Transition Coverage

Input : Deterministic input-enabled Mealy machine $\mathcal{M}$

Output: A list of traces (T) achieving full transition coverage

```

1  $\epsilon \leftarrow 0.2$  // Exploration factor
2  $T \leftarrow \emptyset$  // The set of traces
3  $visited \leftarrow \emptyset$  // The set of visited transitions
4 while  $|visited| < |Transitions(\mathcal{M})|$  do
5    $trace \leftarrow []$  // The current trace
6    $current\_state \leftarrow Initial\_state(\mathcal{M})$  // Every trace starts at the initial state
7    $continue \leftarrow \text{True}$  // A flag controls the length of the current trace
8    $discover\_new \leftarrow \text{False}$  // A flag confirms exploring a new transition
9   while  $continue \vee (|trace| < |States(\mathcal{M})|)$  do
10    // Get all outgoing transitions for the current state
11     $outgoing \leftarrow \text{Get\_outgoing\_transitions}(\mathcal{M}, current\_state)$ 
12     $uncovered \leftarrow \{t \in outgoing \mid t \notin visited\}$  // Exclude unvisited transitions
13    // Action Selection ( $\epsilon$ -greedy)
14    if  $uncovered \neq \emptyset \wedge \text{Random}(0, 1) > \epsilon$  then
15       $current\_transition \leftarrow \text{Random}(uncovered)$  // Select random transition
16    else
17      // If all transitions from the current state are already covered:
18       $input \leftarrow \text{Random}(Input\_set)$  // Select a random input
19       $current\_transition \leftarrow \text{Get\_Transition}(current\_state, input)$ 
20    Append  $current\_transition$  to  $trace$  // Add the transition to the trace
21    if  $current\_transition \notin visited$  then
22       $discover\_new \leftarrow \text{True}$  // Update Coverage Metric for the trace
23       $visited \leftarrow visited \cup current\_transition$  // Update the visited transitions
24       $current\_state \leftarrow \text{Next\_state}(current\_transition)$  // Update the current state
25       $continue \leftarrow (\text{Random}(0, 1) \geq 0.5)$  // Coin_Toss for random termination
26    if  $discover\_new$  then
27      // Confirm trace only if it discovered a new transition
28       $T \leftarrow T \cup trace$ 
29 return  $T$ 

```

A single trace continues sequentially until a stochastic coin toss (Line 22) terminates the sequence, subject to a mandatory minimum path length constraint to prevent premature truncation (Line 9), where “States($\mathcal{M}$)” retrieves the complete state set of $\mathcal{M}$. However, this stopping condi-

tion implicitly assumes that the target model contains cyclic topologies, in which states are mutually accessible and reachable, and includes explicit reset transitions returning from certain operational states to the initial state (excluding sink states designed to handle unexpected inputs). Models lacking cycles and loops fail to efficiently capture real-world software systems, which inherently rely on reset mechanisms and interconnected transitions between distinct functional operations.

The Quality Control Filter at Line 23 keeps the final output clean and minimal; the algorithm doesn't just save every trace it takes. At the end of a path, the trace must reveal a brand-new transition; otherwise, it is discarded. This ensures that only a list of high-value, unique traces that actively contributed to achieving 100% transition coverage is included.

Algorithm 6: Generating Additional Random Traces

Upon establishing this minimal training suite, the generation algorithm (Algorithm 6) synthesises random traces until the dataset reaches the designated target size. This data introduces structural variety, allowing the learner to be more confident about merging decisions and construct a highly accurate model of the system. To generate an individual trace, the algorithm executes the following sequence: the process starts at the system's initial state (Line 3), selects a random input symbol (via `Random(Input_set)`), determines the transition corresponding to the specified input, and finally, appends the new transition to the current trace. The next transition starts at the target state of the last transition. This iterative traversal continues until the trace satisfies a minimum length threshold, defined as twice the number of states in the target model ($|States(\mathcal{M})| \times 2$). This threshold ensures that paths are sufficiently deep to capture complex long-range behaviours rather than terminating prematurely.

Once the trace exceeds the minimum length threshold, the algorithm evaluates a fair coin toss (a stopping condition with a probability of 0.5) before each subsequent step to determine whether to continue the walk or terminate the trace. Ultimately, insert the newly generated trace into the training set (T). This generation process repeats iteratively until the training dataset satisfies the specified target size.

Algorithm 6: Add More Random Traces

Input : Set of existing traces T , Target size $size$, Deterministic input-enabled Mealy machine $\mathcal{M}$

Output: Updated set of traces T extended with random traces

```

1 while  $|T| < size$  do
2    $trace \leftarrow []$  // The current trace
3    $current\_state \leftarrow Initial\_state(\mathcal{M})$  // Every trace starts at the initial state.
4    $continue \leftarrow \text{True}$  // A flag controls the length of the current trace
5   while  $continue \vee |trace| < (|States(\mathcal{M})| \times 2)$  do
6      $input \leftarrow \text{Random}(Input\_set)$  // Select a random input
7      $current\_transition \leftarrow \text{Get\_Transition}(current\_state, input)$ 
8     Append  $current\_transition$  to  $trace$  // Add the transition to the trace
9      $current\_state \leftarrow \text{Next\_state}(current\_transition)$  // Update the current state
10     $continue \leftarrow (\text{Random}(0, 1) \geq 0.5)$  // Coin_Toss for random termination
11  end
12   $T \leftarrow T \cup trace$ 
13 end
14 return  $T$ 

```

Table 4.2 summarises the experimental setup for both use case categories (real-world and randomly generated automata). We run each experiment for each data size 10 times with a different set of traces (generated by Algorithms 5 and 6).

Case Study	Number of Case Studies	Q	I	O	Training Data Size	Number of independent runs per data size
Real World	5	6 to 19	4 to 10	4 to 10	Smallest dataset: The smallest set that achieves the transition cover Increasing the dataset: duplicate the smallest dataset until PAT-DFASAT and DFASAT produce identical results	10
Random Generated	6	7 to 12	3 to 4	3 to 2		

Table 4.2: Summary of the experimental setup

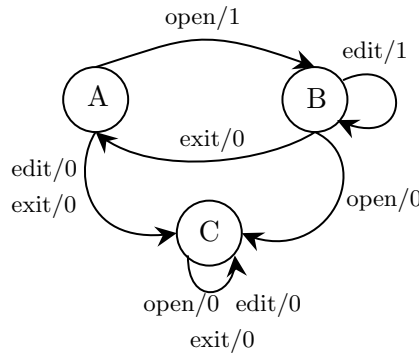
4.3 Evaluation Methodology

Evaluating inferred automata provides a quantitative measure of a learner’s performance, typically determined by how closely the inferred model approximates the target system. Evaluation metrics often vary between methodologies; while some focus on accuracy, others prioritise efficiency, with researchers frequently employing diverse terminology to define these factors.

A model’s accuracy can be assessed through either its language or its structure:

- **Language Evaluation:** This is the most common approach in the literature. It assesses the model’s ability to correctly classify a set of test traces as “positive”: belonging to the system language, or “negative”: not belonging. These test traces should not be included in the training set. The learner’s success is measured by its predictive power over the system’s observable behaviour.
- **Structural Evaluation:** this approach compares the topology of the inferred graph directly against the target model.

If the reference automaton is known, a graph comparison (such as bisimulation or state-mapping) can be utilised. If the target is hidden or unknown, test traces are employed to estimate structural fidelity.

Figure 4.1: The Target Model $\mathcal{M}$ (same as Figure 2.1)

Example

Let us consider three automata: $\mathcal{M}_1$ (Figure 4.2a), $\mathcal{M}_2$ (Figure 4.2b), and $\mathcal{M}_3$ (Figure 4.2c), which

were inferred using three distinct algorithms (For the sake of this example, we assume that the three models are not necessarily inferred using our proposed experimental approaches). All three models represent attempts to learn the identical target reference machine shown in Figure 4.1. The first learner over-merged all states and transitions into a single-state automaton, resulting in $\mathcal{M}_1$. Conversely, the second learner under-merged states, yielding a structure identical to the initial prefix tree, $\mathcal{M}_2$. Finally, the third learner, $\mathcal{M}_3$, captures an intermediate structural state with one transition label error: the self-loop around state S2 in $\mathcal{M}_3$ results in output (0) for the input *edit* while the correct output is (1). To evaluate the language accuracy of these three configurations, we utilise the following test positive and negative traces (see Definition 3):

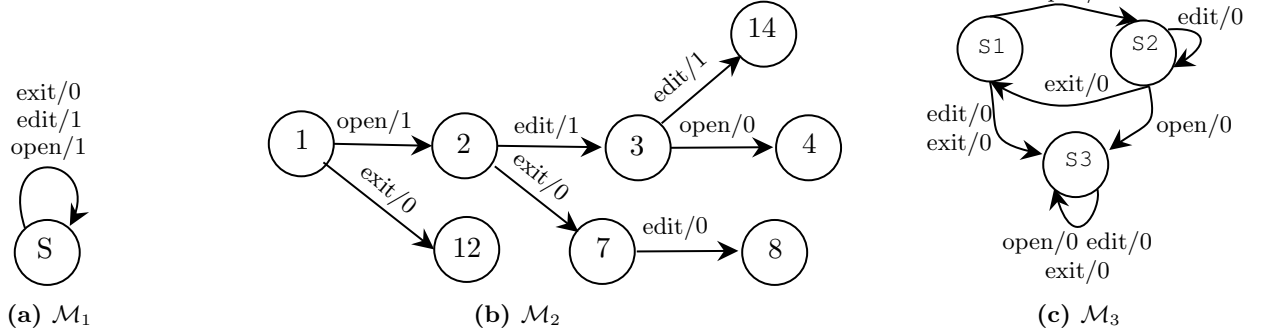


Figure 4.2: Examples of three different learned models

- **Positive Traces:**

$$t_1 = \langle \text{open}/1, \text{edit}/1, \text{edit}/1, \text{exit}/0 \rangle$$

$$t_2 = \langle \text{open}/1, \text{exit}/0, \text{open}/1 \rangle$$

- **Negative Traces:**

$$t_3 = \langle \text{open}/1, \text{edit}/1, \text{edit}/0 \rangle$$

$$t_4 = \langle \text{open}/1, \text{edit}/0 \rangle$$

Evaluating these traces against each model yields the following classification behaviours:

- $\mathcal{M}_1$ correctly accepts t_1 and t_2 , and correctly rejects t_3 and t_4 .
- $\mathcal{M}_2$ erroneously rejects all valid accepted traces (t_1 and t_2) while successfully rejecting the invalid ones (t_3 and t_4).
- $\mathcal{M}_3$ correctly classifies t_2 and t_3 , but misclassifies t_1 and t_4 .

The severe under-merging in $\mathcal{M}_2$ causes the model to reject the valid language space, despite successfully disallowing the rejected strings. Conversely, the extreme over-merging in $\mathcal{M}_1$ generalises the language correctly but fails entirely to capture the underlying topology and constraints of the target structure. Meanwhile, $\mathcal{M}_3$ recovers a nearly accurate automaton structure yet still fails to accurately handle 50% of the evaluation traces.

Evaluating inferred software models requires different metrics depending on the practical goal. Structural metrics measure how cleanly a model represents a system's overall architecture, while behavioural metrics check whether a model accurately follows specific execution rules. Choosing which measure matters most comes down to how the model will be used.

When the goal is system understanding and reverse engineering, structural clarity is most important. Engineers documenting legacy software or learning an unfamiliar system need a clean, readable blueprint. In this scenario, metrics that reward minimal states and clear transitions take priority, even if the model simplifies or misses a few rare edge cases. A highly accurate model with thousands of cluttered, redundant states is useless for human comprehension.

Conversely, when the goal is safety verification, property-driven compliance takes priority over structure. For safety-critical software, like medical devices or autonomous vehicle controllers, testers must prove the system can never reach a dangerous state. In this case, it does not matter if the model is structurally complex or missing non-critical features, as long as it precisely models the safety rules.

Ultimately, structural measures help humans understand how a system is organised, while property measures ensure the system strictly obeys its critical safety boundaries. Aligning the evaluation metric with the practical engineering objective ensures the model delivers real value.

For a fair evaluation, it is critical to evaluate the result with respect to both language and structure. A model may achieve a high classification accuracy (language) while remaining structurally distinct from the reference automaton. Conversely, a model may be structurally similar but fail to classify traces correctly due to a single misplaced transition. The language evaluation is inherently more “forgiving” than the structure comparison in scenarios where numerous states are left unmerged, yet the observed language remains mostly correct. Conversely, the language evaluation may drop significantly while structure comparison remains high if most states are merged correctly, but a few critical errors occur near the initial state, drastically altering downstream trace acceptance. By employing a dual-evaluation strategy, we can gain a comprehensive understanding of the learner’s strengths and limitations, ensuring a fair assessment of the resulting model.

Consequently, we specified the accuracy of our first Research Question **RQ1** as follows:

RQ1-(A) How does incorporating mined patterns within EDSM and DFASAT algorithms improve the language accuracy (F1) of the final model?

RQ1-(B) How does incorporating mined patterns within EDSM and DFASAT algorithms improve the structural accuracy (Structural Difference) of the final model?

4.3.1 Language Evaluation

Generating Random Test Traces

Evaluating the language accuracy of the inferred models (**RQ1-(A)**) involves two categories of test sequences: valid and invalid random traces. These traces are excluded from the training dataset. Notably, while we require the training dataset to achieve full transition coverage (see Section 4.2), we do not impose such coverage requirement on the test traces. The structural layout of the evaluated case studies (see Section 4.1) incorporates a *sink-state*, which is entered whenever an invalid input is supplied (see Section 2.1.1). A sink-state is uniquely defined as a state with no outgoing transitions to any other state, featuring only self-loops for every input in the input alphabet set. Consequently, **for evaluation purposes only**, we consider any trace that terminates at the sink state to be an invalid trace.

All invalid traces of the evaluation set are prefix-closed. A prefix-closed trace is a sequence of valid transitions followed by one invalid transition. Given that any transition entering a sink state is invalid (does not belong to the system’s logic as discussed earlier), a prefix-closed trace is one that terminates either at the sink state or results in output that deviates from the correct definition

for that specified input. Conversely, a valid trace terminates at a valid state (i.e., any state other than the sink-state). The size of each set (valid/invalid) of the test traces is four times the training set size:

$$Valid_Test_Traces_size = Training_size \times 4$$

$$Invalid_Test_Traces_size = Training_size \times 4$$

Algorithm 7 outlines the random generation of valid test traces from a deterministic input-enabled Mealy machine. Each trace originates at the initial state (q_0), as shown in Line 6, and is assigned a randomised minimum length to ensure a balanced distribution of both short ($|Q|/2$) and long ($|Q| \times 2$) execution scenarios (Line 8). As the algorithm traverses the Mealy machine by selecting random inputs, the sink-state is treated as restricted; if a random transition leads to the sink-state, the algorithm discards the step and forces a retry from the current state to preserve path validity (Line 13). Once the path satisfies the minimum length requirement, a stochastic termination condition (Line 16) determines whether to conclude the trace. Before any completed path is appended to the final evaluation suite, it must pass a redundancy filter (Line 17): it can neither duplicate an existing evaluation sequence nor the training set. This ensures that the test set contains only valid traces that are not used for training.

A limitation of Algorithm 7 is that it requires all valid states to maintain outgoing transitions to other valid states. In the edge case where the model contains a state whose outgoing transitions lead exclusively to a sink state, the trace generation process will encounter a deadlock. Consequently, synthesising long evaluation traces under such structural constraints becomes impossible.

Algorithm 7: Random Valid Test-Traces Generator

Input : Target dataset size *size*, Deterministic input-enabled Mealy machine $\mathcal{M}$
1 , Set of training traces *Training_T* **Output:** Set of valid test traces *Valid_Test_T*
2 $S_{sink} \leftarrow \text{find_sink_state}(\mathcal{M})$
3 $Valid_Test_T \leftarrow []$
4 **while** $|Valid_Test_T| < size$ **do**
5 $trace \leftarrow []$ // The current trace
6 $current_state \leftarrow \text{Initial_state}(\mathcal{M})$ // Every trace starts at the initial state
7 $continue \leftarrow \text{True}$
 // Set the minimum length to a random value
8 $current_length_limit \leftarrow \text{Random}(|States(\mathcal{M})|/2, |States(\mathcal{M})| \times 2)$
9 **while** $continue \vee (|trace| < current_length_limit)$ **do**
10 $input \leftarrow \text{Random}(Input_set)$ // Select a random input
11 $current_transition \leftarrow \text{Get_Transition}(current_state, input)$
12 $next_state \leftarrow \text{Next_state}(current_transition)$ // Retrieve next-state
13 **if** $next_state \neq S_{sink}$ **then**
 // Append the transition if next-state is not a sink-state
14 **Append** $current_transition$ **to** $trace$
15 $current_state \leftarrow \text{Next_state}(current_transition)$
16 $continue \leftarrow (\text{Random}(0, 1) \geq 0.5)$
 // Verification against the Training set
17 **if** $trace \notin Training_traces$ **then**
18 $Valid_Test_T \leftarrow Valid_Test_T \cup trace$
19 **return** $Valid_Test_T$

Algorithm 8 outlines the generation of an invalid prefix-closed trace, which requires generating

a sequence of valid transitions and terminating with a final invalid step. Consequently, we use Algorithm 7 to generate a sequence of valid transitions, then execute a final invalid step with Algorithm 8. Algorithm 8 starts by identifying the terminal state in the valid sequence (Line 3). Next, selecting a random input from the input alphabet set for this terminal state. If the transition under the picked input targets the sink-state, this final transition is directly appended to the trace. Otherwise, the algorithm randomly assigns an incorrect output from the output alphabet set that differs explicitly from the correct output defined in the target reference model (Line 9).

Algorithm 8: Adding a Final Invalid Transition to a Test Trace

Input : A valid trace $Valid_T$, $sink_state$, Deterministic input-enabled Mealy machine $\mathcal{M}$

Output: A Prefix-Closed Invalid Trace $Invalid_Test_T$

// An invalid-trace is prefixed with a valid-trace

```

1  $Invalid\_Test\_T \leftarrow valid\_T$ 
2  $current\_state = Get\_Last\_State(Valid\_T)$  // Get the termination state of Valid_T
3  $input = Random(Input\_set)$ 
4  $current\_transition \leftarrow Get\_Transition(current\_state, input)$ 
5  $next\_state \leftarrow Next\_state(current\_transition)$  // Retrieve next-state
6 if  $next\_state = sink\_state$  then
7    $invalid\_transition \leftarrow Get\_Transition(current\_state, input)$ 
8 else
9    $Incorrect\_outputs = \{o \in O \mid o \neq \lambda(current\_state, input)\}$ 
10  if  $Incorrect\_outputs = \emptyset$  then
11    Go to line 3 and try again // If  $|O| = 1$ 
12   $output \leftarrow Random(incorrect\_outputs)$ 
13   $invalid\_transition \leftarrow (current\_state, input, output, None)$ 
14 Append  $invalid\_transition$  to  $Invalid\_Test\_T$ 
15 return  $Invalid\_Test\_T$ 

```

Line 11 handles an edge case where $|O| = 1$. In this scenario, if the randomly selected input transition does not lead to a sink state, the algorithm selects an alternative input and retries the step. This introduces a further structural limitation: if a state lacks invalid transitions (i.e., those leading to a sink state) and the output alphabet contains only a single character, the trace generation process will encounter a deadlock.

F1-Score

To quantify language accuracy, the generated test traces are executed against the inferred model. Each evaluation sequence is classified into one of the following four categories (summarised in Table 4.3):

- **True Positive (TP):** A valid trace by the target model is truly classified as valid by the inferred model.
- **False Positive (FP):** An invalid trace by the target model is incorrectly classified as valid by the inferred model.
- **True Negative (TN):** An invalid trace by the target model is truly classified as invalid by the inferred model.
- **False Negative (FN):** A valid trace by the target model is incorrectly classified as invalid by the inferred model.

Precision represents the ratio of true positives to the total number of positive classifications

		Inferred Model	
		Positive	Negative
Target Model	Positive	Ture Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	Ture Negative (TN)

Table 4.3: The Confusion Matrix

predicted by the model, capturing the proportion of predicted acceptances that are genuinely valid. It provides a response to the query, “How many correct predictions were made out of all positive predictions?” This metric is formally defined in Equation 4.1:

$$\text{Precision} = \frac{TP}{TP + FP} \quad (4.1)$$

Recall, or the true positive rate, measures a learner’s capacity to identify all valid sequences within the target language space. It quantifies the proportion of actual accepted traces that are correctly identified by the inferred model, as calculated via Equation 4.2:

$$\text{Recall} = \frac{TP}{TP + FN} \quad (4.2)$$

The $F1$ -score computes the harmonic mean of precision and recall, as formulated in Equation 4.3. Given the inherent trade-off between precision and recall in model inference, the $F1$ -score serves as a balanced performance metric. A key property of this harmonic mean is that if either precision or recall approaches zero, the overall score drops to zero, thereby penalising poor performance in either individual component. $F1$ -score ranges between 0 and 1.

$$F1 = \frac{(2 * \text{precision} * \text{recall})}{(\text{precision} + \text{recall})} \quad (4.3)$$

4.3.2 Structural Evaluation

The structural evaluation assesses the inferred model based on the specific topology of states and transitions that constitute its internal logic. To address the second half of the first Research Question(**RQ1-(B)**), we evaluate the structural accuracy of the inferred models using the “Structural Difference” (SD) tool [53]. The tool is provided by the open-source framework StateChum [11]. The Structural Difference is the score that indicates the similarity between two state machines in terms of their internal structure (i.e., the similarity between states and their transitions). We use it to calculate similarity scores between states of the inferred model and their corresponding states in the target model. This comparison of behavioural overlap follows a recursive methodology:

- **Local Similarity:** determines the overlap of immediate transitions. This metric represents the ratio of shared adjacent transitions to the total number of adjacent transitions.
- **Global Similarity:** adds weighted similarity of the target and source states of these transitions to the local similarity. By aggregating the similarity scores of successive matched transition pairs, for the broader topological context, a better approximation is established.

After that, the algorithm identifies the state pairs with the highest scores to serve as foundational key pairs. These key pairs act as reference points for the mapping process. Beginning with these highly evident, equivalent nodes, the algorithm evaluates and matches the surrounding states and transitions recursively until no further common structural elements can be identified. The remaining unmapped components define the structural difference between the two machines. The yielding score ranges from 0 (complete divergence) to 1 (perfectly isomorphic automata). For example:

$$SD(\mathcal{M}, \mathcal{M}_1) = 0.22 (\text{Figure 4.2a})$$

$$SD(\mathcal{M}, \mathcal{M}_2) = 0.25 (\text{Figure 4.2b})$$

$$SD(\mathcal{M}, \mathcal{M}_3) = 0.88 (\text{Figure 4.2c})$$

Example

The Structural Difference algorithm takes two models: a target (reference) model and another (inferred in this example) model. Consider the two models in Figure 4.3, where the target model $\mathcal{M}$ is shown in Figures 4.3a and the inferred model $\mathcal{M}_3$ is shown in Figure 4.3b. The algorithm computes a score indicating how closely the internal structure of the inferred model matches that of the target model.

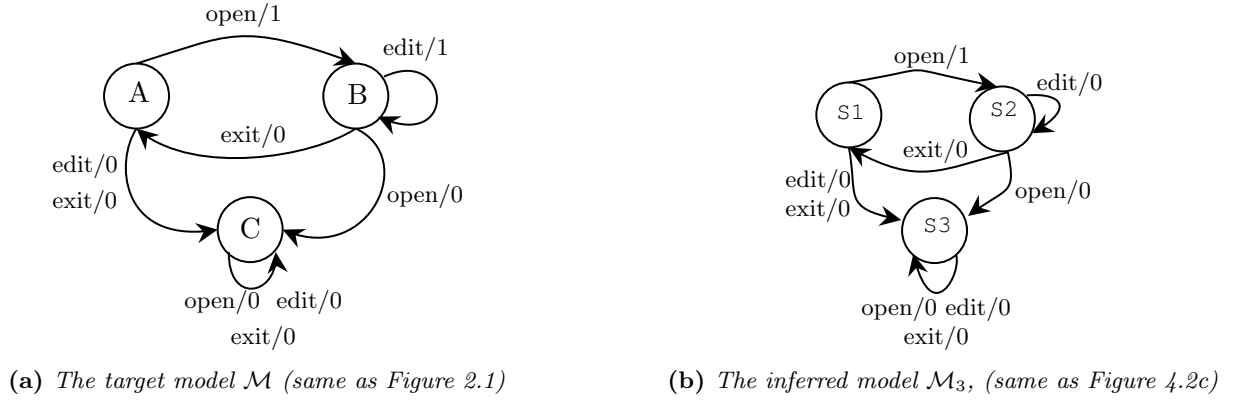


Figure 4.3: The target and the inferred models

The structural difference algorithm starts by comparing all pairs of states between the two models. The total number of pairs is the product of the number of states of the two models ($|Q_{\mathcal{M}}| \times |Q_{\mathcal{M}_3}|$). Each pair is scored based on direct incoming and outgoing transitions (local similarity) and the next states (global similarity). The local similarity (LS) of the pair (x, y) is calculated by:

$$LS(x, y) = \frac{|matching\ adjacent\ transition\ between\ x\ and\ y|}{|all\ adjacent\ transitions\ for\ y|}$$

For example, the local similarity of the outgoing transitions of pair $(A, S1)$ is $LS_{outgoing} = \frac{3}{3} = 1$; S1 has 3 outgoing transitions which match the outgoing transitions from state A. The local similarity of the incoming transitions of pair $(A, S1)$ is $LS_{incoming} = \frac{1}{1} = 1$. For the pair $(B, S1)$, $LS_{outgoing} = \frac{1}{3}$ and $LS_{incoming} = \frac{0}{1}$; S1 and B match one outgoing transition (i.e., exit/0) out of 3, and there are no shared incoming transitions between the two states.

To calculate global similarity, the algorithm extends local similarity by recursively evaluating the similarity of the source and target states connected to each matching transition. For two matched transitions, a higher score is produced if the source/target states of these transitions are almost equivalent, and a lower score if they are dissimilar. An attenuation ratio k is used so that closer neighbour states have a stronger influence than distant ones. For each state pair, an equation is constructed using the attenuation formula:

$$GS(x, y) = \frac{1}{2} \cdot \frac{|mathed transition(x, y)| (1 + k \cdot GS(x1, y1))}{|Unmathed transition(x, y)| + |mathed transition(x, y)|}$$

Where $x1$ and $y1$ are the source/target states for x and y , respectively.

For example, the pair $(A, S1)$ has 3 outgoing transitions match and point to targets $(B, S2)$ via open/1, and $(C, S3)$ via edit/0 and exit/0. using attenuation factor $k = 0.6$, the global similarity of $(A, S1)$ is:

$$GS_{outgoing}(A, S1) = \frac{1}{2} \cdot \frac{(1 + 0.6 \cdot GS_{outgoing}(B, S2)) + 2 \cdot (1 + 0.6 \cdot GS_{outgoing}(C, S3))}{0 + 3}$$

For each pair of states, the global similarity is calculated for both source and target states (i.e., for incoming and outgoing transitions). Calculating the global similarity for the source states is similar to that for the target states. The overall similarity score for each state pair is computed by averaging the similarity of incoming and outgoing matching transitions.

Among all pairs, the pairs with the highest scores are chosen as key pairs and used as reference points to compute the differences between the two models. For example, state S1 in $\mathcal{M}_3$ is paired with every state in $\mathcal{M}$. The same applies to states S2 and S3 in $\mathcal{M}_3$.

Among these pairs, the highest-scoring pairs are marked as key pairs. Determining the key pair depends on two factors:

- The score of the key pair must be one of the top 25% of all scores
- The second score of the key pair (the score to the second matching pair) is at least 50% less than the highest score.

In this example, state S1 is marked as a key pair. Figure 4.4 shows the scoring between each state in $\mathcal{M}_3$ and equivalent states in the reference model $\mathcal{M}$. For example, state S1 is marked with the notation $(K\ 12, 5 = A)[A]$, where the score for the pair $(S\ 1\ in\ \mathcal{M}_3, A\ in\ \mathcal{M}) = 12$, “K” means that the pair $(S1, A)$ is a key pair, and the digit 5 is the second highest score of another pair of S1 and another state (not known from the visualisation).

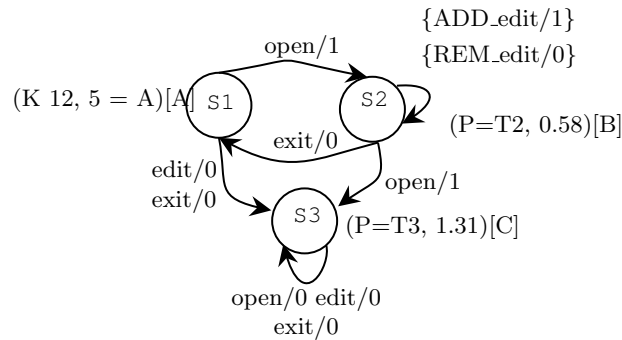


Figure 4.4: Visualising the Structural Difference between $\mathcal{M}$ and $\mathcal{M}_3$ shown in Figure 4.3

The key pairs are used to compare the surrounding areas until no further pairs of states or transitions can be found. Every adjacent state to S1 in $\mathcal{M}_3$ is compared to every adjacent state to state A in $\mathcal{M}$. For example, beside S3, the notation $(P=T3, 1.31)[C]$ in Figure 4.4 means S3 in $\mathcal{M}_3$ is the most similar to state C in $\mathcal{M}$; the similarity score for the pair $(S3, C) = 1.31$, which is the highest score among all compared pairs for state S3. The same applies to S2 in $\mathcal{M}_3$ and its most similar state B in $\mathcal{M}$.

For S2, there is a notion of the self-loop: ADD.edit/1 and REM.edit/0, which means that for $\mathcal{M}_3$ to be identical to $\mathcal{M}$, a self-loop for input edit and output 1 must be added, and the existing self-loop for input edit and output 0 must be removed.

The structural difference score is determined by the size of the minimum number of edit operations (adding or removing transitions and states) required to make the models identical. The structural difference score (edit distance) between these two models is $SD(\mathcal{M}, \mathcal{M}_3) = 0.88$.

4.3.3 Pairwise Evaluation

Vargha–Delaney (A12)

The Vargha–Delaney statistic [50], also known as A_{12} , is a non-parametric effect size statistic widely used in software engineering experiments, empirical studies, and data analysis. It quantifies how often values from one distribution (e.g., treatment A) are greater than values from another distribution (e.g., treatment B). In other words, having two groups of values (A and B), A12 gives us a number that indicates which group has higher values as follows:

- Groups are statistically similar if $A_{12} = 0.5$
- Group A tends to have larger values if $A_{12} > 0.5$
- Group B tends to have larger values if $A_{12} < 0.5$

The A_{12} can be calculated using Equation 4.4, where $|A| = m$, and $|B| = n$

$$A_{12} = \frac{\sum_{i=1}^m \sum_{j=1}^n \phi(a_i, b_j)}{m \times n} \quad (4.4)$$

Where:

$$\phi(a, b) = \begin{cases} 1 & a > b \\ 0.5 & a = b \\ 0 & a < b \end{cases} \quad (4.5)$$

Worked Example: Evaluating Learners via the Vargha–Delaney statistic (A12)

Let us assume we inferred a model using two different learning algorithms (X and Y) and used five different datasets of the same size to evaluate both learners. For each dataset, we measured the $F1$ -score, yielding two sets (X_{F1} and Y_{F1}) where each value represents the $F1$ -score for a given dataset.

$$X_{F1} = \{0.3, 0.3, 0.5, 0.6, 0.3\}$$

$$Y_{F1} = \{0.6, 0.4, 0.3, 0.5, 0.3\}$$

We want to assess how learner X performs compared to learner Y . Since each group has 5 values, there are $5 \times 5 = 25$ total pairings. To evaluate this, we apply the A_{12} metric by comparing every value in X_{F1} to each value in Y_{F1} as shown in Equation 4.5. For instance, comparing the first element of X_{F1} (0.3) against all elements in Y_{F1} yields:

$$\phi(0.3, 0.6) = 0, \quad \phi(0.3, 0.4) = 0, \quad \phi(0.3, 0.3) = 0.5, \quad \phi(0.3, 0.5) = 0, \quad \phi(0.3, 0.3) = 0.5$$

The sum of these comparisons is 1. We repeat this exact process for all $x \in X$ against all $y \in Y$, and then apply Equation 4.4 to find the final result. As shown in Equation 4.6, $A_{12} = 0.44$, which indicates that learner X is slightly inferior to learner Y . Table 4.4 details how the A_{12} value is interpreted.

$$A_{12} = \frac{1 + 1 + 3.5 + 4.5 + 1}{5 \times 5} = 0.44 \quad (4.6)$$

A_{12} value (distance from 0.5)	Interpretation
0.56–0.64 or 0.44–0.36	Small effect
0.64–0.71 or 0.36–0.29	Medium effect
> 0.71 or < 0.29	Large effect

Table 4.4: Interpretation of the A_{12} value

Bootstrap Confidence Intervals for Non-Parametric Effect Sizes

A confidence interval (CI) provides a range of plausible values for a population parameter rather than relying solely on a single point estimate. Standard statistical approaches often rely on parametric assumptions, such as normally distributed data. However, performance metrics in machine learning and software engineering (such as F_1 -score, execution runtime, or Structural Differences) frequently exhibit skewness or bounded scales.

The **Bootstrap Confidence Interval** (Bootstrap CI) [21] is a non-parametric procedure. It estimates the sampling distribution of a statistic by repeatedly resampling from the observed data with replacement. The bootstrap CI is particularly useful because it operates without distributional assumptions and does not require the data to follow a normal distribution or any specific theoretical model. This flexibility makes it ideal for small or non-standard samples where analytical variance formulas are complex, unknown, or unreliable. Furthermore, it serves as a robust companion to non-parametric metrics such as Vargha and Delaney’s A_{12} statistic; since A_{12} evaluates ordinal ranks rather than continuous parameters, traditional parametric confidence intervals do not directly apply, making bootstrapping an effective method for accurately estimating its precision.

Let an observed sample consist of n observations. The percentile bootstrap method proceeds as follows:

1. **Resample with Replacement:** Randomly select n observations from the original dataset with replacement to form a bootstrap sample.
2. **Compute the Statistic:** Calculate the metric of interest (e.g., mean, median, or A_{12}) on this bootstrap sample.
3. **Repeat:** Reiterate steps 1 and 2 a total of B times (typically $B = 1000$ or $B = 2000$) to yield B bootstrapped estimates: $[S_1^*, S_2^*, \dots, S_B^*]$.
4. **Extract the Interval:** Sort the B bootstrapped values in ascending order. For a 95% confidence interval using the percentile method, select:
 - **Lower Bound:** The 2.5th percentile value (position $0.025 \times B$).
 - **Upper Bound:** The 97.5th percentile value (position $0.975 \times B$).

Interpretation for A_{12} effect size

Because the no-effect value for A_{12} is (0.50), the observed performance difference between the two sets of values at the 5% level is:

- **Statistically Significant:** If the no-effect value (0.50) **is not included** in the range of [lower_bound, upper_bound].
 - Systematic superiority of Group 1: When the entire range is > 0.50 (e.g., [0.67, 1.00]). Resampling demonstrates that under nearly every sub-configuration of data, Group 1 holds the upper hand.
 - Systematic superiority of Group 2: When the entire range is > 0.50 (e.g., [0.04, 0.34]). Group 1 wins only a tiny fraction of comparisons.
- **Statistically Not Significant:** If the no-effect value (0.50) **is included** in the range of [lower_bound, upper_bound].

The width of the confidence interval measures precision and stability:

- **Wide Range:** Represents high uncertainty and low precision, where individual data points exert strong leverage on ranks. It is common with small sample sizes or high internal variance. For example, $A_{12} = 0.87$ and $CI=[0.67, 1.00]$ (width = 0.33) means Group 1 is consistently superior, but the exact magnitude of dominance is imprecise. It could be moderate (0.67) or absolute (1.00).
- **Small Range:** Represents high precision and high stability, which indicates large, stable sample sizes or very low variance across resamples. For example, $A_{12} = 0.87$ and $CI=[0.82, 0.91]$ (width = 0.09) demonstrates high confidence in both the existence and the exact scale of the superiority.

Example

This example illustrates how to calculate the Bootstrap CI for the A_{12} effect size. Consider evaluating two learning algorithms (X and Y) across five datasets using $F1$ -score:

$$X_{F1} = \{0.3, 0.3, 0.5, 0.6, 0.3\}$$

$$Y_{F1} = \{0.6, 0.4, 0.3, 0.5, 0.3\}$$

- **Step 1:** Calculate the A_{12} score (as demonstrated previously):

$$A_{12}(X_{F1}, Y_{F1}) = 0.44$$

- **Step 2:** Resampling Iterations (Illustrative)

- **Iteration 1:** Resample $X^{*(1)} = \{0.3, 0.5, 0.5, 0.6, 0.3\}$ and $Y^{*(1)} = \{0.4, 0.3, 0.5, 0.5, 0.3\}$:

$$A_{12}^{*(1)} = 0.60$$

- **Iteration 2:** Resample $X^{*(2)} = \{0.3, 0.3, 0.3, 0.3, 0.5\}$ and $Y^{*(2)} = \{0.6, 0.6, 0.4, 0.5, 0.3\}$:

$$A_{12}^{*(2)} = 0.18$$

- **Step 3:** Interval Extraction ($B = 1000$)

Repeating this procedure 1000 times creates a sorted array of bootstrapped estimates. Extracting the 50th (2.5th percentile) and 1950th (97.5th percentile) values yields an approximate interval of [0.18, 0.68]. Because the confidence interval spans across the no-effect baseline (0.50), the performance difference between learner X and learner Y does not reach statistical significance at the 5% level.

4.3.4 Proportion of Correct Merges

The proportion of correct merges (Research Question **RQ2**) serves as a primary indicator of both structural and language model accuracy. In this evaluation, we focus specifically on the ratio of correctly executed merges to the total number of completed merges during the EDSM phases, given that the downstream SAT solver is responsible for completing the remaining unmerged transitions. Ultimately, the primary objective of the heuristics embedded within both DFASAT and PAT-DFASAT is to strictly suppress incorrect merges during this initial phase, thereby presenting the SAT solver with a highly accurate partial automaton while preventing premature over-generalisation.

To implement this verification mechanism during learning, we instrument the PTA construction process by mapping each state in the PTA back to its unique reference state in the target model. As execution traces are parsed to populate the tree, every sequence path tracks its corresponding state trajectories within the target automaton. To illustrate this, consider the baseline traces and the corresponding PTA shown in Figure 4.5a. The initial state of the target model is $q_0 = A$; the root state of the PTA is mapped directly to state A . In Figure 4.5a, we visually distinguish these target mappings using state border styles, ensuring that each state in the PTA shares a visual identity of its target equivalent. For instance, following the first trace, the input-output event `open/1` triggers a transition from state A to state B in the target model; consequently, state 2 in the PTA is assigned a reference mapping to target state B . To formally evaluate the fidelity of the learning process, every candidate merge is validated against these underlying reference states; a correct merge is logged if and only if both candidate nodes map to the identical target reference state.

Positive Training Traces:

$\langle \text{open}/1, \text{edit}/1, \text{exit}/0, \text{open}/1, \text{open}/0 \rangle$
 $\langle \text{open}/1, \text{exit}/0, \text{edit}/0 \rangle$
 $\langle \text{open}/1, \text{exit}/0, \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{open}/0, \text{open}/0 \rangle$
 $\langle \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{open}/0, \text{exit}/0 \rangle$
 $\langle \text{open}/1, \text{edit}/1, \text{edit}/1 \rangle$

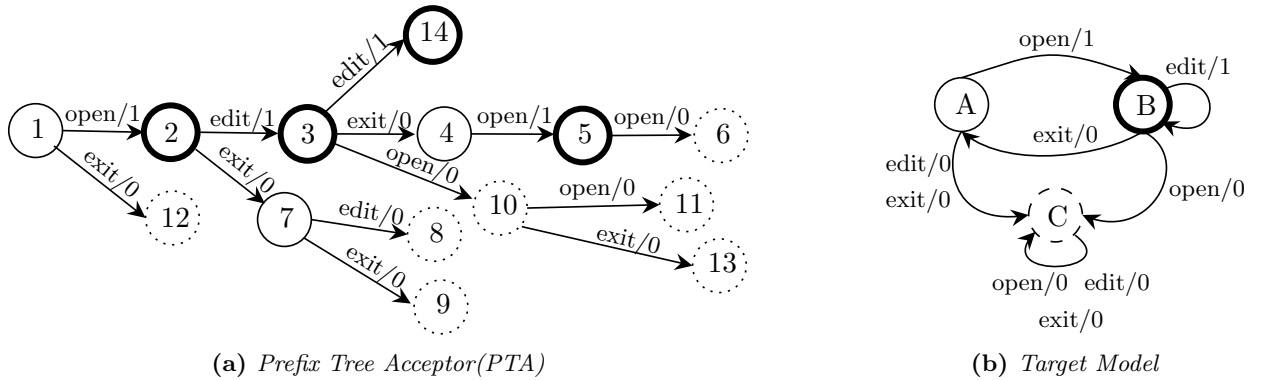


Figure 4.5: A PTA and its target model - (copy of Figure 2.5)

4.3.5 Execution Time

To address Research Question 3 (**RQ3**), we evaluate the execution runtime for each candidate learner. The execution time is the total time required to complete the model inference process. The timing mechanism begins immediately after the initial construction of the PTA, thus encompassing

the time required to mine behavioural patterns, execute heuristic merges, and converge on a final solution via the SAT solver (where applicable). Analysing execution times provides an alternative perspective for comparing learner performance. Specifically, we investigate the precise operational conditions under which a learner operates more efficiently, as well as the underlying structural factors that introduce computational latency.

4.4 Summary

In this chapter, we detail the experimental setup and evaluation methodology designed to systematically address the core research questions of this thesis. The first research question evaluates the performance of the proposed algorithms in terms of both structure and language accuracy. To evaluate the structural topology (**RQ1-(B)**), we utilise a structural difference metric that measures the difference between transitions relative to the target model. Conversely, for the language-level evaluation (**RQ1-(A)**), we assess the models using independent sets of valid and invalid traces to compute an $F1$ -score, which quantifies how precisely the inferred model captures the target language specification.

Furthermore, we measure the proportion of correctly executed state merges (**RQ2**) relative to the total number of merges performed during the learning process. This proportion effectively captures the precise quality and fidelity of our pattern-guided merging heuristics compared to the baseline DFASAT framework. From another perspective, we measure execution time and evaluate (**RQ3**) the performance of the learners based on the total processing time. Finally, we evaluate our algorithms using Vargha and Delaney’s A_{12} effect size for pairwise comparisons. This statistical analysis aims to provide a robust overview of the systematic performance advantages and overall performance disparities between any two inference methods.

Chapter 5

Results and Discussion

This chapter presents the empirical results of the evaluated model-inference frameworks for all experiments conducted in each case study, and critically discusses their performance across various operational contexts. Specifically, we analyse how our primary proposed pattern-based learners, PAT-EDSM and PAT-DFASAT, perform relative to the baseline DFASAT across varying data scales, beginning with real-world software systems and concluding with randomly generated automata. By rigorously evaluating metrics for Structural Difference (SD), language acceptance, merging precision, and runtime, the discussion highlights how pattern-based constraints successfully mitigate the challenges posed by data sparsity. Finally, we evaluate and address potential threats to internal and external validity, thereby contextualising the reliability and robustness of our empirical conclusions.

5.1 Results

To address Research Questions **RQ1** and **RQ2**, we first present the distribution of raw values across our three evaluation metrics: the $F1$ -score for language acceptance, the Structural Similarity (SD) score, and the proportion of correct state merges. Next, we introduce the Vargha and Delaney A_{12} effect-size statistics, visualised via comparative heat maps, to provide a high-level overview of relative performance. While the A_{12} effect sizes offer a high-level statistical synthesis of which learner outperforms another, the underlying metrics provide deeper insight into each algorithm’s efficiency and accuracy.

In the heat-map plots for the A_{12} metric and its 95% bootstrap confidence interval (see Section 4.3.3), each cell displays three distinct values: the calculated A_{12} score, the span of the 95% bootstrap confidence interval, and a statistical significance indicator. The cell colour visually interprets the A_{12} value: a blue hue indicates that the row learner significantly outperforms the column learner. Specifically, an A_{12} score that meets or exceeds the established large-effect-size threshold ($A_{12} \geq 0.64$) indicates that the row algorithm achieves higher values than its competitor. Conversely, orange-to-red shading indicates that the row learner underperforms relative to the column learner, while yellow zones signify statistical parity or no meaningful difference between the two algorithms.

The 95% bootstrap confidence interval is reported as a range between its lower and upper bounds ([lower_bound, upper_bound]). If this interval spans across the no-effect baseline of 0.5, the performance difference is deemed statistically non-significant (denoted by “ns” in the heat map); otherwise, the difference is considered statistically significant (denoted by “*” in the heat map).

Finally, to address Research Question **RQ3**, we evaluate and compare the execution runtime of the respective approaches, thereby assessing the computational efficiency of each learning architecture.

The initial training data, which represent the minimum number of traces that cover all transitions for the use case (target model) (generated by Algorithm 5). The minimum number of traces represented by “Data” in the plots in this chapter is detailed in Table 5.1. Furthermore, the maximum sizes of the evaluated data are detailed in Table 5.2. The maximum number represents the stopping point for doubling the data size at which PAT-DFASAT and the baseline DFASAT produce isomorphic automata (see Section 4.2). Note that the depth refers to the length of the longest trace in the training set, while $|\text{Leaves}|$ refers to the number of terminal states (states with no children) in the PTA.

Table 5.1: *The Initial Training Data for all use cases*

Use Case Name	The Minimum data size			
	$ \text{Traces} $	$ Q $ in PTA	$ \text{Leaves} $	Depth
Bluetooth	14	223	14	21
OpenSSL	28	153	28	11
Coffee Machine	11	58	11	11
TCP	45	625	45	21
Randoom1	9	57	9	9
Random2	24	274	24	19
Random3	21	196	21	15
Random4	19	151	19	16
Random5	20	180	20	16
Random6	23	231	23	20

Table 5.2: *The Maximum size for Training Data for all use cases*

Use Case Name	The Maximum data size			
	$ \text{Traces} $	$ Q $ in PTA	$ \text{Leaves} $	Depth
Bluetooth	869	16,828	600	41
OpenSSL	896	6,102	600	20
Coffee Machine	176	970	99	17
TCP	360	7,585	300	38
Randoom1	144	1,119	98	20
Random2	768	12,424	600	33
Random3	336	5,081	299	26
Random4	152	2,240	149	28
Random5	80	1,185	75	24
Random6	736	11,286	600	31

Details about the target models for all use cases are detailed in Table 5.3.

Table 5.3: *Detailed configuration for all use cases - copy of Table 4.1*

System Name	Library	$ Q $	$ I $	$ O $	$ \mathcal{T} = Q \times I $
Coffee Machine [6]	LearnLib	6	4	3	24
Bluetooth [41]	AALpy	15	7	11	105
TCP [22]	AALpy	14	10	11	140
OpenSSL [19]	LearnLib	6	7	7	42
ABP Sender [48]	LearnLib	19	9	10	171
Random 1	Generated By StateChum [11]	7	3	2	21
Random 2		12	4	3	48
Random 3		10	4	2	40
Random 4		9	4	2	36
Random 5		10	4	2	40
Random 6		11	4	3	44

Bluetooth

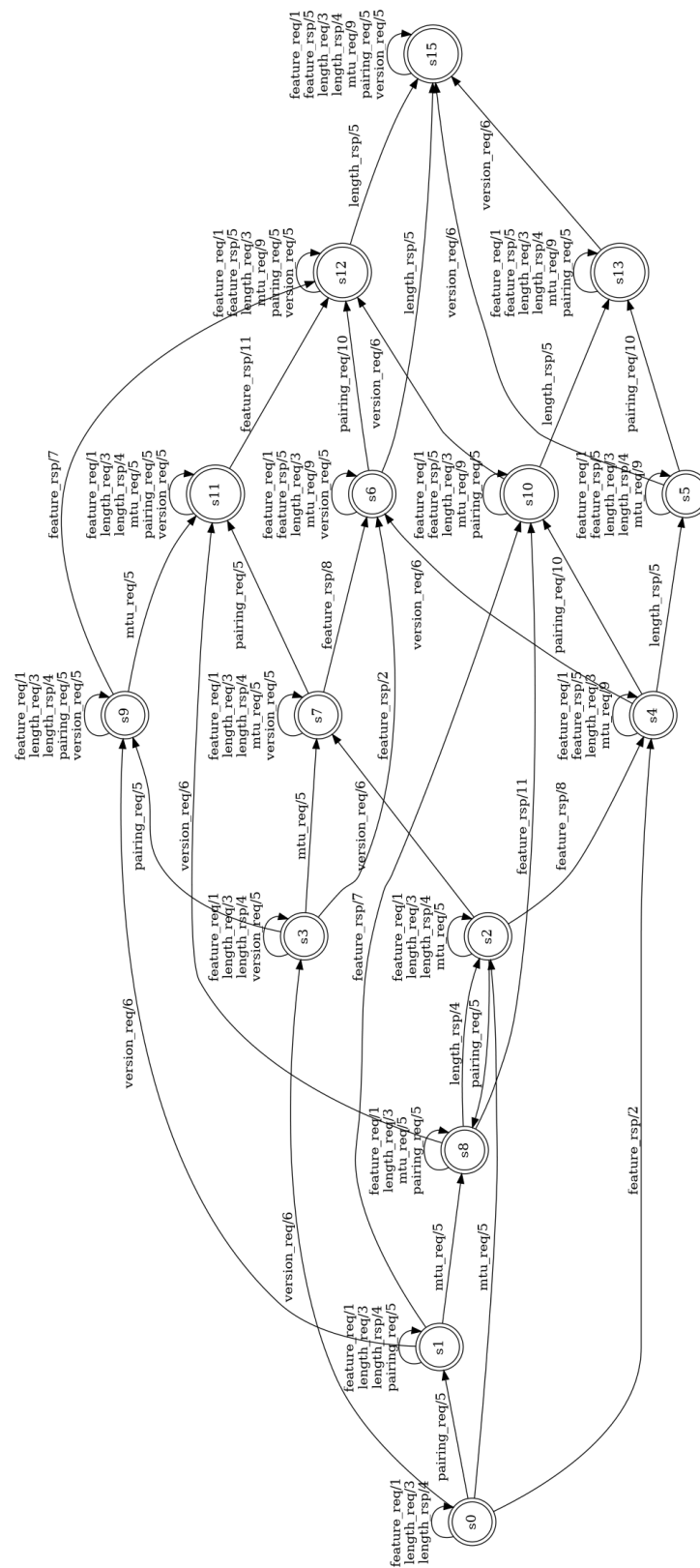


Figure 5.1: *Bluetooth use case - Reference Automata (the initial state is s_0)*

Figure 5.2 illustrates that all candidate learners require a minimum dataset size of $\text{Data} \times 64$ to achieve a 100% proportion of correct merges and perfectly capture the target language specification, eliminating all but minor structural errors in the inferred automata. Notably, at a data density of $\text{Data} \times 32$, PAT-DFASAT significantly outperforms the alternative learners across all evaluation metrics. Although PAT-DFASAT commits a small number of errors during its initial state-merging phase, it maintains a robust $F1$ -score ranging from 0.75 to 0.9 and a high SD-score between 0.70 and 0.95. In contrast, at the same data scale, the baseline DFASAT framework fails to converge on a valid solution within the specified state bound in 50% of the experimental runs. For the remaining 50% of runs where DFASAT does successfully converge, it yields $F1$ -scores between 0.9 and 1 and SD-scores between 0.70 and 0.95. Therefore, while DFASAT’s $F1$ -scores are marginally higher than those of PAT-DFASAT in its successful runs, its overall probability of finding a valid solution drops to a mere 50%. Furthermore, while the PAT-EDSM pipeline shows localised improvements over standard DFASAT at this dataset size, empirical evidence indicates that PAT-DFASAT delivers superior structural and behavioural fidelity overall.

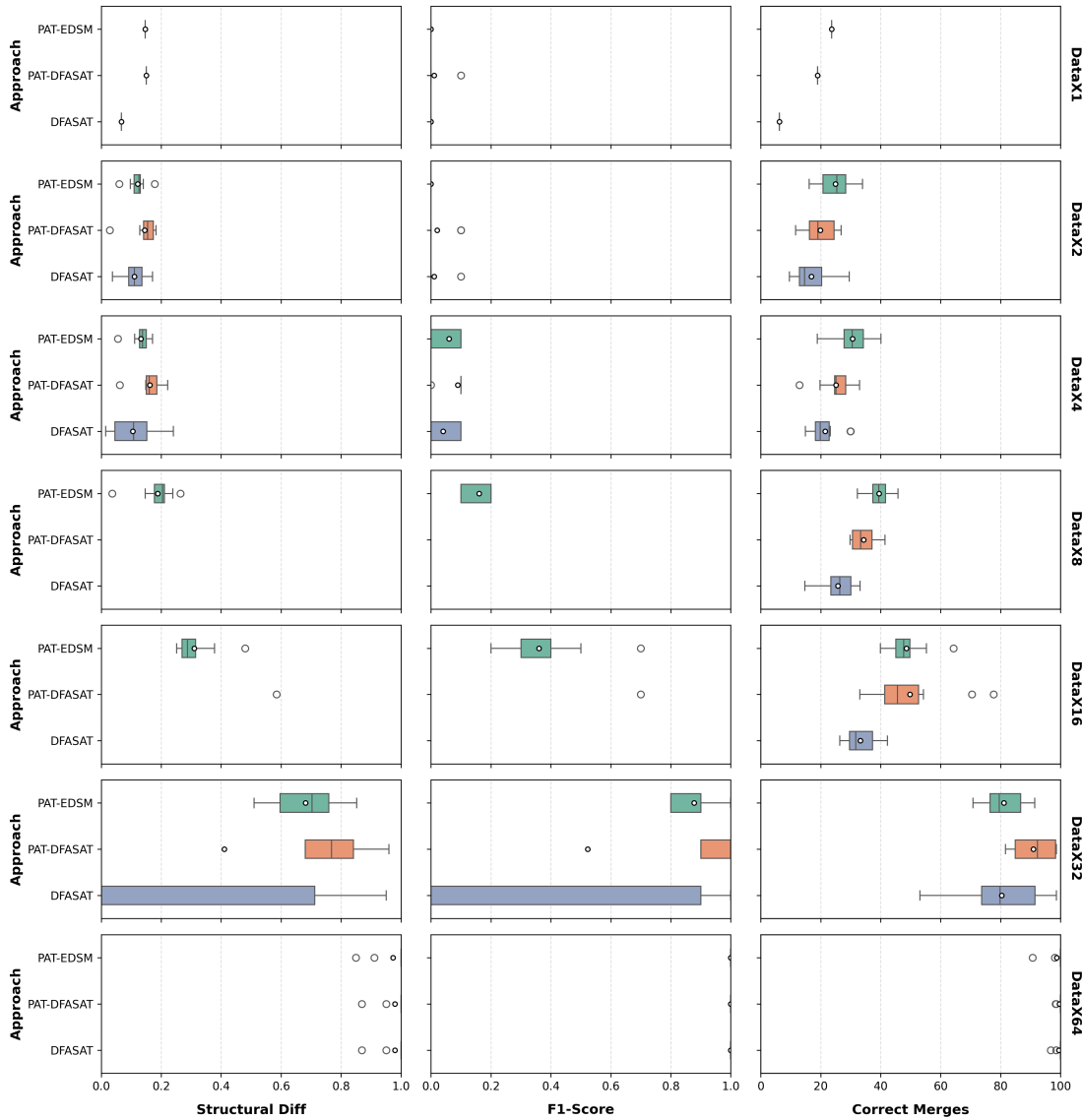


Figure 5.2: Results for the Bluetooth use case with different data sizes

It is worth noting that under extreme data sparsity, the SAT solver fails to identify an optimal solution within the specified upper bound for the maximum number of states. In our implementation of the SAT-based optimisation phase, this upper bound is initialised to at least twice the state cardinality of the target automaton, $|Q_{\text{target}}| \times 2$ (see Section 3.5). Empirically, in the Bluetooth

use case, the SAT solver fails to identify any valid solution at intermediate data sizes ($\text{Data} \times 4$ and $\text{Data} \times 8$), whereas it successfully resolves a solution at smaller or larger data scales.

With severely restricted datasets ($\text{Data} \times 1$ to $\text{Data} \times 2$), the initial EDSM phase commits early structural mistakes due to insufficient evidence, forcing the downstream SAT solver to merge the remaining states into an incorrect partial model. While the resulting automaton is highly sub-optimal, the constrained variable space of the sparse data makes it computationally easier for the solver to fit the remaining states into the erroneous structure within the designated colour upper bound. Conversely, at intermediate data scales ($\text{Data} \times 4$ and $\text{Data} \times 16$), EDSM continues to commit early merge errors, but the model size remains significantly larger. Consequently, the SAT solver struggles to merge these remaining states into the already compromised structure within the rigid colour limits. Ultimately, as the training data size increases further ($\text{Data} \times 32$ and beyond), EDSM acquires sufficient statistical evidence to guarantee high-fidelity merges, thereby producing an initial model with minimal errors or perfectly accurate merges. This structural accuracy provides a high-quality backbone, enabling the downstream solver to seamlessly integrate the remaining states into the target topology.

Figure 5.3 illustrates the execution runtime across the evaluated learning frameworks. At smaller data scales ($\text{Data} \times 1$ and $\text{Data} \times 2$), all candidate learners successfully complete execution within a reasonable time frame of a few minutes. At intermediate scales ($\text{Data} \times 4$ and $\text{Data} \times 8$), execution runtimes increase noticeably; PAT-EDSM requires approximately one hour, whereas PAT-DFASAT and the baseline DFASAT require roughly two hours. This pronounced runtime inflation for the SAT-based approaches relative to PAT-EDSM is directly attributable to their frequent failure to converge on a valid solution within the state bounds at this specific scale, as established in the preceding analysis.

At a data density of $\text{Data} \times 16$, the runtime for PAT-EDSM scales sharply to approximately 4 hours, converging with the execution time of both PAT-DFASAT and DFASAT. At the $\text{Data} \times 32$ scale, PAT-EDSM reaches its peak execution latency, exceeding 11 hours. Conversely, PAT-DFASAT demonstrates superior computational efficiency at this scale, with runtimes ranging from approximately 2.5 to 5.5 hours, whereas the baseline DFASAT runs from 3 to 7 hours. It is critical to contextualise this by noting that DFASAT fails to identify a valid solution in 50% of these experimental runs. Finally, at the maximum data volume ($\text{Data} \times 64$), all three frameworks exhibit a stabilised runtime of approximately 7 hours, with marginal outliers observed in isolated runs of PAT-EDSM and DFASAT.

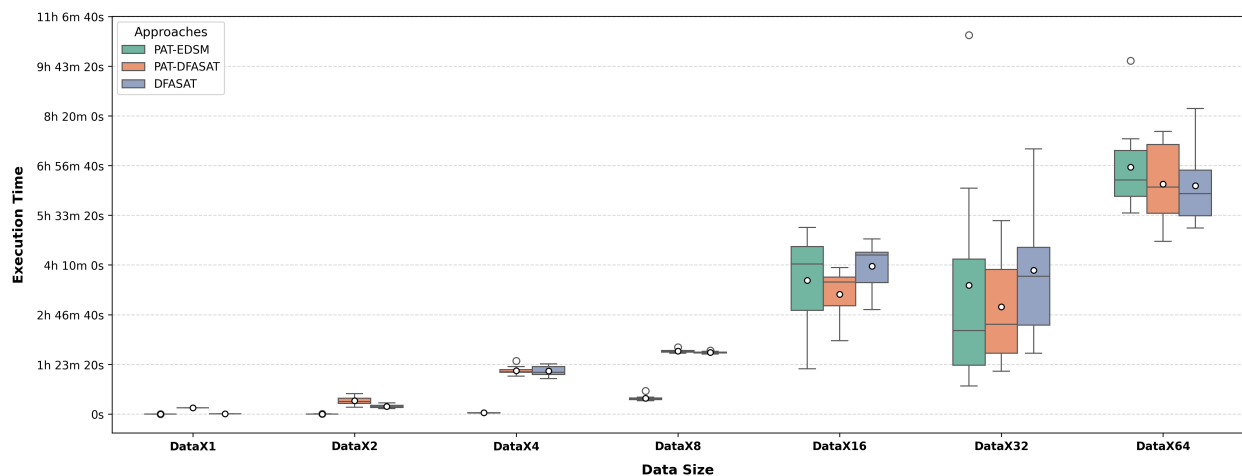


Figure 5.3: *Bluetooth - Runtime results: comparing the learners' performance*

Although the candidate learners fail to achieve high SD or $F1$ -scores at data sizes below $\text{Data} \times 32$, the A12-scores shown in Figure 5.4 reveal clear performance differences at the different scales. Specifically, at the minimum data density ($\text{Data} \times 1$), PAT-EDSM significantly outperforms both alternative learners and is statistically significant. However, as the training data size increases, PAT-DFASAT consistently outperforms both PAT-EDSM and DFASAT. Ultimately, once the data scale reaches $\text{Data} \times 64$, all candidate learners converge to nearly identical scores across all evaluation metrics and show no statistical significance, with a wide confidence intervals indicating that individual data points exert strong leverage on ranks.

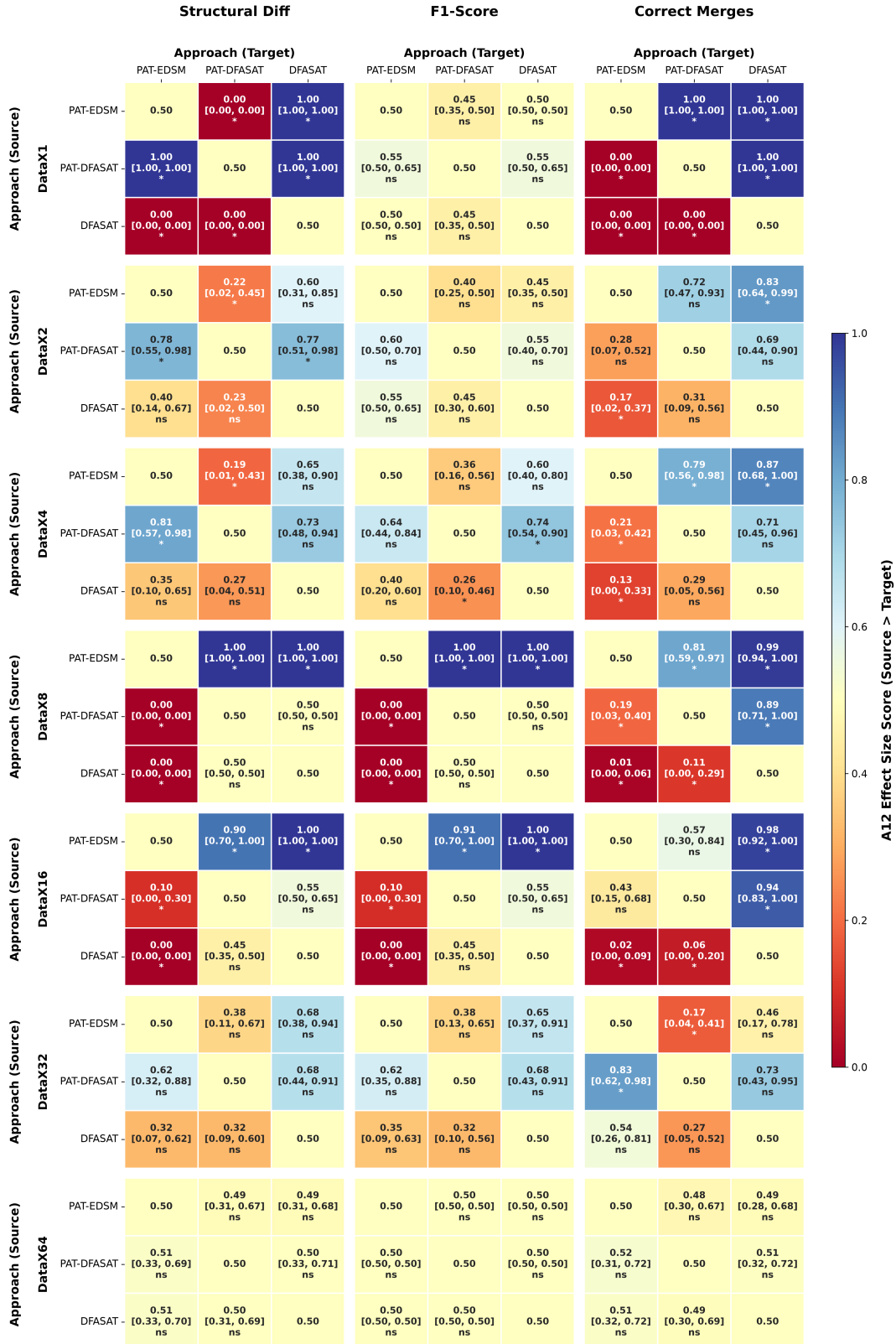


Figure 5.4: Bluetooth - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

OpenSSL

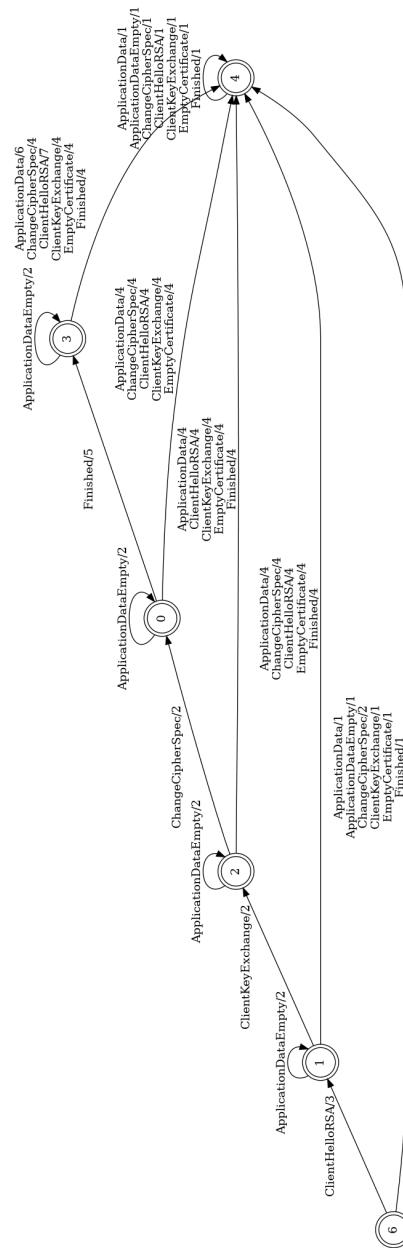


Figure 5.5: *OpenSSL use case - Reference Automata (the initial state is s_0)*

As shown in Figure 5.6, the learners required a massive amount of data to successfully recover the correct language and structure for OpenSSL. Nevertheless, both PAT-DFASAT and DFASAT converge to nearly identical scores at dataset sizes of $\text{Data} \times 8$ and beyond. An interesting result occurs at $\text{Data} \times 4$, where PAT-DFASAT achieves high SD-scores (≈ 0.8) paired with a strong, though wider-ranging, $F1$ -score (spanning 0.7 to 1.0), while executing 100% correct merges. Remarkably, PAT-DFASAT achieves a 100% correct merge ratio even under extreme data sparsity (at $\text{Data} \times 1$), where competing learners falter. For instance, standard DFASAT requires at least $\text{Data} \times 16$, and PAT-EDSM could not achieve 100% correct merge performance among all the evaluated sizes.

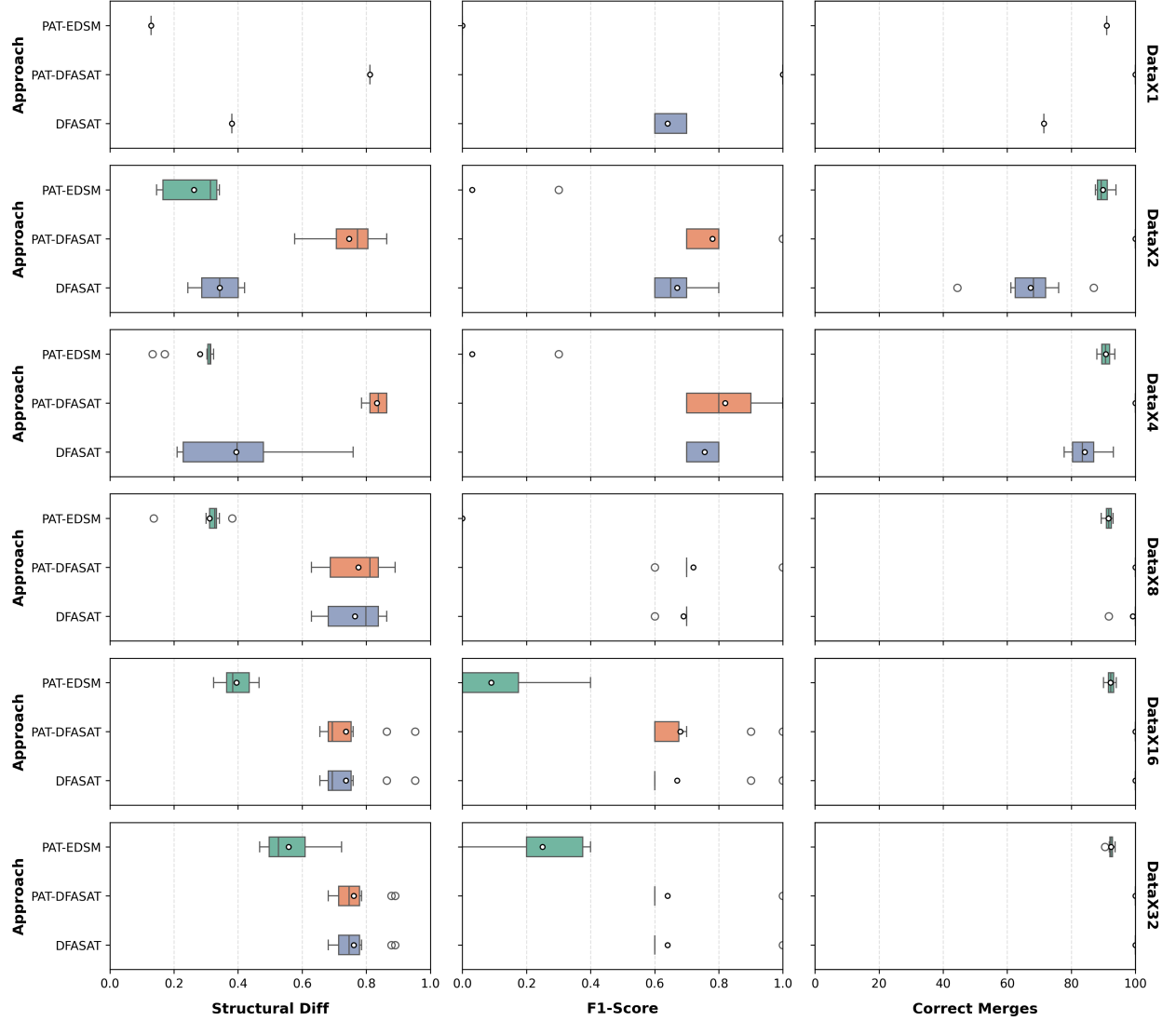


Figure 5.6: Results for the OpenSSL use case with different data sizes

Figure 5.7 demonstrates that all candidate learners exhibit highly comparable execution time. Across all evaluated data scales, the maximum execution time is 17 minutes. Specifically, data sizes ranging from $\text{Data} \times 1$ to $\text{Data} \times 8$ require only a few seconds to complete, whereas execution durations scale to a few minutes at the $\text{Data} \times 16$ density. Runtimes then increase sharply to approximately 13 to 17 minutes at the $\text{Data} \times 32$ scale. It is worth noting that at the $\text{Data} \times 16$ and $\text{Data} \times 32$ scales, PAT-DFASAT and the baseline DFASAT framework perform similarly in both SD and $F1$ -scores; however, their runtimes differ slightly, with DFASAT demonstrating marginal superiority. Conversely, at these scales, PAT-EDSM under-merges states, inflating the remaining state space and consequently increasing overall execution time.

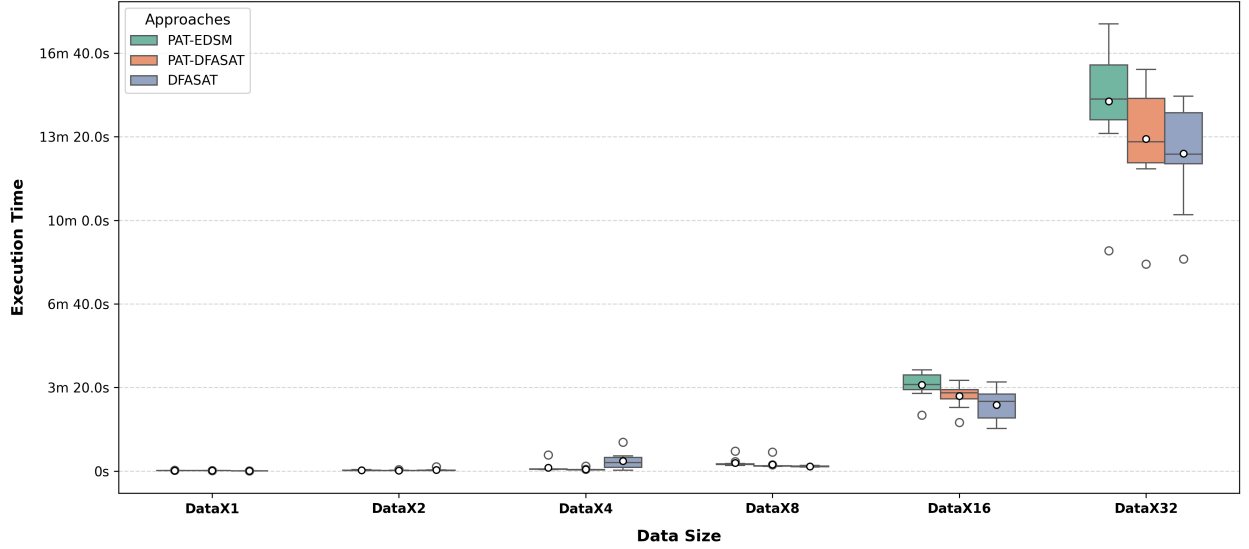


Figure 5.7: *OpenSSL - Runtime results: comparing the learners' performance*

Figure 5.8 demonstrates that both PAT-DFASAT and DFASAT consistently outperform PAT-EDSM across all evaluated data sizes and demonstrate a statistical significance (denoted by *). Furthermore, PAT-DFASAT exhibits a pronounced performance advantage over standard DFASAT at constrained data scales (Data $\times$ 2 to Data $\times$ 4). This performance disparity progressively narrows as the training data size increases, ultimately closing entirely once the data size reaches Data $\times$ 8, at which point both learners achieve nearly identical scores.

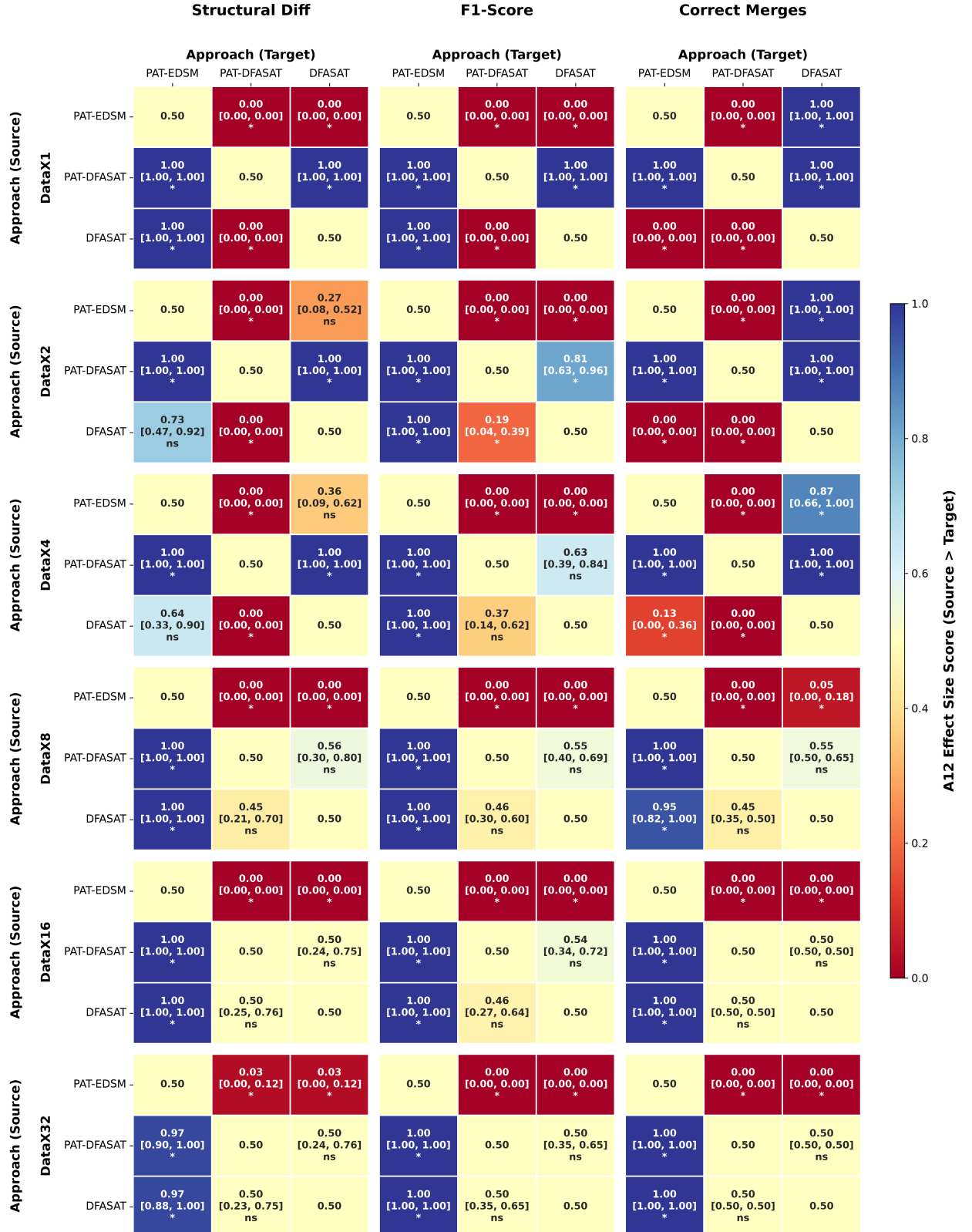


Figure 5.8: OpenSSL - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

Coffee Machine

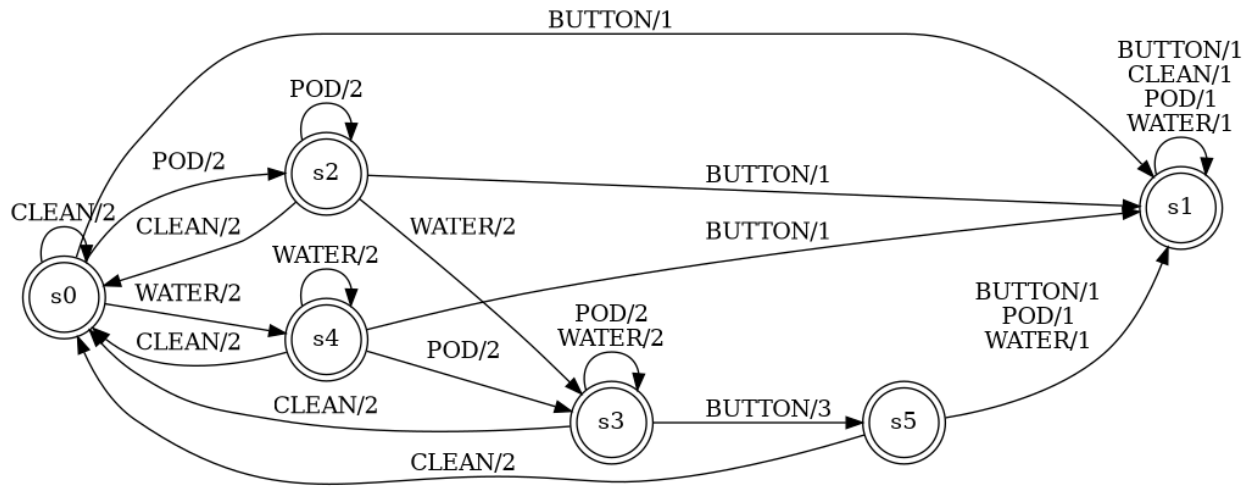


Figure 5.9: Coffee Machine use case - Reference Automata (the initial state is s_0)

Coffee Machine

Figure 5.10 shows that, for the Coffee Machine case study, PAT-EDSM successfully recovers both the correct language and structure once the data size reaches $\text{Data} \times 32$. In contrast, both PAT-DFASAT and the baseline DFASAT framework continue to commit minor structural errors at this scale. Notably, the latter two learners converge, producing isomorphic automata at dataset sizes of $\text{Data} \times 8$ and beyond. At smaller data scales ($\text{Data} \times 2$ and $\text{Data} \times 4$), PAT-DFASAT consistently outperforms standard DFASAT, though its overall performance remains secondary to that of PAT-EDSM. It is worth noting that while PAT-DFASAT does not closely approximate the target model’s structural topology at these lower scales, it nevertheless achieves a relatively high $F1$ -score, indicating strong fidelity in language acceptance despite the structural discrepancies.

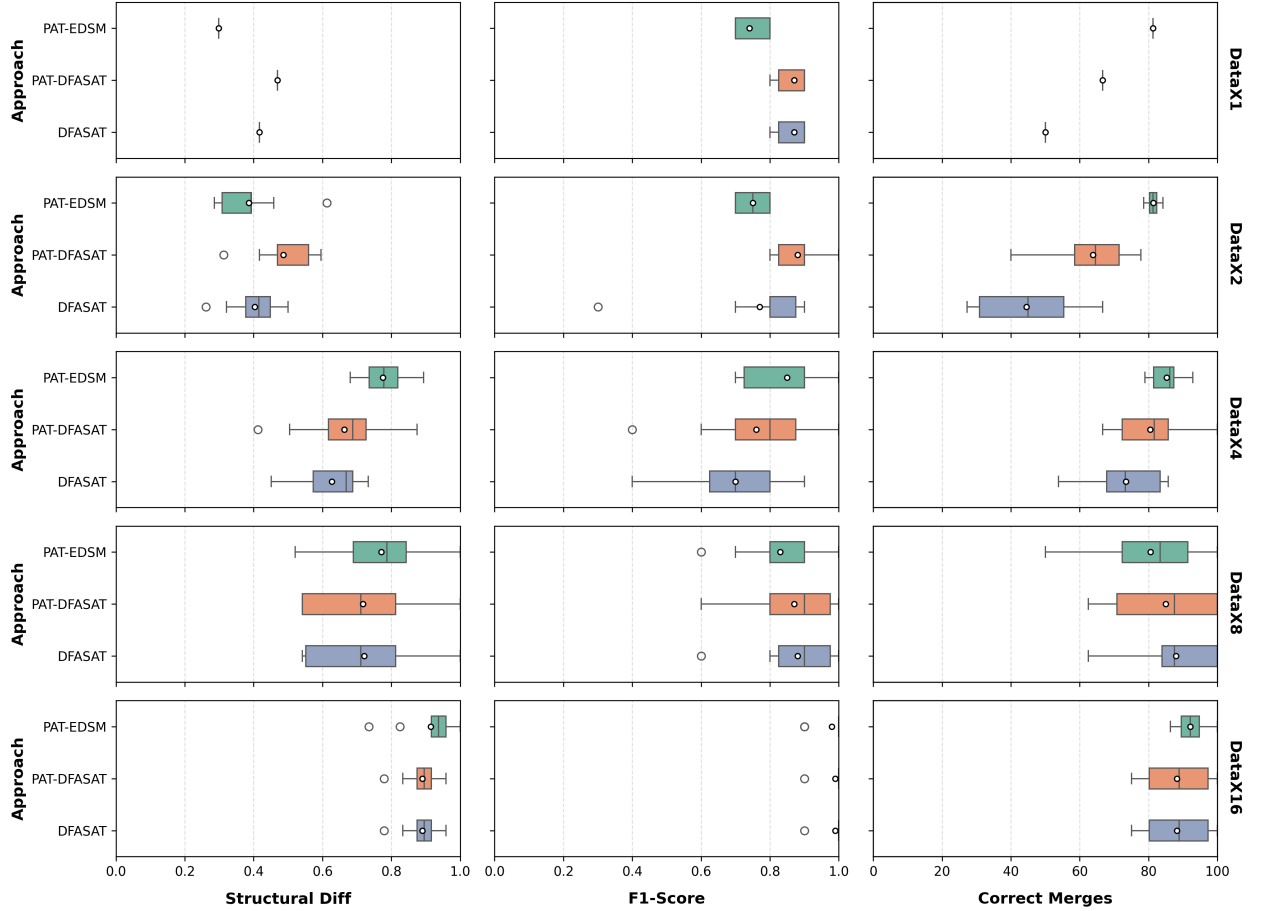


Figure 5.10: Results for the Coffee Machine use case with different data sizes

Figure 5.11 demonstrates that at lower data scales, all candidate learners require only a few seconds to successfully complete execution. However, at the maximum data volume (Data $\times$ 16), the baseline DFASAT framework requires a considerably longer runtime compared to the other two learners. While DFASAT's execution runtime ranges from 10 seconds to over 1 minute, the runtimes for PAT-DFASAT and PAT-EDSM are approximately 10-20 seconds, with PAT-DFASAT emerging as the fastest framework. It is noteworthy that at this specific scale, PAT-DFASAT and DFASAT produce identical metric scores across all evaluation criteria (SD-score, $F1$ -score, and the proportion of correct merges), indicating that PAT-DFASAT achieves the same optimal model structure with significantly greater computational efficiency.

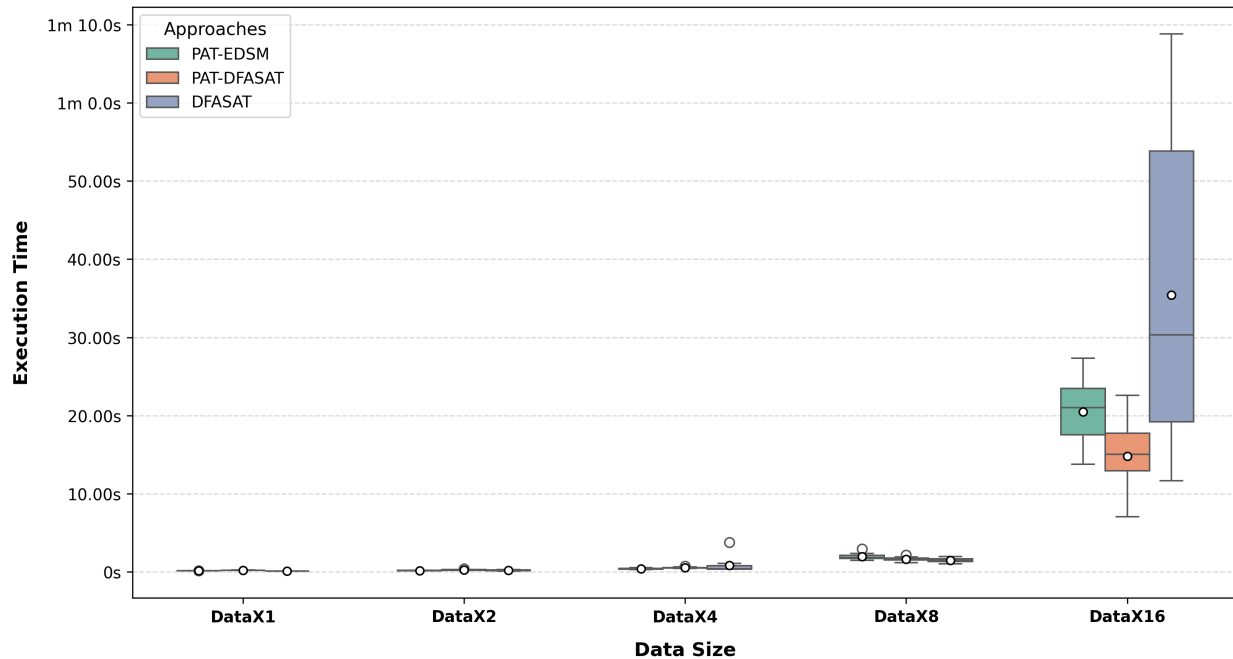


Figure 5.11: *Coffee Machine - Runtime results: comparing the learners' performance*

At the minimum data density (Data $\times$ 1 and Data $\times$ 2), both PAT-DFASAT and DFASAT outperform PAT-EDSM in terms of SD-score and $F1$ -score, while PAT-EDSM outperforms both of them in terms of the correct merge rate while demonstrating a statistical significance. However, at this data scale, all learners failed to learn a reasonably accurate model. As the data size increases (Data $\times$ 4 and beyond), PAT-EDSM reverses this trend, outperforms the other two learners, and demonstrates statistical significance across all metrics. However, the wide range of confidence intervals indicates high uncertainty and low precision. At high data sizes (Data $\times$ 16), PAT-DFASAT and DFASAT achieve $F1$ -scores comparable to PAT-EDSM and demonstrate no statistical significance; however, they still fall short on other metrics, particularly in their SD-scores and the total ratio of correct merges. Comparing PAT-DFASAT to DFASAT shows that under extremely sparse data (Data $\times$ 4 or less), PAT-DFASAT outperforms its competitor across all metrics, whereas they exhibit similar performance as data size increases. Notably, at data sizes (Data $\times$ 16 and Data $\times$ 16), the three learners demonstrate no statistical significance across all metrics.

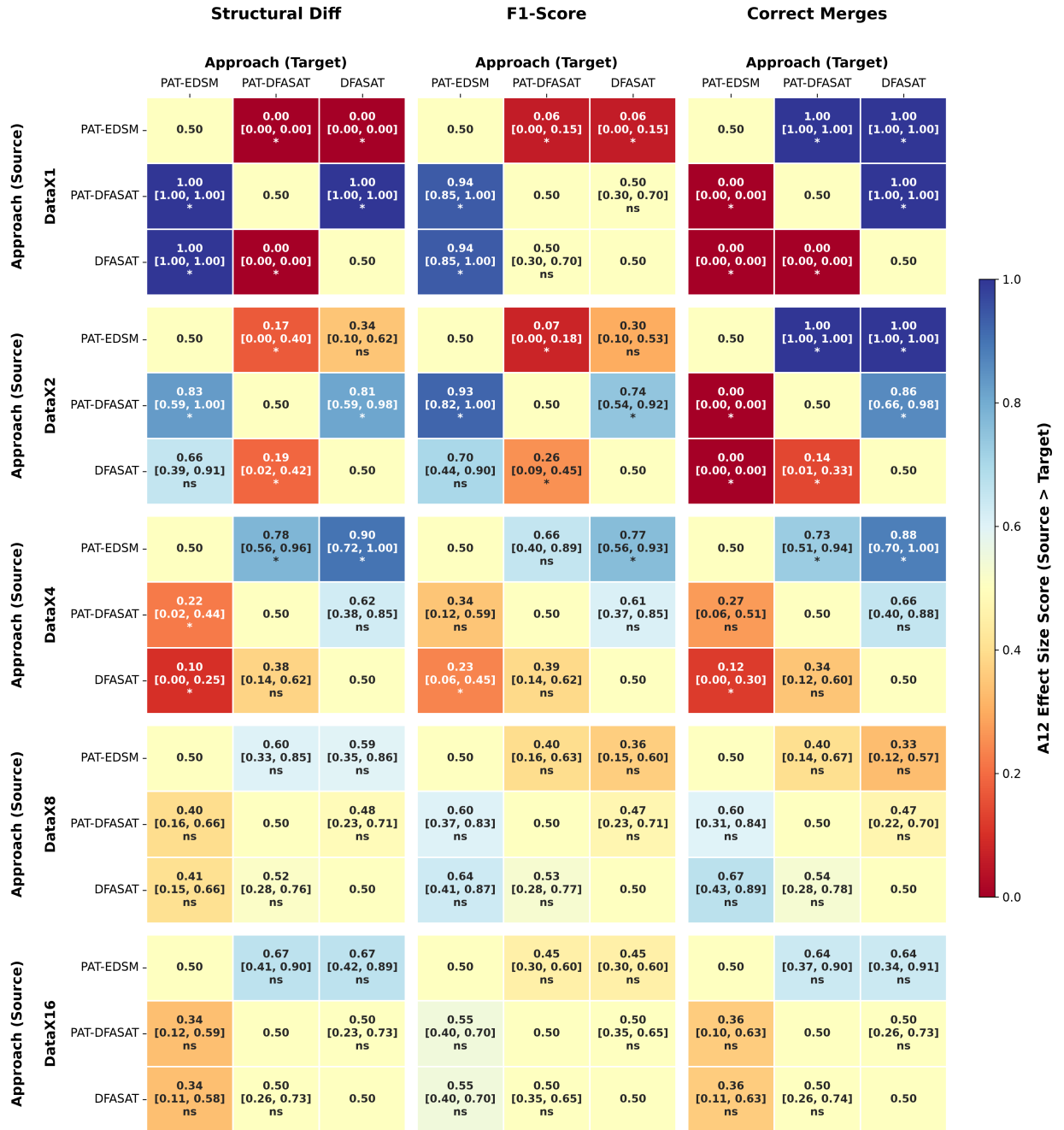


Figure 5.12: Coffee Machine - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

This demonstrates that at a data scale of $\text{Data} \times 32$, both PAT-DFASAT and DFASAT achieve 100% correct merges during their EDSM phase, yet they struggle to identify the exact target model during the SAT solver phase. Conversely, PAT-EDSM successfully converges on the correct target model. At this data scale, both SAT-based learners produce isomorphic automata. Consequently, Figure 5.14 contrasts the target model $\mathcal{M}$ produced by PAT-EDSM (Figure 5.13b) with the approximate model $\mathcal{M}'$ produced by PAT-DFASAT (Figure 5.13a).

In both diagrams, the initial state is denoted by a star. The input-output transitions are formatted with the notation $\{\text{POD}, 2\}$, which denotes an input of POD yielding an output of 2. A structural comparison reveals that $\mathcal{M}$ and $\mathcal{M}'$ are highly similar, adhering to the following state-mapping sequence: $T1 \rightarrow S1$, $T2 \rightarrow S3$, $T5 \rightarrow S4$, $T6 \rightarrow S5$, $T7 \rightarrow S2$, $T4 \rightarrow S6$.

The sole topological discrepancy occurs at state $T6$ in $\mathcal{M}'$ relative to its counterpart $S5$ in $\mathcal{M}$. In $\mathcal{M}$, state $S5$ possesses a transition labelled $\{\text{CLEAN}, 2\}$ to state $S1$. In contrast, the corresponding transition from state $T6$ in $\mathcal{M}'$ erroneously targets state $T7$. This structural deviation stems from the highly symmetrical operational behaviour shared by states $S1$ and $S2$. State $S1$ features a self-loop labelled $\{\text{CLEAN}, 2\}$ alongside an incoming transition from state $S2$ under the same label. Conversely, state $S2$ features a self-loop labelled $\{\text{WATER}, 2\}$ and an incoming transition from state $S1$ under that same label. This tightly coupled relationship allows the system to transition between $S1$ and $S2$ by alternating between $\{\text{CLEAN}, 2\}$ and $\{\text{WATER}, 2\}$ inputs. Because their future behaviours are so symmetrical, the sparse training data provides insufficient evidence to distinguish them. Consequently, the SAT solver cannot structurally differentiate between $S1$ and $S2$, ultimately resulting in incorrect state merging.

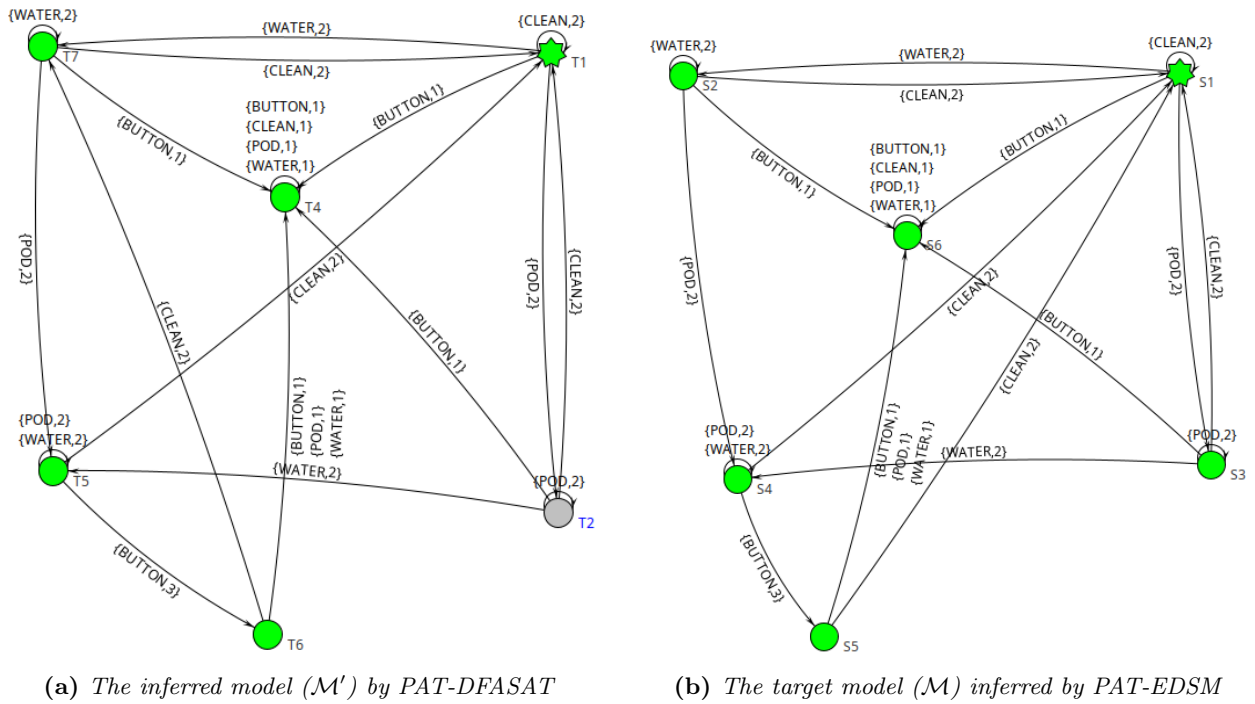
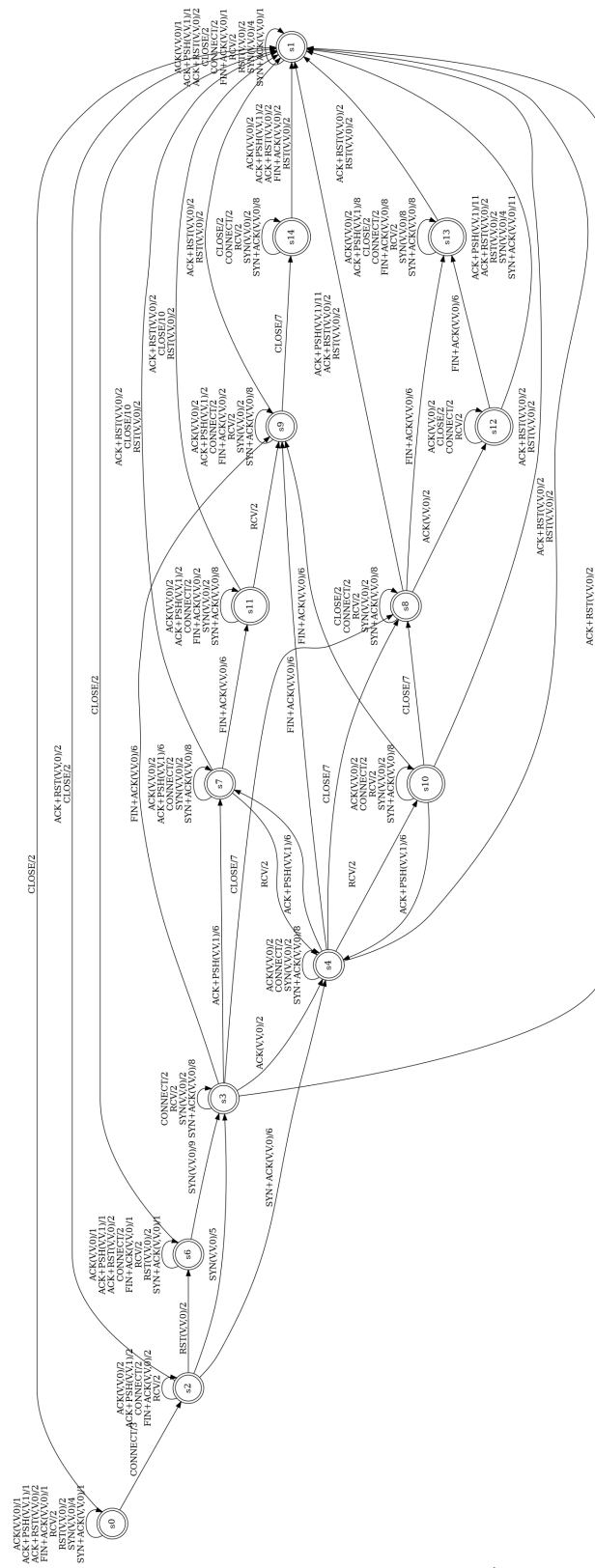


Figure 5.14: Comparing two inferred models using PAT-EDSM and PAT-DFASAT for the Coffee Machine use case at size ($\text{Data} \times 32$)

TCP

Figure 5.15: TCP use case - Reference Automata (the initial state is s_0)

TCP

For the TCP use case, Figure 5.16, PAT-EDSM struggled to outperform PAT-DFASAT and DFASAT across all data sizes. However, PAT-DFASAT and DFASAT achieved similar scores with extremely sparse data sets (Data $\times$ 4 or less), and identical scores with more data. Notably, with a small data size (Data $\times$ 1 or $\times$ 2), PAT-DFASAT made fewer mistakes in initial correct merges than DFASAT, but still struggled to achieve high SD and $F1$ scores.

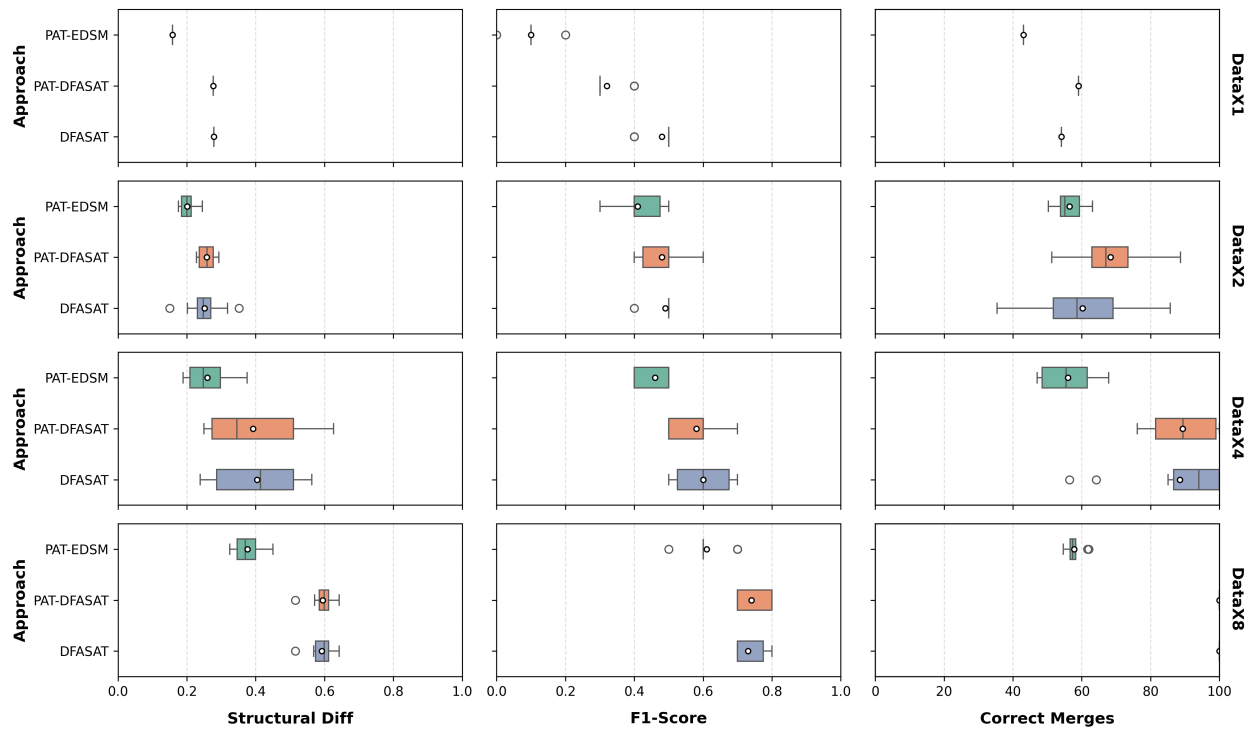


Figure 5.16: Results for the TCP use case with different data sizes

Figure 5.17 illustrates that at small data scales (Data $\times$ 4 or less), PAT-EDSM requires less execution time than the SAT-based learners. At these lower densities, the SAT-based frameworks execute multiple computationally expensive search iterations before converging on a valid solution. Conversely, with larger data volumes (Data $\times$ 8), the runtime of the SAT-based learners optimises significantly, allowing them to execute faster than PAT-EDSM. Notably, at this larger scale, the runtimes of both SAT-based frameworks become highly comparable. This computational alignment reflects the convergence observed in their other evaluation metrics, specifically their SD-score, $F1$ -score, and the proportion of correct merges. Conversely, at smaller data scales, PAT-DFASAT executes a higher proportion of correct merges than the baseline DFASAT, which directly influences their respective execution times. Consequently, PAT-DFASAT requires less runtime than standard DFASAT at these lower densities, even though both frameworks ultimately struggle to accurately approximate the target model under such sparse data. Moreover, we observed that at very small data scales, the variance (or spread) of the execution runtimes is minimal. At intermediate data ranges, however, this variance increases sharply, indicating that the learner struggles to navigate the search space efficiently. Conversely, introducing a larger amount of data shrinks the runtime variance once again. This trend serves as an indicator, revealing when the training data is too sparse to guide the algorithm effectively and when it becomes dense enough to ensure computational efficiency.

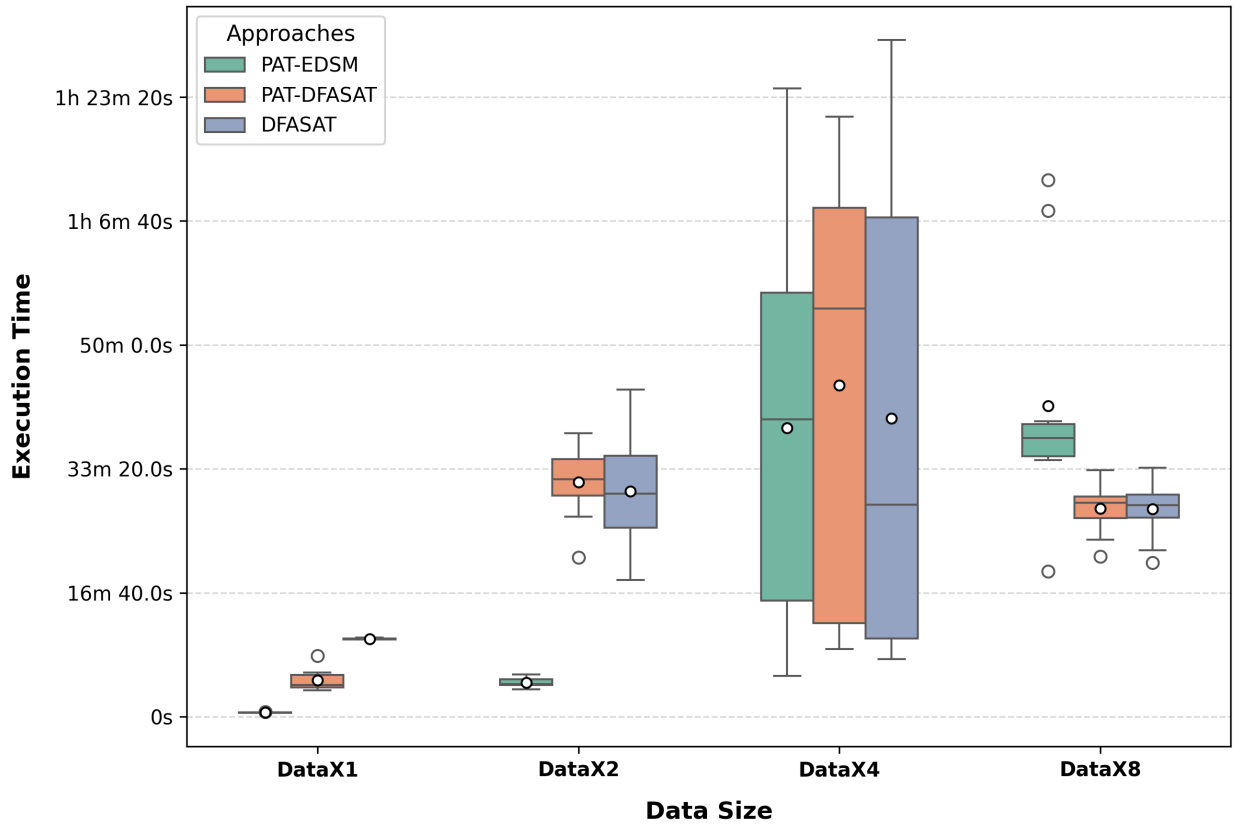


Figure 5.17: TCP - Runtime results: comparing the learners' performance

Across all evaluated data sizes, none of the candidate learners successfully approximated the exact target model. However, from a broader perspective, PAT-DFASAT and the baseline DFASAT consistently outperform PAT-EDSM and demonstrate statistical significance. While PAT-DFASAT and standard DFASAT yield comparable performance across all metrics at data scales (Data $\times$ 4 and Data $\times$ 8) while demonstrating no statistical significance, their behaviour diverges at the smaller scale (Data $\times$ 2), where PAT-DFASAT slightly outperforms DFASAT yet demonstrates no statistical significance. At the minimum data scale (Data $\times$ 1), DFASAT achieves higher values than PAT-DFASAT in terms of both SD-score and $F1$ -scores; however, PAT-DFASAT reverses this advantage by outperforming DFASAT in the proportion of correct merges.

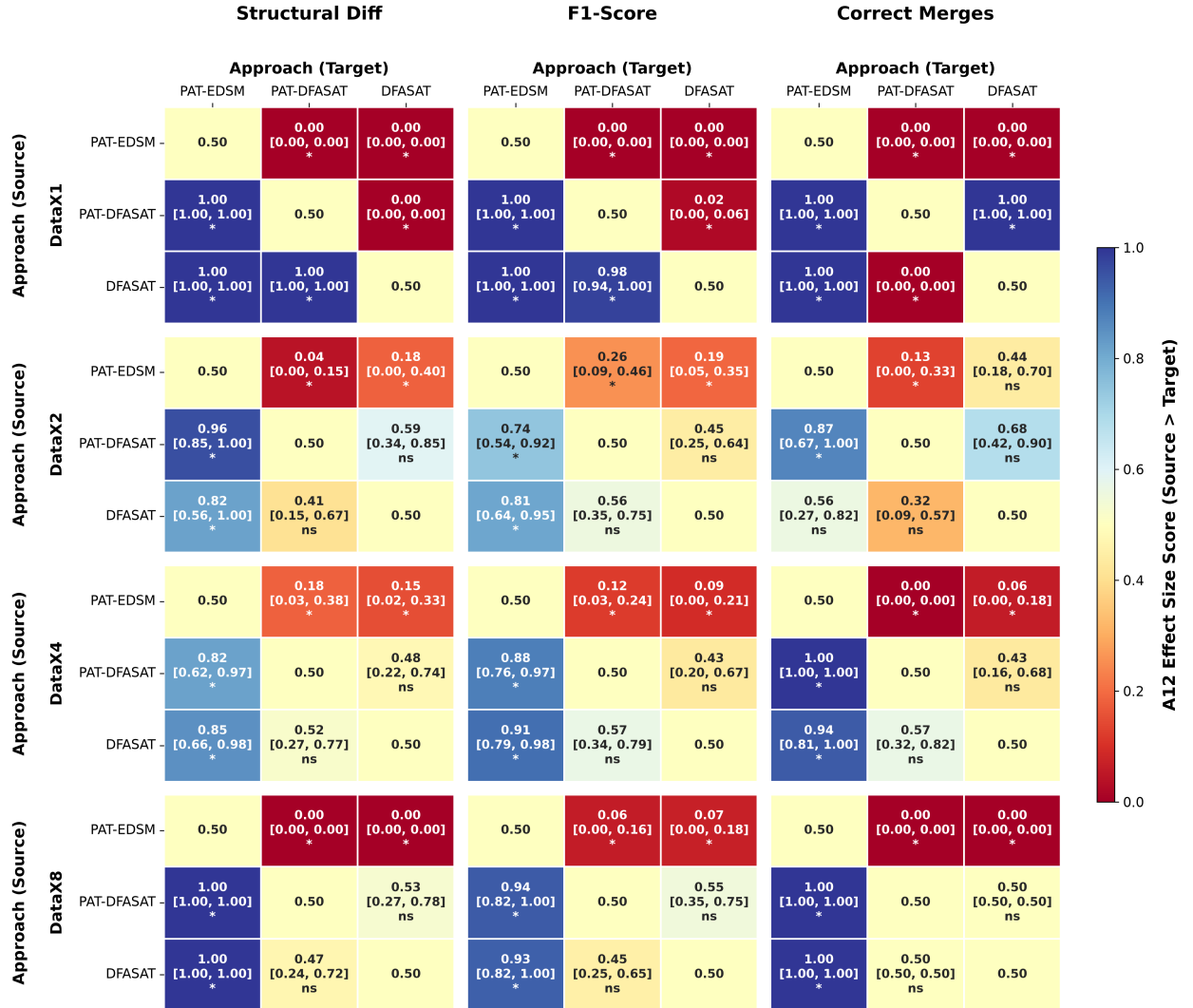


Figure 5.18: TCP - A12 results: comparing the learners' performance

ABP Sender

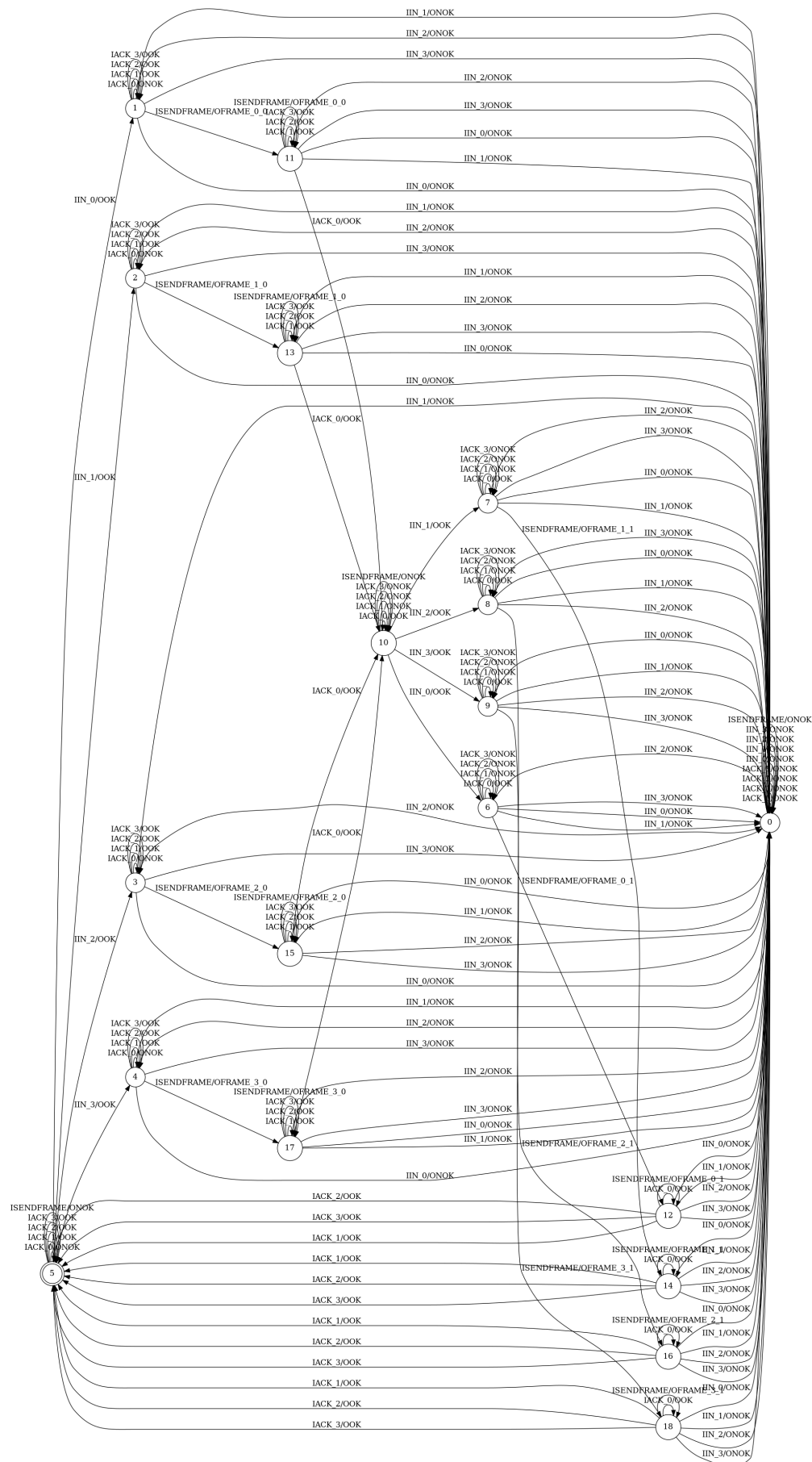


Figure 5.19: *ABP Sender use case - Reference Automata (the initial state is (5))*

ABP Sender

Figure 5.20 shows that, for the ABP Sender case study at small data scales (Data $\times$ 1 to Data $\times$ 4), PAT-DFASAT outperforms both PAT-EDSM and DFASAT in terms of structural difference and the proportion of correct merges. However, all learners failed to capture the target language accurately at these lower scales. At Data $\times$ 4, PAT-DFASAT achieved a high proportion of correct merges (90–100%). Although PAT-DFASAT outperformed its competitors, its SD-score remained moderate (≈ 0.7), and $F1$ -scores across all learners were low. At a larger data scale (Data $\times$ 8), both PAT-DFASAT and DFASAT achieved a high proportion of correct merges ($\approx 100\%$) along with converging SD- and $F1$ -scores, although DFASAT exhibited a wider dispersion in its $F1$ -score distribution. At data scale (Data $\times$ 16), PAT-DFASAT and DFASAT yielded similar performance across all evaluated metrics while consistently outperforming PAT-EDSM.

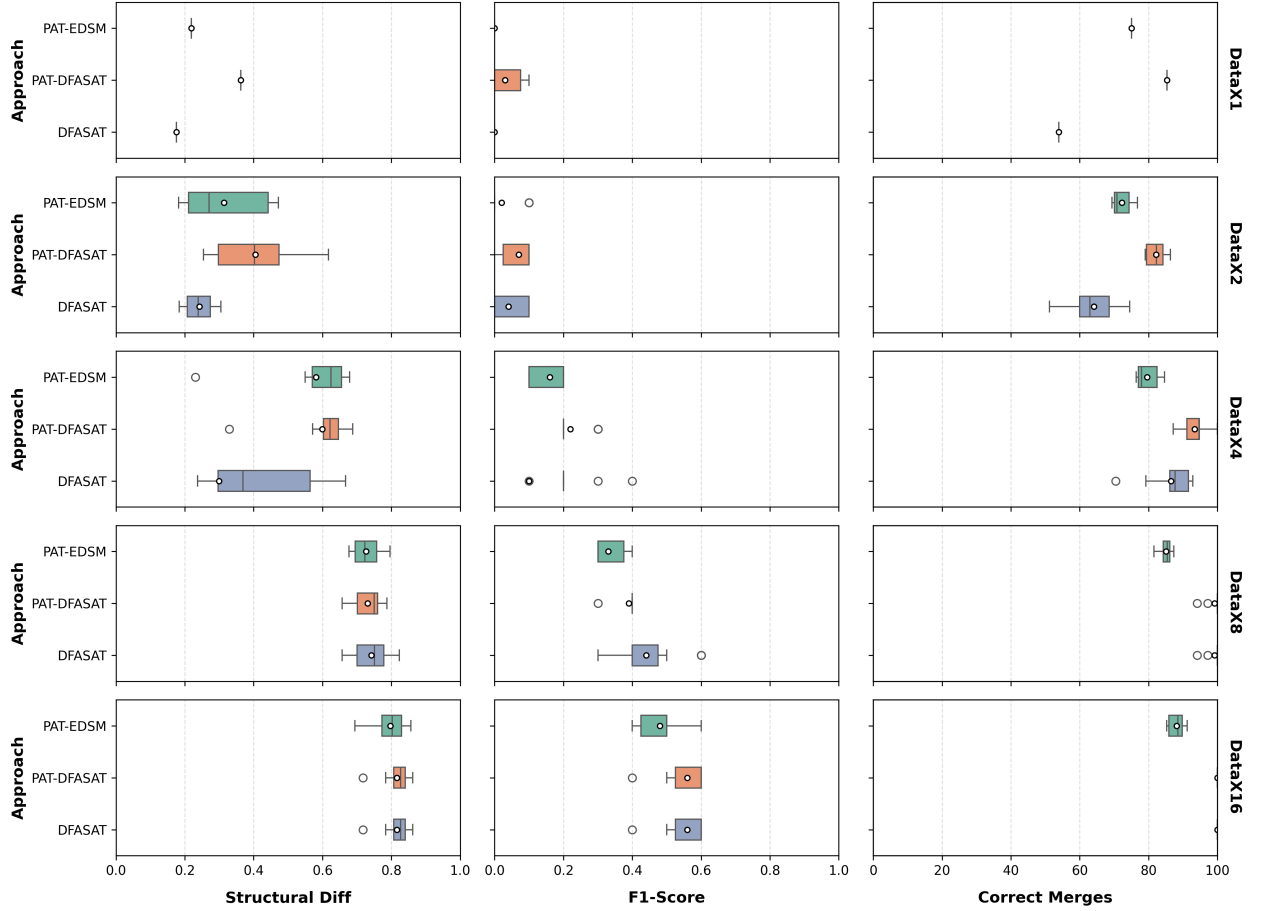


Figure 5.20: Results for the ABP Sender use case with different data sizes

Figure 5.21 illustrates that all learners require several hours to complete the inference process for the ABP Sender case study. The peak execution time occurred at the $(\text{Data} \times 4)$ scale for the DFASAT learner, which exhibited an outlier exceeding two days. At this scale, all learners display substantial variance in execution time across independent runs. However, as data size increased, execution-time variance narrowed significantly; at $\text{Data} \times 8$ and $\text{Data} \times 16$, both PAT-DFASAT and DFASAT showed stable execution times of approximately 3 hours and 7 hours, respectively.

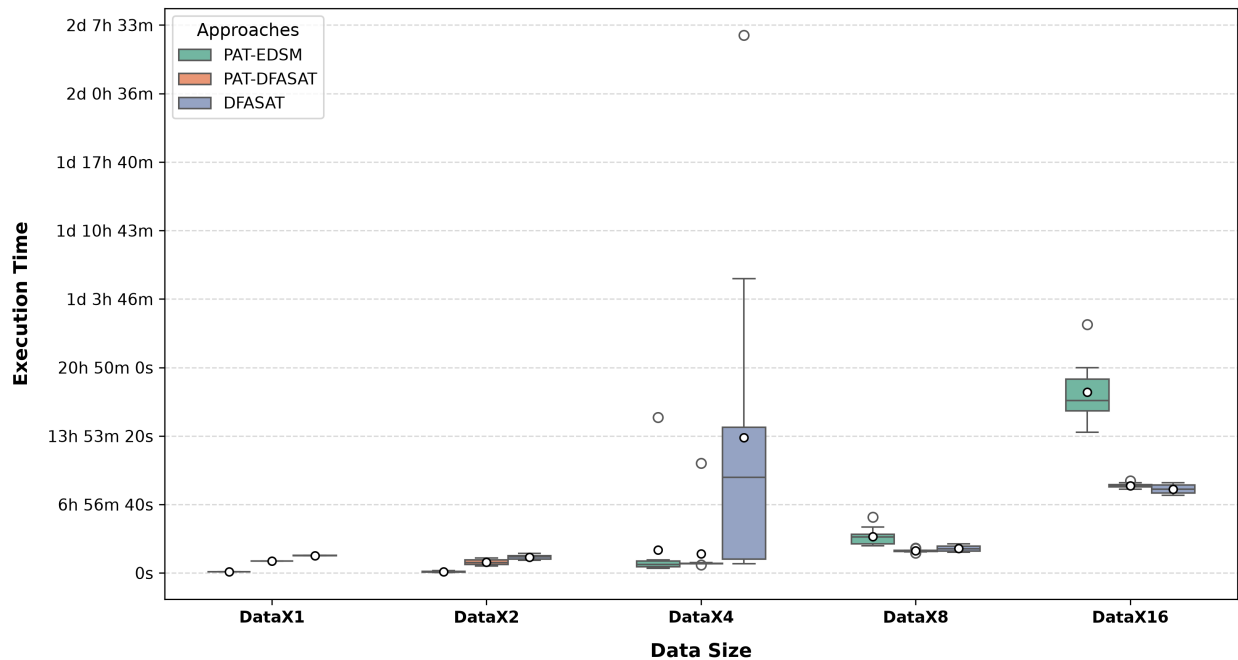


Figure 5.21: *ABP Sender - Runtime results: comparing the learners' performance*

Figure 5.22 shows that PAT-DFASAT outperforms both PAT-EDSM and DFASAT at small data scales (Data $\times$ 1 and Data $\times$ 2). However, at these lower scales, none of the learners successfully approximated the target model. At scale Data $\times$ 4, PAT-DFASAT marginally outperforms PAT-EDSM in terms of SD- and F_1 -scores; however, this advantage lacks statistical significance for the SD-score, while the 95% confidence interval for the F_1 -score indicates statistical uncertainty. At the same scale, PAT-DFASAT significantly outperforms DFASAT in SD-score with statistical significance (albeit accompanied by wide confidence intervals), while demonstrating slight superiority in F_1 -score without statistical significance. Crucially, for the proportion of correct merges, PAT-DFASAT exhibits substantial and statistically significant outperformance over both PAT-EDSM and DFASAT with high confidence.

At larger data scales (Data $\times$ 8 and Data $\times$ 16), PAT-DFASAT and DFASAT exhibit convergent performance across all metrics. Both algorithms consistently outperform PAT-EDSM, demonstrating statistical significance with high confidence in the proportion of correct merges, while showing no statistically significant difference or notable confidence-interval overlap in their SD- and F_1 -scores.

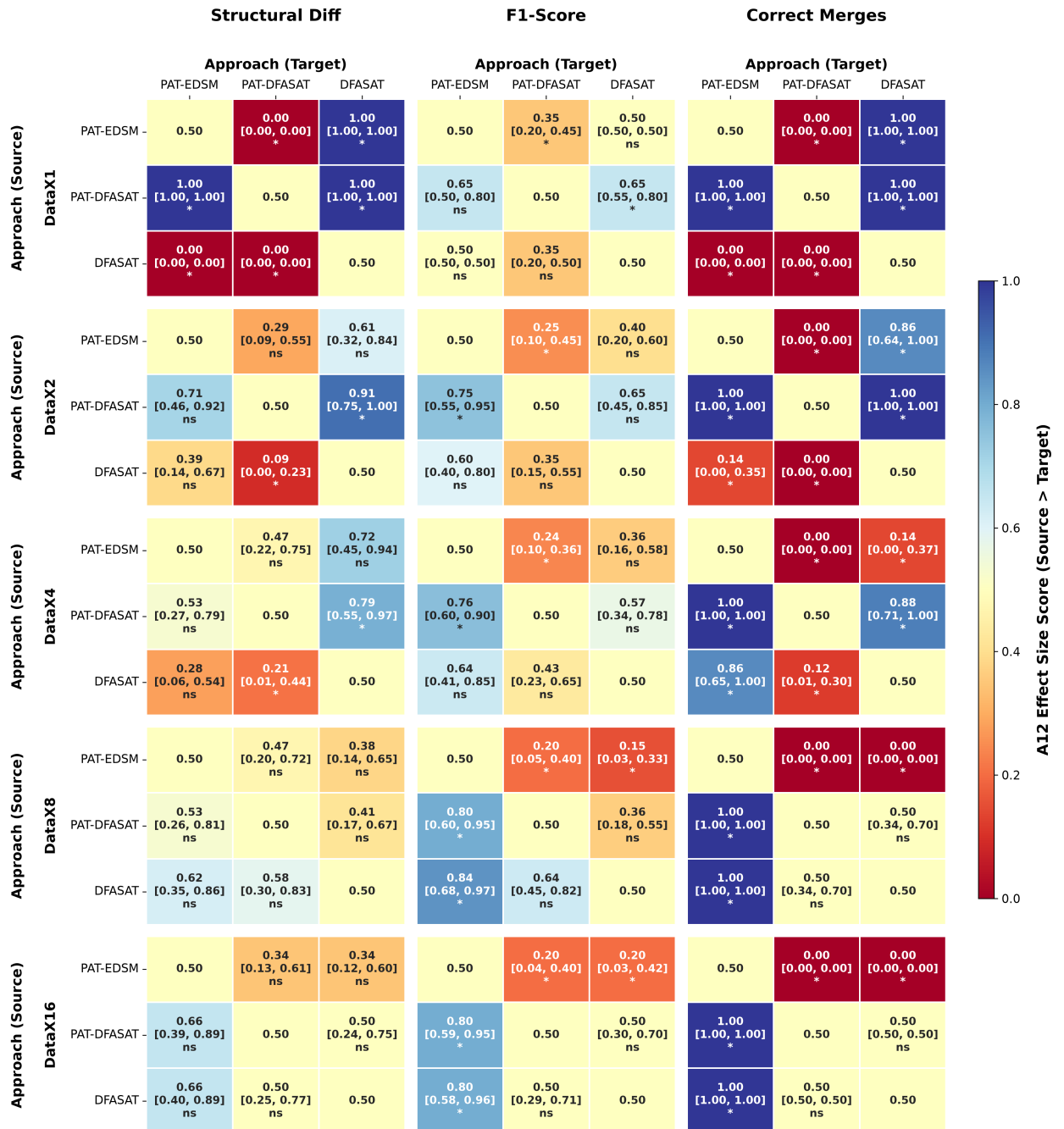


Figure 5.22: ABP Sender - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

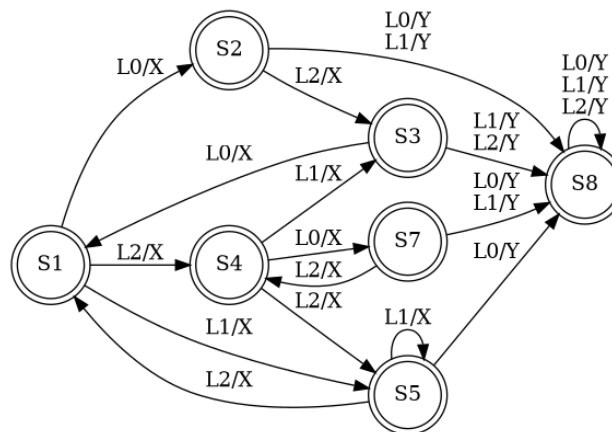
Random1

Figure 5.23: *Random1 use case - Reference Automata (the initial state is S1)*

5.1.2 Randomly Generated Use Cases

Random1

As illustrated in Figure 5.24, PAT-DFASAT outperforms the alternative learners in the Random1 case study. At a data scale of $\text{Data} \times 8$, PAT-DFASAT successfully identifies the exact target language while achieving high SD-scores (≥ 0.90). At this scale, PAT-DFASAT executes 100% correct merges in 90% of the experimental runs, whereas DFASAT introduces errors. Notably, the remaining 10% of cases in which PAT-DFASAT commits merge errors correspond to highly challenging instances where all evaluated learners failed to avoid incorrect merges. Conversely, while PAT-EDSM also achieves a 100% correct merge rate in 90% of the runs at this same data scale, it fails to capture the correct language specification. This limitation stems from severe under-merging, leaving many valid state pairs unmerged. Ultimately, when the data size increases to $\text{Data} \times 16$, all candidate learners converge on the correct target automata.

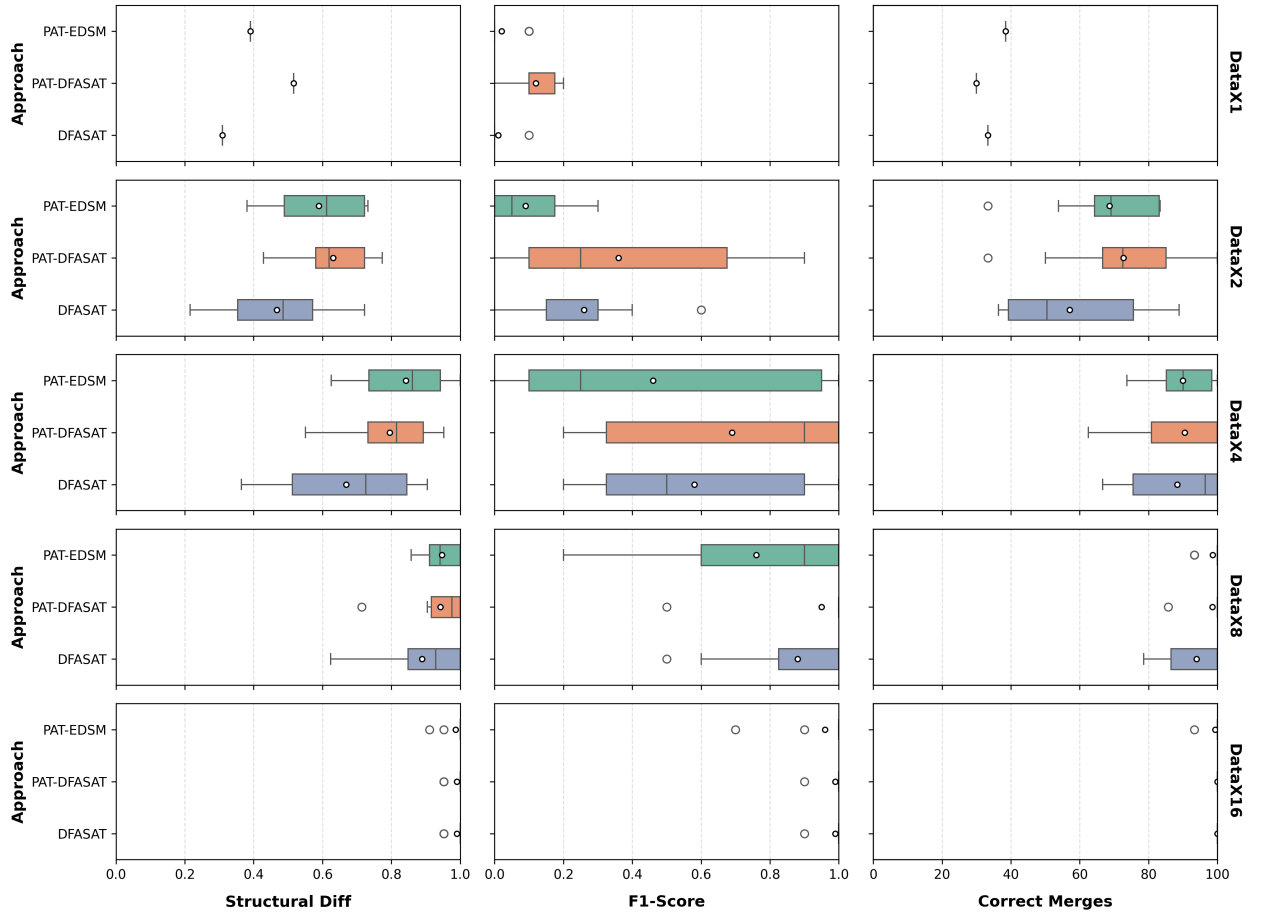


Figure 5.24: Results for the Random1 use case with different data sizes

In general, across most data sizes, all candidate learners have comparable execution times, completing in just a few seconds. However, an interesting anomaly occurs at the $\text{Data} \times 4$ scale, where DFASAT exhibits a much wider variance in execution latency, ranging from a few seconds to more than one minute. At this scale, DFASAT exhibits a low rate of precise merges; this structural inaccuracy directly inflates the SAT search space, thereby increasing the time required for the solver to converge on a valid solution. Another interesting result emerges at the $\text{Data} \times 16$ scale,

where all evaluated learners correctly identify the target model, yet their execution runtimes differ significantly (DFASAT has a narrower range than PAT-EDSM). For DFASAT, providing a small

volume of training data yields rapid but incorrect results. However, introducing a moderate amount of data complicates the state-merging process, thereby expanding the variance in execution times. This runtime range is reduced again once a sufficient data is provided to adequately constrain the search space.

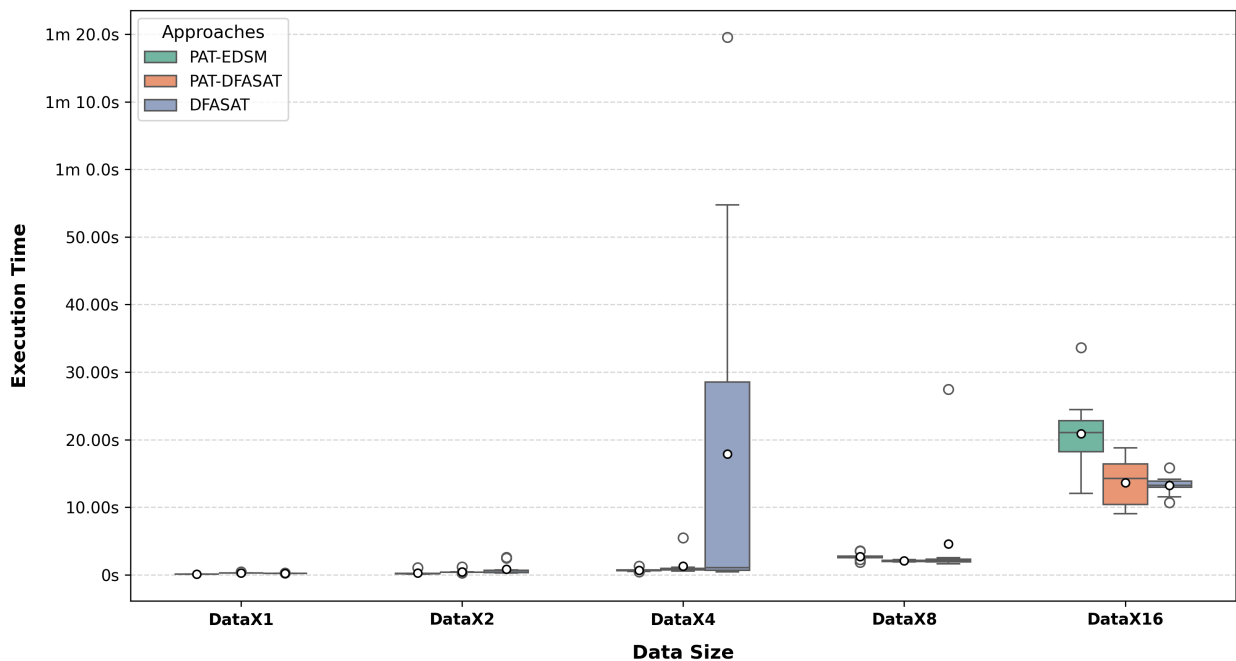


Figure 5.25: *Random1 - Runtime results*

At small data scales (Data $\times$ 1 to Data $\times$ 4), none of the learners approximate the target model. However, with the minimum amount of data, PAT-DFASAT outperforms both DFASAT and PAT-EDSM in SD-score and $F1$ -Score, despite PAT-EDSM yielding a higher proportion of correct merges at this scale. However, at this scale (Data $\times$ 1) the set of values across all metrics demonstrates statistical significance. As data size increases (Data $\times$ 2 and Data $\times$ 4), both PAT-DFASAT and PAT-EDSM achieve superior SD scores and correct merge ratios relative to standard DFASAT and demonstrate no statistical significance. From a language acceptance perspective, however, PAT-DFASAT and DFASAT consistently outperform PAT-EDSM and demonstrate statistical significance with high uncertainty. At a size of (Data $\times$ 8), PAT-DFASAT achieves a slight advantage over all alternative learners across all evaluated metrics with no statistical significance. Ultimately, all learners converge to similar performance with a data size of Data $\times$ 16.

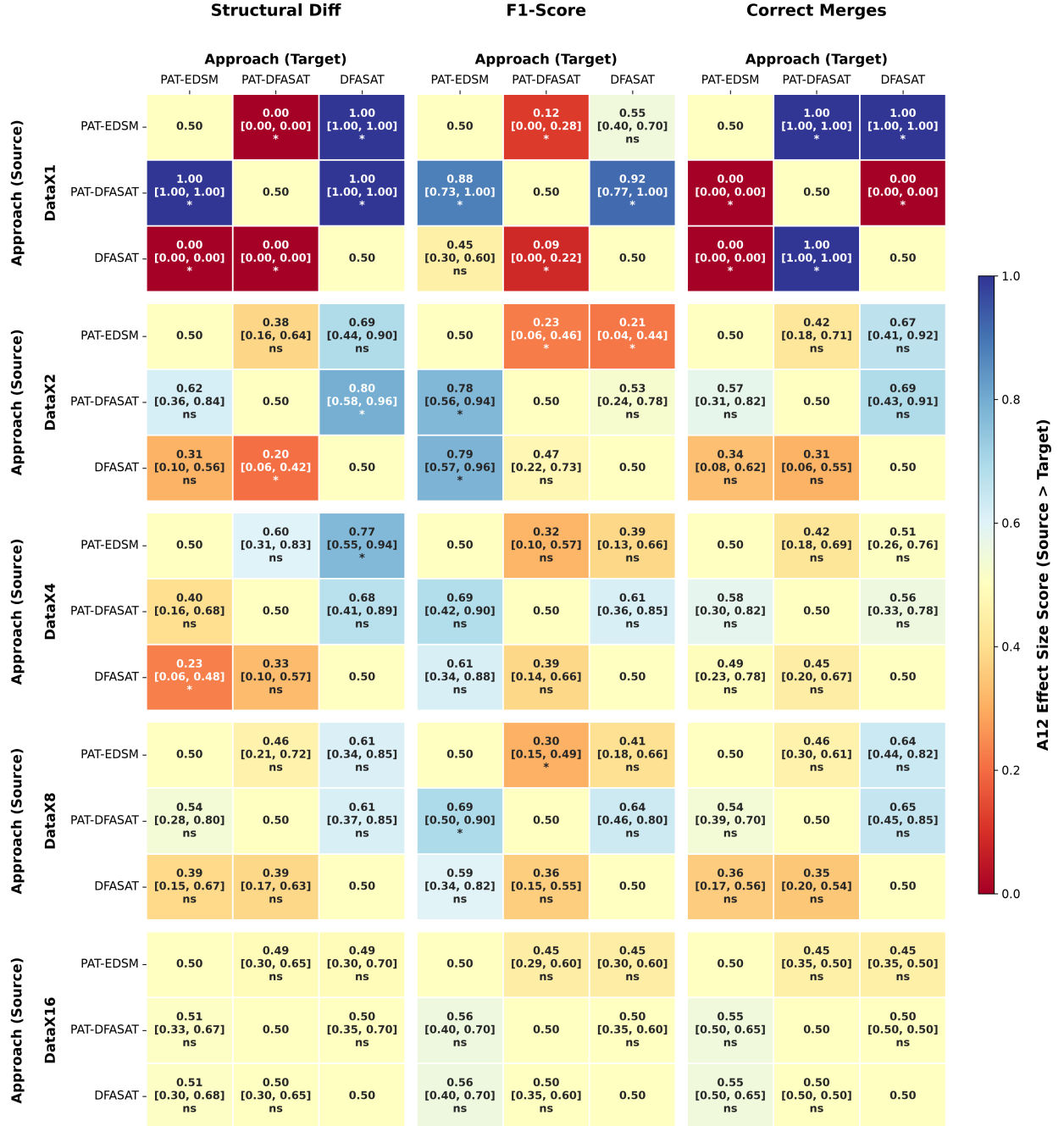


Figure 5.26: Random1 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

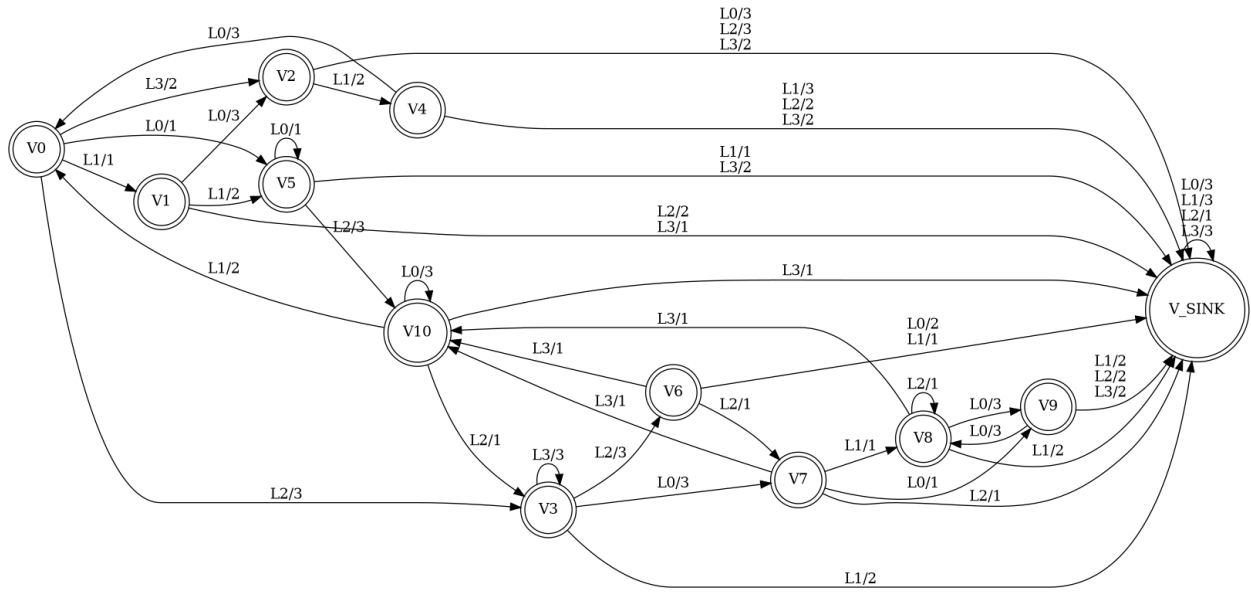
Random2

Figure 5.27: *Random2* use case - Reference Automata (the initial state is V0)

Random2

In the Random2 use case (Figure 5.28), PAT-DFASAT performs merges with fewer mistakes than DFASAT under extremely sparse data (Data $\times$ 2 or less). Nevertheless, the SAT solver remains prone to error due to data sparsity. Even when PAT-DFASAT achieves a 100% correct merge rate (such as at Data $\times$ 4), the subsequent SAT solver layer produces automata exhibiting high SD-scores but sub-optimal $F1$ -scores. Notably, as the training data scales, the performance of PAT-DFASAT and DFASAT converges, becoming identical across all metrics. Conversely, PAT-EDSM consistently underperforms relative to the other two learners across all data sizes.

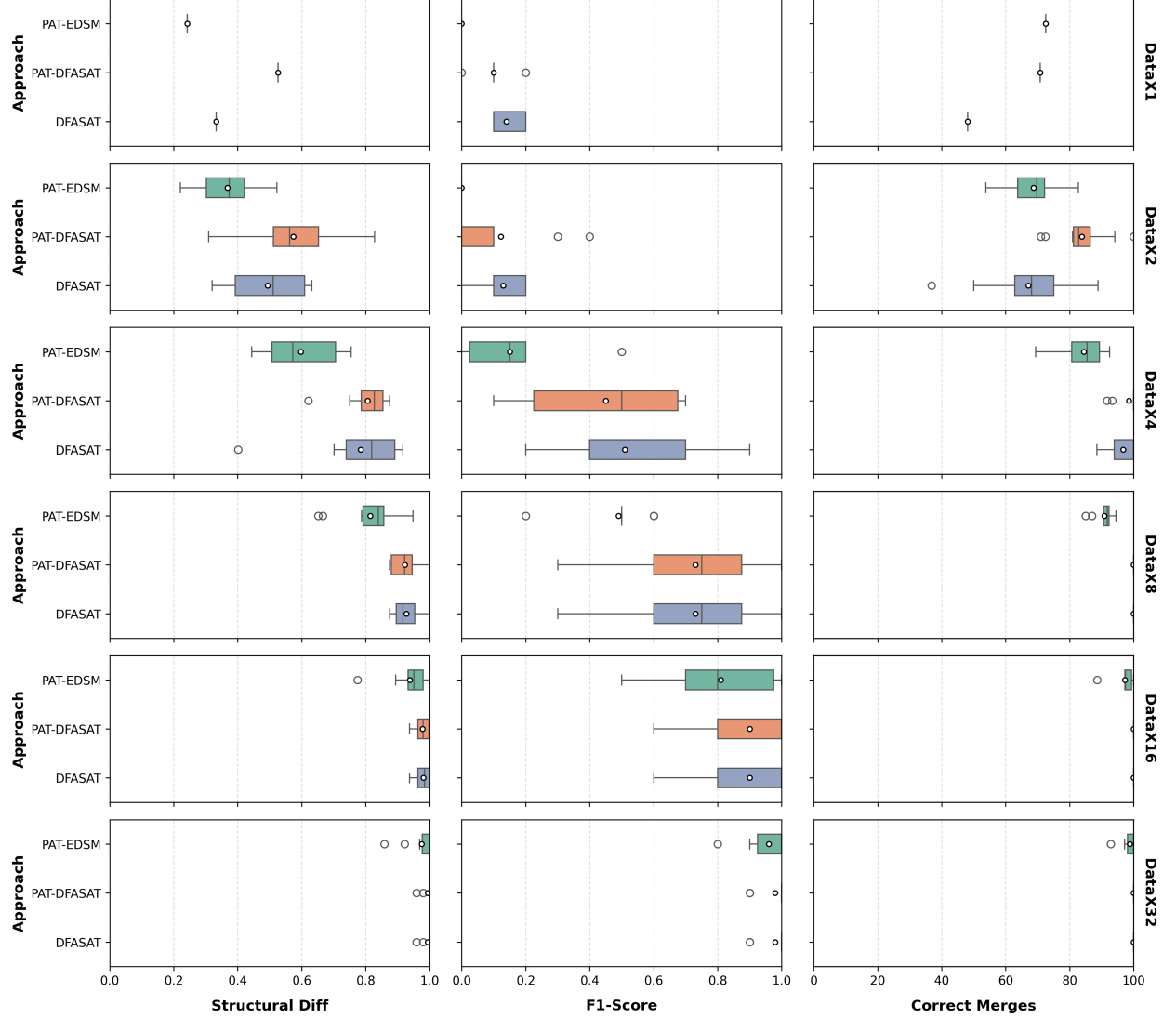


Figure 5.28: Results for the Random2 use case with different data sizes

Across all evaluated data sizes, PAT-EDSM is the fastest learner; however, at the largest scale (Data $\times 32$), it becomes the slowest framework. Meanwhile, PAT-DFASAT and DFASAT consume approximately the same execution time at data scales of Data $\times 4$ and beyond, a trend that aligns with the convergence observed in their statistical performance metrics. Conversely, for smaller data sizes (Data $\times 1$ and Data $\times 2$), PAT-DFASAT has a faster runtime than standard DFASAT. This performance divergence directly reflects the precision of their initial state merges, as PAT-DFASAT introduces fewer heuristic errors during the early stages of induction. Notably, the variance in execution times increases at moderate data scales (e.g., Data $\times 2$), a trend that is particularly pronounced for DFASAT and PAT-DFASAT. This variance then gradually decreases as a larger amount of data is provided to the learners.

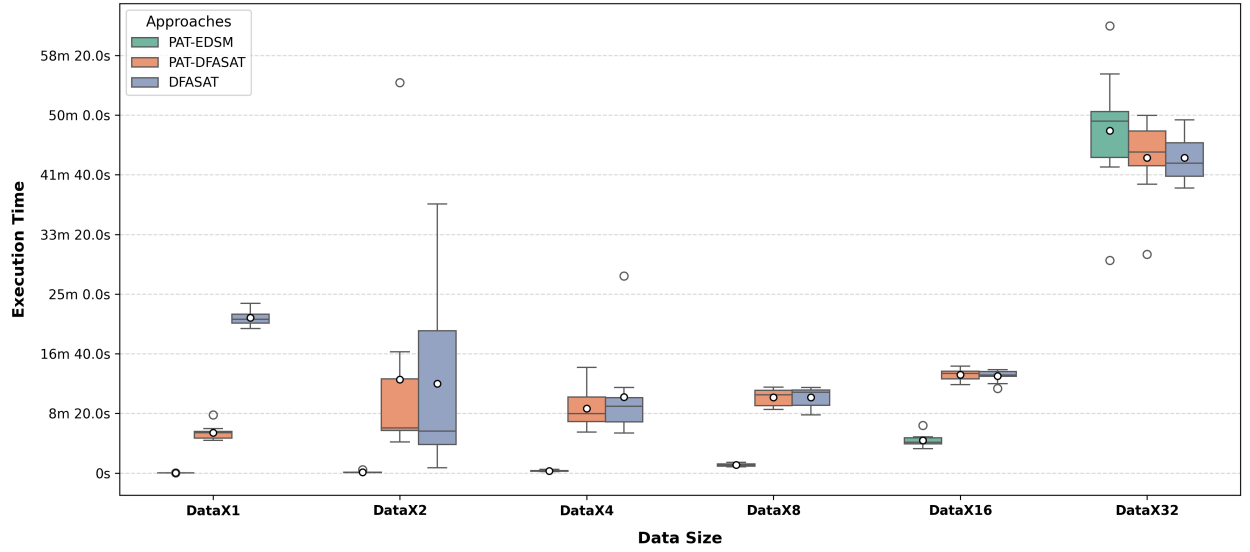


Figure 5.29: *Random2 - Runtime results*

Notably, at a data density of $\text{Data} \times 4$, PAT-DFASAT constructs automata that closely resemble the target model's structural topology, yet fail to accurately approximate the language. For instance, Figure 5.31 illustrates an inferred automaton achieving a high structural similarity (SD) score of 0.88, yet yielding a low F_1 -score of 0.50. In this visualisation, green edges denote missing transitions that must be added, whereas red edges represent incorrect transitions that must be removed to recover the exact target model. The majority of the states and transitions are correctly identified, with only five transitions incorrectly directed to erroneous target states. This language-structure decoupling is driven by misdirected transitions affecting early sequence paths. Specifically, the majority of accepted traces starting with the input-output sequence $\langle \{L2, 2\}, \{L2, 2\}, \dots \rangle$ are incorrectly rejected because they lead to state $S7$, which lacks the subsequent valid paths. Similarly, traces beginning with the prefix $\langle \{L2, 2\}, \{L0, 2\}, \dots \rangle$ transition into state $S8$, where they are also erroneously rejected. For example, the accepted trace $\langle \{L2, 2\}, \{L2, 2\}, \{L3, 1\}, \{L3, 1\}, \{L3, 1\} \rangle$ is rejected by the inferred model. The last three transitions of the trace correspond to a self-loop around state $S7$, whereas the trace leads to the sink state in the inferred model. Recall that, for evaluation purposes, we consider traces that terminate at the sink state to be rejected. Because these early, high-frequency paths are blocked by just a few misplaced transitions, the F_1 -score drops drastically despite the overall state graph being nearly isomorphic to the target machine.

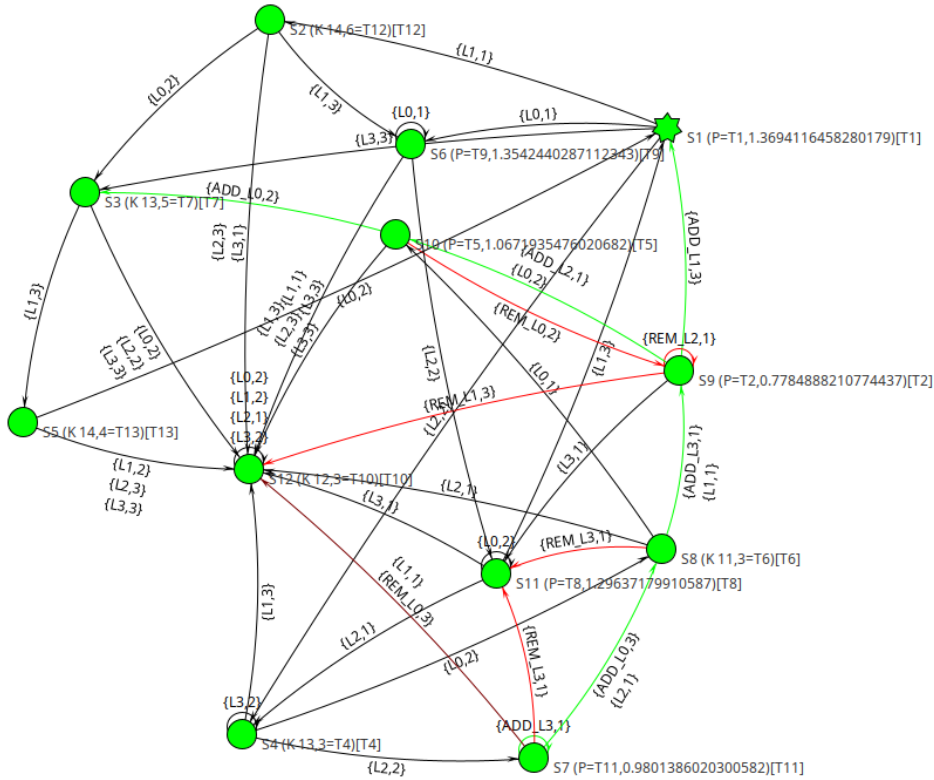


Figure 5.31: Inferred model by PAT-DFASAT for the *Random2* use case

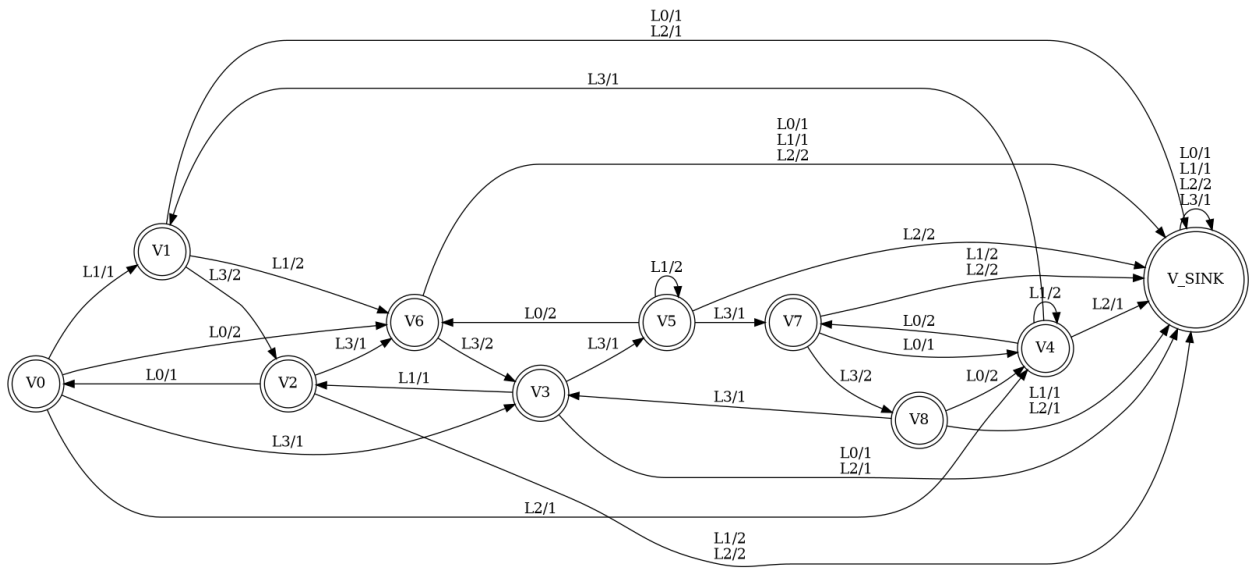
Random3

Figure 5.32: *Random3* use case - Reference Automata (the initial state is V0)

Random3

Similar to the Random2 use case, Figure 5.33 demonstrates that, in the Random3 use case, PAT-DFASAT executes state merges with significantly fewer mistakes than DFASAT with sparse data size ($\text{Data} \times 4$ or less). This initial merging phase enables the downstream solver to construct solutions with high SD-scores (in $\text{Data} \times 4$) ranging from 0.70 to 0.90, though these solutions exhibit a high variance in their $F1$ -scores, spanning a wide range from 0.10 to 0.90. At a data size of $\text{Data} \times 8$, PAT-DFASAT performs correct merges, yet the solver makes mistakes. As the data increases ($\text{Data} \times 16$ and beyond), both PAT-DFASAT and DFASAT achieve identical metric values. Conversely, the PAT-EDSM pipeline consistently demonstrates the lowest performance across all evaluated data dimensions.

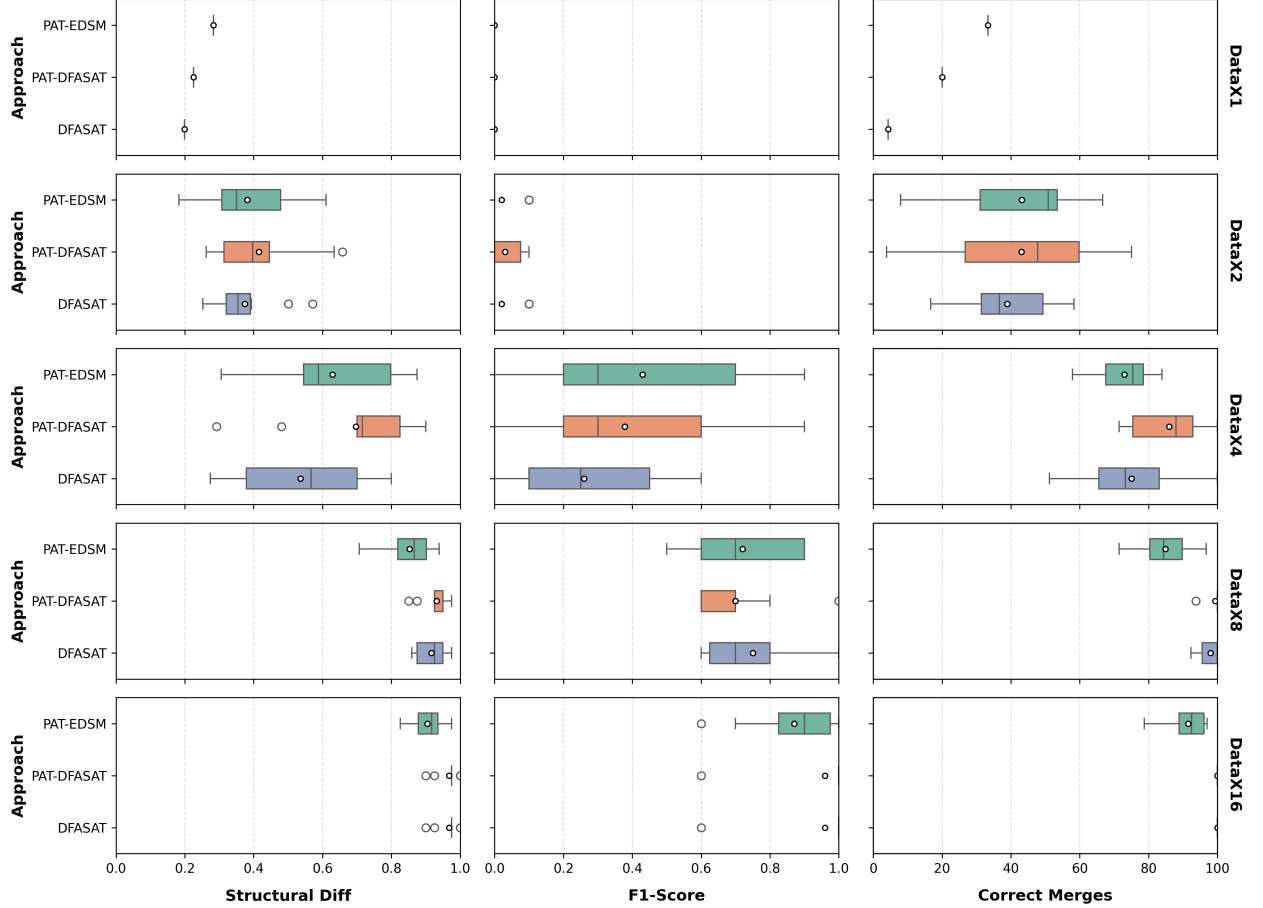


Figure 5.33: Results for the Random3 use case with different data sizes

As illustrated in Figure 5.34, at smaller data sizes (Data $\times$ 1 to Data $\times$ 4), the DFASAT operates as the slowest candidate learner. At these scales, DFASAT executes a greater number of incorrect state merges during its initial phase. As the data size increases, DFASAT optimises precision toward a higher proportion of correct merges; this structural improvement is directly reflected in its runtime, which scales down to match that of the other learners. Notably, although PAT-EDSM fails to outperform the SAT-based learners across the other evaluation metrics, it maintains a highly comparable execution time across all evaluated data scales. Notably, the variance in execution times increases sharply at moderate data scales (e.g., Data $\times$ 2 and Data $\times$ 4), a trend that is particularly pronounced for the DFASAT and PAT-DFASAT frameworks. This variance subsequently contracts as a larger amount of data is fed to the learners.

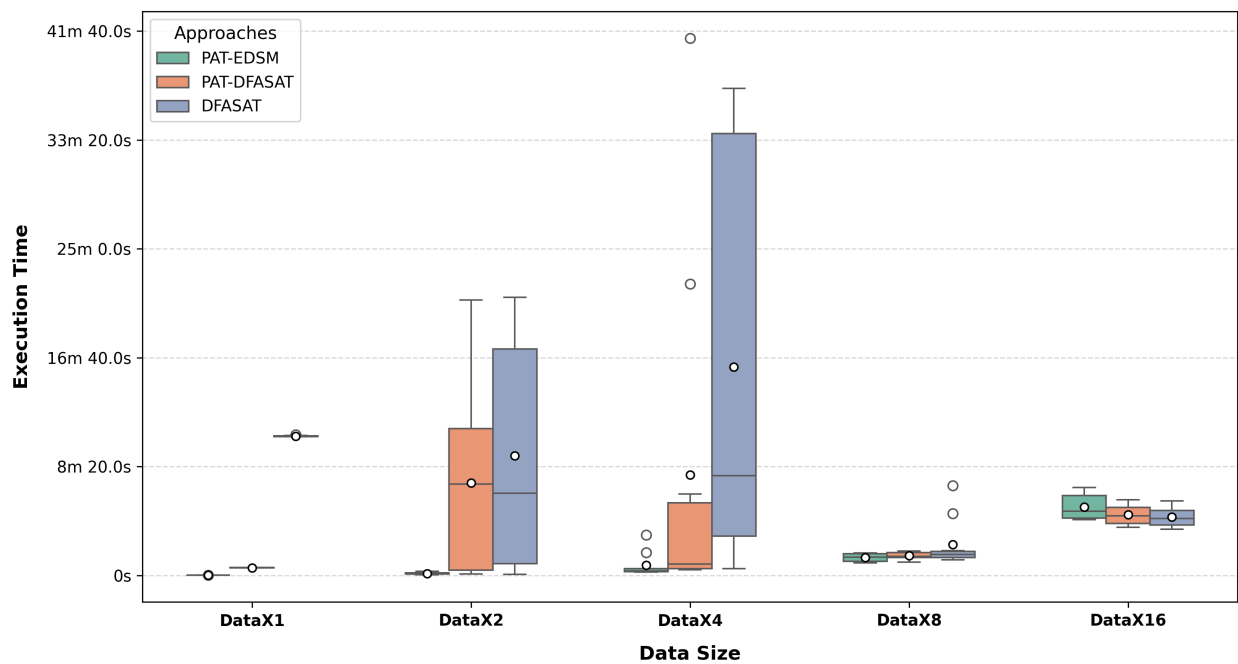


Figure 5.34: *Random3 - Runtime results*

At the minimum data density ($\text{Data} \times 1$), PAT-EDSM outperforms both PAT-DFASAT and DFASAT in SD-scores and the proportion of correct merges; however, all three learners fail to approximate the target model. At intermediate data scales ($\text{Data} \times 4$ and $\text{Data} \times 8$), PAT-DFASAT slightly outperforms DFASAT and PAT-EDSM across all evaluated metrics with no statistical significance. Once the training data size increases ($\text{Data} \times 16$ and beyond), PAT-DFASAT and DFASAT converge to produce isomorphic automata, while PAT-EDSM consistently fails to match their performance ceiling.

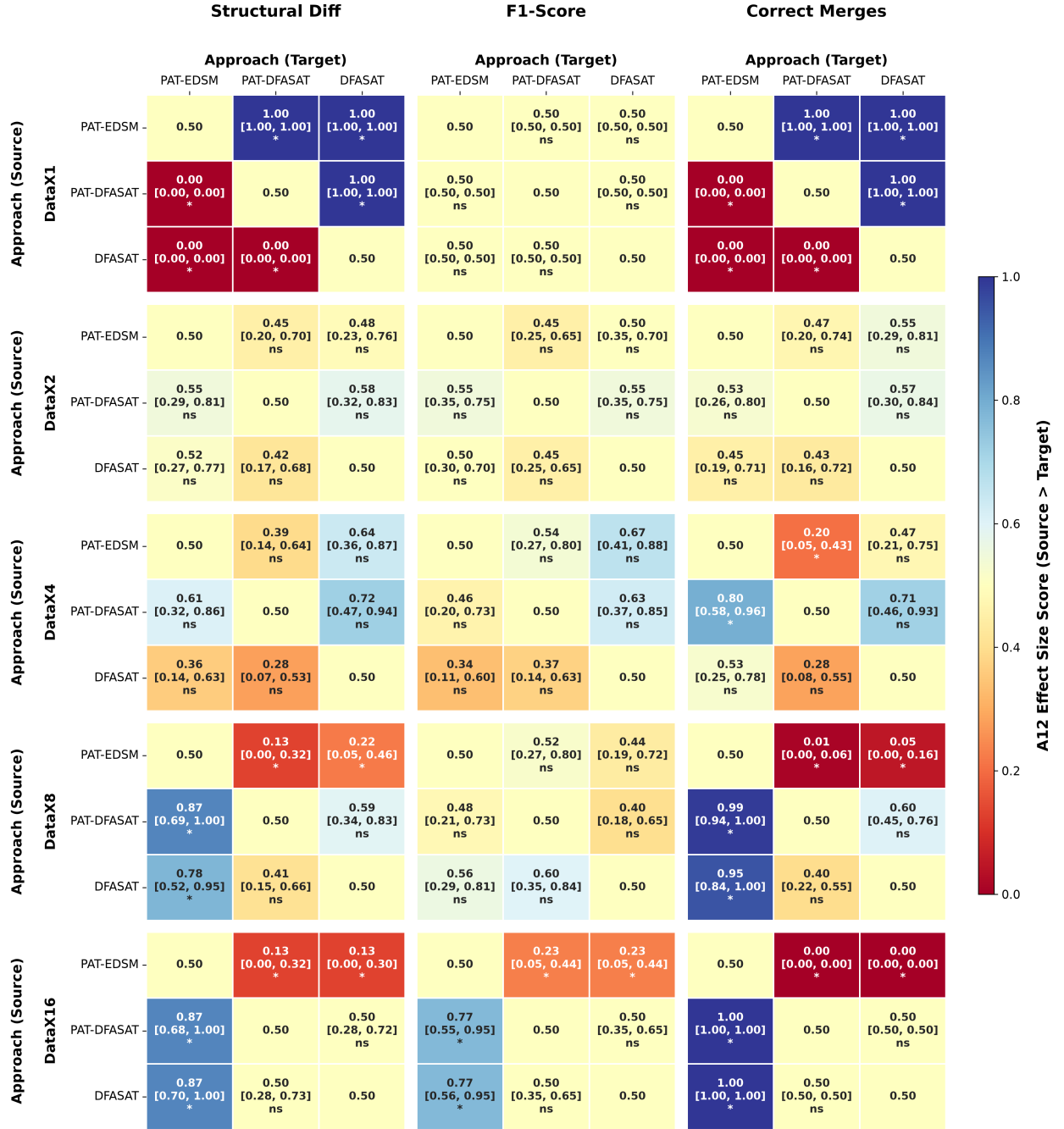


Figure 5.35: *Random3* - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

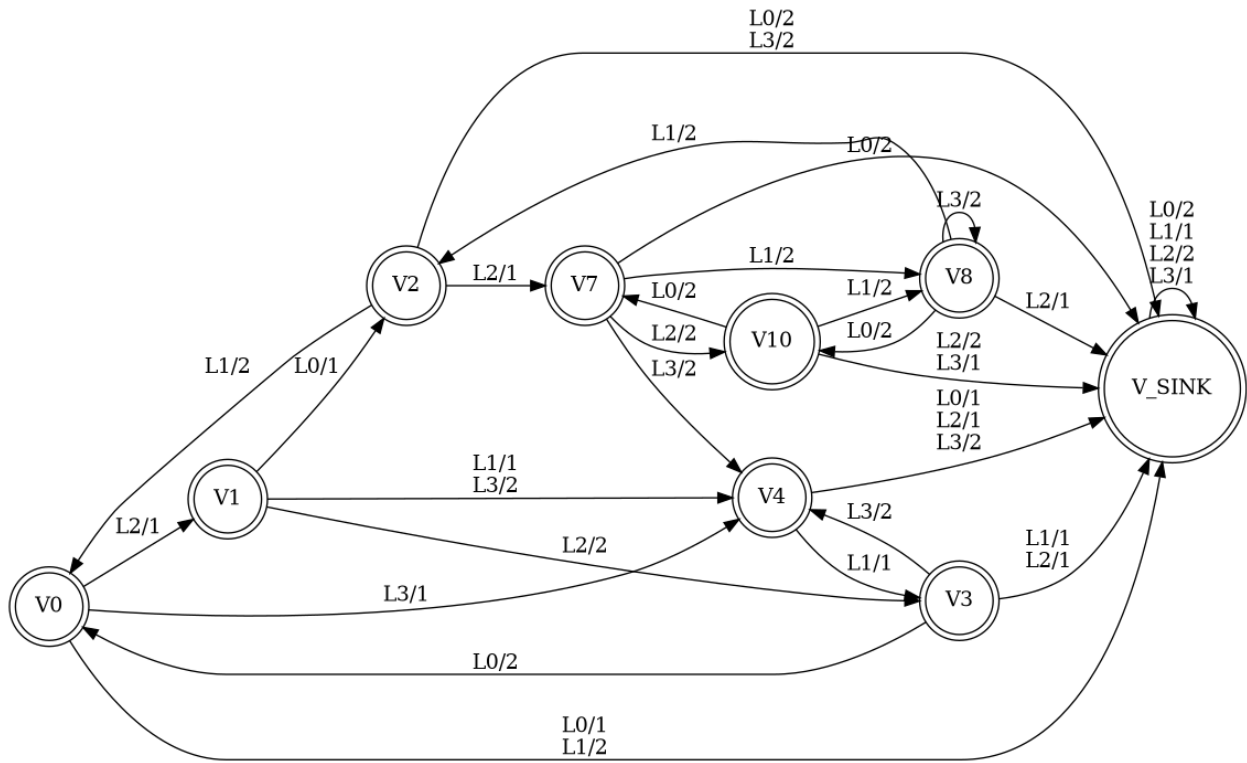
Random4

Figure 5.36: *Random4* use case - Reference Automata (the initial state is V0)

Random4

In the Random4 use case (Figure 5.37), both PAT-DFASAT and DFASAT exhibit comparable scores at data size of (Data $\times$ 4) and beyond. However, with extremely sparse data (Data $\times$ 2), PAT-DFASAT demonstrates a wider dispersion of values than DFASAT. Specifically, PAT-DFASAT had SD-scores between 0.5 and 0.8, $F1$ -scores between 0.1 and 0.9, and 90-100% correct merge rate. Meanwhile, DFASAT at the same scale exhibits SD-scores between 0.5 and 0.7, $F1$ -scores between 0.1 and 0.4, and a matching correct merge rate of 90% to 100%. As the data size increased, the performance of both learners converged to identical scores, whereas the PAT-EDSM pipeline consistently achieved the lowest performance across all evaluated data sizes.

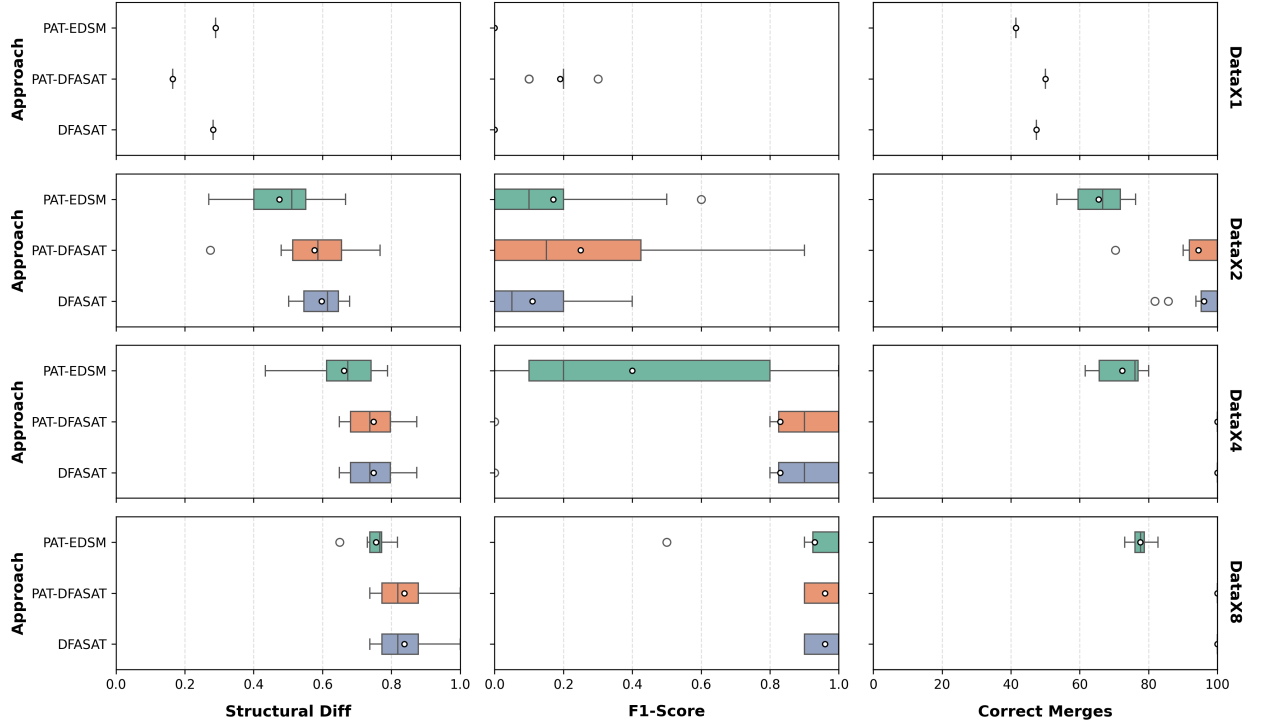


Figure 5.37: Results for the Random4 use case with different data sizes

Figure 5.38 illustrates that across almost all evaluated data sizes, all candidate learners exhibit highly comparable execution profiles. However, an interesting anomaly occurs at the minimum data scale (Data $\times$ 1), where PAT-DFASAT incurs a significantly longer runtime than the other frameworks. While PAT-DFASAT requires more than 5 minutes to complete execution under these highly constrained conditions, PAT-EDSM and the baseline DFASAT framework require only a few seconds. At the Data $\times$ 2 scale, an anomalous outlier was observed, with the execution runtime

substantially higher than that of all other data points. This latency directly reflects an inflation of the search space for the SAT solver, combined with a scarcity of evidence to guide early, precise state merges. This explanation is further supported by the correspondingly low SD and $F1$ -scores observed at this specific scale.

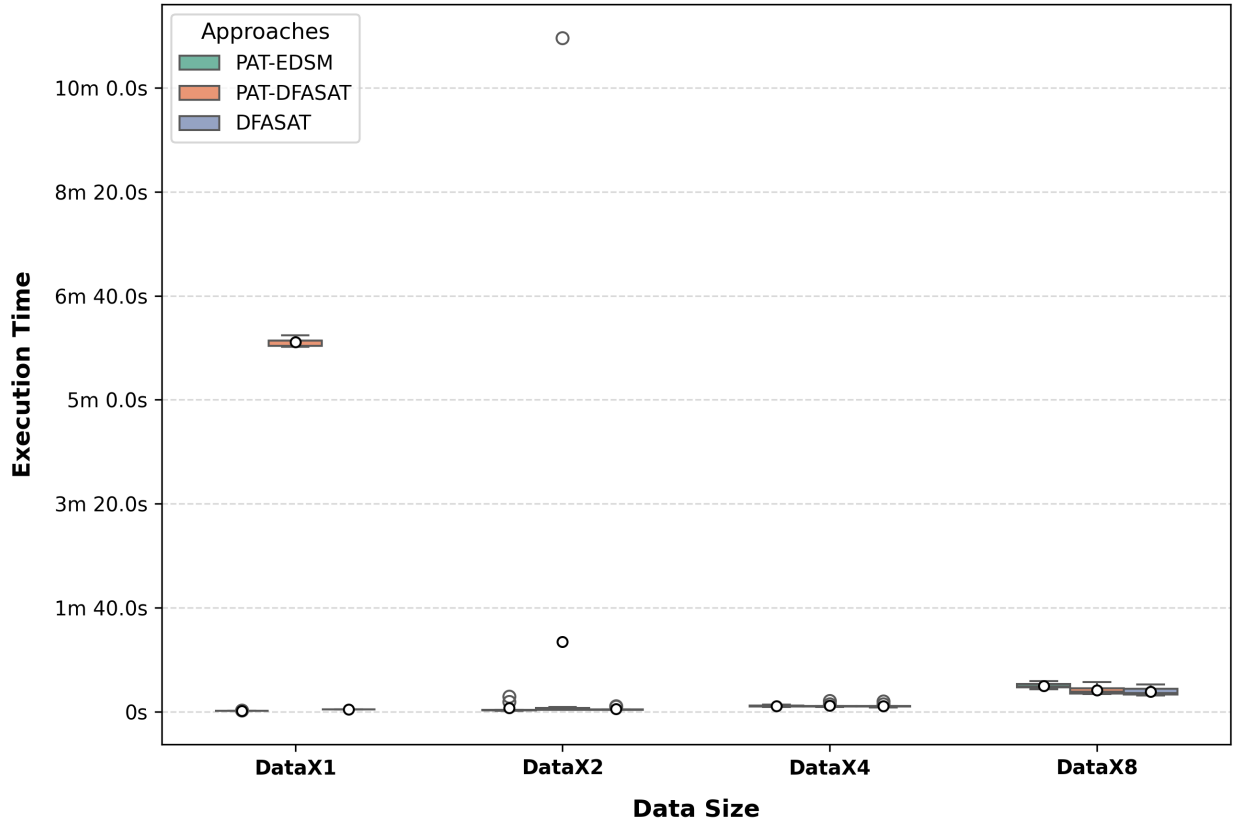


Figure 5.38: *Random4* - Runtime results

Although none of the learners is able to detect the target model across the evaluated data, Figure 5.39 demonstrate the overall performance among them. With the smallest data set (Data $\times$ 1), PAT-EDSM structurally outperforms both PAT-DFASAT and DFASAT, yielding higher SD-scores. Conversely, at this data scale, PAT-DFASAT exhibits superior language accuracy and produces a higher proportion of correct merges than either of the other learners. With more data, both PAT-DFASAT and DFASAT outperform PAT-EDSM across all metrics, while simultaneously converging toward similar performance relative to one another. The 95% confidence intervals demonstrate either no statistical significance or uncertainty.

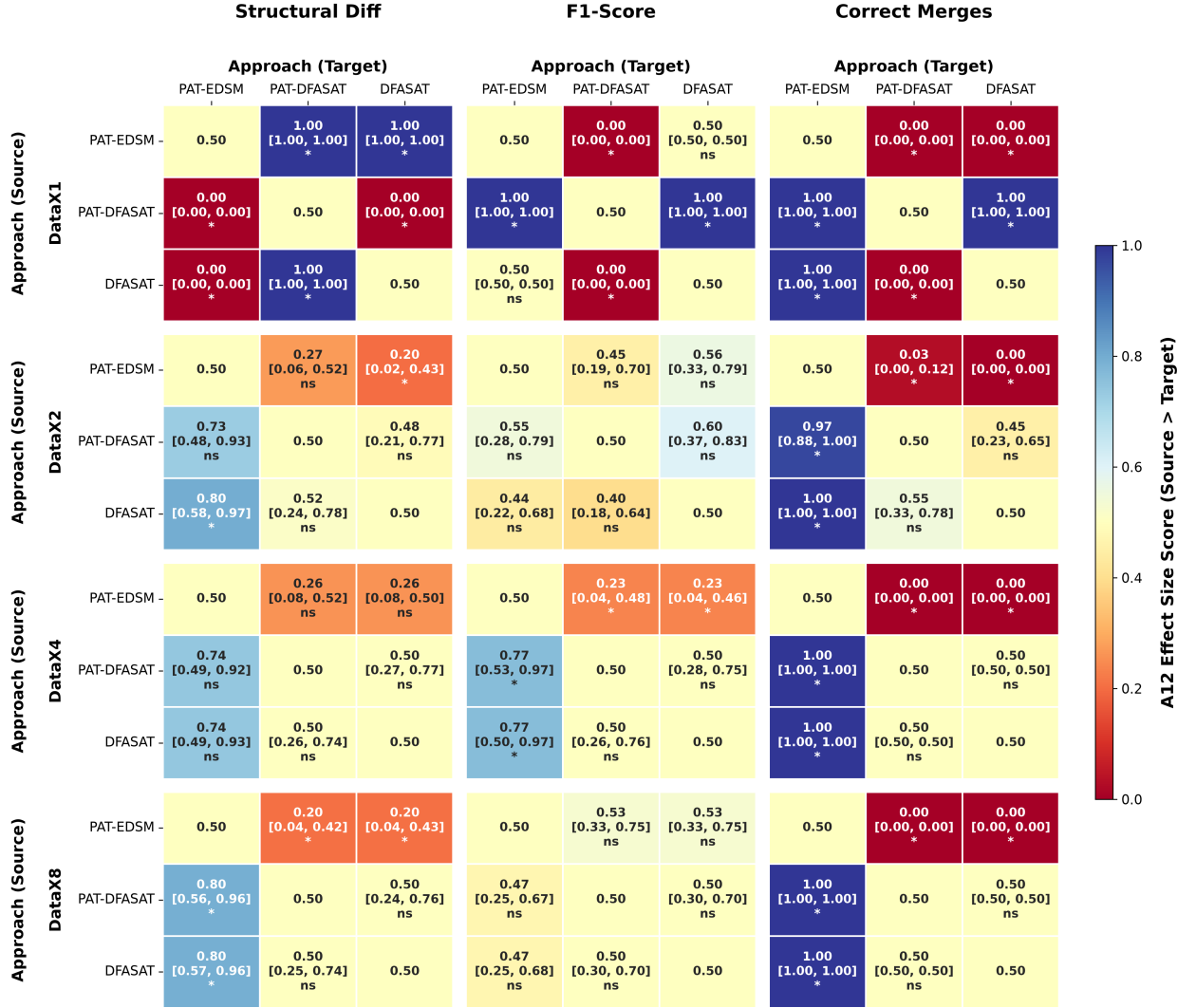


Figure 5.39: *Random4* - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

Figure 5.38 illustrates that across almost all evaluated data sizes, all candidate learners exhibit highly comparable execution profiles. However, an interesting anomaly occurs at the minimum data scale ($\text{Data} \times 2$), where SAT-based learners incur a significantly longer runtime than PAT-EDSM. While SAT-based methods require time ranging from a few seconds to over 10 minutes, the PAT-EDSM framework requires only a few seconds. Notably, at this scale, the SAT-based learners make some initial merging errors that affect SAT search time. However, this optimises as data scales at ($\text{Data} \times 4$) when both learners perform only correct merges. Notably, the variance in execution times increases sharply at moderate data scales (e.g., $\text{Data} \times 2$), a trend that is particularly pronounced for the DFASAT and PAT-DFASAT frameworks. This variance subsequently contracts as a larger amount of data is fed to the learners.

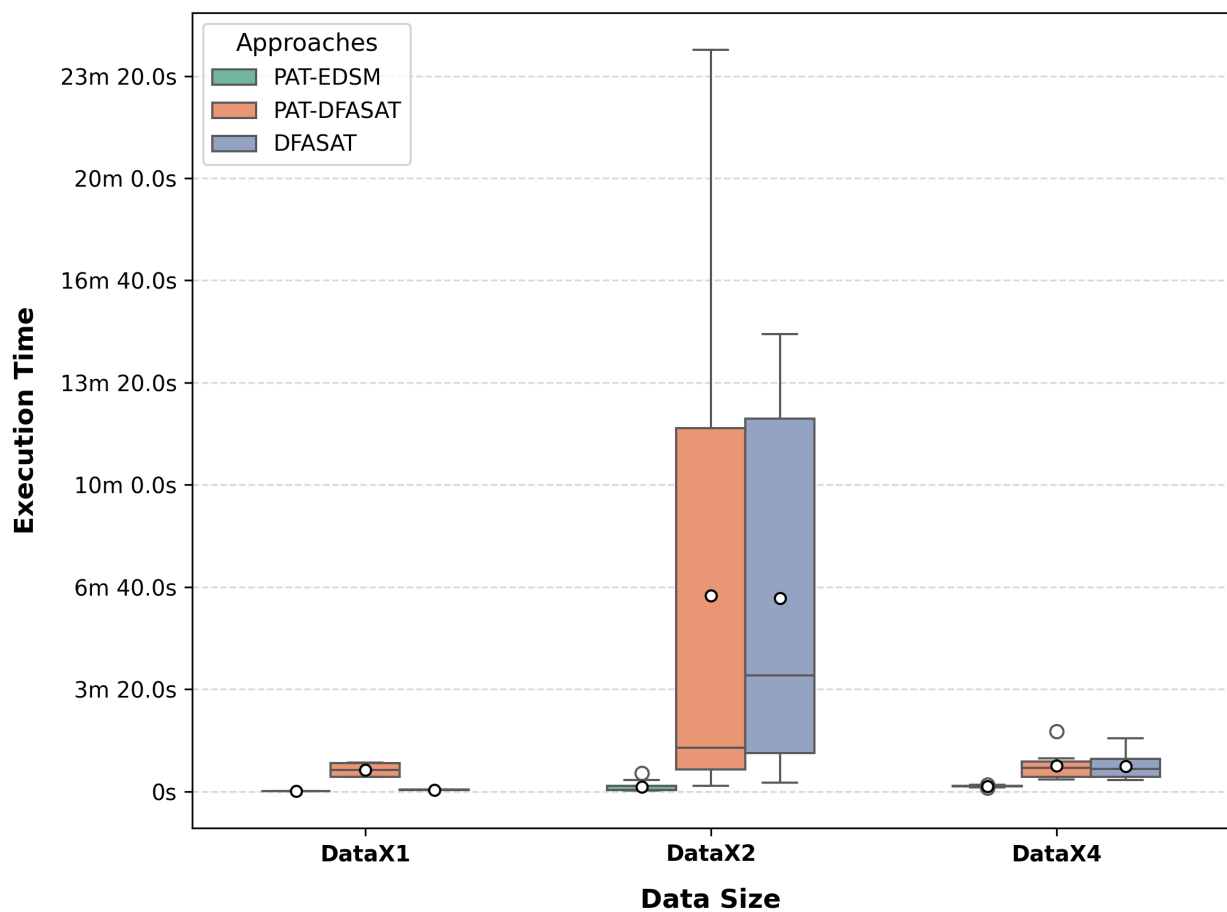


Figure 5.42: *Random5 - Runtime results*

Although none of the learners is able to detect the target model across the evaluated data, Figure 5.43 demonstrate the overall performance among them. At the smallest data size, DFASAT outperforms both PAT-DFASAT and PAT-EDSM in SD-scores and $F1$ -scores. However, at the same data size, PAT-DFASAT performs a higher proportion of correct merges than the alternative learners. As data size increases ($\text{Data} \times 4$), the performance of PAT-DFASAT and DFASAT converges, yielding highly comparable results. Ultimately, both configurations outperform the PAT-EDSM pipeline across all evaluated metrics. The 95% confidence intervals demonstrate either no statistical significance or uncertainty.

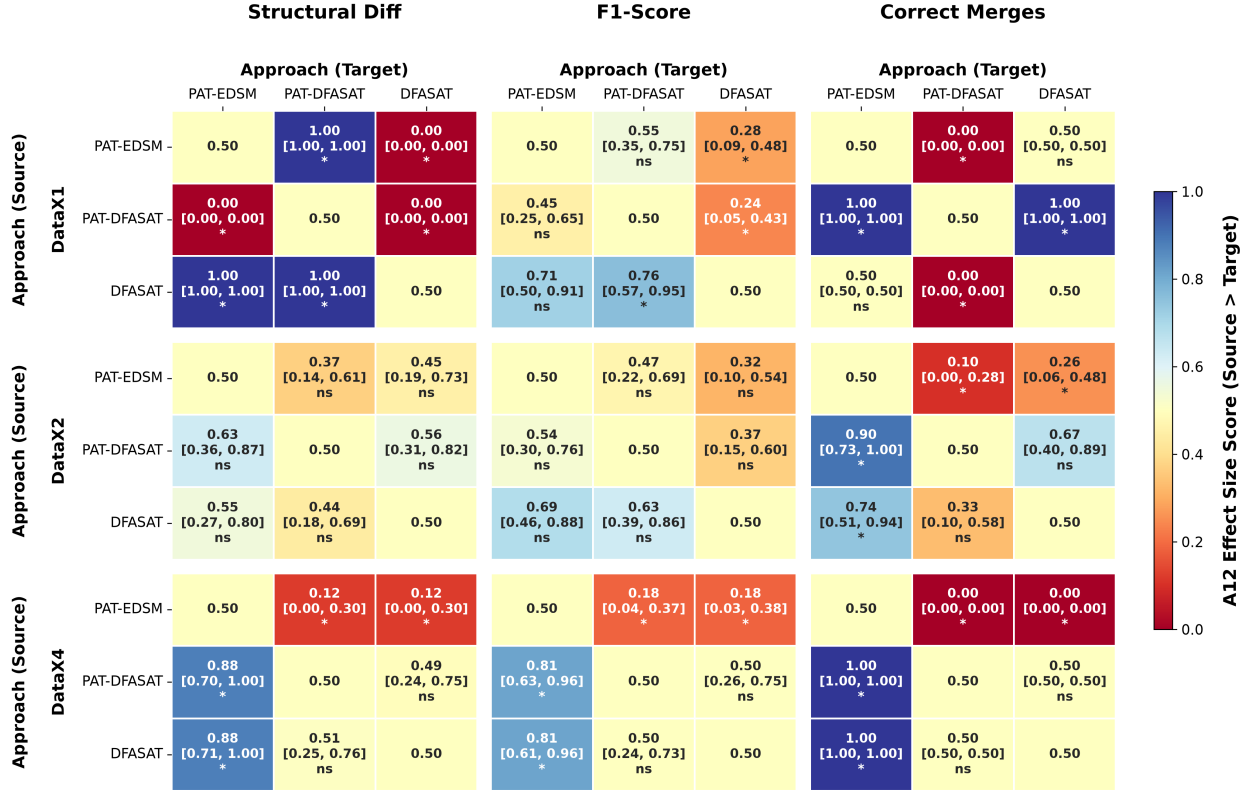
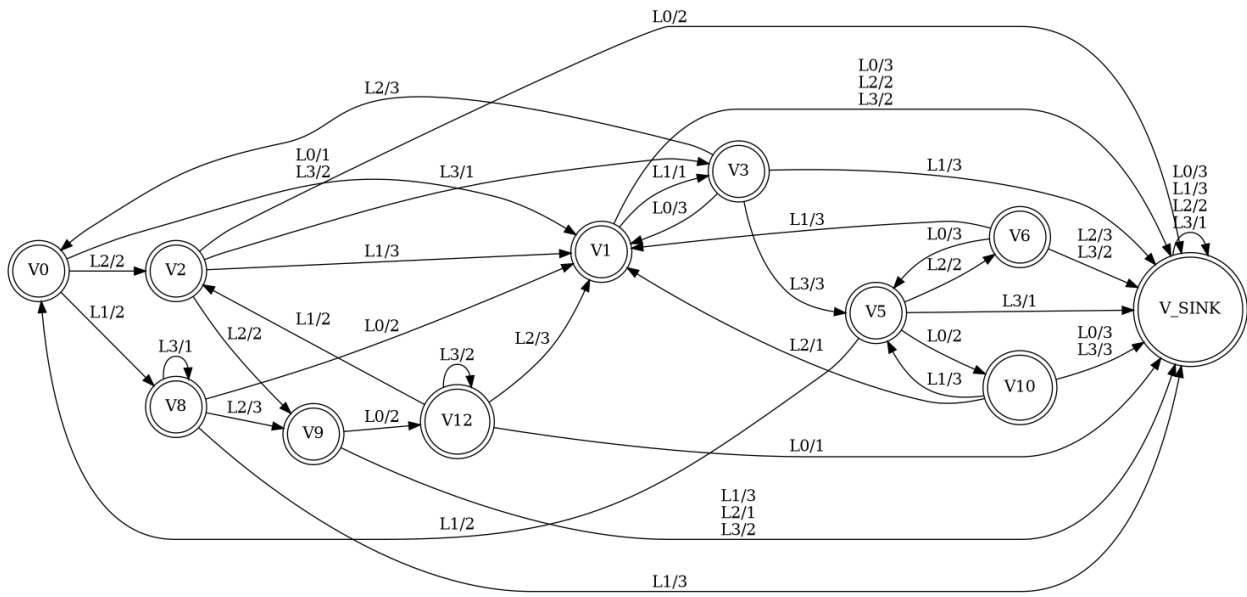


Figure 5.43: *Random5 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)*

Random6**Figure 5.44:** *Random6 use case - Reference Automata (the initial state is V0)*

Random6

Random6 use case is similar to Random4 and Random5. PAT-DFASAT and DFASAT outperform PAT-EDSM across all data sizes. They also exhibited either similar performance (with small data sizes) or identical scores (larger data sizes). At a data size of ($\text{Data} \times 4$), they start to reach a high proportion of correct merges (≥ 0.90) while struggling to approximate the target model. At a data size of ($\text{Data} \times 8$) and beyond, the SAT-based learners achieve 100% correct merges, yet the SAT makes mistakes. Ultimately, at the size of ($\text{Data} \times 32$), both PAT-DFASAT and DFASAT produce isomorphic automata.

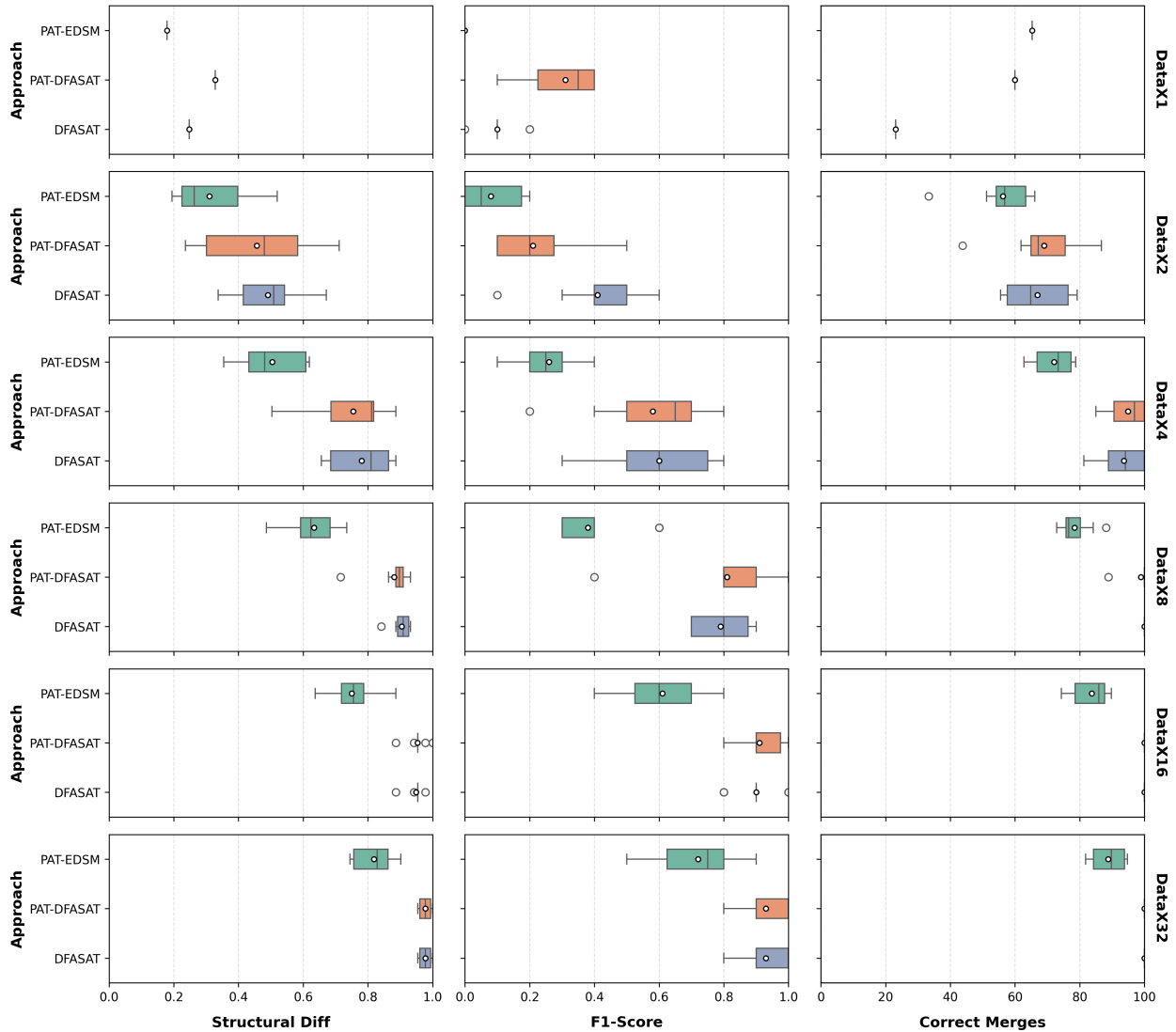


Figure 5.45: Results for the Random6 use case with different data sizes

Across nearly all evaluated data scales, PAT-EDSM executes faster than alternative frameworks, despite achieving the lowest accuracy when approximating the target model. Conversely, the SAT-based learners require a considerably longer runtime to complete execution. However, achieving a high proportion of correct merges during the initial heuristic phase significantly optimises their computational performance, specifically at data scales of $\text{Data} \times 8$ and beyond, allowing them to converge more efficiently. Notably, the variance in execution times increases sharply at moderate

data scales (e.g., $\text{Data} \times 2$ and $\text{Data} \times 4$), a trend that is particularly pronounced for the DFASAT and PAT-DFASAT frameworks. This variance gradually decreases as a larger amount of data is provided to the learners.

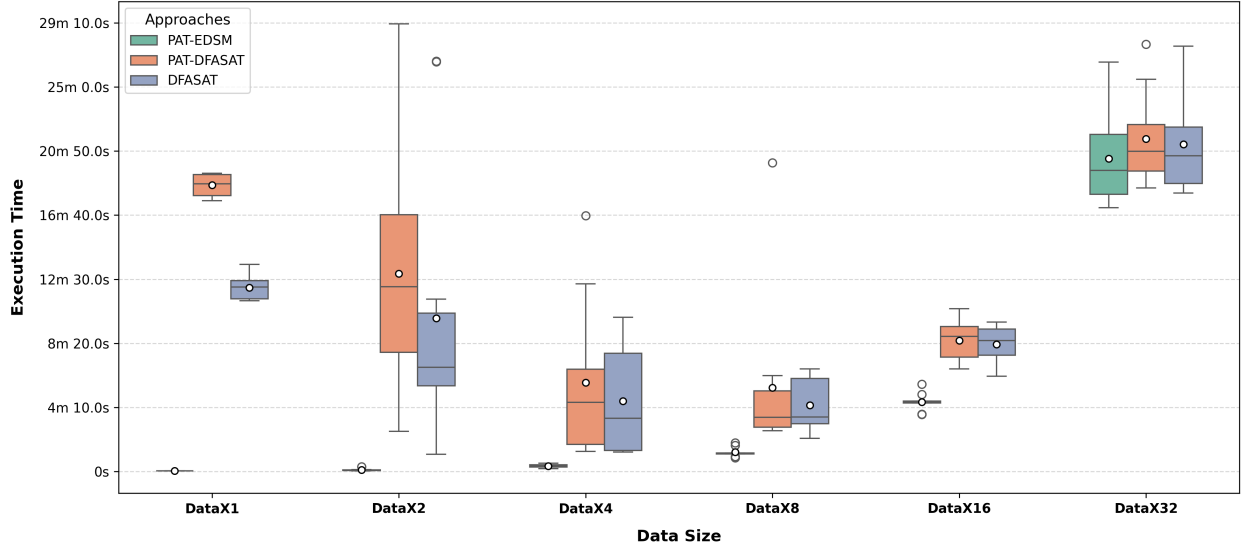


Figure 5.46: *Random6 - Runtime results*

Across all data sizes, both PAT-DFASAT and DFASAT consistently outperform PAT-EDSM across all metrics with statistical significance (and high precision and high stability at data size (Data $\times$ 8 and more). At a data size of (Data $\times$ 2), although none of the learners approximates the target model, DFASAT outperforms PAT-DFASAT in terms of the $F1$ -score and achieves a marginally superior SD-score. However, this trend does not hold for the proportion of correct merges, in which PAT-DFASAT exhibits superior merge accuracy to DFASAT. Ultimately, as the training data size scales further (Data $\times$ 2 and beyond), both learners converge toward a statistically comparable performance profile. The 95% confidence intervals between PAT-DFASAT and DFASAT demonstrate no statistical significance.

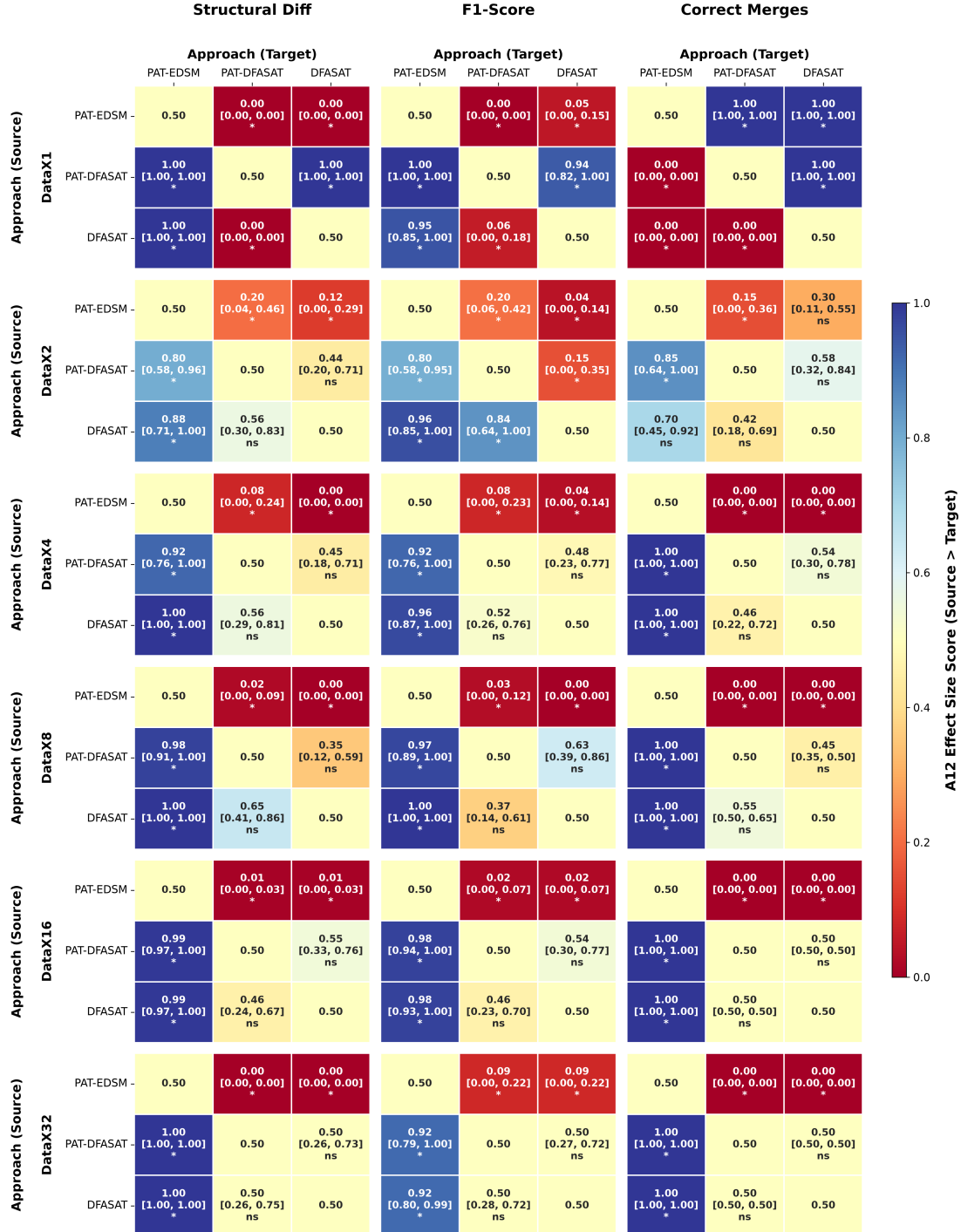


Figure 5.47: Random6 - A12 results: comparing the learners' performance (ns: statistically non-significant - *: statistically significant)

5.1.3 Results Summary

Across the diverse structural benchmarks and training data scales evaluated in this work, the primary empirical findings can be summarised as follows:

Results for Real-World Automata Benchmarks

- TCP Framework:** Under extreme data sparsity ($\text{Data} \times 1$), all three candidate learners failed to achieve significant SD-scores or $F1$ -scores. Nevertheless, even within these sparse data bounds, PAT-DFASAT demonstrated a superior capability to detect and reject incorrect merges compared to the alternative frameworks. As the training data size increases, the performance of PAT-DFASAT and the baseline DFASAT converge to yield identical metric values, whereas the PAT-EDSM pipeline consistently fails to achieve parity with the SAT-based approaches. The runtime for the TCP use case varies from a few minutes to more than one hour. The precision of early merges affects the SAT solver time for PAT-DFASAT and DFASAT; incorrect early merges increase the search time to find a valid solution. When the learner performs correct merges, the runtime is reduced significantly compared to a small data size with inaccurate merges.
- Bluetooth Protocol:** Both PAT-EDSM and PAT-DFASAT systematically outperform standard DFASAT. At a data density of $\text{Data} \times 32$, the baseline DFASAT framework fails to generate a valid solution in 50% of the experimental runs, whereas PAT-DFASAT exhibits robust, superior performance across all three evaluation criteria. Crucially, at intermediate data scales ($\text{Data} \times 4$ and $\text{Data} \times 8$), the downstream SAT solver fails to resolve a satisfiable model within its designated upper bound. At this specific scale, the initial EDSM phase introduces early merge errors because the residual model space remains large; the solver struggles to compute a valid state mapping consistent with that compromised initial topology. The overall runtime for the Bluetooth use case is the longest among all evaluated use cases. On average, the learners require a few hours to successfully complete the process. However, SAT-based learners take more time (compared to PAT-EDSM) when the initial merges are incorrect. With more accurate initial merges, all learners operate in convergent timing.
- OpenSSL State Machine:** PAT-DFASAT achieves the highest overall SD score among all evaluated learners. While PAT-EDSM fails to accurately capture either the language or structural accuracy — despite achieving a high proportion ($\approx 90\%$) of correct merges — both DFASAT and PAT-DFASAT exhibit a high variance in their recorded $F1$ -scores. Notably, under highly constrained data conditions ($\text{Data} \times 4$), PAT-DFASAT leverages its pattern heuristics to successfully orchestrate accurate initial merges, whereas both standard DFASAT and PAT-EDSM introduce early errors. OpenSSL requires a few seconds to process a small amount of data ($\text{Data} \times 8$ and less). However, execution time increases sharply as data size increases ($\text{Data} \times 16$ and beyond).
- Coffee Machine Control System:** The empirical results indicate that PAT-EDSM surpasses both SAT-based methods across different data sizes. However, at the size of $\text{Data} \times 32$, PAT-EDSM outperforms both SAT-based methods in terms of SD-scores, while all three configurations successfully converge to capture the complete target language. In terms of state-merging precision, at the same scale, both DFASAT and PAT-DFASAT exhibit an identical distribution of values ranging from 75% to 100%, while PAT-EDSM minimises structural deviations with a tighter precision interval of 85% to 100%. Regarding the runtime, the Coffee Machine use case requires only a few seconds among all learners. However, DFASAT shows a wide range of execution times for data sizes of $\text{Data} \times 16$. At this scale, both PAT-DFASAT and DFASAT produce isomorphic automata, yet they differ in their execution time. DFASAT ranges from a few seconds to more than a minute, while its opponent, PAT-DFASAT, does

not exceed 20 seconds. This variance indicates that PAT-DFASAT has a more efficient state-merging strategy than DFASAT.

- **ABP Sender Protocol:** The empirical evaluation of the ABP Sender case study demonstrates distinct structural and computational performance characteristics across the evaluated learners. At lower data scales ranging from Data $\times$ 1 to Data $\times$ 4, PAT-DFASAT consistently outperforms both PAT-EDSM and the baseline DFASAT in terms of structural difference and the proportion of correct merges. Despite this relative superiority, extreme data sparsity at these lower thresholds prevents all learners from accurately capturing the target language.

Notably, at the Data $\times$ 4 scale, PAT-DFASAT achieves a high proportion of correct merges, falling between 90% and 100%. However, this high precision in state merging does not yield a fully accurate model, as its structural difference score remains moderate at approximately 0.7, and F_1 -scores across all evaluated frameworks remain low. This data scale (Data $\times$ 4) had enough information to mine patterns that prevent incorrect initial merges by PAT-DFASAT. However, the partially merged automata and the remaining unmerged states lack the information needed for the SAT solver to produce an accurate model in terms of structure and language.

Regarding computational overhead, the inference process for the ABP Sender requires several hours to complete across all algorithmic variants. The overall peak execution time occurs at the Data $\times$ 4 scale for the unconstrained DFASAT learner, which exhibits extreme search-space explosion resulting in outlier runs exceeding two days. At this specific data scale, all learners display substantial variance in execution time across independent experimental runs. As the available trace volume increases, this computational variance narrows significantly. At higher scales such as Data $\times$ 8 and Data $\times$ 16, execution times for both PAT-DFASAT and DFASAT stabilise into predictable performance baselines of approximately three hours and seven hours, respectively.

Results for Synthetic Random Automata

The six random automaton configurations are categorised into three primary representative use cases: Random1, Random2, and Random4 (with the behavioural dynamics of Random3 aligning with Random2 and Random6, and Random4 mirroring Random5). Across all data sizes and evaluation scenarios, the PAT-EDSM framework demonstrates an inferior performance relative to the SAT-based pipelines across all three metrics. Consequently, these three core use cases summarise the results of PAT-DFASAT relative to the baseline DFASAT framework as follows:

1. **PAT-DFASAT Dominance (Random1):** The mined structural patterns provide an advantage at a data density of Data $\times$ 8, enabling the initial EDSM phase to execute high-fidelity state merges (achieving a 100% correct merge rate in 90% of the experimental runs). Concurrently, this high-quality partial model prunes the downstream search space, allowing the SAT solver to construct highly accurate automata ($SD \geq 0.90$) while reliably preserving the language specification.
2. **High-Ratio Correct Merges with Solver-Induced Errors (Random2):** At a restricted data size (Data $\times$ 4), PAT-DFASAT achieves a 100% correct merge rate during its initial phase, yet the downstream SAT solver remains susceptible to optimisation errors due to severe data sparsity. The solver requires additional trace constraints to assign structural labels accurately, though both DFASAT and PAT-DFASAT eventually converge to structurally identical automata as data increases.
3. **Statistical Parity across learners (Random4):** PAT-DFASAT and the baseline DFASAT framework learn isomorphic automata overall. However, under extreme data sparsity (Data $\times$

2), PAT-DFASAT occasionally yields a superior SD-scores, even when it fails to fully capture the underlying language.

In general, the execution times for the randomly generated automata are highly comparable among the evaluated learners. However, when SAT-based approaches fail to avoid incorrect state merges during the initial heuristic phase, the downstream SAT solver struggles to identify a valid solution within a reasonable time frame. This structural inaccuracy results in a significant divergence between the computational runtime of PAT-EDSM and that of SAT-based frameworks, with the latter exhibiting pronounced execution latency. Moreover, the results show that the high variance of runtime values at a specific data size indicates that the learners need more data

Results for 95% Bootstrap Confidence Interval The calculation of the 95% bootstrap confidence interval for the A_{12} effect size across all evaluated case studies yields two distinct empirical scenarios:

- **Large Effect Size ($A_{12} \geq 0.9$):** When A_{12} indicates a large effect size of 0.9 or higher, the corresponding bootstrap confidence interval demonstrates statistical significance, exhibiting a narrow range bounded by high values ($\approx [0.9, 1.0]$). This tight interval, coupled with elevated A_{12} scores, confirms the systematic superiority of Group 1.
- **Moderate Effect Size ($0.3 \leq A_{12} \leq 0.8$):** When A_{12} reflects a moderate effect size between 0.3 and 0.8, the confidence interval reveals either a lack of statistical significance or a substantial interval width. This observed variance is attributable to the modest sample size per group (10 independent runs per data scale), which naturally yields wider confidence bounds under non-parametric resampling.

5.1.4 Discussion

Research Objective 1 (**RO1**) addresses the utilisation of domain knowledge to advance passive model inference. In this thesis, we proposed a lightweight pattern-mining strategy and evaluated two methodologies for integrating these constraints into state-of-the-art passive inference algorithms, specifically EDSM and DFASAT. Our empirical results demonstrate that integrating a SAT solver into the pattern-based model inference process systematically improves the learner’s overall performance.

Across all evaluated use cases, the SAT-based learners consistently outperform the PAT-EDSM framework. The singular exception occurs in the *Coffee Machine* use case, where the downstream SAT solver fails to distinguish between two nearly identical, structurally equivalent states. However, such behavioural similarity between states in the target model is difficult for an automated learner to detect directly, typically requiring a domain expert to identify the equivalent states. If this domain knowledge is available a priori, selecting the PAT-EDSM framework over the SAT-based approaches becomes the more effective design choice.

Research Objective 2 **RO2** addresses the impact of mined patterns on inferred model accuracy in terms of $F1$ -scores (**RQ1-(A)**) and Structural Difference (**RQ1-(B)**). Our findings illustrate that PAT-DFASAT consistently outperforms PAT-EDSM across these evaluation criteria while either outperforming or maintaining statistical parity with the baseline DFASAT framework.

Research Question 2 (**RQ2**) explicitly investigates the effect of the mined patterns on the fidelity of early state merges between DFASAT and PAT-DFASAT. Crucially, the initial EDSM phase remains vital; it not only prunes the search space for the downstream solver but also provides

essential structural constraints that prevent the solver from over-merging states or failing to find a reasonable solution.

The primary architectural divergence between these two SAT-based approaches lies in their respective heuristics for pruning or blocking incorrect state merges during the initial EDSM phase. Empirically, our mined pairwise patterns guide the learning process toward a significantly higher proportion of correct initial merges under conditions of extreme data sparsity. Conversely, the baseline DFASAT heuristic is inherently restrictive; while this strictness facilitates high-fidelity decisions given high data density, its efficacy degrades markedly as training data becomes increasingly sparse.

In scenarios characterised by an extensive alphabet size and severely constrained training data, the probability of multiple transition traversals decreases sharply as the dataset shrinks. This effect is especially present for deep states situated far from the root or initial state. Under these sparse-data conditions, the standard DFASAT heuristic acts as an artificial bottleneck, inhibiting the learner’s ability to perform valid state merges due to an apparent data-induced lack of similarity between target state pairs. Because all traces originate from the initial state, they naturally share a small subset of transitions at the beginning of the sequences before rapidly diverging onto distinct path trajectories as they traverse deeper into the graph.

Our mined pattern-based approach is an effective alternative for mitigating this constraint. However, the downstream SAT solver remains susceptible to over-merging states when the data is exceptionally sparse, a behavioural drift that manifests even when PAT-DFASAT successfully performs accurate initial merges. Despite this solver-level limitation, optimising the initial merging phase remains vital to overall inductive accuracy, as the SAT solver cannot backtrack to undo earlier erroneous merge decisions. Consequently, by securing higher-fidelity early merges, our pattern-based methodology systematically achieves superior structural and behavioural accuracy overall.

Research Objective 3 **RO3** (addressed by Research Question 3 **RQ3**) explicitly addresses the effect of the mined patterns on the performance of the model-inference process in terms of execution time. The empirical results demonstrate that the precision of the initial merges in the SAT-based approaches directly affects the downstream search time required for the solver to find a valid solution. Specifically, when the underlying EDSM heuristic introduces structural errors, the SAT solver struggles to successfully merge the remaining states with the partially incorrect model. Consequently, the state-space exploration becomes computationally expensive, leading to a substantial inflation in overall execution time. Conversely, when the initial merges are guided correctly, the solver rapidly identifies a valid (and typically more accurate) solution, achieving runtimes that are either highly comparable to or faster than those of PAT-EDSM.

Furthermore, we observed a correlation between the variance of execution runtimes and the scale of the training dataset. This variance indicates whether a given amount of data is sufficient for learners to converge efficiently. This trend is particularly pronounced within the PAT-DFASAT and DFASAT frameworks, manifesting consistently across all evaluated case studies, with the sole exceptions of the *coffee machine* and *OpenSSL* use cases.

Under conditions of extreme data sparsity, the learners process different execution runs within highly comparable time frames, regardless of the ultimate accuracy of the inferred models. Conversely, when the provided data volume is sufficient, learners also complete the induction process within a similarly tight time frame and successfully converge on highly accurate models. The most notable runtime variance manifests between these two data-scale extremes. At moderate data points, the learners are provided with larger amounts of data that are nonetheless insufficient to establish robust structural evidence for precise state merging. In these scenarios, the learners are prone to making strategic mistakes during the early heuristic stages, which substantially inflates the time required to evaluate and select subsequent state pairs. Ultimately, these early, incorrect

structural merges compound downstream, severely hindering the convergence speed and overall execution time of the SAT solver.

We also investigated the direct integration of mined patterns into the SAT solver phase. The results illustrate no significant performance variance between configurations that incorporate the patterns at the solver level and those that omit them. In practice, patterns are extracted directly from the existing training data; when the mined patterns fail to adequately capture the true behaviour of the target model, integrating these uninformative patterns into the solver yields no optimisation benefits. Conversely, when the dataset is sufficiently rich to guide the learner toward correct state reductions, the downstream solver inherently possesses adequate logical constraints to merge the remaining states accurately without requiring supplementary pattern heuristics.

Given that the baseline DFASAT heuristic remains prone to early merging errors under sparse data conditions, we also evaluated the concurrent integration of mined patterns into both the DFASAT heuristic and the subsequent SAT phase. The evaluations revealed no statistically significant improvements when combining patterns with the baseline DFASAT heuristic. This lack of variance stems from the fact that the standard DFASAT heuristic is structurally more rigid than our pattern set; any candidate merge rejected by the baseline DFASAT framework is always rejected by our patterns, but the reverse does not hold. Additionally, appending the patterns to the SAT phase after applying the DFASAT heuristic provided no material benefit, for the reasons detailed previously.

In summary, among all algorithmic pipelines evaluated in this thesis, PAT-DFASAT represents the most robust and statistically superior framework for passive automata induction within highly constrained, sparse data environments.

5.2 Threats to Validity

In this section, we analyse the structural and execution configurations in our experimental setup that, if altered, could affect the empirical outcomes. These considerations are categorised into *external threats*, which pertain to the structural characteristics of the target models, and *internal threats*, which relate to the algorithmic configurations and implementation details of the induction pipelines.

5.2.1 External Threats to Validity

1. **Sink-State vs Self-Loop Topologies:** In our experimental design, all evaluated automata handle the input-enabled property utilising a sink-state architecture (see Section 2.1.1). Implementing alternative structural configurations, such as explicit self-loops, could potentially influence the behavioural profile and final evaluation results of the learners.
2. **Scale and State Cardinality:** The evaluated use cases are restricted to relatively small models, with the state count $|Q|$ ranging between 6 and 15 states. Inductive inference on models characterised by higher state cardinalities might alter the scalability and precision metrics observed in this study.
3. **Alphabet and Input/Output Dimensions:** The evaluated benchmark use cases feature input and output alphabet sizes bounded by $|I| \approx |Q| \times [0.5, 1.1]$ and $|O| \approx |Q| \times [0.5, 1.1]$, respectively. Transitioning to systems with massive alphabets relative to the number of states $|Q|$, such as $|Q| \times 2$, could affect learning performance.

Consequently, the scope of our empirical generalisations is bounded by small-to-medium-scale automata sharing a uniform topological handling of input completeness via a sink state.

5.2.2 Internal Threats to Validity

1. **Strictly Positive Merge Scoring Thresholds (score > 0):** We restrict state-merging operations exclusively to candidate pairs demonstrating positive evidence (score > 0). In practice, a score of 0 denotes a lack of overlapping evidence rather than a logical contradiction, meaning it does not violate any mined patterns and could represent a valid merge. Allowing zero-score merges might alter the final structural topology; however, it introduces an aggressive risk of early incorrect merges when no shared transitions exist between states near the root.
2. **SAT Solver Execution Triggers:** The hybrid pipeline executes initial EDSM merges until the size of the inferred model is reduced to half the cardinality of the original Prefix Tree Acceptor (PTA). Altering this execution policy — such as triggering the SAT solver when the highest pair score falls below a specific threshold, or when the model reaches a pre-specified absolute size (as implemented in the standard DFASAT algorithm) — could affect the overall evaluation results.
3. **Solver Timeouts and Termination Criteria:** Our pipeline enforces a strict five-minute computational timeout per solver instance to optimise processing resources during the satisfiability search. Altering this threshold, expanding the execution window, or removing the limit entirely could affect final induction results.
4. **DFASAT Minimum Colour Lower Bounds:** Our custom pipeline initialises the search space at an absolute structural minimum of 2 states, allowing the solver to search for the smallest model. Adjusting this initialisation floor upward or utilising a more restrictive threshold mechanism could contract the search space and alter the behavioural accuracy of the final automata.
5. **SAT maximum colour upper bounds:** To prevent the SAT solver from unbounded expansion and to manage computational complexity, we impose a strict termination limit where the search bounds are capped at twice the state cardinality of the target automaton ($|Q_{\text{target}}| \times 2$). If the solver cannot identify a satisfying assignment within this colour ceiling, it reports that no solution was found and terminates the search. Modifying this multiplier or setting a less restrictive upper bound could allow the discovery of larger, over-fitted state machines, thereby changing the recorded evaluation results.
6. **Increasing Independent Runs per Case Study:** A potential threat to statistical conclusion validity arises from the sample size employed per experimental condition (10 independent runs per data scale). When the A_{12} statistic reflects a moderate effect size (i.e., between 0.36 and 0.64), the associated 95% bootstrap confidence intervals occasionally exhibit substantial interval widths or span the no-effect baseline of 0.5. This variance is primarily attributable to the modest number of independent runs, which inherently yields broader confidence bounds under non-parametric resampling. Consequently, marginal performance differences between learners may appear statistically non-significant despite underlying trend differences. While this sample size was selected to balance computational feasibility across extensive runtime requirements, conducting a higher number of independent runs per case study could narrow these confidence intervals, increase statistical power, and further solidify the statistical conclusions drawn.

Chapter 6

Related work

As discussed in Chapter 2, automata learning has garnered significant interest within the research community and has evolved into a highly prevalent field of study. The primary contribution of this thesis is a methodology for enhancing the performance of EDSM and DFASAT by leveraging mined patterns.

A substantial number of techniques already exist to refine EDSM, including the DFASAT approach covered in Section 2.2.4, and various methods employ patterns for model inference. However, a gap remains in the literature. To the best of our knowledge, no existing passive reverse-engineering techniques infer Mealy machines using the specific pattern-based framework presented in this work (see Chapter 3).

There have, however, been numerous attempts to utilise alternative state-machine formalisms to guide or constrain the inference process. In this chapter, we explore these established methodologies and examine their application to passive automata learning.

This chapter is organised as follows: Section 6.1 examines the k -tail algorithm, a foundational passive state-merging methodology. Section 6.2 explores the $L^\#$ framework, an active learning algorithm that utilises state-merging principles to infer Mealy machines. Section 6.3 reviews various scholarly advancements and algorithmic optimisations designed to extend the EDSM framework, focusing specifically on enhancements to classification accuracy and computational scalability. Finally, Section 6.4 investigates how existing literature integrates temporal and structural invariants to provide heuristic guidance for the inference process.

6.1 The K-tail Algorithm

A foundational approach in the state-merging technique is the k -tails algorithm, originally proposed by Biermann and Feldman [10]. The k -tails method facilitates the state-merging technique based on the similarity of their future behaviours. Specifically, two states within a prefix-tree are considered equivalent and candidates for merging if they share identical paths or “tails” of a predefined length, k .

Formally, the parameter k determines the depth of the look-ahead used to establish state equivalence. One of the primary challenges in applying k -tails is selecting an optimal value for k . As there is no universally “perfect” value, k is often determined manually by the developer [15]. The choice of k introduces a significant trade-off in learning the resulting model:

- Over-merge (k is too small): If k is set to a low value, the algorithm becomes overly aggressive in its merging. This often leads to an over-generalised DFA, where distinct operational states are collapsed into a single state, potentially resulting in a trivial model that accepts a language far broader than the target system.
- Under-merge (k is too large): Conversely, if k is set to a high value, the criteria for equivalence become excessively strict. In such cases, the learner may fail to identify valid merge candidates due to insufficient evidence in the training data. This results in an under-merged model that remains close to the original prefix tree, failing to capture the underlying recursive structure of the system.

This sensitivity to the k parameter remains a central consideration in the design of the k -tail inference approach, as the accuracy of the resulting model is inherently tied to the structural depth of the observed execution traces.

The mechanics of the k -tail approach are illustrated in Figure 6.1. In this scenario, with the parameter set to $k = 1$, the algorithm identifies states that exhibit identical one-step future behaviours. This equivalence is determined by comparing the outgoing transition labels and subsequent states for each candidate pair.

It is important to note that the k -tail algorithm typically prioritises early merges. In scenarios where multiple potential state pairs satisfy the equivalence criteria, the algorithm employs a top-down or breadth-first selection strategy; pairs located closer to the root of the prefix tree are selected for merging before those at greater depths. While this approach promotes model minimality, the aggressive nature of early merging can occasionally lead to over-generalisation if the initial trace sample is insufficient.

1. Initial Merge ($k = 1$): State 0 and State 4 are identified as merge candidates because they share the same outgoing transition labelled *open/1*.
2. Deterministic Closure: The merging of 0 and 4 necessitates the merging of their respective successor states, 1 and 5, to maintain the determinism of the resulting automaton.
3. Recursive Progression: Subsequently, State 1 and State 2 are merged based on their shared *edit/1* path. This operation triggers a merge of State 6 into State 1 to ensure consistency across the transition function.

Through this iterative process, the original, highly specific prefix tree is abstracted into a more concise, generalised Mealy machine. This example demonstrates how a small k value can significantly reduce the state-space cardinality by identifying common functional patterns within the traces.

Both the k -tail and Evidence-Driven State Merging (EDSM) algorithms utilise a look-ahead mechanism to evaluate state equivalence. Each algorithm selects candidate state pairs for merging based on the similarity of their subsequent event sequences (future behaviours). However, they differ significantly in how they define the scope of this “look-ahead”:

- k -tail Algorithm: This approach employs a predefined, fixed-length sequence (k). Two states are considered equivalent if they share identical paths for exactly k subsequent steps. While computationally efficient, this static constraint can miss deeper structural similarities.
- EDSM Algorithm: In contrast, EDSM does not rely on a fixed parameter. Instead, it dynamically compares all available future sequences from two candidate states. The algorithm calculates a “score” based on the amount of overlapping evidence. It prioritises merging

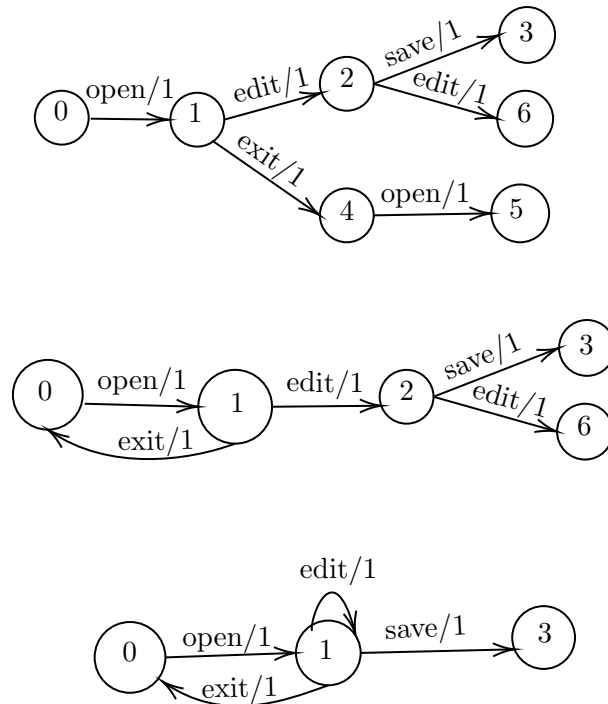


Figure 6.1: *Merging states using k -tail strategy*

the pair with the highest degree of shared evidence, effectively selecting the states with the longest matching sequences.

EDSM has been empirically proven as a reliable and practical approach in passive automata learning, consistently outperforming other heuristics in competitions such as Abbadingo [31]. Because of its established robustness and accuracy, we have selected EDSM as the foundational algorithm for this thesis.

6.2 The L-sharp algorithm:

The $L^\#$ (L-sharp) algorithm [49] is an active inference algorithm for Mealy machines. In contrast to traditional active learning frameworks, such as the foundational L^* algorithm, $L^\#$ incorporates the state-merging paradigm to construct hypothesis automata.

In classical active learning, the learner utilises Membership Queries to determine if a specific word belongs to a language and Equivalence Queries to verify if the hypothesis model is isomorphic to the System Under Learning (SUL). However, within the context of Mealy machines, the binary “accept/reject” nature of membership queries is replaced by input-output mappings. Consequently, $L^\#$ adapts the membership query into an Output Query. Here, the learner provides an input sequence to an Oracle (the teacher) and receives the corresponding sequence of outputs produced by the SUL. This allows the learner to map the functional response of the system rather than just its language boundaries.

The core innovation of $L^\#$ is the establishment of apartness, which is a constructive form of inequality between states. Formally, two states in a Mealy machine are considered “apart” if, when provided with an identical sequence of inputs, they produce diverging output sequences. Throughout the inference process, the learner constructs a prefix tree (i.e., PTA/APTA) based on the results of output queries. By applying the apartness relation, the learner can determine

which nodes in the tree cannot be mapped to the same state in the hidden Mealy machine. This process yields a hypothesis automaton, which is then submitted to the teacher for an equivalence query. If the model is incorrect, the teacher provides a counterexample. The learner treats this counterexample as vital new evidence, incorporating it into the prefix tree to refine the apartness relations and initiate a new iteration of the learning cycle.

While the $L^\#$ algorithm utilises the concept of apartness to identify state pairs that cannot be merged, our approach relies on mined rules extracted from the initial traces. In the $L^\#$ framework, apartness is established through evidence of observable differences in future behaviour. In contrast, our methodology enforces hard constraints based on temporal patterns; we strictly prohibit any state merge that violates these predefined invariants. Furthermore, a fundamental distinction lies in the learning paradigm: $L^\#$ is an active learning algorithm that interacts with a teacher to refine its hypothesis, whereas our hybrid process is a purely passive inference algorithm. Although both methodologies employ state merging in their initial stages, they diverge significantly in their secondary phases:

- Stage 1 (State Merging): Both approaches use state merging, though they apply different restrictions (apartness vs. pattern-based blocking).
- Stage 2 (Validation vs. Resolution): In $L^\#$, the second stage involves performing equivalence queries to verify the hypothesis automaton. Our approach, however, transitions to a SAT solver to resolve remaining logical constraints and identify the minimal automaton.

Potential for Methodological Adaptation

Our initial pattern-based state-merging stage could be adapted to the $L^\#$ framework to create a “warm-start” active learner. In such a configuration, a hypothesis automaton could be generated using our enhanced EDSM approach before being submitted to an equivalence query. If the query fails, the resulting counterexample can be used to refine the mined patterns, iteratively improving the constraints until a correct model is identified.

6.3 Improvements to EDSM

While DFASAT (Section 2.2.4) is one of the most sophisticated algorithms designed to address the inherent limitations of EDSM, several other notable refinements have been proposed in the literature. These methodologies aim to enhance the state-merging process by introducing alternative heuristics, search strategies, or hybrid formalisms. In this section, we discuss various algorithms built on the EDSM framework to improve its performance.

6.3.1 Biased EDSM score using Markov Model

Our approach builds upon the foundational concept of using behavioural patterns to guide automata inference. This strategy was also explored by Alsaedi [3]. Alsaedi utilised the Markov Model to predict the likelihood of an “impossible event” following a specific sequence of n events. In his framework, if a proposed state-merge introduces a transition that violates these statistical predictions, the EDSM score for that merge is proportionally reduced (biased).

While our methodology shares the goal of leveraging patterns, it differs from Alsaedi’s work in three fundamental aspects:

1. Hard vs. Soft Constraints: Alsaeedi employs a “soft” biasing strategy to reduce the EDSM score but still potentially allows the merge. Whereas we implement hard constraints. In our framework, any merge that violates a mined pattern is strictly blocked, preventing the learner from exploring structurally inconsistent paths.
2. Automata Formalism: Alsaeedi’s research focuses on Labelled Transition Systems (LTS). In contrast, our work targets Mealy machines, which necessitate handling deterministic input-output pairs rather than simple state transitions.
3. SAT-Based Integration: We further extend this logic by incorporating a SAT solver into the inference engine. This allows the mined patterns to serve as logical clauses that help the SAT solver navigate the search space more efficiently, thereby identifying consistent, minimal models that satisfy both the traces and the temporal patterns.

6.3.2 Shared Evidence-Driven State Merging

The scoring function in EDSM plays a crucial role in selecting which path to adopt among alternative mergers. Although this evidence seems promising for performing a good merge first, Abela *et al.* [1], a high score may block some future good merges. EDSM has to perform a “correct” merge at each algorithm iteration to locate the target model. Moreover, EDSM is extremely sensitive to errors introduced by early merges and cannot undo the bad merges. Generally speaking, it is impossible to pinpoint the exact moment of a “bad” integration. EDSM frequently converges to a model that is significantly bigger than the intended DFA. This is presumably due to the initial “bad” merging that EDSM makes [1].

Shared Evidence-Driven State Merging (S-EDSM), [1] tries to “look ahead” for future evidence to increase the merging scores and avoid bad merges as much as possible. It begins by considering all valid merges (Mv) of the hypothesis DFA H . Then, for each Mv , it checks the other merges that could be performed after it. Those are called compatible merges. Mv_1 is compatible with Mv_2 , if we perform Mv_1 then we still could perform Mv_2 and vice versa. The merging score is the sum of the scores of all compatible merges. The performed merge has the highest score.

Incompatible merges are also retained to enable backtracking. A list of incompatible merges is maintained for each valid merge when the final hypothesis is large (too many states). In such a case, returning to a merge that blocked others could be worthwhile. Since the merge with the highest scores does not necessarily indicate the best option, one could revisit other options.

Although S-EDSM significantly reduces incorrect merges, it is designed to exhaustively explore all possible future merges for each candidate state pair before committing to a decision. Furthermore, in scenarios involving large target automata, backtracking introduces a significant computational overhead. This “trial-and-error” approach scales poorly as the state space grows, as each backtrack requires re-evaluating numerous potential merge trajectories.

In contrast, our approach seeks to mitigate the frequency of incorrect merges by extracting structural and temporal rules from the initial training data. Rather than relying on expensive post-hoc backtracking, we use these mined rules to proactively guide the merging process. By filtering out inconsistent merges at the point of proposal, we reduce the branching factor of the search and decrease the likelihood of entering a state-space configuration that would necessitate a costly reversal. This transition from an exploratory search (S-EDSM) to a constrained search (our approach) provides a more direct path to the minimal automaton while maintaining high accuracy with lower computational intensity.

6.3.3 The ED-BEAM Algorithm

The Evidence-Driven BEAM Search (ED-BEAM) [30] algorithm couples the EDSM approach with beam search. The search control approach employed by the algorithm enables it to revisit any decision made at the top of the search tree. To put it simply, ED-BEAM explores options near the tree’s root, then uses the EDSM heuristic to identify optimal options farther out. This heuristic is justified by the idea that early decisions are more likely to be incorrect than later ones. As a result, returning to the first backtracking spots is preferable to returning to the later ones [12].

The Beam is a heuristic search algorithm that chooses the most promising node in a limited set. This will select the first pair to merge, and then the EDSM will be applied to the rest of the tree. If the final automaton were not the desired one, the algorithm undoes the merges, returns to the first pair, and selects another pair. The process is repeated until a solution is found.

While ED-BEAM relies on an extensive backtracking framework to explore alternative merge paths, it executes its initial merge based on global evidence (the beam width) before reverting to local EDSM rules for subsequent steps. In contrast, our proposed approach introduces a lightweight evaluation mechanism that simultaneously assesses each state pair using both global and local deterministic evidence. Crucially, our global evidence, which is derived from mined patterns, evaluates every potential merge candidate holistically rather than restricting global analysis to the initial choice. Furthermore, we leverage the distinct strengths of a SAT solver to entirely bypass deep, computationally expensive backtracking pipelines: the PAT-EDSM phase handles the merging of highly evidential state pairs, while the SAT solver efficiently resolves pairs characterised by low similarity or ambiguous evidence.

This architectural difference opens a clear path for hybrid adaptation. The lightweight evidence evaluation and the SAT-driven resolution phases from our algorithm could be integrated directly into ED-BEAM’s beam selection phase. By doing so, ED-BEAM could prune unviable paths at the root level much faster, significantly reducing its heavy backtracking overhead.

6.3.4 Genetic Algorithm and EDSM

Genetic Algorithms (GAs) are meta-heuristic search techniques frequently used to produce high-quality solutions for complex optimisation and search problems. In the context of automata learning, GAs are particularly well-suited to identifying a target Deterministic Finite Automaton (DFA). The biological metaphors are mapped to the learning process in which the gene, the atomic unit, represents a single state-merge operation. The chromosome (Individual) is a complete sequence of merges that results in a candidate hypothesis model. The population is a collection of these candidate models (individuals).

The GA operates by encoding a solution as a sequence of valid merges to be applied to the prefix tree. It iteratively explores various merge sequences, evolving a population of candidate models toward an optimal solution. However, the quality of a model produced by state-merging is highly sensitive to initial choices; a single suboptimal merge early in the process can impose rigid constraints on all subsequent operations, often leading to a local optimum rather than a global one.

To mitigate this sensitivity, Guillaumier et al. [26] proposed a hybrid evolutionary strategy designed to locate a high-quality starting set of merges. Their approach utilises a GA to perform an initial sequence of merges, resulting in a partial hypothesis. This partial model is then expanded and completed using the deterministic power of EDSM to produce a final hypothesis of size m . By combining the stochastic global search capabilities of a GA with the evidence-driven refinement of EDSM, this strategy significantly increases the likelihood of finding a minimal, accurate model.

While Genetic Algorithms (GA) and beam-search variations such as ED-beam can identify high-quality merge sequences, finding a consistently correct path remains computationally intensive. The primary distinction between these approaches and our own lies in how they integrate the EDSM heuristic into their search strategies:

- **S-EDSM:** This approach calculates a cumulative score by looking ahead through an entire sequence of potential future merges before committing to the first one.
- **Genetic Algorithms (GA):** These identify an optimal sequence of merges through evolutionary selection and then apply standard EDSM to the resulting intermediate model.
- **ED-beam:** This strategy performs the single best merge identified by a restricted search (the “beam”) before reverting to standard EDSM for the remainder of the process.

In contrast to these multi-stage or evolutionary techniques, our approach integrates pattern-based constraints directly into the initial stage of EDSM. By exercising “extra care” during the selection of merge pairs (by blocking any pair that violates a mined hard pattern), we ensure that the initial reduction of the search space is guided by the mined patterns.

Although our methodology does not currently incorporate a backtracking strategy, it offers a significant advantage in terms of computational efficiency. By proactively preventing incorrect merges through rule enforcement rather than post hoc correction, we avoid the exponential growth in processing time associated with the exhaustive look-ahead of S-EDSM or the iterative generation of a GA. Consequently, our approach achieves a high-fidelity model in a fraction of the time required by more exhaustive search-based competitors.

6.4 Pattern-based Automata Learning

Our research analyses the given training data and observes the absence of pairwise labels. Those observations guide the EDSM toward eliminating the state pairs that introduce new behaviour to the training data. In this section, we compare other approaches that have integrated patterns into the model inference process to our proposed methods.

A closely related approach to this thesis is presented by Lo *et al.* [36] (introduced in Section 2.2.5), who mined patterns that subsequently guide the K -tail [10] state-merging algorithm, thereby reducing the likelihood of over-generalisation. While our proposed methodology shares a similar structural pipeline, it diverges from their work in three fundamental ways:

- **Pattern Complexity:** Lo *et al.* extract complex, long-sequence temporal properties, whereas our approach focuses on a more lightweight form of pairwise patterns.
- **Core Heuristic:** Their approach relies on the localised k -tail heuristic, while we adapt the EDSM framework specifically tailored for Mealy machines.
- **Exact Resolution:** We integrate a constraint-based SAT solver phase following the initial pattern-guided merges to robustly resolve ambiguous states, a phase absent in their heuristic-only approach.

Several studies have adapted composing a model from small components (or patterns) [44, 43, 23] (introduced in Section 2.2.5). The general approach involves extracting “mini-patterns” from the provided data and combining them to construct the final automaton. Our approach shares the use of simple patterns but diverges significantly in its execution and objectives:

1. Streamlined Pattern Mining: The aforementioned frameworks [43, 44, 23] maintain and query a diverse library of patterns in stages. Conversely, we consolidate the initial stages into a single, direct step by mining patterns exclusively and directly from the training traces. We utilise a single pattern type rather than a library. This significantly reduces the computational overhead of the mining process and simplifies the overall architectural complexity.
2. Heuristic Guidance vs Constructive Assembly: Unlike approaches in [43, 44, 23], which rely entirely on the patterns to build the final automaton, our mined patterns serve a guiding role for the EDSM algorithm. This distinction is critical in scenarios involving sparse data, where mined patterns may be incomplete or incorrect. By using patterns as a secondary constraint rather than the primary building block, our learner remains robust even when the initial pattern-mining stage fails to capture the system’s full complexity.
3. Prohibitive Behaviour Constraints: Despite these differences, we share the use of negative patterns as prohibited behaviours with Reger’s method [43]. In our method, if merging two states results in the acceptance of a known negative pattern, the learner proactively blocks the merge. Meanwhile, in Reger’s method, when the composition of two patterns would permit a sequence identified in a negative pattern, that specific composition is disallowed. By treating patterns as a “merge blocker” within a state-merging framework, we leverage their power to prevent over-generalisation without the need for the complex structural assembly required by Reger’s approach.

Walkinshaw *et al.* [52] utilised user-defined Linear Temporal Logic (LTL) to reduce the reliance on massive execution trace sets (introduced in Section 2.2.5). They employ a model checker to verify that intermediate hypothesis machines do not violate these established properties. If a violation is detected, the model checker generates a counterexample, which is appended to the initial trace set before restarting the entire inference process.

Our work diverges from Walkinshaw *et al.*’s approach in several fundamental aspects:

- Pattern Type: We utilise lightweight, pairwise temporal properties rather than full Linear Temporal Logic (LTL) specifications.
- Pattern Origin: We dynamically mine patterns directly from the provided dataset, whereas their approach relies strictly on pre-defined, user-specified constraints.
- Data Enrichment vs Guidance: They continuously enrich the training dataset with runtime counterexamples, whereas we extract inherent system properties to directly guide the learner during state merging.
- Algorithmic Complexity: Their loop-based refinement requires multiple costly iterations of the learning algorithm, whereas our framework operates in a straightforward, single-pass pipeline.

Mining Complex Patterns

Pattern-based specification mining typically focuses on small, straightforward patterns to ensure efficient trace checking, which is the core focus of this thesis. However, effective specification mining often requires extracting more complex patterns. To bridge this gap, researchers have developed various methods to mine sophisticated behavioural constraints. In this section, we explore the previous work on mining complex patterns, which can be used in several approaches to model inference.

Yang *et al.* [58] focused on mining simple cause-and-effect patterns from execution traces, wherein each cause must be followed by a corresponding effect, and an effect cannot manifest without a prior cause. In practice, they observed that evaluating isolated pairs of events that satisfy this criterion rarely yields a meaningful understanding of a program’s complex temporal behaviour. To address this limitation, they incorporated pattern scopes to introduce a layer of context and structural complexity [58]. A pattern scope formally defines the specific region or interval within an execution trace where a given property must hold.

Taking a different architectural approach, Reger *et al.* introduced open automata as a formal abstraction for sequential patterns. An open automaton is a Non-deterministic Finite Automaton (NFA) that incorporates a unique “hole” symbol ($\bullet$) into its transition definition. This symbol acts as a wildcard, representing any character or event that does not belong to the defined pattern alphabet. Specifically, for a pattern template P and a set of runtime events φ , an instantiated pattern $P(\varphi)$ is generated by mapping components of φ to the corresponding literal symbols in P . The hole symbol ($\bullet$) subsequently subsumes any event outside of the subset φ .

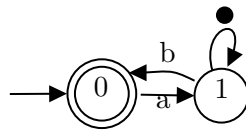


Figure 6.2: *Resource ownership pattern*

Consider the resource ownership pattern illustrated in Figure 6.2 defined over an alphabet $\Sigma = \{\text{open}, \text{edit}, \text{exit}\}$. An instantiated pattern for $\varphi = \{\text{open}, \text{exit}\}$ (where $a = \text{open}$ and $b = \text{exit}$) successfully satisfies the given trace $t = \langle \text{open}, \text{edit}, \text{exit}, \text{open}, \text{exit}, \text{open}, \text{edit}, \text{edit}, \text{exit} \rangle$. In this instance, the $\bullet$ symbol encapsulates the unmapped event subset $\Sigma \setminus \varphi = \{\text{edit}\}$. By leveraging this “hole” abstraction, open automata grant localised patterns a more sophisticated topology, allowing them to scale seamlessly across broader alphabets. Furthermore, this notation simplifies composing multiple primitive patterns into a single complex pattern. This compositionality helps the learner efficiently verify pattern validity across training traces while providing an abstract formalism that isolates the core features or events required for a specific validation task.

Similarly, Li *et al.* [34] attempted to construct complex behavioural specifications by merging parallel simple patterns or chaining sequential ones. Their framework considers four distinct pattern templates, each restricted to an alphabet of size 2. They first mine all valid instances that match these templates, recording a precise timestamp for every event within each pattern. Next, they apply event merging, pattern chaining, and graph composition to synthesise highly informative, complex properties. State merging is applied to concurrent primitive patterns that share identical timestamps, whereas pattern chaining unifies patterns with sequential temporal intervals. Finally, the graphs resulting from these merging and chaining operations are structurally combined to construct the final specification model.

6.4.1 Markov Model in automata learning

The literature employed the Hidden Markov Model (HMM) to extract patterns from traces. By observing the provided traces, the model predicts the subsequent event in a sequence. The Markov model assumes an underlying, hidden Finite State Machine (FSM) with a known initial state and provided behavioural traces. Each transition carries a probability representing the likelihood of moving from one state to another. Because these probabilities remain constant over time, the future state is determined solely by the current state (the Markov property). Generally, an n^{th} -order Markov model defines state transition probabilities based on the previous n states of the process.

Applying Neural Networks to Model Inference

Cook and Wolf [16] explored the application of recurrent neural networks (RNNs) to model inference. They trained an RNN on positive execution sequences to dynamically predict the conditional probability distribution of the next event given a history of preceding events. Rather than relying on static Markov chains, where an n^{th} -order property maps directly to an isolated matrix, the network leverages its internal hidden-layer state space to compress and maintain this temporal context across variable-length histories.

To map the learned continuous weights back into a discrete formal model, the authors developed a systematic extraction and structural resolution pipeline. A baseline transition graph is constructed using first-order event sequences that exceed a user-defined probability or frequency threshold. In this initial topology, nodes represent discrete events, and directed edges represent the permitted immediate sequence flow. The first-order graph is systematically verified against the higher-order probability constraints extracted from the trained network’s deeper contextual representations. Whenever the local branching structure of the graph violates these higher-order conditional probabilities, a state-splitting operation is executed on the conflicting node. This refinement process iterates recursively until the structural pathways of the graph fully satisfy the network’s predictive distributions across all sequence depths. The finalised graph is transformed into a formal automaton by mapping its edges to discrete states and pairs of incident edges to valid transition functions. If the resulting structure exhibits non-determinism (see Definition 2), standard state-merging operations are applied to enforce structural determinism.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

In contemporary software engineering, software has become a foundational component of critical infrastructure and daily life. Deploying low-quality software can result in severe financial losses. While software testing and quality assurance serve as primary mechanisms to mitigate these risks, their efficacy depends significantly on the availability of accurate system specifications. Formal software models, such as finite state machines, facilitate these specifications by providing a visual structure that guides developers throughout the software development life cycle. However, for these verification techniques to remain valid, the underlying models must remain complete, consistent, and continuously synchronised with the source code. In practice, such up-to-date models are rarely available, as developers frequently prioritise immediate code delivery over time-consuming manual documentation. To bridge the gap between the need for precise specifications and the practical constraints of manual upkeep, passive model inference has emerged as a vital reverse-engineering paradigm for automatically inferring behavioural models directly from execution traces.

Passive model inference typically constructs state-machine abstractions from software execution traces. The baseline framework in passive automata learning is EDSM [31], which iteratively merges states in the PTA by calculating a local similarity score to select the best state pairs. Although EDSM demonstrated significant capabilities by winning the Abbadingo One DFA Learning competition [31], its purely greedy heuristic remains highly error-prone under data-sparse conditions.

To overcome these limitations, the DFASAT framework [27] integrates preliminary EDSM merges with SAT solvers. By translating the partially merged model into a CNF formula, DFASAT successfully resolved the most complex benchmarks in the Abbadingo competition, which EDSM failed to resolve. DFASAT controls early-stage errors by employing a rigid blocking heuristic during its initial merging phase, strictly prohibiting state merges that would introduce new transition labels to existing “red” states.

However, empirical evaluation demonstrates that the standard DFASAT heuristic is overly restrictive, frequently bypassing valid merge candidates. Missing a correct merge at an early stage restricts the search space suboptimally and causes a cascade of incorrect structural choices in subsequent iterations. The primary limitation of DFASAT lies in its local scope: it evaluates compatibility based solely on direct, immediate outgoing transitions. In data-sparse environments featuring large input alphabets, states within a PTA rarely share outgoing labels, which often leads to the erroneous blocking of correct merges. While the exact SAT solver can theoretically recover from these localised heuristic omissions, the resulting inflation of the search space dramatically increases

computational complexity. Furthermore, while DFASAT prohibits adding new labels directly between the immediate state pairs under evaluation, it permissively allows unobserved labels to be appended to any downstream states within the remaining subtrees during non-determinism resolution. Consequently, while DFASAT successfully minimises the risk of early, irreversible merges, it remains prone to structural inaccuracies when training data is sparse.

To address this rigidity, this thesis proposed a more flexible paradigm that balances the permissive, error-prone nature of the EDSM scoring algorithm against the overly restrictive constraints of local blocking mechanisms. We designed, implemented, and evaluated a novel framework that extracts lightweight pairwise sequential patterns directly from the training traces of a Mealy machine (Contribution **C1**). Subsequently, we used these mined patterns as global constraints to guide state selection. Under this approach, a proposed merge is strictly rejected if it induces a behavioural violation of any mined pattern. The extraction engine operates by systematically evaluating every distinct input-output transition label to identify the absolute absence of subsequent events; for instance, if an input-output pair $a/1$ is never followed by $b/2$ anywhere within the PTA, the ordered pair $P(a/1, b/2)$ is formally established as a pattern.

Our proposed pattern-guided framework was integrated into several algorithmic pipelines (Contribution **C2**):

- **PAT-EDSM:** Augments the standard greedy EDSM algorithm by incorporating a pattern-restricted pair-selection heuristic to proactively block invalid merges (Contribution **C2-1**).
- **PAT-DFASAT:** executes a preliminary sequence of merges via PAT-EDSM before encoding the remaining, highly optimised state graph into a SAT solver (Contribution **C2-3**).
- **Pattern-guided DFASAT:** integrates the mined sequential patterns directly into the baseline DFASAT local blocking heuristic (Contribution **C2-5**).
- **Exact Constraint SAT Encodings:** translates the mined patterns into propositional clauses for a CNF formula to constrain the SAT solver (Contribution **C2-4**). This was evaluated across two distinct configurations:
 - PAT-DFASAT+PAT-SAT: Integrating mined patterns into EDSM and then into the SAT solver (Contribution **C2-6**).
 - Pattern-guided DFASAT+PAT-SAT: Integrating mined patterns into the standard DFASAT heuristic and then into the SAT solver (Contribution **C2-7**).

Each proposed configuration was evaluated against the baseline DFASAT framework across a variety of industrial case studies, including Bluetooth, OpenSSL, and TCP protocol, as well as collections of randomly generated automata (Contribution **C3**). Performance was rigorously quantified using the following core evaluation metrics:

- **Language Acceptance ($F1$ -score):** evaluates the accuracy of the inferred model using independent testing traces excluded from the training dataset (Research Question **RQ1-(A)**).
- **Structural Difference (SD) Score:** evaluates the topological similarity (in terms of states and transitions) of the inferred model relative to the true target model (Research Question **RQ1-(B)**).
- **Proportion of Correct Merges:** measures the ratio of correct merge decisions executed relative to the total number of merges performed during the heuristic phase (Research Question **RQ2**).

- **Execution time:** measures the total time required to learn a system (Research Question **RQ3**).

Across all evaluated configurations and data scales, PAT-DFASAT represents the most robust and statistically superior framework for passive model inference, successfully mitigating the challenges of extreme data sparsity to deliver high-fidelity behavioural models (see Chapter 5). However, when states of the target automata exhibit highly identical behaviour, PAT-EDSM outperforms PAT-DFASAT. In those specific scenarios, the PAT-EDSM heuristic effectively avoids incorrect merges that the downstream SAT solver struggles to differentiate. Nevertheless, such similarity remains difficult for an automated learner to detect directly and typically necessitates domain-expert intervention.

Furthermore, our results demonstrate that the precision of early heuristic merges directly impacts the total execution time, particularly regarding the downstream solver’s convergence speed. When the learner introduces structural errors at an early stage, an incorrect partially merged model is passed to the SAT solver. Given moderate data sizes, it becomes computationally expensive for the solver to successfully merge the remaining states with these flawed structures.

Moreover, a relationship exists between execution runtime variance and the scale of the training dataset. In most instances, the variance contracts significantly at both operational extremes: very sparse data and highly sufficient data sizes. With highly sparse data, learners rapidly converge on a solution within a concise time-frame, even if the resulting model is far from the target model. With large datasets, the rich constraint space enables the learner to infer the model efficiently. The most notable runtime variance occurs in the intermediate data ranges, where the training traces alternate between being straightforward to process and being structurally ambiguous to the learner.

7.2 Future Work

Several promising research directions emerge from this study to further enhance the scalability and applicability of pattern-based passive automata learning:

Expanding Independent Runs to Enhance Statistical Power

When the A_{12} statistic indicates a moderate effect size (i.e., between 0.36 and 0.64), the associated 95% bootstrap confidence intervals frequently exhibit a lack of statistical significance or substantial interval widths. This variance is primarily attributable to the sample size employed per experimental condition (10 independent runs per data scale), which inherently yields broader confidence bounds under non-parametric resampling. To address this limitation, future work will expand the dataset by executing a higher number of independent runs for each case study. Re-evaluating the empirical results over larger sample sizes will narrow these confidence intervals, thereby increasing statistical power, clarifying marginal effect sizes, and providing firmer conclusions regarding algorithm performance.

Predictive Trace Sufficiency and Early Termination

A critical question in passive inference is determining whether the available training traces are sufficient to yield a structurally correct model. Future work could focus on predicting SAT solver failures prior to the encoding phase, thereby reducing unnecessary computational overhead. For instance, if the EDSM heuristic constructs a large cluster of “red” states whose potential unifications systematically violate either our mined-pattern constraints or the deterministic Mealy properties, then further SAT exploration becomes redundant.

We can establish an upper bound based on the cardinality of this red state cluster; if $|\text{red-cluster}| \geq$

$|Q_{\text{target}}| \times 2$, it indicates that the SAT solver cannot satisfy the formula using a smaller state budget. Because our framework bounds the solver from exceeding this limit, the process can be safely terminated early to avoid exhaustive SAT computations. Additionally, this red-state cluster can be used to derive tight lower bounds on the state-colouring configuration in the SAT solver, provided these states are shown to be pairwise incompatible under Mealy and heuristic constraints.

Logarithmic State-Budget Search Acceleration

To reduce the time required for the SAT solver to converge on a valid solution, the current linear search strategy can be optimised. Presently, the state-budget search sets a lower bound of 2 states and an upper bound of $|Q_{\text{target}}| \times 2$, incrementing the target state count (or colour budget) sequentially by one each time an UNSAT result is returned. Future implementations could replace this linear traversal with a binary search strategy. The solver would evaluate the midpoint between the lower and upper bounds; if the formula is SAT, this midpoint becomes the new upper bound; if UNSAT, it marks the new lower bound. Adopting this logarithmic (log) exploration strategy would significantly reduce the number of required solver invocations.

Advanced Pattern Mining and Complex Topologies

The framework can be extended by integrating more sophisticated, higher-order temporal patterns beyond binary invariants, such as LTL-style properties or k-consecutive event dependencies. Furthermore, future research will explore how to adapt the pattern-guided inference engine to support a broader taxonomy of model types, including non-input-enabled Mealy machines, concurrent state infrastructures, and parallel distributed software systems.

Learning Without a Reference Model

Finally, we can adapt the learning process into a more practical approach where a reference model is entirely unavailable. In industrial settings, complete reference models are rarely available, leaving execution traces as the sole source of information about system behaviour. In such a case, we can partition the provided execution traces into a training set (90%) and a testing set (10%). The learner induces the model using the 90% training allocation, and its performance is subsequently evaluated against the remaining 10% testing subset. If the learner fails to learn effectively (i.e., the test traces are incorrectly classified), the framework leverages the previously acquired knowledge and retries the process with a new, randomly generated 90/10 partition. Through successive iterations, the learner incrementally improves its accuracy until it successfully classifies the test set. This iterative refinement approach allows the framework to autonomously learn and validate its inferred models without requiring access to an actual target model.

Bibliography

- [1] ABELA, J., COSTE, F., AND SPINA, S. Mutually compatible and incompatible merges for the search of the smallest consistent dfa. In *International Colloquium on Grammatical Inference* (2004), Springer, pp. 28–39.
- [2] AICHERNIG, B. K., TAPPLER, M., AND WALLNER, F. Benchmarking combinations of learning and testing algorithms for automata learning. *Formal Aspects of Computing* 36, 1 (2024), 1–37.
- [3] ALSAEEDI, A. *Improving Software Model Inference by Combining State Merging and Markov Models*. PhD thesis, University of Sheffield, 2016.
- [4] ANGLUIN, D. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106.
- [5] ANGLUIN, D., AND LAIRD, P. Learning from noisy examples. *Machine Learning* 2 (1988), 343–370.
- [6] BERNARDO, M., AND ISSARNY, V. *Formal Methods for Eternal Networked Software Systems*, vol. 6659. Springer, 2011.
- [7] BIERE, A., CIMATTI, A., CLARKE, E., AND ZHU, Y. Symbolic model checking without bdds. In *International conference on tools and algorithms for the construction and analysis of systems* (1999), Springer, pp. 193–207.
- [8] BIERE, A., FALLER, T., FAZEKAS, K., FLEURY, M., FROLEYKS, N., AND POLLITT, F. CaDiCaL, Gimsatul, IsaSAT and Kissat entering the SAT Competition 2024. In *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions* (2024), vol. B-2024-1 of *Department of Computer Science Report Series B*, University of Helsinki, pp. 8–10.
- [9] BIERE, A., JÄRVISALO, M., AND KIESL, B. Preprocessing in sat solving. *Handbook of Satisfiability* 336, 2021 (2021), 391–435.
- [10] BIERMANN, A. W., AND FELDMAN, J. A. On the synthesis of finite-state acceptors. Tech. rep., 1970.
- [11] BOGDANOV, K. Statechum. <https://github.com/kirilluk/statechum>, 2026.
- [12] BUGALHO, M., AND OLIVEIRA, A. L. Inference of regular languages using state merging algorithms with search. *Pattern Recognition* 38, 9 (2005), 1457–1467. Grammatical Inference.
- [13] CHOI, W., NECULA, G., AND SEN, K. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.
- [14] CLARKE, E. M. Model checking. In *International conference on foundations of software technology and theoretical computer science* (1997), Springer, pp. 54–56.
- [15] COHEN, H., AND MAOZ, S. The confidence in our k-tails. In *the 29th ACM/IEEE international conference on Automated software engineering* (2014), pp. 605–610.

- [16] COOK, J. E., AND WOLF, A. L. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 3 (1998), 215–249.
- [17] CZERNER, P., ESPARZA, J., KRASOTIN, V., AND KRAUSS, A. A resolution-based interactive proof system for unsat. *Logical Methods in Computer Science* 22 (2026).
- [18] DAMASCENO, C. D. N., MOUSAVI, M. R., AND SIMAO, A. D. S. Learning by sampling: learning behavioral family models from software product lines. *Empirical Software Engineering* 26 (2021), 1–46.
- [19] DE RUITER, J., AND POLL, E. Protocol state fuzzing of TLS implementations. In *24th USENIX Security Symposium (USENIX Security 15)* (Washington, D.C., Aug. 2015), Springer, pp. 193–206.
- [20] DUPONT, P., MICLET, L., AND VIDAL, E. What is the search space of the regular inference? In *International Colloquium on Grammatical Inference* (1994), Springer, pp. 25–37.
- [21] EFRON, B. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 569–593.
- [22] FITERĂU-BROȘTEAN, P., JANSSEN, R., AND VAANDRAGER, F. Combining model learning and model checking to analyze tcp implementations. In *Computer Aided Verification* (Cham, 2016), S. Chaudhuri and A. Farzan, Eds., Springer International Publishing, pp. 454–471.
- [23] GABEL, M., AND SU, Z. Javert: fully automatic mining of general temporal properties from dynamic traces. In *16th ACM SIGSOFT International Symposium on Foundations of software engineering* (2008), pp. 339–349.
- [24] GOLD, E. M. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320.
- [25] GROCE, A., PELED, D., AND YANNAKAKIS, M. Adaptive model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference* (2002), Springer, pp. 357–370.
- [26] GUILLAUMIER, K., AND ABELA, J. Learning dfas by evolving short sequences of merges. In *International Conference on Grammatical Inference* (2021), PMLR, pp. 217–236.
- [27] HEULE, M. J., AND VERWER, S. Software model synthesis using satisfiability solvers. *Empirical Software Engineering* 18, 4 (2013), 825–856.
- [28] ISBERNER, M., HOWAR, F., AND STEFFEN, B. The open-source learnlib. In *Computer Aided Verification* (Cham, 2015), D. Kroening and C. S. Păsăreanu, Eds., Springer International Publishing, pp. 487–495.
- [29] KANG, Z., PAN, H., HOI, S., AND XU, Z. Robust graph learning from noisy data. *IEEE Transactions on Cybernetics* 50 (2018), 1833–1843.
- [30] LANG, K. J. Faster algorithms for finding minimal consistent dfas. *NEC Research Institute, Tech. Rep* (1999).
- [31] LANG, K. J., PEARLMUTTER, B. A., AND PRICE, R. A. Results of the abbadingo one dfa learning competition and a new evidence-driven state merging algorithm. In *International Colloquium on Grammatical Inference* (1998), Springer, pp. 1–12.
- [32] LEE, D., AND YANNAKAKIS, M. Principles and methods of testing finite state machines—a survey. *IEEE* 84, 8 (2002), 1090–1123.

- [33] LESKOVEC, J., KLEINBERG, J., AND FALOUTSOS, C. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007).
- [34] LI, W., FORIN, A., AND SESHIA, S. A. Scalable specification mining for verification and diagnosis. In *47th design automation conference* (2010), pp. 755–760.
- [35] LO, D., KHOO, S.-C., LIU, C., AND HAN, J. Specification mining: A concise introduction. In *Mining Software Specifications*. CRC Press, 2011, pp. 17–44.
- [36] LO, D., MARIANI, L., AND PEZZÈ, M. Automatic steering of behavioral model inference. In *7th Joint Meeting Of The European Software Engineering Conference and the ACM SIGSOFT symposium on The foundations of software engineering* (2009), pp. 345–354.
- [37] MUŠKARDIN, E., AICHERNIG, B., PFERSCHER, A., AND TAPPLER, M. Aalpy: an active automata learning library. *Innovations in Systems and Software Engineering* 18 (03 2022), 1–10.
- [38] OHANNESSIAN, M. I., AND DAHLEH, M. A. Large alphabets: Finite, infinite, and scaling models. In *2012 46th Annual Conference on Information Sciences and Systems (CISS)* (2012), pp. 1–5.
- [39] PELED, D., VARDI, M. Y., AND YANNAKAKIS, M. *Black Box Checking*. Springer US, Boston, MA, 1999, pp. 225–240.
- [40] PELLEN, B., RODRÍGUEZ, M. B., VAANDRAGER, F., AND BOS, P. v. d. Systematic evaluation of black-box checking for fast bug detection. *arXiv preprint arXiv:2512.07434* (2025).
- [41] PFERSCHER, A., AND AICHERNIG, B. K. Fingerprinting bluetooth low energy devices via active automata learning. In *International Symposium on Formal Methods* (2021), Springer, pp. 524–542.
- [42] RAMANATHAN, M. K., GRAMA, A., AND JAGANNATHAN, S. Static specification inference using predicate mining. *ACM SIGPLAN Notices* 42, 6 (2007), 123–134.
- [43] REGER, G., BARRINGER, H., AND RYDEHEARD, D. A pattern-based approach to parametric specification mining. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2013), IEEE, pp. 658–663.
- [44] SHIN, D., BIANCULLI, D., AND BRIAND, L. Prins: scalable model inference for component-based system logs. *Empirical Software Engineering* 27, 4 (2022), 87.
- [45] ULYANTSEV, V., ZAKIRZANOV, I., AND SHALYTO, A. Symmetry breaking predicates for sat-based dfa identification. *arXiv preprint arXiv:1602.05028* (2016).
- [46] ULYANTSEV, V. I., AND TSAREV, F. N. Extended finite-state machine induction using sat-solver. *IFAC Proceedings Volumes* 45, 6 (2012), 236–241.
- [47] UTTING, M., AND LEGEARD, B. *Practical model-based testing: a tools approach*. Elsevier, 2010.
- [48] VAANDRAGER, F., EBRAHIMI, M., AND BLOEM, R. Learning mealy machines with one timer. *Information and Computation* 295 (2023), 105013. Special Issue: Selected papers of the 15th International Conference on Language and Automata Theory and Applications, LATA 2021.
- [49] VAANDRAGER, F., GARHEWAL, B., ROT, J., AND WISSMANN, T. A new approach for active automata learning based on apartness. In *International conference on tools and algorithms for the construction and analysis of systems* (2022), Springer, pp. 223–243.

- [50] VARGHA, A., AND DELANEY, H. D. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.
- [51] WAGNER, F., SCHMUKI, R., WAGNER, T., AND WOLSTENHOLME, P. *Modeling software with finite state machines: a practical approach*. Auerbach Publications, 2006.
- [52] WALKINSHAW, N., AND BOGDANOV, K. Inferring finite-state models with temporal constraints. In *23rd IEEE/ACM International Conference on Automated Software Engineering* (2008), IEEE, pp. 248–257.
- [53] WALKINSHAW, N., AND BOGDANOV, K. Automated comparison of state-based software models in terms of their language and structure. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 2 (2013), 1–37.
- [54] WALKINSHAW, N., BOGDANOV, K., DAMAS, C., LAMBEAU, B., AND DUPONT, P. A framework for the competitive evaluation of model inference techniques. In *First International Workshop on Model Inference In Testing* (2010), pp. 1–9.
- [55] WALSH, T. Sat v csp. In *International Conference on Principles and Practice of Constraint Programming* (2000), Springer, pp. 441–456.
- [56] WEYUKER, E. J. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5, 4 (1983), 641–655.
- [57] YAHAV, E., SHOHAM, S., FINK, S., AND PISTOIA, M. Static specification mining using automata-based abstractions. *Mining Software Specifications: Methodologies and Applications* (2011), 159.
- [58] YANG, J., AND EVANS, D. Dynamically inferring temporal properties. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (2004), pp. 23–28.
- [59] YANG, J., EVANS, D., BHARDWAJ, D., BHAT, T., AND DAS, M. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering* (2006), pp. 282–291.
- [60] ZAKIRZAYANOV, I., SHALYTO, A., AND ULYANTSEV, V. Finding all minimum-size dfa consistent with given examples: Sat-based approach. In *Software Engineering and Formal Methods* (2018), Springer, pp. 117–131.

Appendices

Appendix A

Generating Random Automata

Several frameworks have been proposed to generate random complex models. Among these, the Forest-Fire algorithm introduced by Leskovec et al. [33] is particularly salient for its ability to produce directed graphs that exhibit the nature of real-world software models [54]. The algorithm operates through an iterative growth process where each iteration adds a new node and new edges to the existing nodes. When a tree is burned in a forest (a node is added), the fire spreads into the neighbouring trees (connect the new node to some neighbour). Consider the graph in Figure A.1. When a new state v is added, it is linked to an existing state w and then linked to some other states.

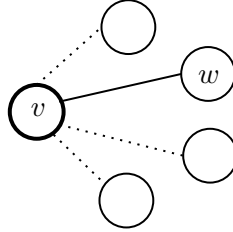


Figure A.1: Adding a new node with the Forest-Fire algorithm - dashed lines represent new edges and the solid line represents the addition of a new state

The algorithm is governed by three primary parameters:

1. *Forward-burning probability f* : represents the probability of having incoming edges. The number of incoming edges is calculated by:

$$x = \frac{f}{1 - f} \quad (\text{A.1})$$

2. *Backward-burning ratio b* : represents the probability of having outgoing edges. The number of outgoing edges is calculated by:

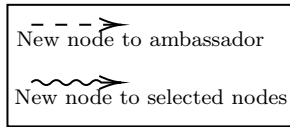
$$y = \frac{fb}{1 - fb} \quad (\text{A.2})$$

3. *Total number of nodes n* .

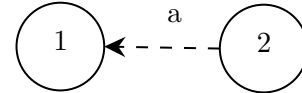
At each iteration, when a new node v is added:

1. The node v initiates its attachment by linking to a randomly selected ambassador node w , where $v \neq w$.
2. State v selects $x+y$ unvisited neighbours of w (x and y are calculated by Equation A.1 and A.2 respectively).
3. State v initiates out-edges to the selected nodes. Steps (2) and (3) are recursively applied until no more nodes can be reached.

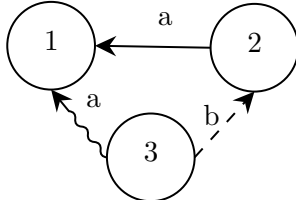
Figure A.2 illustrates the forest-fire algorithm for generating random automata. The target automaton is a 4-state automaton with alphabet $= \{a, b\}$. Adding a new node to an empty graph is trivial (Figure A.2a). Note that the dashed edge points to the randomly selected ambassador, while the wavy arrow points to the ambassador's neighbours. When adding the second node (Figure A.2b), the ambassador becomes available, but its neighbours are unavailable. An edge with label a connects 2 to 1 (the ambassador). Adding the third node (Figure A.2c) makes it clearer how the algorithm works. Node 2 is randomly selected as an ambassador; node 3 forms an outgoing edge to node 2 with a label b . Noting that node 2 is linked to node 1, node 3 will be linked to 1 with the same label. The ambassador for node 4 also happens to be node 2 (Figure A.2d). Now, node 2 has an outgoing edge to node 1 and an incoming edge from node 3; nodes 1 and 3 became the selected nodes for the new node. The last step is to connect node 4 to the selected nodes (i.e. nodes 1 and 3) via outgoing edges using the same labels.



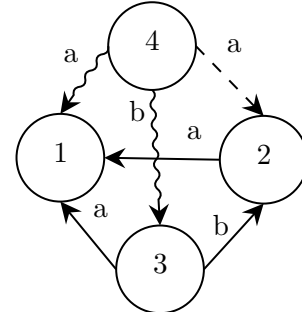
(a) Adding a new node to an empty graph



(b) Adding a second node



(c) Adding a third node



(d) Adding a fourth node

Figure A.2: Example of Generating a 4-state random automaton using the forest fire algorithm

While the Forest-Fire model is robust for general network synthesis, its direct application to generate state machines is precluded by the reachability from the initial state limitation. The algorithm's growth mechanism does not inherently preserve reachability from the initial state. In our example (Figure A.2d, state 1 (i.e. the initial state) cannot reach any other state.

To bridge the gap between general network structure and state machines, Walkinshaw *et al.* [54] applied the following modifications to the generative framework:

- **Alphabet Encoding and Determinism:** When a new edge is added, its label is randomly selected from a subset of the input alphabet. This subset is controlled by a parameter a . The selection process is constrained by a pruning mechanism to ensure the resulting transitions remain deterministic. When adding a new outgoing edge from the new state v , if the edge's

label has already been assigned to another outgoing transition from s , the edge addition is aborted.

- **Connectivity and Reachability:** To satisfy the reachability requirement from the initial state q_0 , the traditional Forest-Fire edge orientation is inverted. Rather than the new state v initiating a link to the ambassador node w , the edge is reverse directed $w \rightarrow v$, thereby extending the reachability during each iteration.
- **Self-loops:** The structural restriction against self-loops is bypassed through a parameter s , which defines the probability of establishing a self-loop edge.

Figure A.4 illustrates how the modifications to the forest-fire algorithm bypass all the limitations of the original algorithm. When the first node is added, the parameter s allows the addition of a self-loop labelled b (Figure A.3a). The second node (Figure A.3b), initiates an edge from the ambassador to the new node (node 2) with the remaining label a . Likewise, adding the third node (Figure A.3c), the random ambassador (node 2) points to node 3. Note that the new node still connects to all the ambassador's parents and children via outgoing transitions, as in the original algorithm. Additionally, the algorithm selects node 3 to be an accepted node. The ambassador for node 4 is node 2 (Figure A.3d). Thus, node 2 points to node 4 by a transition label a (since b is already taken in the previous step). Two outgoing edges with labels a and b connect node 4 to 1 and 3 respectively, as node 1 is the parent of 2 and node 3 is a child of 2.

Comparing the two generated automata: Figure A.2d that was generated by the original algorithm and Figure A.3d that was generated by the modified algorithm, we can clearly see that the modified algorithm bypasses all the above limitations. Every state is reachable from the initial state (state 1). The generated state machine is deterministic (See state 4 in both automata). The notions of final states and self-loops are applied (state 3 in Figure A.3d).

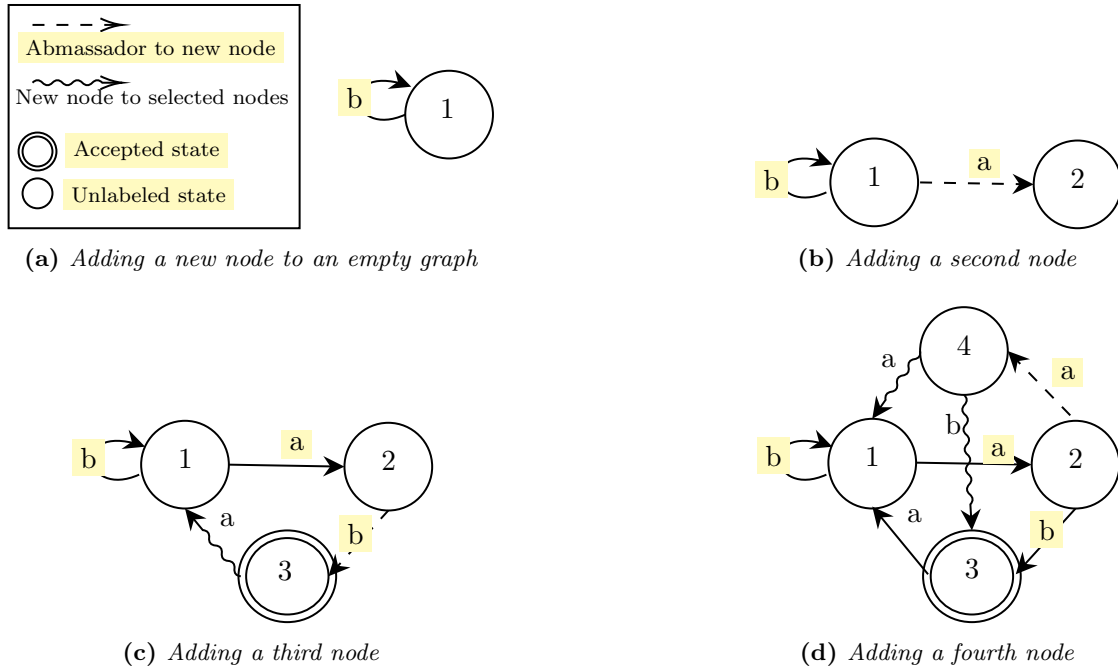


Figure A.4: Generating a 4-state random automaton using the modified forest-fire algorithm

Finally, after adding the transitions, two more steps are applied:

1. Enforcing reachability from the initial state: there is at least one path from the initial state to every other state.
2. Maintain minimality (optional): find the smallest set of states that represent the same generated automaton by removing equivalent states. If the minimal automaton is relatively small (far from the requested size), it is discarded, and a new automaton is generated.

Appendix B

APTA to CNF Formula

Example:

In this example, we are constructing a CNF formula for the PTA in Figure B.1. Let $C=\{0, 1\}$, $Q=\{0, 1, 2, 3, 4\}$, $I=\{\text{edit, exit, open}\}$, $O=\{0,1\}$, and $E=\{(0,3), (0,1)\}$.

The first step is to define the variables. For readability, all variable numbers will be written between round brackets (i.e. $x_{1,0} \equiv x(0,1)$).

1- Colouring Variables:

$x(0,0), x(0,1)$
 $x(1,0), x(1,1)$
 $x(2,0), x(2,1)$
 $x(3,0), x(3,1)$
 $x(4,0), x(4,1)$

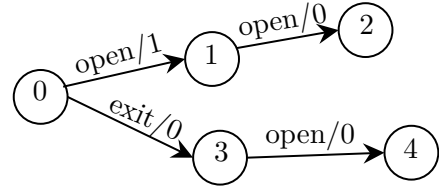


Figure B.1: PTA

2- Parenting Variables:

$y(\text{edit}, 0, 0, 0), y(\text{edit}, 0, 0, 1), y(\text{edit}, 0, 1, 0), y(\text{edit}, 0, 1, 1)$
 $y(\text{edit}, 1, 0, 0), y(\text{edit}, 1, 0, 1), y(\text{edit}, 1, 1, 0), y(\text{edit}, 1, 1, 1)$
 $y(\text{exit}, 0, 0, 0), y(\text{exit}, 0, 0, 1), y(\text{exit}, 0, 1, 0), y(\text{exit}, 0, 1, 1)$
 $y(\text{exit}, 1, 0, 0), y(\text{exit}, 1, 0, 1), y(\text{exit}, 1, 1, 0), y(\text{exit}, 1, 1, 1)$
 $y(\text{open}, 0, 0, 0), y(\text{open}, 0, 0, 1), y(\text{open}, 0, 1, 0), y(\text{open}, 0, 1, 1)$
 $y(\text{open}, 1, 0, 0), y(\text{open}, 1, 0, 1), y(\text{open}, 1, 1, 0), y(\text{open}, 1, 1, 1)$

We can now build the clauses for the formula. Note that each clause is linked to the next one with a conjunction ($\wedge$).

1- Every state has at least one colour

$(x(0,0) \vee x(0,1)) \wedge$
 $(x(1,0) \vee x(1,1)) \wedge$
 $(x(2,0) \vee x(2,1)) \wedge$
 $(x(3,0) \vee x(3,1)) \wedge$
 $(x(4,0) \vee x(4,1))$

2- Every state has at most one colour

$(\neg x(0,0) \vee \neg x(0,1)) \wedge$
 $(\neg x(1,0) \vee \neg x(1,1)) \wedge$

$$\begin{aligned}
&(\neg x(2, 0) \vee \neg x(2, 1)) \wedge \\
&(\neg x(3, 0) \vee \neg x(3, 1)) \wedge \\
&(\neg x(4, 0) \vee \neg x(4, 1))
\end{aligned}$$

3- A parent relation is set when a state and its parent are coloured

For the transition from state p to state q with input a and output b , if state p has colour i and state q has colour j , then the final model must have a transition from state coloured i to a state coloured j with input a and output b .

$$x(p, i) \wedge x(q, j) \Rightarrow y(a, b, i, j)$$

$$\text{By De-Morgan's law: } \neg x(p, i) \vee \neg x(q, j) \vee y(a, b, i, j)$$

Transition from state 0 to 1 with input *open* and output 1:

$$\begin{aligned}
&(y(\text{open}, 1, 0, 0) \vee \neg x(0, 0) \vee \neg x(1, 0)) \wedge \\
&(y(\text{open}, 1, 0, 1) \vee \neg x(0, 0) \vee \neg x(1, 1)) \wedge \\
&(y(\text{open}, 1, 1, 0) \vee \neg x(0, 1) \vee \neg x(1, 0)) \wedge \\
&(y(\text{open}, 1, 1, 1) \vee \neg x(0, 1) \vee \neg x(1, 1))
\end{aligned}$$

Transition from state 0 to 3 with input *exit* and output 0:

$$\begin{aligned}
&(y(\text{exit}, 0, 0, 0) \vee \neg x(0, 0) \vee \neg x(1, 0)) \wedge \\
&(y(\text{exit}, 0, 0, 1) \vee \neg x(0, 0) \vee \neg x(1, 1)) \wedge \\
&(y(\text{exit}, 0, 1, 0) \vee \neg x(0, 1) \vee \neg x(1, 0)) \wedge \\
&(y(\text{exit}, 0, 1, 1) \vee \neg x(0, 1) \vee \neg x(1, 1))
\end{aligned}$$

Transition from state 1 to 2 with input *open* and output 0:

$$\begin{aligned}
&(y(\text{open}, 0, 0, 0) \vee \neg x(1, 0) \vee \neg x(2, 0)) \wedge \\
&(y(\text{open}, 0, 0, 1) \vee \neg x(1, 0) \vee \neg x(2, 1)) \wedge \\
&(y(\text{open}, 0, 1, 0) \vee \neg x(1, 1) \vee \neg x(2, 0)) \wedge \\
&(y(\text{open}, 0, 1, 1) \vee \neg x(1, 1) \vee \neg x(2, 1))
\end{aligned}$$

Transition from state 3 to 4 with input *open* and output 0:

$$\begin{aligned}
&(y(\text{open}, 0, 0, 0) \vee \neg x(3, 0) \vee \neg x(4, 0)) \wedge \\
&(y(\text{open}, 0, 0, 1) \vee \neg x(3, 0) \vee \neg x(4, 1)) \wedge \\
&(y(\text{open}, 0, 1, 0) \vee \neg x(3, 1) \vee \neg x(4, 0)) \wedge \\
&(y(\text{open}, 0, 1, 1) \vee \neg x(3, 1) \vee \neg x(4, 1))
\end{aligned}$$

5- A parent relation forces a state once the parent is coloured

$$y(a, b, i, j) \wedge x(p, i) \Rightarrow x(q, j)$$

$$\text{By De-Morgan's law: } \neg y(a, b, i, j) \vee \neg x(p, i) \vee x(q, j)$$

If the final model has a transition from colour i to colour j with input a and output b , and p (the parent) was assigned colour i , then q (the child) must be assigned colour j .

Transition from state 0 to 1 with input *open* and output 1:

$$\begin{aligned}
&(\neg y(\text{open}, 1, 0, 0) \vee \neg x(0, 0) \vee x(1, 0)) \wedge \\
&(\neg y(\text{open}, 1, 0, 1) \vee \neg x(0, 0) \vee x(1, 1)) \wedge \\
&(\neg y(\text{open}, 1, 1, 0) \vee \neg x(0, 1) \vee x(1, 0)) \wedge
\end{aligned}$$

$$(\neg y(open, 1, 1, 1) \vee \neg x(0, 1) \vee x(1, 1))$$

Transition from state 0 to 3 with input *exit* and output 0:

$$\begin{aligned} &(\neg y(exit, 0, 0, 0) \vee \neg x(0, 0) \vee x(3, 0)) \wedge \\ &(\neg y(exit, 0, 0, 1) \vee \neg x(0, 0) \vee x(3, 1)) \wedge \\ &(\neg y(exit, 0, 1, 0) \vee \neg x(0, 1) \vee x(3, 0)) \wedge \\ &(\neg y(exit, 0, 1, 1) \vee \neg x(0, 1) \vee x(3, 1)) \end{aligned}$$

Transition from state 1 to 2 with input *open* and output 0:

$$\begin{aligned} &(\neg y(open, 0, 0, 0) \vee \neg x(1, 0) \vee x(2, 0)) \wedge \\ &(\neg y(open, 0, 0, 1) \vee \neg x(1, 0) \vee x(2, 1)) \wedge \\ &(\neg y(open, 0, 1, 0) \vee \neg x(1, 1) \vee x(2, 0)) \wedge \\ &(\neg y(open, 0, 1, 1) \vee \neg x(1, 1) \vee x(2, 1)) \end{aligned}$$

Transition from state 3 to 4 with input *open* and output 0:

$$\begin{aligned} &(\neg y(open, 0, 0, 0) \vee \neg x(3, 0) \vee x(4, 0)) \wedge \\ &(\neg y(open, 0, 0, 1) \vee \neg x(3, 0) \vee x(4, 1)) \wedge \\ &(\neg y(open, 0, 1, 0) \vee \neg x(3, 1) \vee x(4, 0)) \wedge \\ &(\neg y(open, 0, 1, 1) \vee \neg x(3, 1) \vee x(4, 1)) \end{aligned}$$

6- Each parent relation must target at least one colour

$$\begin{aligned} &(y(edit, 0, 0, 0) \vee y(edit, 0, 0, 1)) \wedge \\ &(y(edit, 0, 1, 0) \vee y(edit, 0, 1, 1)) \wedge \\ &(y(exit, 0, 0, 0) \vee y(exit, 0, 0, 1)) \wedge \\ &(y(exit, 0, 1, 0) \vee y(exit, 0, 1, 1)) \wedge \\ &(y(open, 0, 0, 0) \vee y(open, 0, 0, 1)) \wedge \\ &(y(open, 0, 1, 0) \vee y(open, 0, 1, 1)) \end{aligned}$$

7- Each parent relation can target at most one colour

$$\begin{aligned} &(\neg y(edit, 0, 0, 0) \vee \neg y(edit, 0, 0, 1)) \wedge \\ &(\neg y(edit, 0, 1, 0) \vee \neg y(edit, 0, 1, 1)) \wedge \\ &(\neg y(exit, 0, 0, 0) \vee \neg y(exit, 0, 0, 1)) \wedge \\ &(\neg y(exit, 0, 1, 0) \vee \neg y(exit, 0, 1, 1)) \wedge \\ &(\neg y(open, 0, 0, 0) \vee \neg y(open, 0, 0, 1)) \wedge \\ &(\neg y(open, 0, 1, 0) \vee \neg y(open, 0, 1, 1)) \end{aligned}$$

8- All determinisation conflicts are explicitly added as clauses

The colouring graph makes it clear what states must not have the same colour. This can be translated into SAT to prevent any conflict. Let p and q be linked in the colouring graph. If state p is coloured i , then state q must not be coloured with the same colour.

$$x(p, i) \Rightarrow \neg x(q, i)$$

By De-Morgan's law: $\neg x(p, i) \vee \neg x(q, i)$

$$\begin{aligned} &(\neg x(0, 0) \vee \neg x(3, 0)) \wedge \\ &(\neg x(0, 1) \vee \neg x(3, 1)) \wedge \\ &(\neg x(0, 0) \vee \neg x(1, 0)) \wedge \end{aligned}$$

$$(\neg x(0, 1) \vee \neg x(1, 1))$$

Appendix C

Results

Adding Patterns to the SAT Solver In this section, we compare closely related approaches:

1. PAT-DFASAT vs. (PAT-DFASAT with encoding pattern into SAT)
2. Pattern-guided DFASAT vs. (Pattern-guided DFASAT with encoding pattern into SAT)
3. Pattern-guided DFASAT vs. Standard DFASAT

The primary objective of this comparison is to determine whether integrating the patterns into the SAT solver or into DFASAT heuristic improves the learner’s performance.

Our results indicate that adding hard patterns to the SAT solver provides no significant benefit. In fact, this approach substantially increases the number of clauses in the Boolean formula, potentially causing the learner to reach the system’s memory limits before a solution is found.

To quantify these findings, we utilised the A12 effect size metric to compare the two approaches. As a reminder, the A12 metric is interpreted as follows:

- 0.4 to 0.65 indicates no significant difference between the two approaches.
- Above 0.65 indicates that the first approach performs significantly better than the other.
- Below 0.4 indicates that the first approach performs significantly worse than the other.

As illustrated in the following results, the majority of the values fall within the 0.4 to 0.65 range, confirming that the additional complexity of the patterns does not yield a proportional improvement in model accuracy. Note that the evaluation included different data sizes; each A12 value refers to the comparison between two learners at the same data size.

Comparing PAT-DFASAT vs. (PAT-DFASAT with encoding pattern into SAT)

PAT-DFASAT: integrates the mined patterns into the EDSM and then uses the SAT solver. PAT-DFASAT with encoding pattern into SAT: integrates the mined patterns into the EDSM and then integrating them again into the SAT solver.

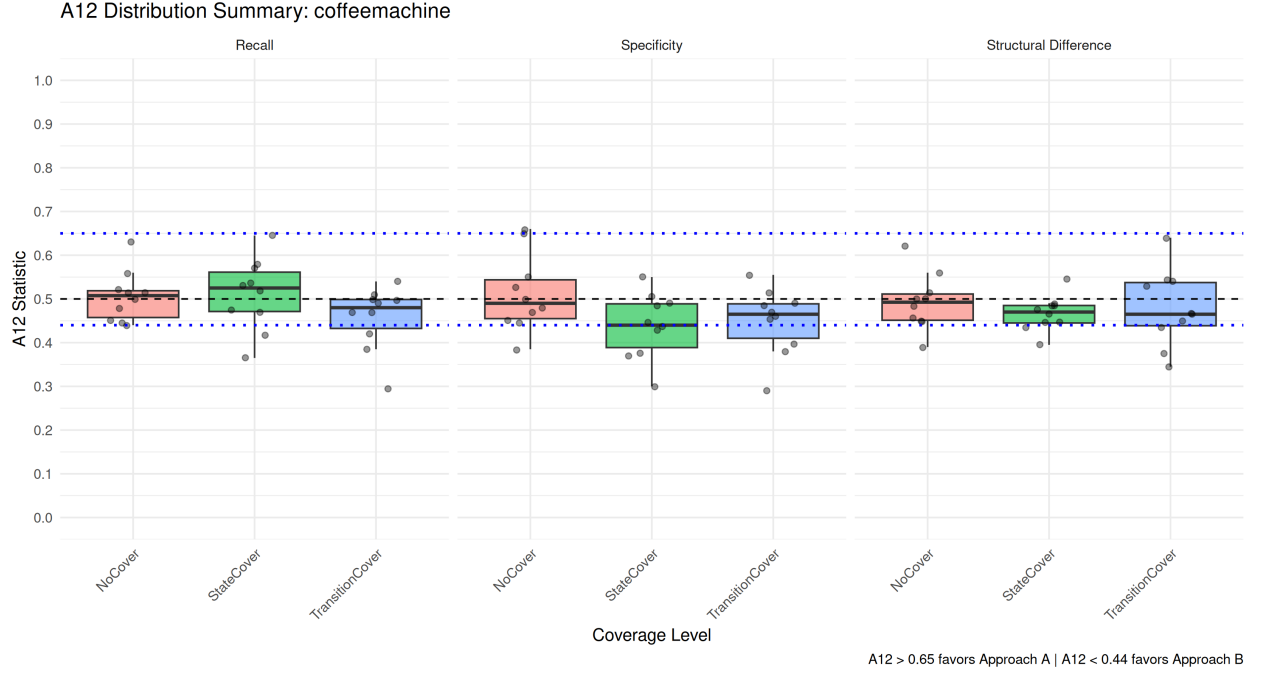


Figure C.1: *Coffee Machine: PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)* - Data size: $|Q|$ between 130 and 1300

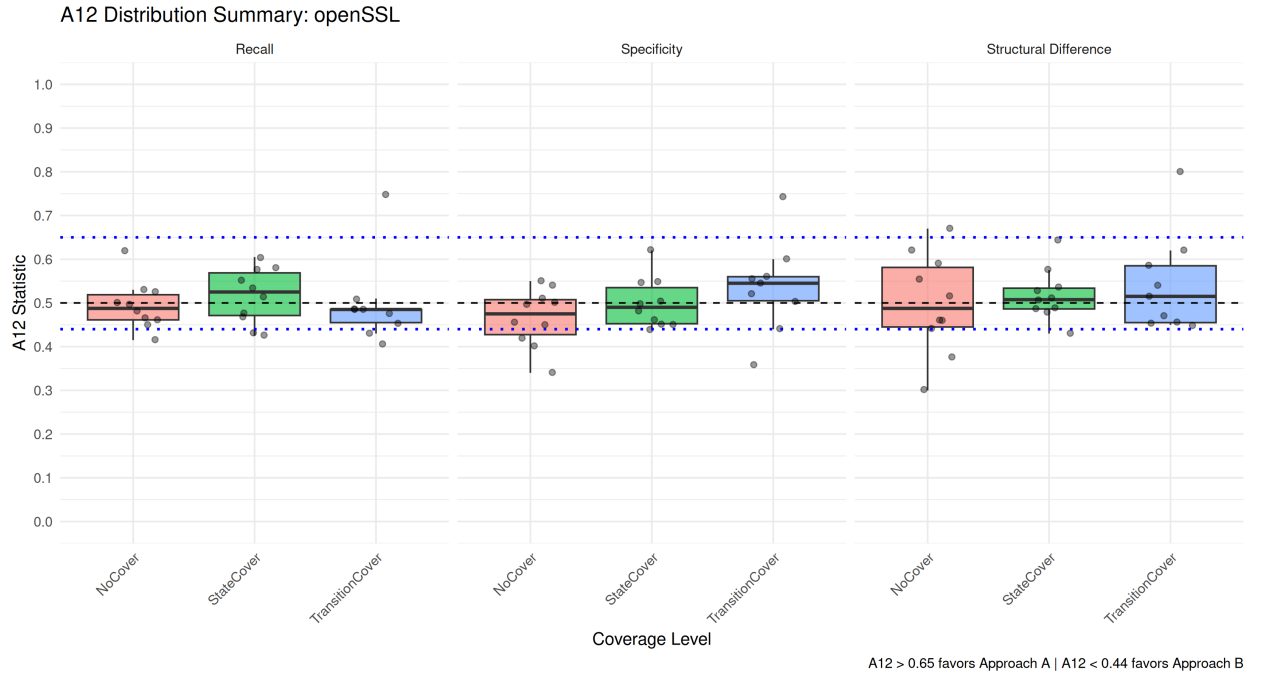


Figure C.2: *OpenSSL - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)* - Data size: $|Q|$ between 1200 and 4100

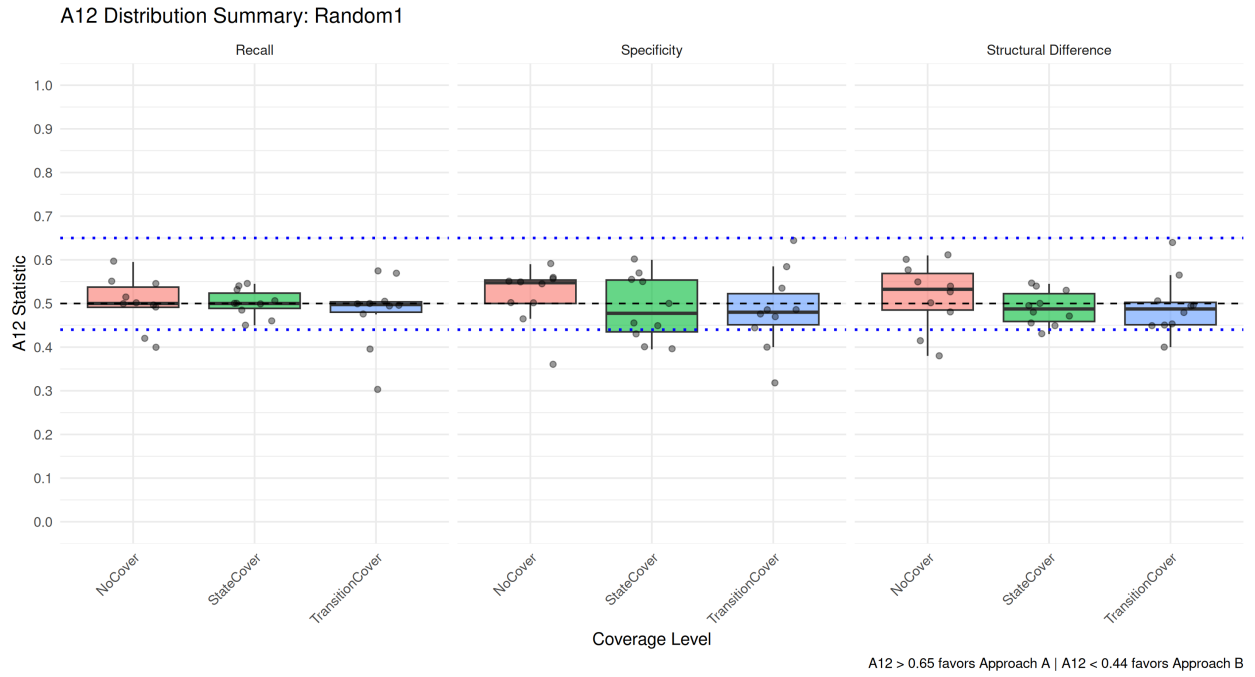


Figure C.3: *Random1 - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)* - $|Q|$ between 150 and 1500

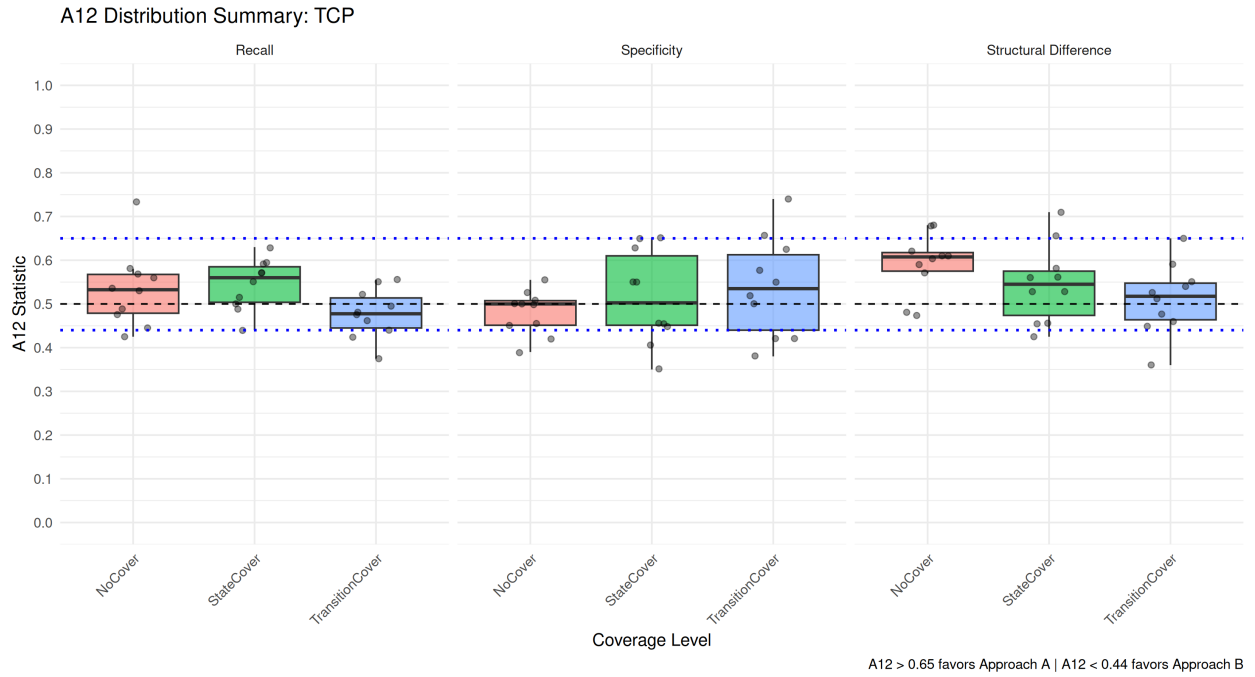


Figure C.4: *TCP - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)* - $|Q|$ between 2500 and 9900

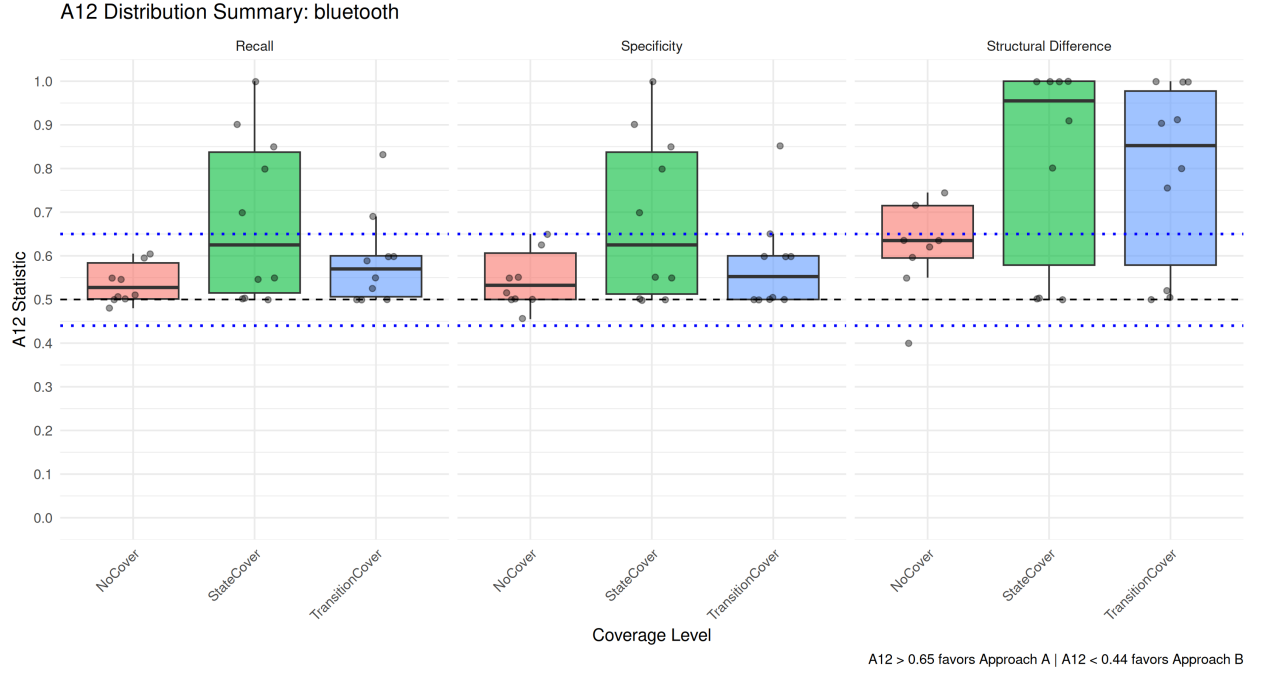


Figure C.5: *Bluetooth - PAT-DFASAT vs. (PAT-EDSM+SAT with pattern constraints)*

Comparing Pattern-guided DFASAT with (Pattern-guided DFASAT with encoding pattern into SAT)

Pattern-guided DFASAT: integrating mined Patterns into the standard DFASAT heuristic and then using the SAT solver. Pattern-guided DFASAT with encoding pattern into SAT: integrating mined Patterns into the standard DFASAT heuristic and then integrating them again into the SAT solver.

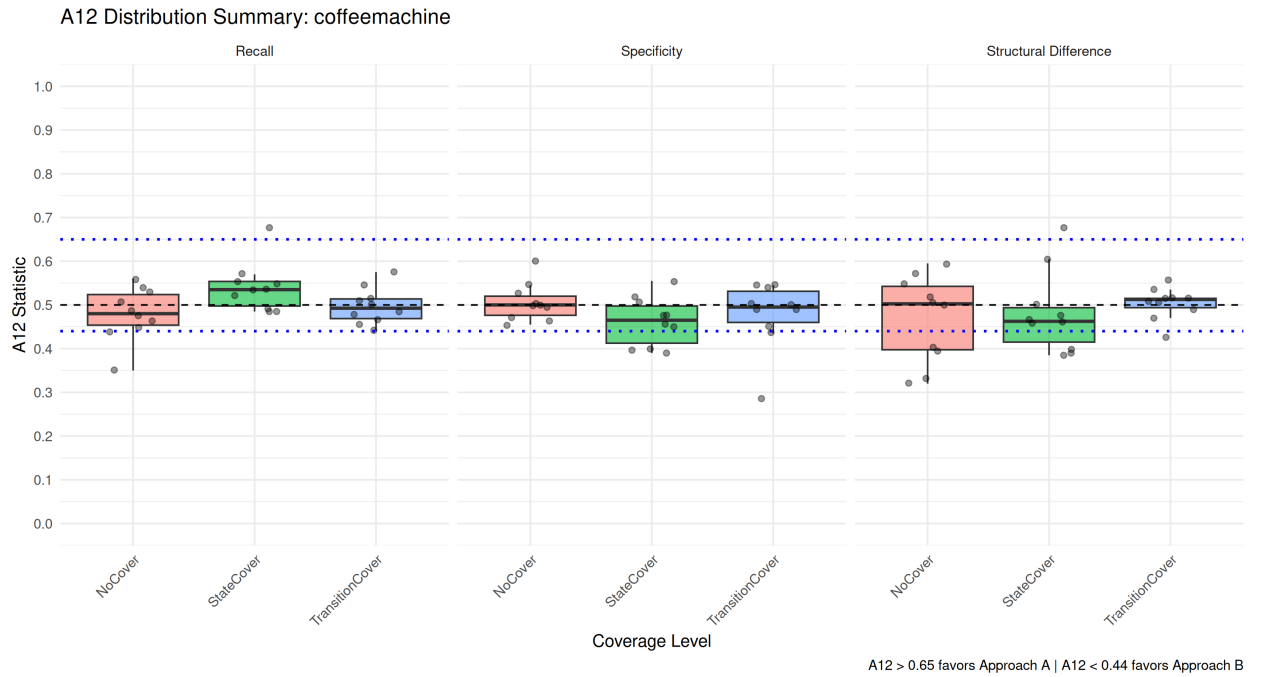


Figure C.6: *Coffeemachine - BiasedSiccoSAT vs BiasedSiccoSATPAT - Data size: $|Q|$ between 130 and 1300*

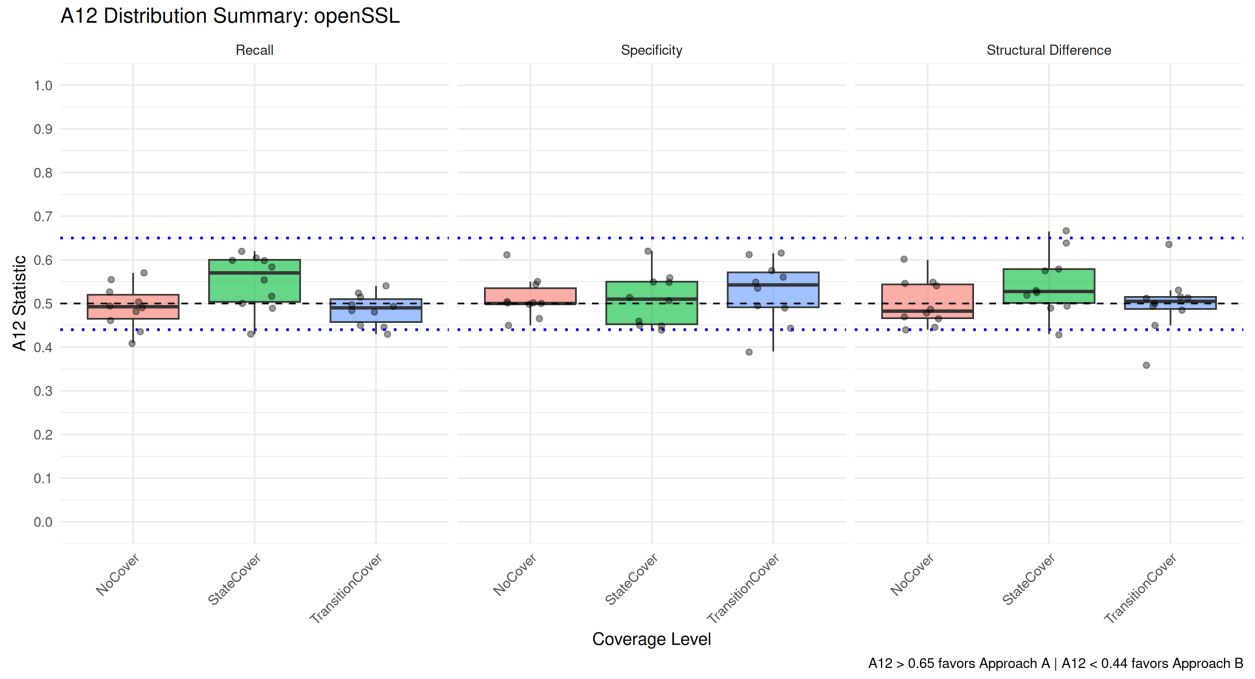


Figure C.7: *openssl* - *BiasedSiccoSAT* vs *BiasedSiccoSATPAT* Data size: $|Q|$ between 1200 and 4100

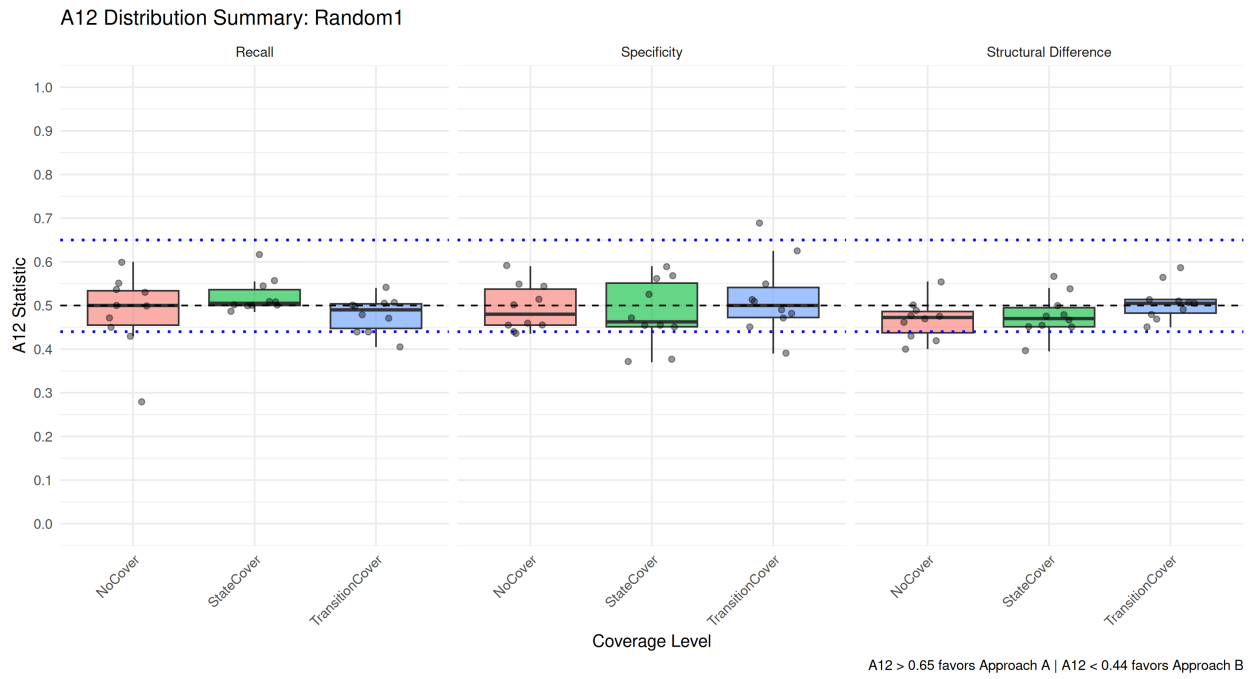


Figure C.8: *Random1* - *BiasedSiccoSAT* vs *BiasedSiccoSATPAT* - $|Q|$ between 150 and 1500

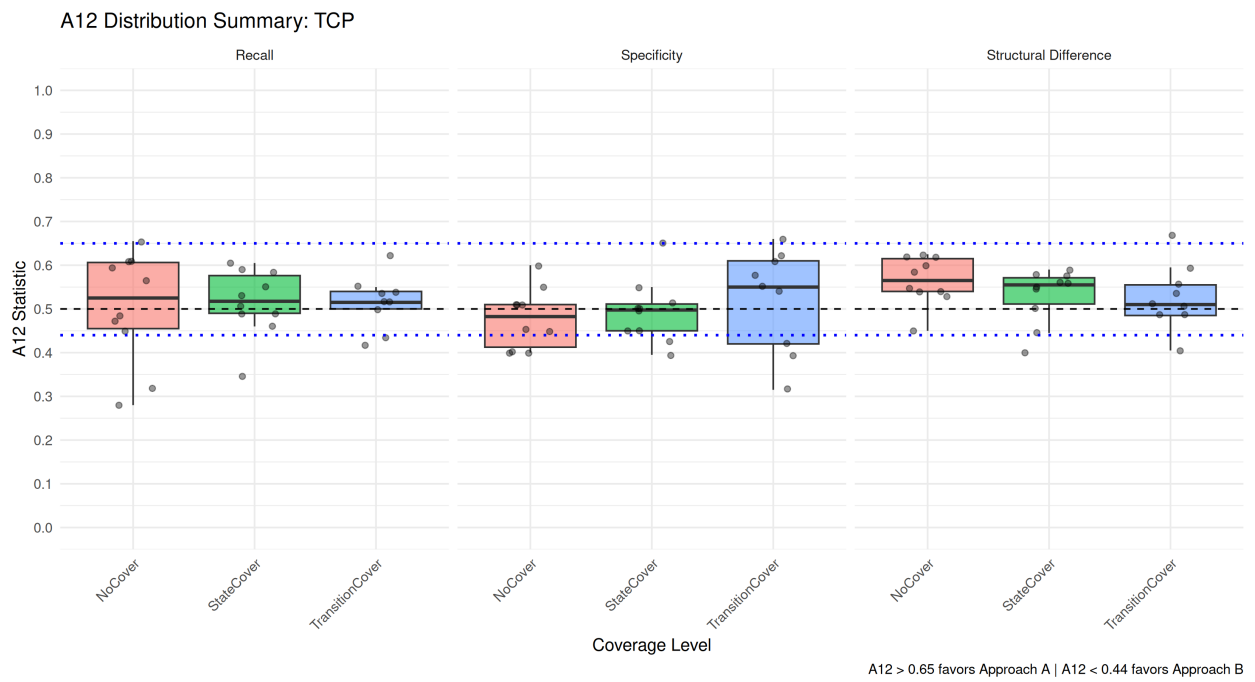


Figure C.9: *TCP* - *BiasedSiccoSAT* vs *BiasedSiccoSATPAT* - $|Q|$ between 2500 and 9900

Pattern-guided DFASAT vs. Standard DFASAT

Pattern-guided DFASAT: integrating mined Patterns into the standard DFASAT heuristic and then using the SAT solver. Our results indicate that adding patterns to the DFASAT heuristic for state merging provides no significant benefit.

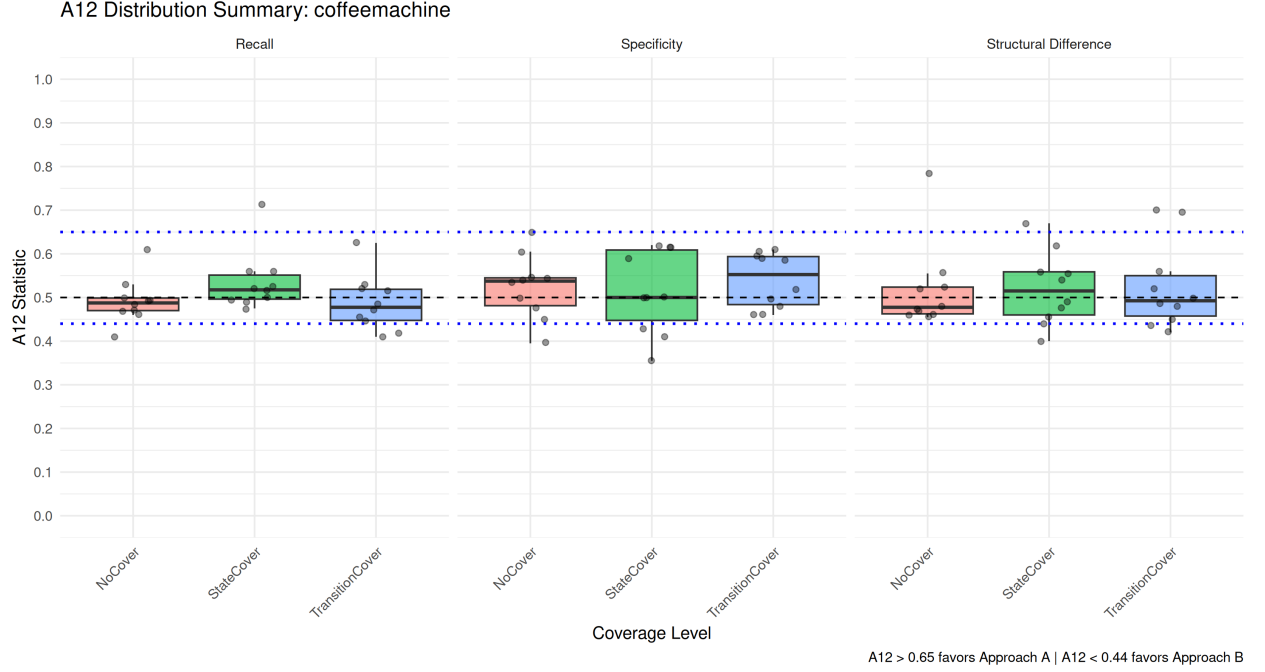


Figure C.10: *Coffee Machine - Pattern-guided DFASAT vs. Standard DFASAT - Data size: $|Q|$ between 130 and 1300*

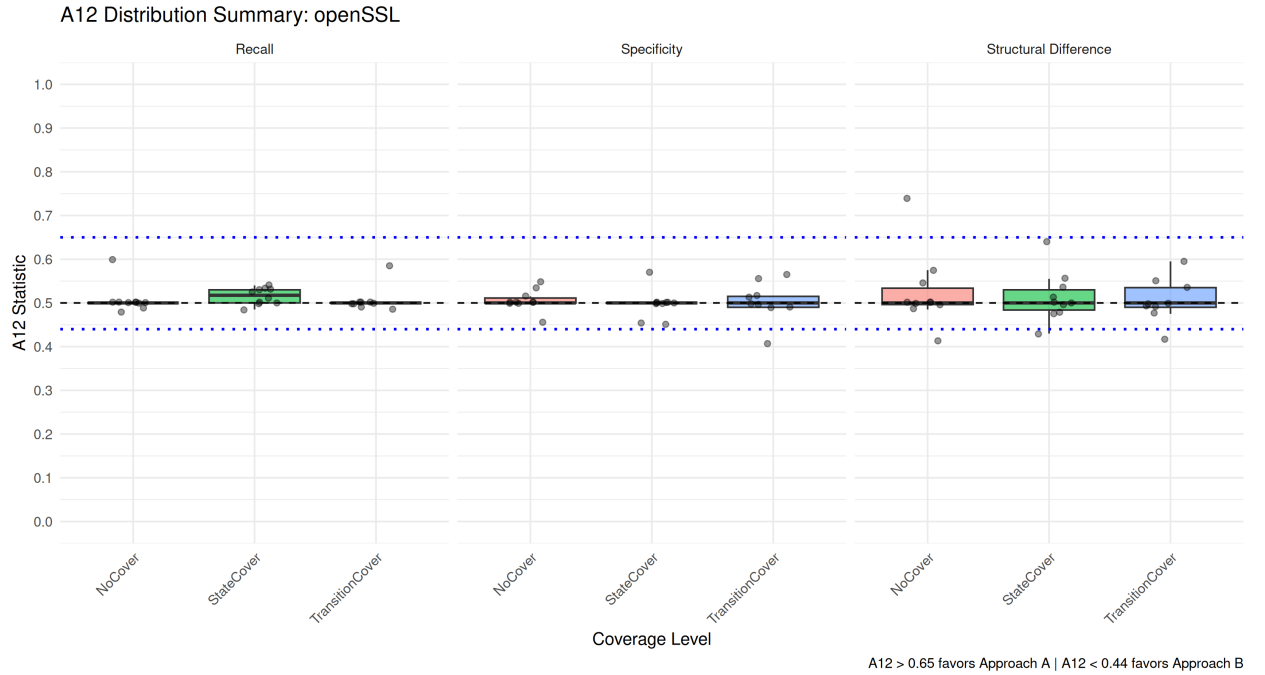


Figure C.11: *OpenSSL - Pattern-guided DFASAT vs. Standard DFASAT - Data size: $|Q|$ between 1200 and 4100*

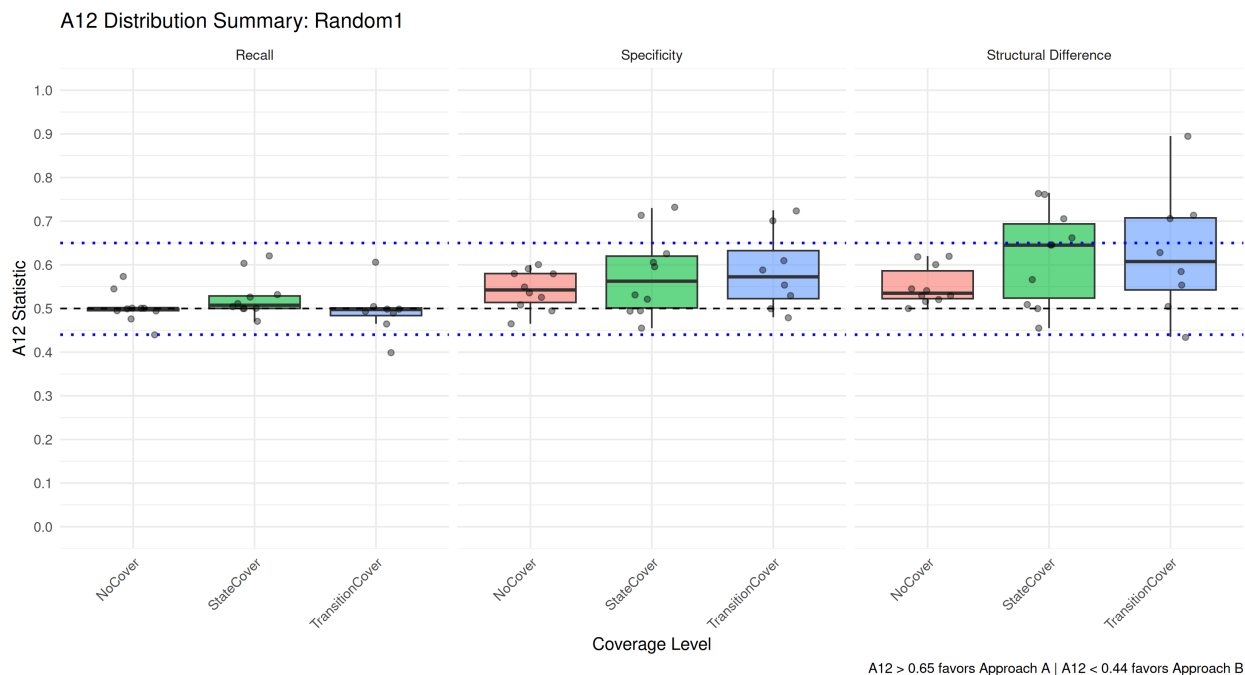


Figure C.12: *Random1* - Pattern-guided DFASAT vs. Standard DFASAT - $|Q|$ between 150 and 1500

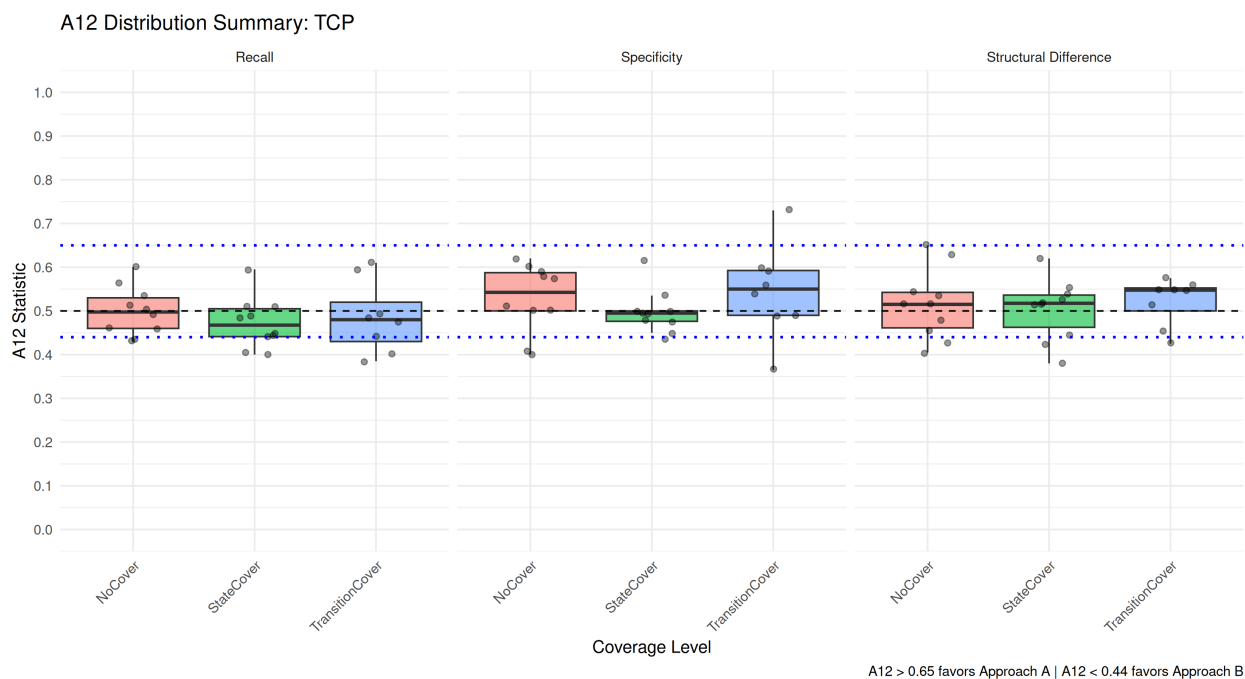


Figure C.13: *TCP* - Pattern-guided DFASAT vs. Standard DFASAT - $|Q|$ between 2500 and 9900