Statistical Machine Learning for Automata-Based Modelling of Simulink Autopilot System

by

Armina Faghihi

Thesis submitted to the University of Ottawa

in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

School of Electrical Engineering and Computer Science

University of Ottawa

Ottawa, Ontario, Canada

Abstract

Aircraft autopilot controllers are often developed as Simulink models. Verifying such controllers is challenging because their behaviour is mainly observed through numeric simulation traces, and there is no explicit behavioural model showing how the controller responds under different conditions. In this thesis, we use automata learning to derive state machines from simulation data to support analysis and verification of a Simulink-based aircraft autopilot. A key difficulty is that standard automata learning assumes a finite alphabet, whereas the model's inputs and outputs are numeric signals.

We address this with an ML-enhanced passive automata-learning approach (MELA) that combines machine learning with automata learning. Feature-importance analysis selects informative signals, and decision tree based range abstraction partitions their numeric ranges into intervals. These intervals are then used to abstract the time-series traces before applying passive automata learning. We apply this pipeline to a closed-loop Simulink aircraft autopilot and learn Moore machines that capture its behaviour.

Our evaluation compares MELA with a Manual baseline that uses the same data generation and learning procedures but relies on manually chosen variables and numeric abstractions. Across four learning sets and six abstraction configurations, MELA reduces the number of states and transitions in the learned automata by an average of 11.6% and improves accuracy by an average of 18.5% compared to the Manual baseline. The learned state machines support verification and exploration: by expressing the autopilot's requirements as temporal queries and evaluating them on the models, we can check whether these requirements are satisfied and identify behaviours that were not known in advance.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Mehrdad Sabetzadeh, whose guidance, support, and constructive feedback throughout this thesis have greatly shaped and strengthened this work. I am also grateful to Professor Shiva Nejati, whose invaluable guidance, feedback, and continuous encouragement have been significant in completing this work.

I am also deeply thankful to my parents, Zahra and Ahmad, for their endless love and belief in me and for always being a source of inspiration to me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Embedded control systems, such as aircraft autopilots, are used in safety applications where faults can affect people and equipment. Model-based development is commonly used to design and validate such controllers before implementation, with Simulink widely used to model both the feedback control logic and the physical system. As these models increase in size and contain many interacting signals, their behaviour is often explored only through simulation traces, without an explicit behavioural model or formal specification that summarizes how the controller behaves across operating conditions. Analysis and verification techniques such as model-based testing, monitoring, and debugging support have therefore been proposed for Simulink models [8]. Automata learning provides a complementary approach: it infers finite-state models from executions of a system, yielding explicit behavioural models that can be inspected, tested, and checked against requirements [1, 39].

1.1 Challenges

Simulink is widely used for the model-based development of embedded control systems in domains such as aerospace, automotive, and industrial automation. Controllers are typically modeled as block diagrams that interact with simulated models and operate on high-dimensional, time-varying signals. As prior work on analysis of Simulink models has shown, understanding why a model satisfies or violates its requirements can be difficult when the behaviour is encoded implicitly in the blocks and signal flows [8]. For complex controllers such as aircraft autopilots, this lack of a clear behavioural description makes it hard to explain how the controller behaves across different operating conditions, to reason about its requirements, and to compare alternative design or configuration choices.

Automata learning offers a way to infer finite-state models from observed executions of a system, and has been applied to a variety of software and control-intensive systems to obtain models suitable for inspection, testing, and verification [1, 22, 39]. In Simulink-based control models, the natural data are numeric time-series signals, sometimes combined with mode indicators and other discrete variables. Directly applying automata learning to such raw data is difficult because the range of possible values is very large, leading to models that are overly complex and fail to generalize well. Recent work on ML-enhanced passive automata learning (MELA) has shown that combining data-driven abstraction with automata learning can address this problem in a network security setting by automatically deriving abstractions for numeric signals before learning [7]. Existing case studies on model learning for complex systems have focused on communication protocols [1], cyber-physical systems [39], and networked intrusion-detection systems [7]. There are also a few

works that learn automata or hybrid automata from Simulink models, for example using automata learning to infer a state machine for a lane-change module in an autonomous driving system [40], but these rely on active automata learning or hybrid-specific techniques and do not combine ML-based numeric abstraction with passive automata learning in the style of MELA. These observations motivate generalizing an ML-enhanced, passive automata-learning approach, previously studied in a networked intrusion-detection setting, to Simulink controllers such as aircraft autopilots, and examining the size and accuracy of the learned models in this new domain.

1.2 Research Contribution

This thesis builds on the MELA framework of Ayoughi et al. [7], which combines supervised abstraction, passive automata learning, and temporal-logic analysis to derive interpretable state machines from numeric time-series data in a network security context. In MELA, the system under learning is a commercial intrusion-detection router, and its behaviour is captured from network-traffic logs. Our work adopts and adapts the same overall framework to a Simulink model of an aircraft autopilot. As in the network domain, we work with numeric time-series data, but here these traces arise from a closed-loop plant-controller model rather than from network flows. This change of domain requires adjustments to the data-generation setup, abstraction choices, and way temporal properties are formulated and evaluated.

Concretely, the contributions of this thesis are as follows.

- **Time-segmented modelling of controller behaviour.** We introduce a time-segmented view of the Simulink inputs and outputs, so that the learned automata capture distinct phases of the controller behaviour instead of a flat sequence of samples.

- **Recurrent and feedforward temporal query patterns for control systems.** Instead of a single temporal-query pattern, we support two complementary patterns: a recurrent pattern, where the system may remain in its current state or advance to the next one, and a feedforward pattern, where each step must move to a strictly "next" state. This distinction helps us interpret query results as pass, fail, or vacuous and find the states and transitions responsible for each outcome.

- **Feature-importance refinement in decision-tree abstraction.** In our work, we encountered decision trees where several features were assigned very similar feature importance, so we extend the abstraction to incorporate additional importance metrics and to explore alternative signal combinations in order to identify the most impactful inputs.

- **Robustness-based evaluation of learned models.** In addition to the evaluation used in the original MELA network case study, we also compute robustness values for the temporal-logic requirements and compare MELA and the Manual baseline using a Mann–Whitney U test, so that the learned automata are assessed not only by size and accuracy but also by how strongly they satisfy the requirements.

Taken together, these contributions provide an empirical demonstration that the MELA

framework, originally developed for network intrusion detection [7], can be transferred to Simulink control models, and that its supervised abstraction and passive learning components are effective for deriving interpretable finite-state models from cyber–physical time-series data.

1.3 Organization

The remainder of this thesis is organized as follows.

Chapter 2: Background - This chapter introduces the core concepts used in the thesis, including automata and automata learning, finite-state machine models, machine-learning classifiers, key Simulink fundamentals, and temporal query checking.

Chapter 3: Related Work - This chapter surveys related work on automata learning, model-based testing of Simulink and machine-learning–based abstraction and analysis.

Chapter 4: Approach - This chapter presents the MELA-based approach for the Simulink autopilot case study, describing the model, data generation, trace creation and abstraction, and the application of automata learning to obtain behavioural models.

Chapter 5: Empirical Evaluation - This chapter reports the empirical evaluation of MELA on the autopilot, detailing the implementation, experimental setup, research questions, evaluation metrics, and results, including verification via temporal queries.

Chapter 6: Conclusion and Future Work - The final chapter summarizes the key findings of the research, discusses the implications of the results, and outlines potential directions for future research.

Chapter 2

Background

This chapter introduces the main concepts needed for this research. It first presents finite-state machines as behavioural models and then reviews active and passive automata learning, which are used to infer state machines from system executions. It then summarizes the machine-learning techniques used for trace abstraction and the basic Simulink notions needed to understand the autopilot model. Finally, it outlines the temporal query checking framework used to evaluate properties on the learned state machines. Together, these topics provide the background required to understand the methods used in this thesis.

2.1 Finite-State Machines

Finite-state machines provide a foundational formalized model to the control logic of reactive systems, such as protocol handlers, controllers, and other components that respond to sequences of input events [4, 16, 20, 45]. In this section, we recall the standard models

of deterministic and nondeterministic finite automata, as well as Mealy and Moore machines. These models form the semantic basis for the learned automata used later in this thesis [43, 45].

2.1.1 Deterministic Finite Automata (DFA)

A deterministic finite automaton (DFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite, non-empty set of states, Σ is a finite input alphabet, $\delta : Q \times \Sigma \to Q$ is a (total) transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of accepting states [20]. A DFA processes a word $w = a_1 a_2 \dots a_n \in \Sigma^*$ by starting in q_0 and applying δ step by step, giving a unique run $q_0, q_1, \dots, q_n$ with $q_i = \delta(q_{i-1}, a_i)$ for $1 \leq i \leq n$. The word w is accepted if the final state q_n belongs to F, and the language $L(A)$ recognized by A is the set of all accepted words.

DFAs give an operational representation of regular languages, and every regular language has a unique minimal DFA, up to isomorphism [20]. In cyber–physical and safety-critical applications, random forests have been used, e.g., to classify executions of autonomous driving controllers with respect to requirement satisfaction [44] and to support learned models of complex software components in conjunction with automata-based techniques [7, 39]. Figure 2.1 illustrates a simple DFA together with the discrimination tree used by an active learning algorithm, showing both the automaton itself and the data structure used to distinguish its states [4].

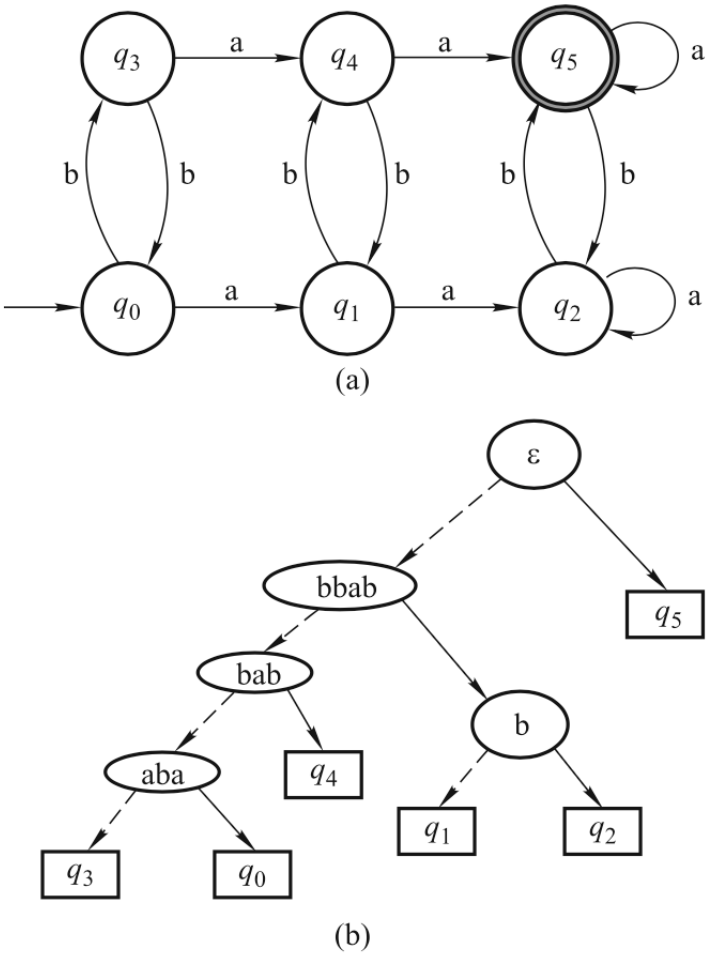


Figure 2.1: A simple deterministic finite automaton and its discrimination tree, adapted.

2.1.2 Nondeterministic Finite Automata (NFA)

Nondeterministic finite automata extend DFAs by allowing several possible successor states for a given state and input symbol. Formally, a nondeterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0 and F are as before, but the transition function

is now $\delta : Q \times \Sigma \to 2^Q$, so that $\delta(q, a)$ is a set of successor states [20]. Many presentations also allow ε-transitions that move between states without using input symbols. These can be handled by extending the domain of δ to $(\Sigma \cup \{\varepsilon\})$ and using suitable ε-closure constructions.

An NFA accepts a word $w = a_1 a_2 \dots a_n$ if there exists at least one run $q_0, q_1, \dots, q_n$ that follows the transition relation and ends in an accepting state $q_n \in F$ [20]. NFAs and DFAs are equivalent in expressive power: every NFA recognizes a regular language, and for every NFA there exists a DFA that recognizes the same language. The classical subset construction turns an NFA into an equivalent DFA whose states are subsets of Q. This translation can be exponential in the number of states, which explains why NFAs can be more compact than DFAs [20].

Nondeterminism offers an abstract way to model underspecification or uncertainty in observed behaviour. Several learning algorithms can be seen as implicitly resolving nondeterministic choices into deterministic hypothesis automata [4, 45].

2.1.3 Mealy Machines

For modelling input–output behaviour, it is often more convenient to work with transducers that produce outputs instead of acceptor automata. A (deterministic) Mealy machine is a finite-state transducer that labels transitions with outputs. Formally, a Mealy machine is a tuple $M = (Q, \Sigma, \Lambda, \delta, \lambda, q_0)$, where Q is a finite set of states, Σ is a finite input alphabet, Λ is a finite output alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $\lambda : Q \times \Sigma \to \Lambda$ is an output function, and $q_0 \in Q$ is the initial state [45, 46]. When the machine reads an input

symbol $a \in \Sigma$ in state $q \in Q$, it moves to state $\delta(q, a)$ and emits the output symbol $\lambda(q, a)$.

For an input word $w = a_1 \dots a_n$, a Mealy machine produces an output word $o_1 \dots o_n$, where each o_i is determined by the current state and the current input symbol. Thus, a Mealy machine defines a function from Σ^* to Λ^* that maps each input word to an output word of the same length.

Mealy machines are a standard model for synchronous digital controllers and communication protocols, in which the output at each step depends on both the current state and the current input [4, 45]. In automata learning, they are widely used to model the behaviour of reactive software components and controllers whose outputs depend directly on the current input, for example in protocol state machine inference or interface automata learning [4, 43, 45]. Many recent extensions of active learning algorithms target Mealy-type models with richer timing or data features [12, 46].

2.1.4 Moore Machines

A Moore machine is a closely related model in which outputs are attached to states rather than transitions. Formally, a (deterministic) Moore machine is a tuple $M = (Q, \Sigma, \Lambda, \delta, \gamma, q_0)$, where Q, Σ, Λ, δ and q_0 are as before, and $\gamma : Q \to \Lambda$ is a state output function. When the machine is in state q, it continuously produces the output symbol $\gamma(q)$; reading an input symbol $a \in \Sigma$ causes a state change to $\delta(q, a)$, which may in turn change the output. Therefore, for an input word $w = a_1 \dots a_n$, a Moore machine produces an output word of length $n+1$ if we include the initial output in q_0 [47].

Mealy and Moore machines are equivalent in the sense that, for every Mealy machine,

there is an equivalent Moore machine that realizes the same input–output relation up to a fixed shift, and vice versa [45, 46]. The choice between them is therefore mainly a matter of convenience for a given modelling task. However, it can have practical consequences for modelling and learning. In particular, Moore machines often match naturally with sampled-data controllers and synchronous block diagrams where outputs are associated with discrete system modes or configurations, as in many embedded control and Simulink-style models [45, 47]. Figure 2.2 illustrates this idea using a pattern-filtering example: the NFA on the left recognises the regular expression `aa*b`, while the Moore machine on the right produces pass/mask outputs to indicate which input positions are relevant for matching the pattern [47].

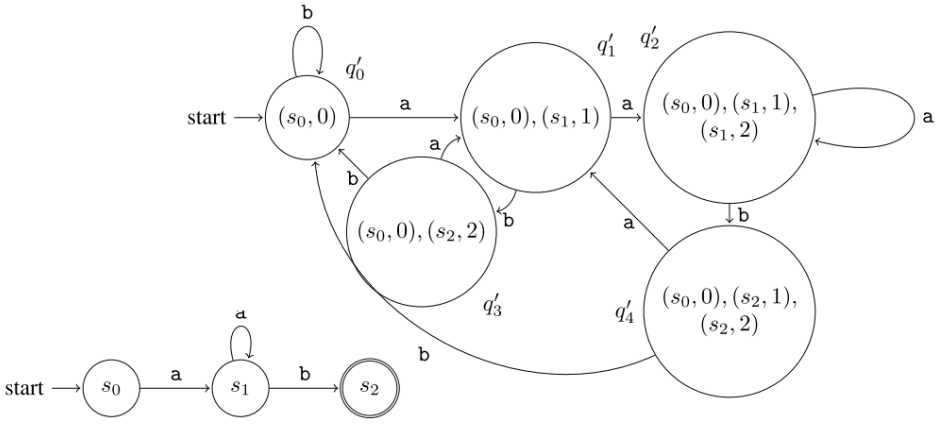


Figure 2.2: Left: NFA for the pattern `aa*b`. Right: The corresponding Moore-machine filter (right) [47].

2.2 Automata Learning

Finite-state automata provide a simple but powerful way to model the observable behaviour of reactive systems [4, 26, 45]. In such models, states represent abstract configurations of the system and transitions describe how the system responds to input symbols over time. Automata-based models have been used to describe and analyze many software and hardware systems, including communication protocols, controllers, and other input–output components [26,45]. Automata learning builds on this view by inferring such models automatically from observations or experiments, instead of constructing them manually [4, 45].

In this thesis, we use finite-state automata as behavioural models for the Simulink aircraft autopilot and focus on learning such automata from data, following existing work on automata-based modelling and verification of control systems [2, 41]. We distinguish between *active* automata learning, where a learner can interact with a system under learning through queries, and *passive* automata learning, where the learner receives a fixed set of observed traces and infers a model from them [4,45]. Our case study uses passive automata learning, but both viewpoints are important for understanding related work and the design of our approach.

2.2.1 Automata as Behavioural Models

Finite-state automata, as introduced in Section 2.1, give a simple but expressive way to model the observable behaviour of reactive systems [4, 26, 45]. States capture abstract

configurations of the system, while transitions describe the change from one state to another. Automata-based models have been applied to a wide range of software and hardware systems, including communication protocols, controllers, and other input–output components [26, 45].

For modelling input–output behaviour, it is often convenient to use Mealy or Moore machines instead of pure acceptor automata. As recalled in Section 2.1.3 and Section 2.1.4, a Mealy machine labels each transition with an output symbol, while a Moore machine associates an output symbol with each state. Both models give a finite-state description of how outputs evolve in response to sequences of inputs and are widely used in automata-based modelling and learning of software components and controllers [41, 43]. In our work, we use Moore machines to represent the abstracted behaviour of the Simulink autopilot [2, 41].

Automata as behavioural models strike a balance between expressiveness and analyzability. They can be visualized as state diagrams that are relatively easy to understand and can be analyzed using standard verification and testing techniques, such as model checking, conformance testing, and coverage analysis [17, 26, 45].

2.2.2 Active Automata Learning

Active automata learning assumes that the learner can interact with a *system under learning* (SUL) by asking queries. In Angluin's classical L^* algorithm [5], the learner can ask membership queries of the form "does the SUL accept word w?" and receives yes/no answers. In addition, the learner can propose a hypothesis automaton and obtain coun-

terexamples through equivalence queries if the hypothesis does not match the SUL. By repeatedly refining its hypothesis based on counterexamples, L^* constructs a minimal DFA that is equivalent to the unknown target automaton, assuming that an oracle can answer equivalence queries [5]. The overall architecture of this query-based setting is illustrated in Figure 2.3, where a learner interacts with a system under learning via membership and equivalence queries, often through a teacher component that may internally rely on testing or model checking [4].

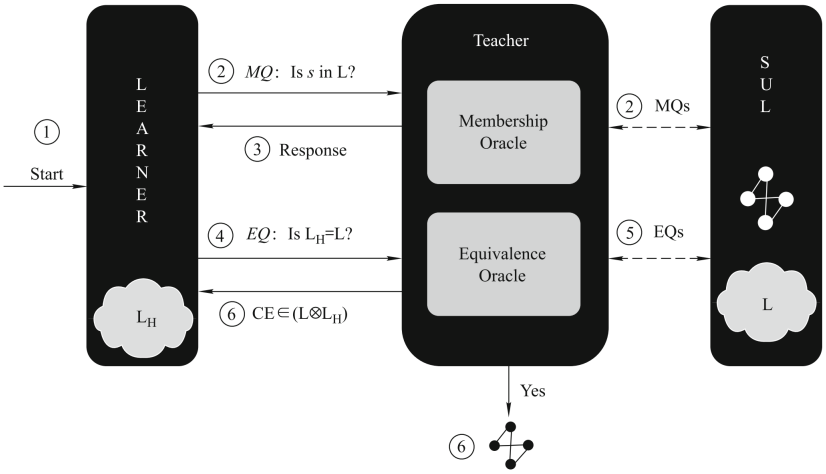


Figure 2.3: Query-based model learning architecture with a minimally adequate teacher, adapted from Fig. 1 in Ali et al. [4].

Later work extended active learning from DFAs to richer models such as Mealy and Moore machines, register automata, and models with timing or data parameters [21, 43]. These extensions have enabled applications to real-world systems, including communication protocols, network stacks, and industrial controllers, where the SUL is tested as a black box and queries are answered by executing the implementation [15, 17, 41, 45].

Active learning is appealing because it can, in principle, guide the exploration of the SUL in a systematic way and construct small, accurate models when queries are answered reliably [21, 43, 45]. However, its assumptions do not always hold in practice. Some implementations do not support reliable resetting, and running the large number of experiments needed for active learning can be time-consuming or require considerable computational and engineering effort [15, 17, 45]. Moreover, repeated executions of the same input sequence may not always produce identical outputs, for example due to nondeterminism or measurement noise. In such settings, it can be more natural or more practical to learn from passively collected traces, as discussed next [4].

2.2.3 Passive Automata Learning

Passive automata learning, sometimes called grammatical inference, assumes that the learner receives a fixed sample of observed behaviour and must infer an automaton that is consistent with this sample [4]. For example, in DFA learning, the sample may consist of sets of positive and negative example words, and the goal is to infer a DFA that accepts all positive words and rejects all negative ones [25]. Unlike active learning, the learner cannot ask new queries and must work only with the data already available [4, 5, 45].

A common family of passive learning algorithms is based on *state merging*. These algorithms start from a prefix-tree acceptor that exactly represents the positive samples and then iteratively merge states while keeping consistency with the labelled examples [4, 42]. The well-known RPNI (Regular Positive and Negative Inference) algorithm is a canonical example of this approach, and many later algorithms can be seen as refinements of its

merging strategy [4]. Figure 2.4 illustrates this state-merging process: starting from a prefix-tree acceptor that accepts exactly the positive sample, compatible states with similar future behaviour are gradually merged to obtain a smaller DFA that generalises beyond the observed examples [25].

Figure 2.4: State-merging DFA induction starting from a prefix-tree acceptor. [25].

Further work has proposed refined merging heuristics and strategies, such as mandatory-merge variants, to improve generalization and robustness to noise [25]. More recent research has also studied passive learning for models beyond plain DFAs, including extended automata with data values or probabilistic annotations [4, 29].

Passive learning is especially useful when system behaviour is observed through logs, simulations, or experiments that cannot easily be repeated on demand, or when direct control over the system's inputs is limited [4]. In such cases, the quality of the learned automaton depends strongly on the coverage and representativeness of the available traces, and on how the raw observations are abstracted into a finite alphabet [4, 29].

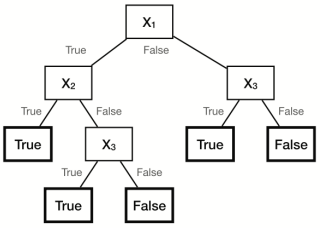
2.3 Machine Learning Classifiers

In addition to finite-state models, this thesis uses supervised machine-learning classifiers to support runtime decisions and validation tasks on execution traces. In particular, we rely on standard classifiers, decision trees, random forests, support vector machines, and logistic regression, that are widely used in software engineering and cyber–physical systems because they provide a good balance between predictive performance, robustness, and interpretability [9, 11, 13, 28]. These classifiers have been successfully employed in requirements-driven test generation and model-based analysis for systems with complex dynamics [7, 18, 39, 40, 44].

2.3.1 Decision Tree Classifier

Decision tree classifiers recursively partition the feature space into regions associated with class labels. Each internal node tests a predicate on a single feature (e.g. whether a signal-based feature x_j is below or above a threshold), each outgoing edge corresponds to the outcome of that test, and each leaf node stores a class prediction or class distribution. A sample is classified by following the unique path determined by these tests from the root to a leaf [11]. As an example, Figure 2.5 shows a simple Boolean function and two decision trees that compute it, making explicit how each root-to-leaf path corresponds to a sequence of tests leading to a class prediction [11].

X_1	X_2	X_3	Y
True	True	True	True
False	True	True	True
True	False	True	True
False	False	True	True
True	True	False	True
False	True	False	False
True	False	False	False
False	False	False	False



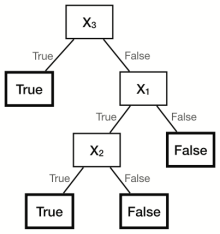


Figure 2.5: Two decision trees representing the Boolean function $Y = X_1 \wedge X_2 \vee X_3$. [11].

Modern decision tree learners typically grow the tree greedily by selecting splits that maximise some impurity reduction criterion, such as information gain or Gini decrease, and then control overfitting via pre-pruning or post-pruning strategies [11]. Recent work emphasises decision trees as an interpretable yet competitive model class, and discusses their role in responsible and explainable AI, including complexity control and constraint-aware learning [11]. In verification and testing contexts, decision trees are particularly useful because each root-to-leaf path corresponds to a human-readable rule over trace features, which can be inspected by engineers and used to justify runtime decisions or to derive guards for transitions in automata-based models [18, 44]. To quantify node purity and feature relevance, decision tree learners rely on several standard measurement criteria such as Gini impurity, information gain, impurity-based (Gini) feature importance, and model-agnostic feature importance measures such as permutation and drop-column importance [27, 36, 37].

2.3.1.1 Gini Impurity

Gini impurity is one of the most widely used node-impurity measures in decision tree learning, in particular in CART-style algorithms [37]. Let t be a node with an empirical class distribution $p_1, \ldots, p_K$ over K classes. The Gini impurity of t is defined as

$$G(t) \;=\; 1 - \sum_{k=1}^{K} p_k^2.$$

The impurity is 0 when all samples at the node belong to a single class, and it is maximised when the classes are uniformly mixed. For a candidate split of node t into child nodes $t_1, \ldots, t_m$, the quality of the split is measured by the impurity decrease

$$\Delta G \;=\; G(t) - \sum_{i=1}^{m} \frac{n_i}{n}\, G(t_i),$$

where n is the number of samples at t and n_i is the number of samples at child t_i. The split with the largest ΔG is selected [37].

2.3.1.2 Information Gain

Information gain is an alternative impurity measure based on Shannon entropy. For a node t with class probabilities $p_1, \ldots, p_K$, the entropy is

$$H(t) \;=\; -\sum_{k=1}^{K} p_k \log_2 p_k.$$

Given a candidate split of t into child nodes $t_1, \ldots, t_m$, the information gain is defined as

$$\mathrm{IG}(t) \;=\; H(t) - \sum_{i=1}^{m} \frac{n_i}{n}\, H(t_i),$$

with n and n_i as above. Intuitively, $\mathrm{IG}(t)$ measures how much uncertainty about the class label is reduced by the split. Classic top–down induction methods such as ID3 and C4.5 rely on entropy-based criteria, and modern surveys compare entropy and Gini-based measures in terms of bias, stability, and computational cost [37].

2.3.1.3 Impurity-Based (Gini) Feature Importance

For a trained decision tree or tree ensemble, impurity-based feature importance provides a simple internal measure of how often and how effectively a feature contributes to reducing impurity [27]. Let n be the total number of training samples, and let n_t be the number of samples reaching node t. For a feature j, its Gini importance is computed as

$$\mathrm{Imp}_{\mathrm{Gini}}(j) \;=\; \sum_{t\,\text{that split on}\,j} \frac{n_t}{n}\Big(G(t) - \sum_{i} \tfrac{n_{t_i}}{n_t} G(t_i)\Big),$$

where $G(\cdot)$ is the Gini impurity and t_i are the children of t. This quantity is often normalised across features to obtain relative importances.

2.3.1.4 Permutation and Drop-Column Feature Importance

Permutation importance is a model-agnostic technique that estimates how much a feature contributes to predictive performance by measuring the degradation in performance when

that feature is randomly permuted [36]. Given a trained classifier and a validation set, one first computes a baseline performance. For each feature j, the values of x_j in the validation set are then randomly permuted, the model is evaluated again, and the decrease in performance is taken as an estimate of the importance of feature j.

Drop-column (or retraining) importance follows a similar idea, but instead of permuting the feature at evaluation time, it removes the feature entirely and retrains the model without it [36]. For each feature j, a new model is trained on $X \setminus \{x_j\}$, and the performance difference with respect to the full model is used as the importance score. This method can provide a stronger notion of importance, as it accounts for the feature during training, but it is considerably more computationally expensive because it requires retraining multiple models.

Permutation and drop-column importance complement impurity-based measures: the former two are model-agnostic and directly tied to predictive performance, while the latter is model-specific and efficient. In practice, all three are often used together when analysing and explaining decision trees and random forests [27, 36].

2.3.2 Random Forest

Random forests are ensemble methods that combine the predictions of many decision trees to improve accuracy and robustness [9]. Each tree is trained on a bootstrap sample of the training set (bagging), and at each split only a random subset of features is considered, which decorrelates the trees and reduces variance. For classification, the random forest typically predicts the class receiving the majority of votes across trees, or averages class

probabilities when probabilistic outputs are needed.

From a statistical viewpoint, random forests are non-parametric models that can approximate complex, nonlinear decision boundaries while retaining many of the interpretability advantages of tree-based methods, such as feature-importance measures and partial dependence plots [9]. They require relatively little hyperparameter tuning and perform well on heterogeneous, high-dimensional data, which has contributed to their wide adoption in machine learning and data mining [9]. In cyber–physical and safety-critical applications, random forests have been used, e.g., to classify executions of autonomous driving controllers with respect to requirement satisfaction [44] and to support learned models of complex software components in conjunction with automata-based techniques [7, 39].

2.4 Simulink Fundamentals

Simulink is a widely used environment for model-based design of cyber–physical systems, particularly in domains such as automotive, aerospace, and industrial control [30, 32, 44]. Models are expressed as executable block diagrams that combine continuous-time plant dynamics with discrete-time control and supervisory logic. These diagrams can be simulated, subjected to systematic testing, or used as a basis for automated code generation for embedded targets [32, 44].

2.4.1 Block-Diagram Modelling

Simulink models are organized as directed graphs of blocks connected by *signals*. Each block represents a computation or component (e.g., gain blocks, integrators, lookup tables, or custom logic), while each signal represents a time-varying value that flows from the output port of one block to the input port of another [30]. Blocks can be hierarchically grouped into subsystems, enabling large models to be decomposed into manageable components that reflect the structure of the underlying controller or plant [32, 44].

During simulation, Simulink evaluates blocks in an order that is consistent with the signal dependencies: at each simulation step, inputs are propagated through the block diagram, intermediate results are computed, and outputs are produced and logged as signals over time [30]. This block-diagram representation is particularly suitable for control and signal-processing applications, where engineers reason in terms of signal flow, control loops, mode logic, and feedback paths. Industrial case studies show that complex automotive and aerospace control software is routinely specified and validated at the Simulink level before implementation [32, 39, 40, 44].

2.4.2 Discrete vs Continuous Execution

Simulink supports both continuous-time and discrete-time dynamics within a single model, which is essential for accurately representing cyber–physical systems. Continuous blocks, such as integrators or transfer functions, are defined by differential equations and are evaluated using ODE-based fixed-step or variable-step solvers over a continuous time base [30]. Discrete blocks, such as zero-order holds, discrete filters, or digital controllers, are eval-

uated at specific sampling instants and maintain internal state that is updated at those instants [30].

The combination of continuous and discrete elements gives rise to hybrid models in which a continuous plant interacts with discrete-time control software [32, 44]. In typical workflows, engineers first construct high-fidelity simulation models that emphasise continuous dynamics and physical realism, and later derive more discretised "code-generation" models whose structure is closer to the final embedded implementation [32, 44]. This distinction between simulation-oriented and code-generation-oriented models is reflected in Simulink's solver configuration and in the use of continuous versus discrete block libraries [30].

The presence of both continuous and discrete behaviours complicates verification and testing, as techniques must account for numerical integration, sampling, and nonlinear dynamics. Nevertheless, recent work has shown that systematic test generation and model checking can be effectively applied to Simulink models with mixed time bases, including models with significant continuous dynamics [32, 44].

A key concept in Simulink is the *sample time* of each block, which determines how frequently it executes and may be continuous, discrete with a fixed period, or inherited from upstream signals [30]. In multi-rate controllers, different components operate at different sample times, leading to a scheduling structure with multiple execution tasks. During simulation, Simulink computes an execution order that respects both data dependencies and sample-time constraints, distinguishing between major steps, where discrete states update, and minor steps, which some solvers use to refine continuous integration [30]. In

fixed-step, purely discrete models, this behaviour closely resembles periodic real-time tasks. Correct sample-time configuration is essential: inconsistent rates or poor discretisation can cause aliasing, missed events, or unrealistic control behaviour, as observed in studies evaluating faults and requirement violations in industrial Simulink models [18, 32, 44].

2.4.3 Autopilot Model

The case study in this thesis is based on a Simulink model of an aircraft autopilot. Such models follow a standard structure in flight-control engineering, separating the continuous aircraft dynamics from the discrete-time control logic and mode management. The plant subsystem encodes the evolution of key state variables, such as altitude and speed, using differential equations integrated by Simulink's continuous-time solvers, while the controller subsystem implements guidance laws, feedback controllers, and mode-switching logic that is executed at one or more discrete sample times [30, 32].

Aircraft autopilot models have been used extensively in the verification and validation literature as benchmarks for temporal-logic verification, model checking, and test generation in a Simulink or state-machine setting [2, 32, 44]. They capture realistic interactions between continuous plant dynamics and discrete supervisory control, including safety-critical requirements such as maintaining altitude and attitude envelopes or respecting actuator limits. Similar industrial-style control models have also been used as targets for automata learning and other model-inference techniques in the broader cyber–physical systems domain [7, 39, 40]. In this case study, we use the Autopilot Demo Simulink model [35] as our reference controller. Figure 2.6 shows its top-level Simulink block diagram.

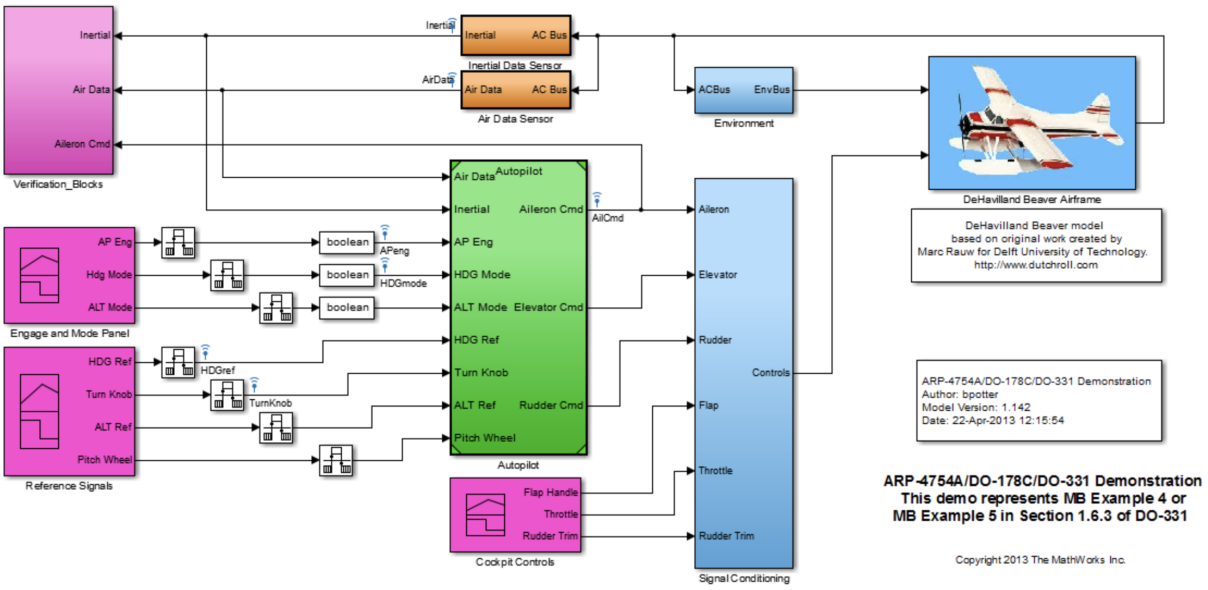


Figure 2.6: Top-level Simulink block diagram of the aircraft autopilot demo used as the case study in this thesis, based on the MathWorks Autopilot Demo model [35].

2.5 Temporal Logic and Query Checking

Temporal logics provide a formal language for specifying how system properties evolve over time and are a standard basis for expressing requirements on reactive and cyber–physical systems [14, 34]. In particular, they are widely used to capture safety, liveness, and timing constraints for software controllers and models of physical processes, and they underpin many model-checking and testing techniques used in practice [14, 32, 44]. This section recalls linear temporal logic (LTL), introduces the idea of query checking as a way to explore families of temporal properties over finite-state models, and discusses vacuous satisfaction and related interpretation issues.

2.5.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a propositional temporal logic interpreted over linear time, i.e., over execution traces or paths of a system [14, 34]. Given a set AP of atomic propositions, the syntax of LTL formulas is defined by

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \mathbf{X}\varphi \mid \varphi_1 \,\mathbf{U}\, \varphi_2,$$

where $p \in AP$, $\mathbf{X}$ is the *next* operator, and $\mathbf{U}$ is the *until* operator. Other temporal operators are defined as abbreviations, for example $\mathbf{F}\varphi := \mathit{true}\,\mathbf{U}\,\varphi$ (eventually) and $\mathbf{G}\varphi := \neg\mathbf{F}\neg\varphi$ (always).

Semantically, LTL formulas are interpreted over infinite paths $\pi = s_0 s_1 s_2 \ldots$ of a Kripke structure or labelled transition system, where each state s_i is labelled with the set of atomic propositions that hold in that state [14,34]. Satisfaction is defined inductively; for example, $(\pi, i) \models \mathbf{X}\varphi$ if and only if $(\pi, i+1) \models \varphi$, and $(\pi, i) \models \varphi_1 \,\mathbf{U}\, \varphi_2$ if and only if there exists $j \geq i$ such that $(\pi, j) \models \varphi_2$ and $(\pi, k) \models \varphi_1$ for all $i \leq k < j$. Typical specifications include safety properties such as

$$\mathbf{G}\,\neg \mathit{error},$$

which asserts that an error condition never occurs, and response properties such as

$$\mathbf{G}\big(\mathit{request} \rightarrow \mathbf{F}\,\mathit{ack}\big),$$

which assert that every request is eventually followed by an acknowledgement [14, 34].

In many practical applications, including testing and runtime analysis, one works with finite execution traces rather than infinite ones. In such cases, either an infinite-word semantics is adopted by implicitly extending finite traces (e.g., by stuttering at the last state), or specialised finite-trace variants such as LTL over finite traces are used. Similar ideas underlie signal-based temporal logics such as Signal Temporal Logic (STL), which extend atomic propositions to predicates over real-valued signals. Industrial studies show that many requirements for cyber–physical models can be captured using a small number of recurring temporal patterns, such as invariance, response, and bounded response over given time scopes [32, 44]. These patterns provide a practical way to structure requirement libraries and to apply temporal logic systematically in verification and testing.

2.5.2 Query Checking over Finite-State Systems

Classical model checking answers the question whether a given temporal-logic formula holds in a given model and, if not, produces counterexample traces that witness violation [14]. Query checking generalises this idea by allowing *queries*, that is, temporal formulas that contain one or more unknowns or placeholders, and by computing all instantiations of these placeholders for which the resulting properties are satisfied by the model. Intuitively, instead of checking a single fixed specification, query checking explores a structured family of temporal properties in a systematic way.

A temporal query typically has the form $Q[x]$, where x is a placeholder that ranges over some domain of candidate subformulas, such as propositional combinations of atomic propositions. Given a finite-state model M and a query $Q[x]$, the goal is to compute all

instances ψ of x such that $M \models Q[\psi]$. For example, one may consider a response pattern

$$\mathbf{G}\big(x \rightarrow \mathbf{F}\, y\big),$$

where x and y stand for unknown state predicates, and then ask for all pairs (x, y) for which the resulting property holds in the model. The resulting instantiations can reveal which events or abstract modes are systematically followed by others and can suggest candidate requirements that are already satisfied.

Algorithmically, many query-checking procedures reduce the problem to repeated calls to standard model checking by treating candidate instantiations as parameters and exploring a search space of substitutions [14]. In data- and trace-driven settings, one can formulate related problems over finite sets of execution traces, where the goal is to mine temporal properties that hold for all observed behaviours. Pattern-based requirements engineering for Simulink and other cyber–physical models, as in recent work on STL specifications and model checking versus testing, can be viewed as a restricted form of such query exploration, in which templates for invariance, response, and timing patterns are instantiated over different signals and thresholds and then evaluated against the model or its traces [6, 32, 44].

Search-based test generation and coverage-driven selection further connect temporal properties with systematic exploration. For instance, temporal logic formulas can be used as objectives when generating simulation-based tests that exercise specific behaviours or push the system towards violation of a requirement, as demonstrated for autonomous-vehicle controllers and other cyber–physical systems [6, 44]. In this view, temporal specifi-

cations act as queries about the existence of behaviours with particular temporal patterns, and solving these queries involves both logical reasoning and numerical search in the space of input scenarios.

2.5.3 Vacuity and Result Interpretation

When interpreting the outcome of model checking, query checking, or temporal-property evaluation over traces, it is important to distinguish meaningful satisfaction from *vacuous* satisfaction. Informally, a specification is satisfied vacuously if it holds for a trivial reason that does not reflect the intent of the requirement [14]. A standard example is the response property

$$\mathbf{G}\big(\mathit{request} \rightarrow \mathbf{F}\, \mathit{ack}\big),$$

which is trivially true in any model where *request* never occurs. In such a case, the property does not provide genuine assurance that requests are eventually acknowledged, because the triggering condition is never activated.

Vacuity can arise in several ways, such as an antecedent of an implication never becoming true, some subformulas being irrelevant to satisfaction, or the system never leaving a restricted portion of its state space [14]. For temporal queries, vacuity may manifest as instantiations that formally satisfy the query but do so in an uninformative way, for example by referring to events that never occur in the model or in the traces under analysis. Detecting such situations is important, because vacuously true properties can give a false sense of correctness.

Existing approaches to vacuity detection typically identify subformulas whose replace-

ment by a weaker or stronger formula does not affect overall satisfaction, or compute alternative valuations of atomic propositions that preserve truth of the specification [14]. In practice, vacuity analysis helps engineers refine their temporal properties, uncover missing assumptions, and focus on specifications that provide informative constraints on system behaviour.

In data-driven and cyber–physical settings, vacuity is closely related to issues of coverage and representativeness of the available executions, as well as to the abstraction used when mapping raw signals to propositions. A property may be satisfied vacuously because certain operating modes are never exercised in the collected traces, or because the abstraction masks behaviours that would falsify the property. Empirical studies on verification, testing, and debugging of Simulink models and other cyber–physical systems highlight the importance of combining temporal-property evaluation with systematic test generation, robustness-oriented search, and explanation techniques in order to avoid misleading conclusions [6,8,23,32,44]. Overall, careful interpretation of temporal-logic and query-checking results, together with explicit vacuity analysis, is essential to obtain trustworthy assurance about system behaviour.

Chapter 3

Related Work

In this chapter, we compare our work with three strands of related research: (1) automata learning and verification for cyber–physical and networked systems, (2) verification and testing of Simulink and cyber–physical models, and (3) supervised abstraction and rule mining for numeric systems.

3.1 Automata learning and verification for cyber–physical systems

Ayoughi et al. [7] propose MELA, a framework that combines supervised abstraction, passive automata learning, and temporal-logic analysis to derive interpretable behavioural models for a network-security case study. They show that, for systems with time-series inputs and outputs, automata learning yields effective, interpretable models that complement

statistical learning: interpretable classifiers such as decision trees are used to obtain predicates over numeric features, while learned Moore machines capture temporal relationships between abstract states and events. Our work follows the same high-level rationale and is an extension of this line of research from network logs to Simulink control models. Instead of network traffic logs, we work with traces generated from a closed-loop plant–controller aircraft autopilot model and study how supervised abstractions and passive learning behave in a continuous-control setting.

Automata learning has also been applied to other cyber–physical and software systems. Smeenk et al. [41] and Schammer et al. [39] use active learning to infer models of embedded control software, while Selvaraj et al. [40] learn automata from autonomous driving software and analyse the resulting models. These studies demonstrate that automata learning can yield useful models of complex control software in cyber–physical settings. Our work adopts a similar preference for interpretable, learned models but targets a Simulink-based aircraft autopilot and relies on passive learning from simulation traces. Moreover, in contrast to approaches that integrate full model checking, we use the learned automata primarily as a basis for query-based analysis of autopilot behaviours.

3.2 Verification and testing of Simulink and cyber–physical models

A substantial body of work investigates verification, testing, and debugging of Simulink and related model-based designs for cyber–physical systems. Nejati et al. [32] introduce

an industrial Simulink benchmark, classify common requirement patterns, formalise them in Signal Temporal Logic (STL), and compare model testing with model checking for finding requirement violations. Tuncali et al. [44] use STL and search-based test generation to exercise autonomous-vehicle controllers, while Gaaloul et al. [18] analyze environment assumptions for software components using machine learning over simulation traces. CPS-Debug [8] supports debugging of Simulink and Stateflow models by combining testing, specification mining, and failure explanation.

These approaches operate directly on Simulink models and their real-valued signals and typically reason about STL properties on continuous-time traces. Our work is complementary: rather than analyzing only signal-level behaviour, we first derive finite-state behavioural models from autopilot traces by applying supervised abstraction and passive automata learning, and then perform query-based analysis over the learned automata. This provides an additional, automata-based view of Simulink controller behaviour that can support specification interpretation and exploration alongside existing STL and simulation-based techniques.

3.3 Supervised abstraction and rule mining for numeric systems

Several recent studies combine supervised learning with testing and analysis to better understand numeric and hybrid systems. Jodat et al. [23, 24] combine machine learning with adaptive random testing to identify test inputs that lead to non-robust or failing behaviours

in industrial control software. Their work uses interpretable models to highlight critical regions of the input space and to explain observed failures. More broadly, research on decision trees, ensembles, and feature-importance methods emphasises the role of interpretable classifiers as a bridge between raw numeric data and higher-level predicates over system behaviour [11, 36].

Our approach relies on supervised abstraction in a similar way: we use interpretable classifiers to partition continuous and categorical signals from the aircraft autopilot model into a finite alphabet of abstract modes that preserve behaviourally relevant distinctions. However, in contrast to work that stops at rule-level explanations or robustness analysis, we use these abstractions as input to passive automata learning and subsequent query checking on the learned models. In this way, supervised rule mining and abstraction form the front end of a pipeline that ultimately yields finite-state behavioural models suitable for structured, query-based reasoning about a Simulink aircraft autopilot.

Chapter 4

Approach

This chapter presents our approach for enhancing automata learning with machine learning in the Simulink case study. Figure 4.1 provides an overview of MELA. The input to the model is a system `S`, denoting the system under learning (SUL). MELA treats `S` as a black box that accepts time-series data as input and generates time-series data as output. The approach begins with `Data Generation` from the Simulink model M to collect input/output time series. We then perform `Trace Creation` to sample these data into traces. Next, we apply `Trace Abstraction`, which includes *Variable Selection* and *Range Abstraction*, to obtain finite-alphabet sequences, and finally run `Automata Learning` to infer a state machine summarizing M's behavior.

4.1 Assumptions and contributions

This chapter presents the workflow used to learn interpretable Moore-machine models from autopilot simulations. Since the workflow is data-driven and depends on discretizing numeric signals, the learned automata should be interpreted as abstractions of observed behaviours under the specific simulation and modelling choices of this study. We therefore summarize the main assumptions and clearly state what is reused from prior work versus what is contributed in this thesis.

4.1.1 Assumptions and scope

We follow the MELA-style pipeline in Figure 4.1, treating the system under learning (SUL) as a black box that maps time-series inputs to time-series outputs. The learned automata summarize the behaviours exercised by the simulation harness and captured by the sampled traces, after abstraction into a finite alphabet.

1. **Segmented inputs and phase structure:** Each run is organised into a small number of equal-duration segments, with piecewise-constant inputs within each segment. This choice is intended to capture phase-like controller behaviour such as early response, correction, and settling, while keeping the resulting traces compact and interpretable.

2. **Fixed discretization of time:** Continuous signals are converted into discrete traces using a fixed sampling rate. This discretization defines what a step means in the

learned automata and in the temporal queries, and it ties the verification results to the selected sampling rate.

3. **Finite-alphabet abstraction:** Passive automata learning operates on symbols rather than continuous values, so numeric signals are discretized into a finite alphabet via supervised variable selection and range partitioning.

4. **Outcome-state definition:** Moore-machine outputs are defined using four categorical outcome states (Certain Fail, Boundary Fail, Boundary Pass, Certain Pass), derived from the relation between the aircraft behaviour and the commanded altitude reference. These outcome labels provide the state meaning used throughout learning and verification.

5. **Passive learning setting:** The models are learned from collected traces without interactive querying of the Simulink model. As a result, the learned model structure and requirement satisfaction results depend on the diversity and representativeness of the available traces.

Passive learning vs. active learning We employ passive automata learning because the autopilot case study is accessed through batch Simulink simulations, where behaviour is observed from offline generated traces. Active learning would require an interactive setup in which the learner repeatedly resets the Simulink model, executes many different input sequences, and treats each simulation run as a query. In our setting, each query corresponds to a complete closed-loop simulation, so adopting active learning would substantially increase both computation time and engineering effort. In addition, small

numerical differences arising from simulation settings and discretization can change the sampled traces, which reduces the repeatability expected by query-based learning. For these reasons, we focus on passive learning from an offline trace set that can be generated once, controlled, and analyzed consistently under a fixed abstraction.

4.1.2 Contributions

Adopted Approach We follow the overall structure of MELA: simulation-based data generation, trace construction, supervised abstraction via variable selection and range partitioning, and passive automata learning to infer an interpretable Moore machine. As in MELA, abstraction is the bridge that makes numeric time-series behaviour suitable for symbolic learning and temporal analysis.

Thesis Extensions We adapt this workflow to a Simulink aircraft autopilot case study and make it phase-aware by structuring each run into equal-duration segments aligned with the input scheduling and the available autopilot signals. We also strengthen the abstraction stage by using feature importance not only to rank variables, but also to guide refinement when scores are close, so we can compare plausible signal combinations and see how abstraction choices affect both the learned model structure and the verification outcomes. Finally, beyond reporting model size and classification accuracy, we evaluate requirement satisfaction from a robustness perspective by checking how consistent the results remain across learning sets and signal selections, supported by statistical comparisons.

Manual Configuration For comparison, we include a manual configuration that keeps data generation and trace construction the same as the MELA-style approach but replaces learning-based abstraction with manually chosen variables and fixed range partitions. This comparison isolates the effect of the learning-based abstraction stage from the rest of the workflow.

4.2 Simulink Model

Simulink is a data-flow-based visual language that can be executed using Matlab and consists of blocks, ports and connections. Blocks typically represent operations and constants, and are tagged with ports that specify how data flow in and out of the blocks. Connections establish data-flows between ports. To simulate a Simulink model M, the simulation engine receives signal inputs defined over a time domain and computes signal outputs at successive time steps over the same time domain used for the inputs. Following the formalization of signal-based test inputs and outputs, we denote a test input for S as $I = (i_1, i_2...i_m)$ and a test output for S as $O = (o_1, o_2...o_n)$ where m is the number of system inputs, n is the number of system outputs, and each i_j and each o_j is a time-series vector for some input and some output of S, respectively. Given a time-series vector $v : [0..T] \rightarrow D$, the time-series range D can be either discrete, i.e., $D \subseteq \mathbb{N}$, or continuous, i.e., $D \subseteq \mathbb{R}$. Discrete time-series ranges can be enumerate such as $D = \{\text{AP-Engagement}\}$, or numeric such as $D = \{\text{Throttle}, \text{Pitch-Wheel}\}$. Automata learning algorithms are not able to abstract and generalize numeric data ranges, whether continuous or discrete. Figure 4.3 illustrates time-series vectors over the time domain $[0..T]$ for the inputs of AUTOPILOT: AP-engagement,

pitch-wheel, and throttle. The AP-engagement input has an enumerated range, whereas the other two inputs are numeric.

We assume that, among the outputs of system S, we use the *current altitude*, which is a continuous signal rather than a pre-defined discrete state. For automata learning, we subsequently abstract this signal into a discrete, enumerable state via range-based categorization relative to a reference (e.g., Certain Pass, Certain Fail), providing feedback on operational status and supporting decision making and control.

The Simulink model used in our study, AUTOPILOT, is a full six-degree-of-freedom simulation of a single-engine, high-wing, propeller-driven airplane with an onboard autopilot. The controller maps time-series pilot commands, engagement status, and setpoints to actions that regulate attitude and airspeed [32]. Figure 4.2 illustrates time-series vectors over the time domain $[0 \ldots 12.5\ \text{s}]$ for the Autopilot inputs: heading hold, altitude hold, heading reference, turn knob, altitude-reference, pitch-wheel, and throttle. The two mode signals (heading hold and altitude hold) are binary, whereas the remaining five inputs vary continuously over time. For context, the figure also plots altitude as the system output, a continuous signal shown alongside the inputs. In our setting we also record AP engagement that is binary and when this signal is 1, the autopilot is engaged, and we take our time window from those engaged segments. The figure includes altitude as the continuous output. We compute the difference between altitude and altitude-reference to obtain a numeric tracking error, which we then discretize into a categorical state. The autopilot model and its requirement set used in this study are taken from an existing industrial Simulink benchmark. In particular, we reuse the Autopilot subject and its provided requirements as reported by Nejati et al. [33]. Our work does not propose new functional requirements; in-

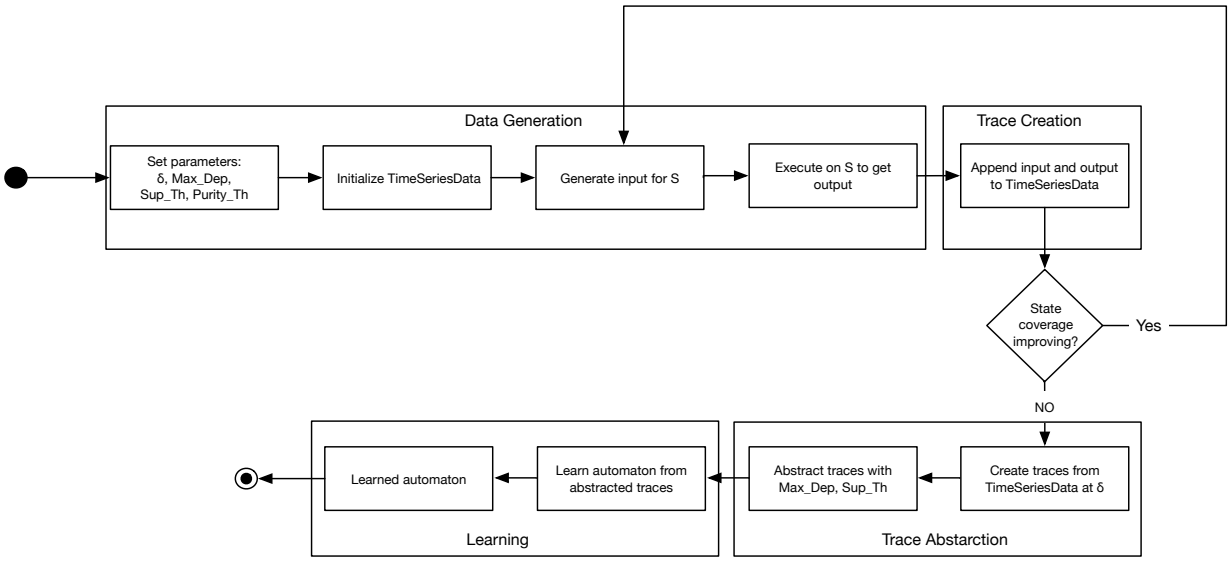


Figure 4.1: ML-enhanced automata learning (MELA) for systems with time-series inputs and outputs.

stead, we transform and analyze this existing requirement set through learned abstractions and Moore-machine models.

4.3 Data Generation

Figure 4.1 begins with a data generation loop, where it iteratively generates test inputs for `S` and executes system `S` to produce test outputs. The purpose of the data generation loop is to produce time-series data to be used for automata learning [23]. In this loop, the test inputs are randomly generated using existing parameterized time-series data generation techniques [6, 44]. In our Simulink model M, each input is piecewise-constant over a small set of equal-duration segments, changing only at segment boundaries. The abstraction preserves this temporal structure, keeping traces time-aware without exposing

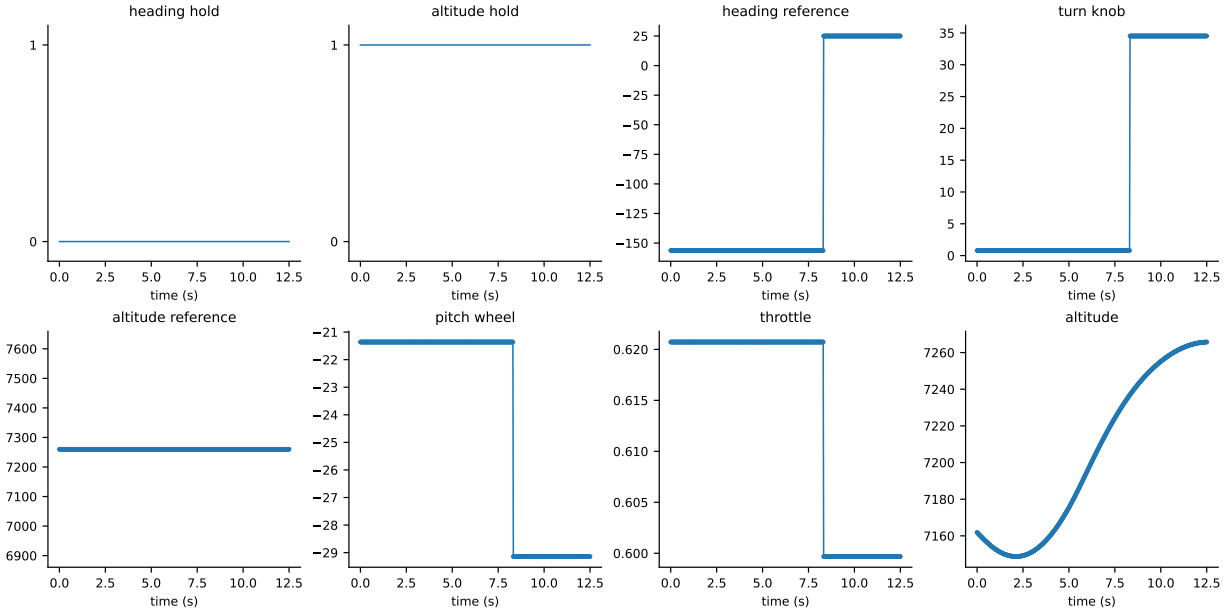


Figure 4.2: A test input consists of time-series vectors over seven autopilot inputs, with altitude as the output.

raw timestamps. To ensure that the learned automata effectively capture the behaviors of SUL, we need to generate test inputs that exercise SUL for different scenarios and yield test outputs that adequately capture SUL's behaviors. To increase the adequacy of the generated data, we employ established black-box test coverage criteria for software testing based on system-state coverage. Specifically, the data generation loop of Figure 4.1 terminates when there is no further improvement in state coverage. This occurs either when the generated outputs cover all system states or when, after several consecutive iterations, the outputs do not cover any new states, indicating that our test generation is unlikely to yield further improvements in state coverage.

4.4 Trace Creation

Figure 4.1 uses the `Trace Creation` routine to convert time-series data vectors into traces to be used for automata learning. Each time-series data vector $v : [0..T] \to D$ is converted into a sequence $v^0, v^1, \ldots, v^k$ of values using the sampling-rate parameter δ, which is an input to Figure 4.1. Specifically, $v^0 = v(0)$, $v^1 = v(\delta), v^2 = v(2 \cdot \delta), \ldots, v^k = v(k \cdot \delta)$, with $k \cdot \delta = T$.

Let $I = (i_1, i_2, \ldots, i_m)$ be a test input of `S`, and $O = (o_1, o_2, \ldots, o_n)$ be a test output of `S`. An input/output trace corresponding to each pair of test input and output of `S` is defined as follows:

$$(i_1^0, \ldots, i_m^0, o_1^0, \ldots, o_n^0),\ (i_1^0, \ldots, i_m^0, i_1^1, \ldots, i_m^1, o_1^1, \ldots, o_n^1),\ \ldots,\ (i_1^0, \ldots, i_m^0, \ldots, i_1^k, \ldots, i_m^k, o_1^k, \ldots, o_n^k)$$

where k is the number of steps with time-step size δ in the time domain $[0..T]$, and for every ℓ, i_ℓ^j is the jth sampled value from input vector i_ℓ, and o_ℓ^j is the jth sampled value from output vector o_ℓ.

For example, Figure 4.3 shows a small excerpt from a trace generated for AUTOPILOT. The values enclosed within "[" and "]" are, respectively, sampled from the following test input vectors of AUTOPILOT: `autopilot engagement`, `turn knob`, `altitude-reference`, `pitch-wheel`, and `throttle`. The last value in each tuple is sampled from the test output vector of AUTOPILOT: `altitude`.

(a) Before trace abstraction

([1, 1, -146.4, 40.1, 7260, -23.2, 0.2], 7160), ([1, 1, -146.4, 40.1, 7260, -23.2, 0.2, 0, 1, 170.4, 5.6, 7260, 2.7, 0.5], 7170), ([1, 1, -146.4, 40.1, 7260, -23.2, 0.2, 0, 1, 170.4, 5.6, 7260, 2.7, 0.5, 0, 0, 135.3, 7.6, 7260, 14.2, 0.7], 7262), ([1, 1, -146.4, 40.1, 7260, -23.2, 0.2, 0, 1, 170.4, 5.6, 7260, 2.7, 0.5, 0, 0, 135.3, 7.6, 7260, 14.2, 0.7, 1, 0, 101.3, 12.3, 7260, 23, 0.9], 7280)

(b) After trace abstraction

([low, low, high], Certain Fail), ([low, low, high, mid, mid, high], Boundary Fail), ([low, low, high, mid, mid, high, high, high, high], Boundary Pass), ([low, low, high, mid, mid, high, high, high, high, high, high, high], Certain Pass)

Figure 4.3: Traces for Autopilot: (a) an example of an actual trace and (b) the same trace after trace abstraction.

4.5 Trace Abstraction

Automata learning approaches assume that traces consist of abstract values only [31]. Hence, raw numerical values should be replaced by categorical or interval-based representations before applying automata learning. To obtain traces consisting of abstract values, we do Trace Abstraction routine in figure Figure 4.1. This routine uses statistical machine learning to refine traces consisting of raw numerical values into a more abstract form. The `Trace Abstraction` routine consists of two steps: first, among all the inputs and outputs of S, we select those that are non-redundant and most correlated with the system state; second, we abstract raw, numeric ranges of the inputs and outputs of S into discrete categories. Below, we describe these two steps; we refer to the first step as *variable selection*

and to the second step as *range abstraction*.

4.5.1 Variable selection

We estimate the importance of each input and output of S for predicting the system state using a random-forest classifier. We first construct a trace table in which each column records the values of a single input or output and the last column records the state. For every variable, we compute three complementary scores: permutation importance, drop-column retraining loss, and impurity-based (Gini) importance [10, 38]. The scores are normalized and combined into a single importance value. We rank these columns based on their importance score and eliminate the lowest-ranked ones (i.e., select the highest-ranked ones). We then refine the traces produced by the Trace Creation step by removing the values related to the eliminated inputs and outputs.

For example, applying variable selection to AUTOPILOT, we discard `heading reference` and `turn knob`, and also drop the binary mode indicators `AP-engagement`, `heading hold`, and `altitude hold`. We retain only `pitch-wheel`, `throttle`, and `altitude-reference`. This choice follows the importance scores, which rank these three controls above the others.

4.5.2 Range Abstraction

We use decision-tree learners to abstract the ranges of numeric inputs and outputs included in our traces after the variable-selection step. For each numeric variable u, we form a two-column table whose first column lists the values of u appearing in the traces and whose

second column lists the corresponding system-state labels. We then train a decision tree with u as the predictor and the system state as the categorical response. The tree's construction is controlled by a stopping criterion (Figure 4.1), which determines the maximum depth to which the tree can grow. In addition, we bound the number of leaves and enforce a minimum leaf-support threshold `Sup_Th` to prune data–sparse partitions. We also apply a leaf–purity gate `Purity_Th` to discard leaves that are insufficiently homogeneous. Each tree leaf represents the following information: (1) the count of samples that are clustered in that leaf (`support`), and (2) the `purity` of the leaf (confidence), which indicates the homogeneity of the samples within the leaf.

Every internal node is linked to its immediate parent node through a condition such as $u \leq c$, where u is the variable and c is a constant in the range of u. Among all the tree leaves, we select those whose number of samples and purity are, respectively, at least the `Sup_Th` and `Purity_Th` thresholds, which are input parameters of Figure 4.1. We consider the conditions $u \leq c$ linking the selected leaves to their immediate parent nodes. Let $c_1 < \cdots < c_k$ denote the distinct split thresholds retained by the tree. We partition the real line into the following intervals: $(-\infty, c_1]$, $(c_1, c_2]$, $\ldots$, $(c_k, +\infty)$. Then, in our traces, we replace the numeric values of u with the categories representing these intervals.

For example, Figure 4.4 shows a decision tree used to abstract the numeric altitude-reference, pitch-wheel, and throttle variables. As shown in Figure 4.3(b), our traces for AUTOPILOT include tuples relating values of altitude-reference, pitch-wheel, and throttle to system–state labels. The decision tree in Figure 4.4 determines, based on these input values and the extracted state labels, how well the inputs predict the system state. For this example, we set `Max_Depth` to 3, cap the tree at six leaves, and use `Sup_Th` as the minimum

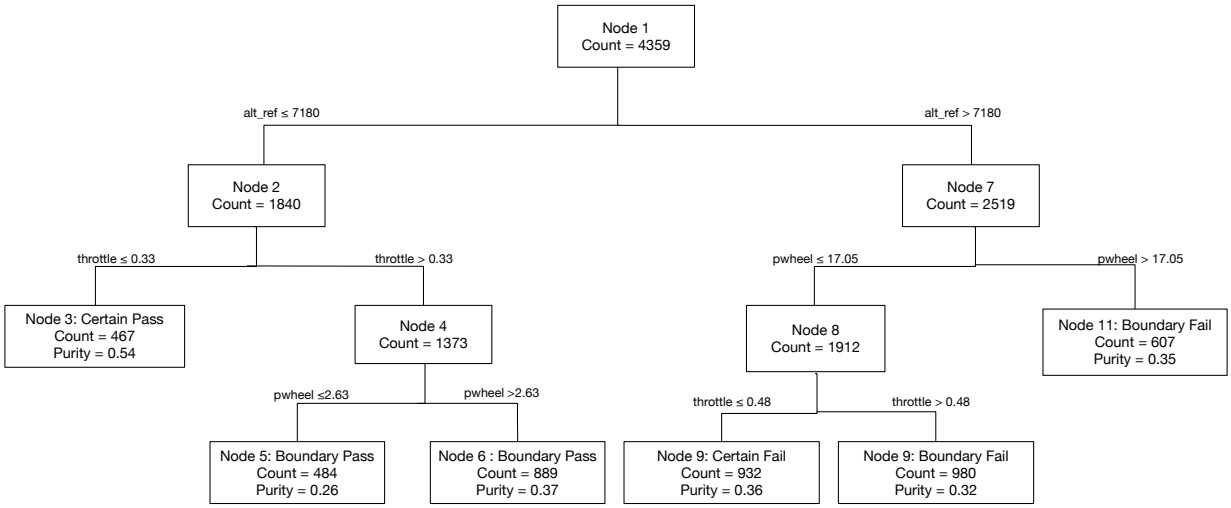


Figure 4.4: Illustrating how a decision tree is used to abstract the decision logic of the autopilot classification model. The numeric ranges of the input attributes altitude-reference, pitch-wheel, throttle are partitioned into categorical regions that determine the output behavior classes: Certain Pass, Certain Fail, Boundary Fail, Boundary Pass. In this abstraction, pitch-wheel and throttle are discretized into [Low, Med, High], while altitude-reference is discretized into [Low, High] only.

leaf support (5% of the training samples) together with a leaf–purity gate `Purity_Th` defined as $1 -$ impurity.

In Figure 4.4, we show the generated tree leaves and their respective number of samples and purity level. We retain only leaves whose sample count and purity both exceed their thresholds. Based on the conditions linking these leaf nodes to their parent nodes, we partition each variable's range into intervals. Concretely, the learned splits induce the following example partitions: for `pitch-wheel`, $(-\infty, 2.63]$, $(2.63, 17.05]$, $(17.05, +\infty)$ "Low", "Med", and "High", for `throttle`, $(-\infty, 0.33]$, $(0.33, 0.48]$, $(0.48, +\infty)$ "Low", "Med", and "High", and for `altitude-reference`, $(-\infty, 7180]$ and $(7180, +\infty)$ "Low" and "High".

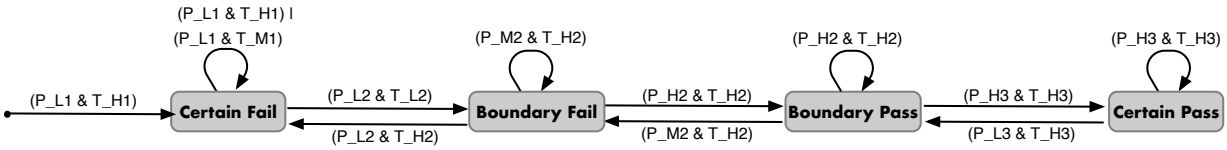


Figure 4.5: A simplified example of a state machine learned for autopilot by our approach(MELA).

4.6 Automata Learning

Figure 4.1 uses the Automata Learning routine to construct an automaton from abstract traces. We realize this routine by passively constructing a Moore machine from the observed data using AALpy [31]. For our case-study system, the Autopilot, we generate Moore machines because each trace step records both the input and the corresponding state. Furthermore, Moore machines map outputs directly to states, making them a suitable representation for Autopilot, as the output of Autopilot reflects the system's state. In the rest of this paper, depending on the context, we interchangeably refer to the outputs of our approach – Moore machines – as either state machines or automata. For example, Figure 4.5 shows the state machine learned for AUTOPILOT, illustrating transitions among *Certain Fail*, *Boundary Fail*, *Boundary Pass*, and *Certain Pass* as a function of pitch-wheel and throttle (Low/Med/High), together with the reference altitude (down/up). Briefly, under an upward reference, sustained high pitch and high power drive the system toward *Certain Pass*; keeping pitch-wheel low tends to remain in *Fail* even with high power; mid-level commands typically yield *Boundary* self-loops. Under a downward reference, the picture mirrors: low pitch-wheel promotes descent toward *Fail*, while increasing pitch-wheel recovers toward *Pass*.

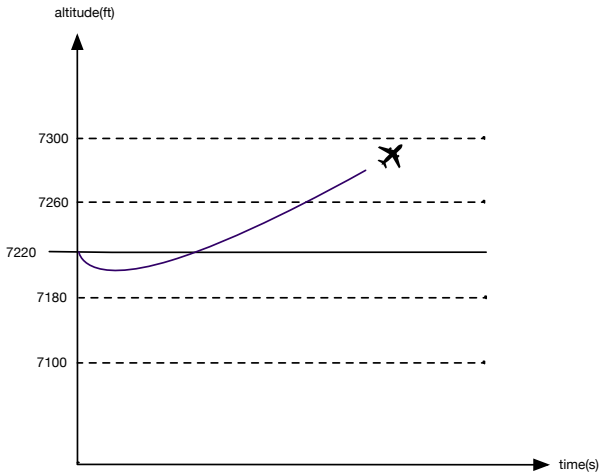


Figure 4.6: Altitude tracking over time in the autopilot example domain.

4.7 Autopilot Domain Example

We motivate our work with a simplified autopilot controller, *Autopilot*, which serves as our running example. Figure 4.5 presents a simplified state machine that shows how the altitude changes when the pitch-wheel and the throttle are at different levels during three successive segments of the run. The initial altitude lies below the reference, indicating a commanded climb.

As noted earlier, we use four categorical outcome states: Certain Fail, Boundary Fail, Boundary Pass, and Certain Pass, and the 25-second run is partitioned into three equal segments. We denote the pitch-wheel by P and the throttle by T, each taking one of three levels $\{L, M, H\}$ indicating low, medium, and high. The segment index is appended in the subscript together with the level. Thus we write, for example, P_{L1}, P_{M2}, T_{H2}, where P_{M2} means "pitch-wheel = medium in segment 2" and T_{H2} means "throttle = high in segment 2".

In the first phase, setting the pitch-wheel low while applying high throttle drives the system directly to Certain Fail. This combination corresponds to a deliberate nose-down command delivered with substantial power, which commits the trajectory to losing altitude despite the climb objective. Once the trajectory has entered Certain Fail during the first phase, any subsequent command that keeps the pitch-wheel low, regardless of whether the throttle is medium or high, maintains Certain Fail.

In the second phase, selecting the low pitch-wheel together with a low throttle places the system in Boundary Fail, meaning the aircraft remains slightly below the altitude reference. From that condition, simply adding power without sufficiently raising the nose is not enough. A medium pitch-wheel with high throttle leaves the system in Boundary Fail, indicating that power alone is insufficient without a larger nose-up command. Breaking out of Fail states requires an increase in attitude combined with power. A high pitch-wheel together with high throttle transitions the system to Boundary Pass, which corresponds to a slight increase above the altitude reference. Once Boundary Pass has been reached in the second phase, repeating the same high pitch-wheel and high throttle command produces a self-loop that holds the aircraft just above the altitude reference. If the autopilot reduces the pitch-wheel back to medium while keeping high power, the system returns to Boundary Fail, illustrating that reducing pitch-wheel removes the small margin and puts the aircraft below the altitude reference again.

In the final phase, maintaining a high pitch-wheel with high throttle leads to Certain Pass, which denotes a stable climb with stable margin above the reference. The diagram also shows that this success is sensitive to sudden attitude changes. From Certain Pass, an sudden move to a low pitch while keeping high throttle immediately pulls the aircraft back

to Boundary Pass. Holding the high pitch-wheel and the high-throttle command maintains Certain Pass through the end of the run. Keeping the high-pitch high-power command preserves Certain Pass until the end of the run.

Chapter 5

Empirical Evaluation

We evaluate MELA by (1) assessing the complexity and accuracy of the generated state machines in representing the behaviours of our case-study system Autopilot and (2) investigating how the learned state machines help with verifying these systems against their requirements and exploring unknown behaviours. Specifically, our evaluation aims to answer the following research questions (RQs):

RQ1 (Complexity and Conformance) How effective is the trace-abstraction component of MELA in reducing the complexity of the generated state machines while maintaining a high level of accuracy? RQ1 assesses how the trace-abstraction component of MELA, impacts the complexity and accuracy of the generated state machines. The primary goal of MELA is to generate state machines that are understandable and abstract, yet highly accurate, ensuring high conformance to the SUL. As a baseline for comparison, we consider an approach that incorporates a trace-abstraction component similar to

MELA, but instead of relying on statistical machine learning, it uses manually defined abstractions based on expert judgment. We refer to this baseline as Manual. Note that in Manual, experts manually abstract numeric value ranges but do not manually create states and transitions; states and transitions are still inferred automatically through automata learning, similar to MELA.

RQ2 (Verification) Do the state machines learned using MELA help determine whether the system meets its requirements and explore its unknown behaviours? We elaborate the system-level requirements of Autopilot into detailed temporal queries. As discussed earlier, the controller's behaviours are not fully known, particularly the conditions under which it transitions among Certain Pass, Boundary Pass, Boundary Fail, and Certain Fail. To explore unknown behaviours, we develop temporal queries, which are queries with placeholders. Temporal queries yield predicates such that, when these predicates replace the placeholder in the query, they form a query that holds over the learned state machine. We report on the evaluation of these temporal queries against the state machines learned in RQ1.

5.1 Implementation

We generate data using a closed-loop Simulink model of an aircraft autopilot. Runs are sampled at a fixed rate, with inputs scheduled as piecewise-constant signals over a small number of equal-duration segments. The AP-engagement is configured so that, once the autopilot is turned on, it stays engaged continuously for a fixed duration of the run [23]. We select multiple altitude references that are either higher or lower than the initial altitude at

which the aircraft starts. The initial altitude is chosen above ground level, which allows us to test both climb and descent behavior within the same setup. We derive discrete outcome states from a continuous fitness signal defined relative to each run's altitude reference . For each reference, we find its maximum and minimum observed fitness values and set two small, reference-specific thresholds as percentages of those extremes. Because the extremes typically differ, these thresholds define asymmetric acceptance margins around the initial reference. Each sample is then assigned, relative to its reference, to one of four outcomes: Certain Fail, Boundary Fail, Boundary Pass, or Certain Pass. Using these labels, we assess input relevance with a random-forest classifier. pitch-wheel, throttle, and altitude reference consistently provide the greatest predictive power. To keep the alphabet compact and the learned model tractable, we retain only these signals for abstraction.

5.2 RQ1: Complexity and Conformance

We discuss the experiment design and the results obtained for RQ1.

5.2.1 Baseline

In this thesis, the term *baseline* refers to an internal reference configuration used to compare against MELA. It is not an established baseline from the literature. The goal is to isolate the contribution of *learning-based abstraction*: the baseline keeps the same data generation, trace creation, automata learning, and verification steps as MELA, but replaces the learning-based abstraction with manually chosen variables and fixed range partitions.

We refer to this configuration as **Manual**.

Manual follows the same trace-collection workflow as MELA, so states and transitions are still inferred automatically from traces (they are not hand-designed). The difference is only in the abstraction stage. Manual applies a two-step abstraction without machine learning: (i) a fixed set of autopilot variables is selected in advance as most relevant for tracking, and (ii) each numeric variable is discretized using predefined thresholds to obtain a finite alphabet.

In our experiments, continuous control inputs are discretized into three equal-width ranges. For example, throttle in $[0, 1]$ is partitioned into $[0, 0.33)$, $[0.33, 0.66)$, and $[0.66, 1]$, which we label as low, mid, and high. Figure 5.1 illustrates the Manual baseline, showing that only the abstraction choices are manual while the rest of the pipeline matches MELA.

5.2.2 Experiment Design

To compare MELA with Manual, we use the same traces for both approaches by applying the data generation and trace creation steps over the testbed described in Section 4.1. We refer to the generated trace sets, which are used to learn automata, as learning sets. We use time-series vectors with time domain $[0..25]sec$ for the input generation routine, and the sampling rate $\delta = 0.025sec$ to convert the time-series data vectors into traces by the trace creation routine, yielding 1000 samples per run. The choice of sampling rate is driven by the need to capture small transients without making the dataset too large. If the rate is too low, fast transitions and state changes may be missed. If the sampling rate is too high, many samples are almost identical, which makes the traces unnecessarily long and leads to

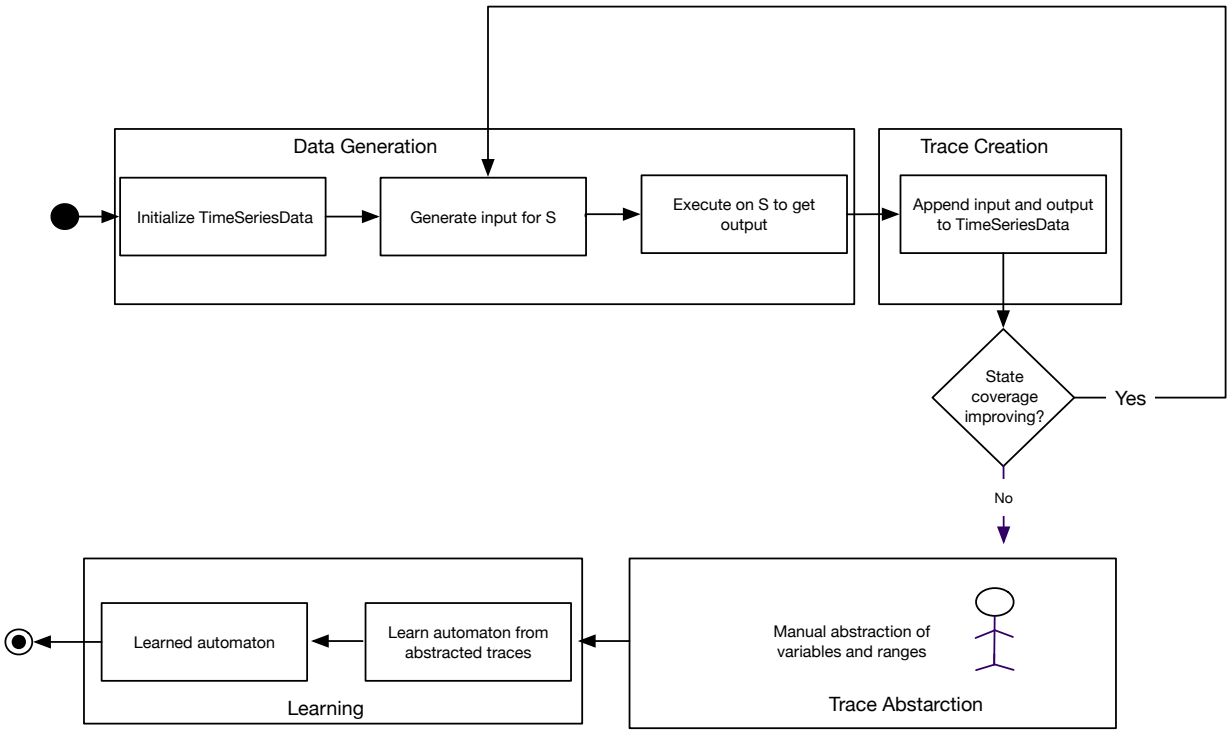


Figure 5.1: MANUAL approach for systems with time-series inputs and outputs.

long self-loops in the learned model without adding useful behavior. The 25-second run is partitioned into three equal segments; inputs remain constant within a segment to allow the aircraft to settle before the next change.

Increasing the run duration increases the number of sampled time steps per run and therefore produces longer traces. Longer traces can improve behavioural coverage by giving the controller more time to exhibit delayed progress, such as the early climb phase when tracking highly above altitude references. This may reduce vacuous query instances and expose late occurring state progressions that do not appear within a shorter window. However, longer runs also include more steady-state behaviour, which increases repeated

symbols and self-loops. This can enlarge the learned automata and increase learning and verification cost without necessarily introducing new behavioural modes. We fixed the run length to 25 s to balance behavioural coverage with tractable trace length and model size. Longer runs would likely benefit scenarios with slower convergence, but should be paired with sampling or segmentation adjustments, or with trace summarization, to avoid unnecessary model growth.

We begin at the initial altitude and define four commanded altitude levels with respect to it: highly below, slightly below, slightly above, and highly above. For each altitude reference, we executed 2,000 closed-loop trials. We run closed-loop simulations in Simulink at each of the four altitude references, using a single altitude reference per run, and record the resulting traces. We then combine the traces into four pairings, each linking one level below the initial altitude with one level above; each pairing serves as a learning set. In our experiments, the initial altitude is 7220 ft. We use 7100 ft as highly below the initial altitude, 7180 ft as slightly below, 7260 ft as slightly above, and 7300 ft as highly above. These choices define four learning sets: $\mathrm{LS}_1 = (7180\,\mathrm{ft}, 7260\,\mathrm{ft})$, $\mathrm{LS}_2 = (7100\,\mathrm{ft}, 7260\,\mathrm{ft})$, $\mathrm{LS}_3 = (7100\,\mathrm{ft}, 7300\,\mathrm{ft})$, and $\mathrm{LS}_4 = (7180\,\mathrm{ft}, 7300\,\mathrm{ft})$. Pairing a lower and a higher altitude reference ensures that each learning set includes both descent and climb, providing the variation needed for the model to distinguish these cases and to analyze how Autopilot input levels vary with the chosen altitude reference.

For the trace-abstraction step of MELA, we perform *variable selection* and *range abstraction*. For variable selection, we rank AUTOPILOT input variables (e.g.,heading hold, altitude hold, pitch-wheel, throttle, etc.) using two complementary importance measures, computed over multiple random seeds relative to the system state: permutation impor-

tance and drop-column retraining loss. Across these measures, *throttle* and *pitch-wheel* consistently rank among the most informative, with *altitude reference* also scoring highly. Accordingly, our evaluation considers six alternative cases focused on the top-ranked variables: (i) single best by permutation (*pitch-wheel*); (ii) next single best alternative (*throttle*); (iii) pairwise controls jointly strong across both measures (*pitch-wheel*, *throttle*); (iv) pairwise mixed (primary control with altitude reference) (*pitch-wheel*, *altitude-reference*); (v) pairwise mixed (alternative control with altitude reference); and (vi) the top-3 set (*pitch-wheel*, *throttle*, *altitude-reference*). Let P denote pitch-wheel, T throttle, and A altitude-reference; then $PT = (P, T)$, $PA = (P, A)$, $TA = (T, A)$, and $PTA = (P, T, A)$. For range abstraction, we use a decision tree to convert numeric ranges into enumerated categories. We set the tree's maximum depth to three (Max_Depth=3), cap the tree at six leaves, and enforce a minimum leaf-support threshold (`Sup_Th`=5%) together with a leaf-purity gate (`Purity_Th`=0.10), with purity defined as $1-$impurity. These settings help avoid overfitting and prevent overly fine partitions of the numeric ranges. Among trees satisfying the structural, support, purity, and split-pattern constraints, we select the best models by held-out accuracy on the test split and report the top configurations. Details of the trace-abstraction component of MELA for our experiments appear in Table 5.1(b).

For the Manual baseline, we specify variable selection and range abstraction manually. Specifically, for variable selection, we select three variables that, in our judgment, most significantly impact the state of system. The variables selected in this way match those identified by our automated approach. For range abstraction, we divide the range of each numeric variable into three equal intervals, labeled Low, Med, and High. The information related to the trace abstraction for the MANUAL baseline is shown in Table 5.1(c). Finally,

Table 5.1: Parameters for our experiments: (a) trace-generation settings used by MELA and the MANUAL baseline; (b) parameters of the trace-abstraction step in MELA; and (c) information about trace abstraction in the MANUAL baseline.

a. Trace Generation for MELA and MANUAL

Altitude-reference pair	Runs	Run length (s)	Sampling rate δ (s)	Segments per run
LS_1	2000	25	0.025	3
LS_2	2000	25	0.025	3
LS_3	2000	25	0.025	3
LS_4	2000	25	0.025	3

b. Trace Abstraction for MELA

Variable selection (permutation importance and drop-column loss)	Range abstraction (decision tree)
`Permutation best`: pitch-wheel	
`Single alternative`: throttle	`Max_Depth`: 3
`Controls pair`: pitch-wheel and throttle	`Max_Leaf_Nodes`: 6
`Mixed pair (primary + altitude reference)`: pitch-wheel and altitude-reference	`Sup_Th`: 5%
`Mixed pair (alternative + altitude reference)`: throttle and altitude-reference	`Purity_Th`: 0.10
`Top-3`: pitch-wheel, throttle, and altitude-reference	

c. Trace Abstraction for the MANUAL Baseline

Manually Variables selected	Manually Range abstraction
`Permutation best`: pitch-wheel	
`Single alternative`: throttle	Let d be the max range of the numeric variable:
`Controls pair`: pitch-wheel and throttle	`Low`: $[0 \dots 0.33 \times d]$
`Mixed pair (primary + altitude reference)`: pitch-wheel and altitude-reference	`Med`: $[0.33 \times d \dots 0.66 \times d]$
`Mixed pair (alternative + altitude reference)`: throttle and altitude-reference	`High`: $[0.66 \times d \dots d]$
`Top-3`: pitch-wheel, throttle, and altitude-reference	

for both MELA and Manual, we use AALpy's passive, prefix-based procedure to infer a Moore machine from the abstract traces.

5.2.3 Metrics

To assess the complexity of the generated state machines, we report the number of states, the number of transitions, and the size of the input alphabet. These three size-based metrics are commonly used in the literature to evaluate the complexity of state machines [19]. To measure the accuracy of the learned models, we follow the established practice in the automata learning literature [3] and compare the models against the ground truth, i.e.,

traces generated by the system's testbed in our context. We note that the learned automata in passive learning are only as good as the traces in the learning sets. Since the learning sets might be incomplete, there could be an accuracy gap between the behaviours of the learned automata and those of the actual system. To measure the accuracy of the learned models, we follow established practice in the automata learning literature and compare them against ground truth, i.e., traces generated by our Autopilot testbed. We note that the learned automata in passive learning are only as good as the traces in the learning sets. Since the learning sets might be incomplete, there could be an accuracy gap between the behaviours of the learned automata and those of the actual system.

To measure the accuracy of the learned automata, we evaluate them on test traces drawn from datasets of about 2000 traces per learning set. For each learning set, we select one trace per distinct input–output transition observed within that set, ensuring coverage of all transition types. Test sets are constructed from held-out data: for each learning set, we sample 20 additional traces uniformly at random, without replacement and excluding any learning traces, from the remainder of that set's dataset, yielding 80 test traces across four representative learning sets. For each learning set, we report the number of test traces accepted by its learned automaton. This protocol emphasizes coverage of distinct transitions while maintaining a strict separation between training and evaluation data.

5.2.4 Results

To answer RQ1, we apply MELA and the Manual baseline to the learning sets for four representative learning sets, and compare the outcomes using the complexity and accu-

racy metrics in Section 5.2.3. Table 2 reports, for each learning set and configuration, the number of states, number of transitions, and alphabet size of the learned automata. Concretely, we evaluate both MELA and the Manual baseline under the six abstraction cases introduced earlier: (i) single best by permutation, (ii) next-best single alternative, (iii) Top-2 controls, (iv) mixed pair—primary control with altitude reference , (v) mixed pair—alternative control with altitude reference , and (vi) Top-3. In total, we obtain 24 automata with MELA and 24 with the Manual baseline, one for each combination of the four learning set and the six abstraction cases. Figure 5.1 shows the accuracy results for the learned automata by MELA and Manual. The accuracy values are computed by evaluating, for each learning set and configuration, the corresponding automaton on that learning set's 20 test traces described in Section 5.2.3, which results in 80 test traces in total across the four learning sets. The plots in the figure show the accuracy distribution of each learned automaton with respect to these test traces. As indicated by Table 2 and Figure 5.1, all the 24 generated automata by MELA have fewer states and transitions than the corresponding automata generated by Manual. Through the ML-based trace abstraction in MELA, we obtain automata that, on average, have 14.7% fewer states, 8.5% fewer transitions and and 1.35% smaller alphabets than automata derived using Manual based abstraction. The accuracy results in Figure 8 show that MELA not only substantially reduces the size of the learned automata but also results in significantly more accurate automata. On average, the automata learned by MELA are 18.5% more accurate than those learned by Manual.

We compute robustness measures for each test based on results obtained from MELA and Manual. Table 5.3 presents the means and standard deviations computed over 20 test inputs for each learning set executed by both approaches. A Mann–Whitney U test with

a significance level of $\alpha = 0.05$ was conducted to compare the two distributions.

The results indicate that, across all configurations, MELA consistently outperformed the Manual baseline. In 20 out of 24 comparisons, the difference between the methods was statistically significant ($p < 0.05$). Effect sizes were predominantly medium to large, indicating that the improvements are practically meaningful. The advantage of MELA was observed across learning set ranges and configurations and is consistent with the robustness measures computed over the 20 test inputs per learning set.

RQ1: Our approach MELA leads to an average reduction of 11.6% in the number of states and transitions of the learned automata, while improving accuracy by an average of 18.5% compared to using Manual based abstractions for automata learning, and these differences were statistically significant under a Mann–Whitney U test in 20 of 24 comparisons with effect sizes predominantly medium to large.

Table 5.2: Comparing the number of states, number of transitions, and the alphabet size for the state machines learned by MELA versus by the MANUAL baseline.

Learning set	Configuration	MELA			MANUAL		
		# States	# Transitions	|Alphabet|	# States	# Transitions	|Alphabet|
LS_1	P	26	81	7	29	82	9
	T	23	65	8	34	85	9
	PT	37	102	15	57	126	18
	PA	43	106	9	48	106	11
	TA	32	80	10	47	103	11
	PTA	52	123	17	67	137	20
LS_2	P	34	94	9	34	99	9
	T	32	90	9	34	101	9
	PT	61	148	18	66	154	18
	PA	52	124	11	58	133	11
	TA	50	120	11	57	132	11
	PTA	75	167	20	85	178	20
LS_3	P	32	85	9	32	90	8
	T	32	87	9	35	93	9
	PT	50	116	18	55	124	17
	PA	40	95	11	47	106	10
	TA	40	97	11	45	102	11
	PTA	55	122	20	61	129	19
LS_4	P	29	78	9	27	70	7
	T	20	52	7	28	70	8
	PT	35	88	16	41	94	15
	PA	36	88	11	38	81	9
	TA	24	59	9	38	83	10
	PTA	40	95	18	51	104	17

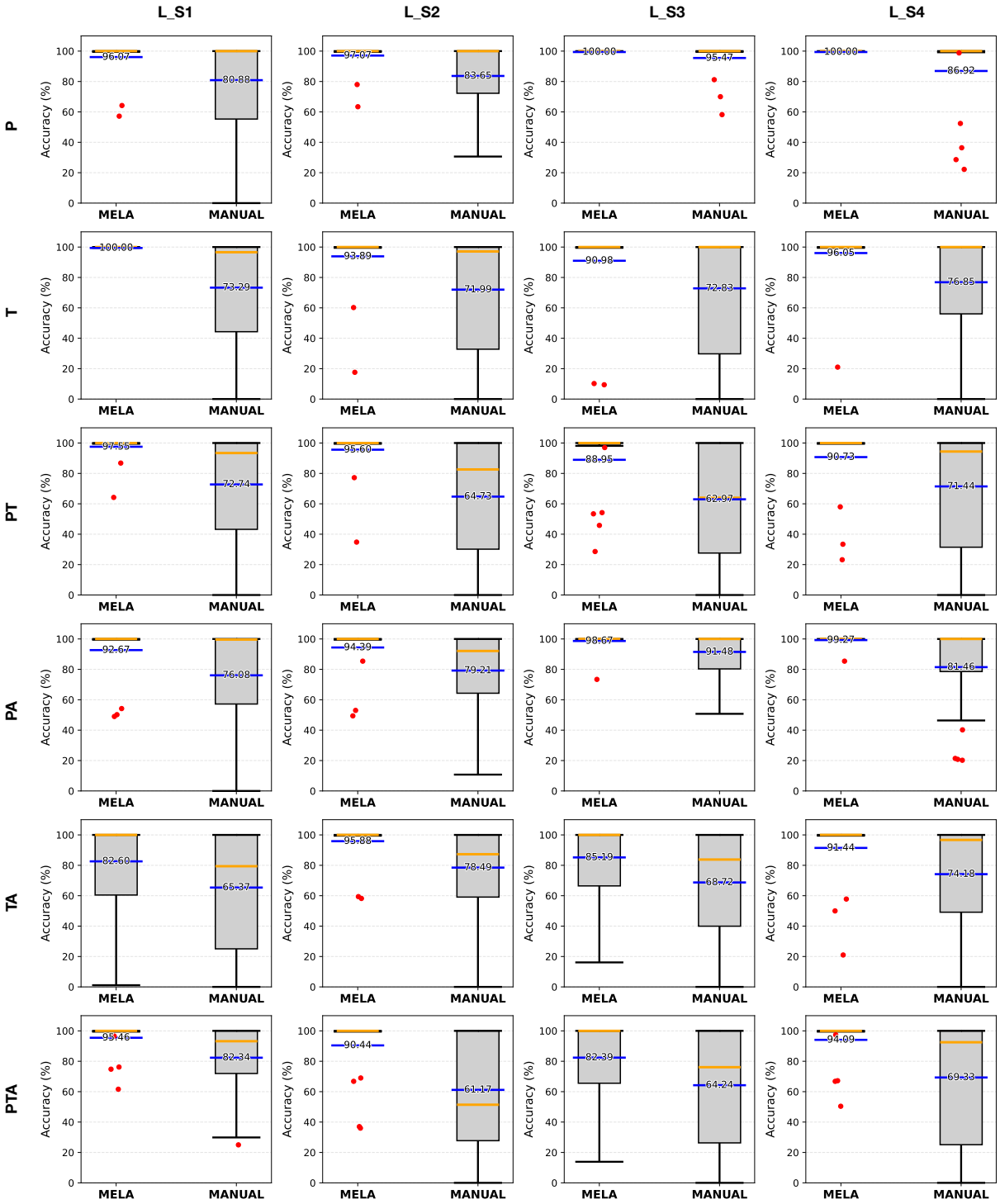


Figure 5.2: Comparing the accuracy of the state machines learned by MELA versus the Manual baseline across four learning sets and six projection configurations (P, T, PT, PA, TA, PTA).

Table 5.3: Manual vs. MELA: means and standard deviations with p-values (Mann–Whitney U) and effect size $\hat{A}_{12}$. Blue p-values indicate cases where MELA significantly outperforms Manual ($\alpha = 0.05$). Effect size magnitude: S (small), M (medium), L (large).

Learning set	Configuration	MELA		Manual		p-value	$\hat{A}_{12}$
		Mean	Standard deviation	Mean	Standard deviation		
LS_1	P	96.07	12.15	80.88	26.65	0.02	0.66 (M)
	T	100.00	0.00	73.29	33.28	0.00	0.75 (L)
	PT	97.55	8.38	72.74	31.94	0.00	0.72 (L)
	PA	92.67	17.92	76.08	33.77	0.03	0.67 (M)
	TA	82.60	28.71	65.37	35.94	0.08	0.65 (M)
	PTA	95.46	10.94	82.34	23.53	0.02	0.68 (M)
LS_2	P	97.07	9.32	83.65	23.56	0.03	0.66 (M)
	T	93.89	20.04	71.99	34.88	0.01	0.69 (M)
	PT	95.60	15.19	64.73	38.87	0.00	0.72 (L)
	PA	94.39	15.14	79.21	25.71	0.02	0.69 (M)
	TA	95.88	12.68	78.49	25.83	0.00	0.74 (L)
	PTA	90.44	20.90	61.17	37.47	0.01	0.72 (L)
LS_3	P	100.00	0.00	95.47	11.68	0.08	0.57 (S)
	T	90.98	27.76	72.83	35.33	0.04	0.65 (M)
	PT	88.95	22.79	62.97	35.79	0.01	0.74 (L)
	PA	98.67	5.95	91.48	13.15	0.02	0.65 (M)
	TA	85.19	24.14	68.72	34.16	0.10	0.64 (M)
	PTA	82.39	28.61	64.24	35.01	0.03	0.69 (M)
LS_4	P	100.00	0.00	86.92	27.18	0.02	0.62 (S)
	T	96.05	17.66	76.85	32.84	0.01	0.67 (M)
	PT	90.73	23.37	71.44	34.52	0.02	0.68 (M)
	PA	99.27	3.26	81.46	30.98	0.02	0.65 (M)
	TA	91.44	21.83	74.18	32.73	0.03	0.67 (M)
	PTA	94.09	14.41	69.33	35.94	0.01	0.71 (L)

5.3 RQ2: Verification

To answer RQ2, we use the state machines learned in RQ1 for the Autopilot case study: for each of the four learning sets (LS_1–LS_4), we verify the corresponding learned state machine under the PTA projection configuration against our temporal queries. We identify two high-level requirements for the system related to reference tracking. In RQ1, we evaluated multiple projection configurations (P, T, PT, PA, TA, PTA) to study how

input selection affects model size and accuracy. For RQ2, we restrict attention to PTA. Our LTL queries explicitly encode the command context via the altitude reference signal, distinguishing climb (A_H) from descent (A_L). Projections that exclude A (i.e., P, T, PT) cannot express this distinction and therefore cannot be used to instantiate the queries derived from R_1 and R_2. Projections that include A but omit one control input such as PA or TA, discard part of the controller action, which weakens the explanatory power of the resulting counterexamples and trace-backed interpretations. The PTA projection preserves both the command context (A) and the two most informative control inputs (P and T), and is therefore the most appropriate configuration for verification in RQ2.

R_1: When a climb command is active, system should progress in stages toward success, moving from Certain Fail to Boundary Fail, from Boundary Fail to Boundary Pass, and from Boundary Pass to Certain Pass.

R_2: When a descent command is active, the system should restore in stages toward failure, moving from Certain Pass to Boundary Pass, from Boundary Pass to Boundary Fail, and from Boundary Fail to Certain Fail.

Transitions in the learned automata are stepwise. Under either command, the next state is either the current state or its immediate neighbor in the commanded direction, and transitions that skip an intermediate state do not occur. We derived temporal queries from R_1 and R_2 listed in the leftmost columns of Tables 5.4 and 5.5. These queries are written in Linear Temporal Logic [6, 14], where "G" is the globally operator and "X" is the next state operator. For succinctness, we use abbreviated names for states: "CF" for Certain Fail, "BF" for Boundary Fail, "BP" for Boundary Pass, and "CP" for Certain Pass.

The temporal queries analyzed in this chapter are based on the Autopilot benchmark requirements [33], which are provided as executable requirement oracles over logged signals such as mode consistency, bounded commands, stability, and reference tracking. In this thesis, we focus exclusively on the altitude reference tracking requirement **R12.1** from the Autopilot benchmark. We do not propose new benchmark requirements. Instead, we use the benchmark oracle for R12.1 and study how its expected behaviours appear in the learned abstractions and Moore machine models. Other benchmark requirements concern behaviours and operating contexts not exercised by our altitude tracking experimental configuration, and are therefore not considered in this evaluation. Our results report both satisfied and violated queries. For presentation, we summarize the expected tracking behaviour into two high level statements, R_1 (climb) and R_2 (descent), and derive the LTL queries in Tables 5.4 and 5.5 from these summaries.

The temporal queries are parameterized by the abstracted inputs shown in Tables 5.4 and 5.5. In particular, we use pitch-wheel P and throttle T with three abstract levels each, denoted by the subscripts L, M, and H, and we use the altitude-reference signal A with two abstract levels, A_L and A_H. Thus P_L, P_M, P_H name the three pitch-wheel levels, T_L, T_M, T_H name the three throttle levels, and A_L, A_H distinguish low and high altitude-reference settings. A setting written as P_H with T_M and A_H means the pitch-wheel is high, the throttle is medium, and the altitude-reference is high for that step. For example, the query $G(? \wedge \text{BF} \Rightarrow X(\text{BP} \vee \text{BF}))$ under $? = P_H T_M A_H$ specifies that when the current abstracted inputs match $P_H T_M A_H$ and the Autopilot is in state BF, the next state must be BP or remain BF. Although pitch-wheel and throttle each have three abstract levels, yielding 9 possible combinations, we report only six settings per requirement. The reason

is that not all pitch levels are meaningful under each command context. Under a climb command (A_H), only P_M and P_H are consistent with upward progression; P_L counteracts the climb direction and is therefore excluded. Under a descent command (A_L), only P_L and P_M are consistent with downward progression; P_H counteracts the descent direction and is therefore excluded. We therefore report only those P–T settings that align with the intended command direction. The queries in Table 5.4(a) and Table 5.5(a), derived from R_1, require the autopilot to move one step in the CF→BF→BP→CP direction on the next transition whenever the altitude-reference is high under A_H. When the system is already in CP and the high altitude-reference setting persists, it is expected to remain in that state. In a dual manner, the queries in Table 5.4(b) and Table 5.5(b), derived from R_2, require one-step movement in the reverse direction, CP→BP→BF→CF, under a low altitude-reference setting under A_L. When the Autopilot is already in CF while the altitude-reference remains low, it is expected to remain in that state.

Table 5.4: Query-checking results under Recurrent Next-Step for the temporal queries derived from system requirements R_1 and R_2.

a Results for temporal queries derived from R1

	Learning set	$? = \mathbf{P}_M\mathbf{T}_M$	$? = \mathbf{P}_H\mathbf{T}_M$	$? = \mathbf{P}_M\mathbf{T}_H$	$? = \mathbf{P}_H\mathbf{T}_H$	$? = \mathbf{P}_H\mathbf{T}_L$	$? = \mathbf{P}_M\mathbf{T}_L$
$G\big(? \wedge \text{UP} \wedge \text{BF} \Rightarrow X(\text{BP} \vee \text{BF})\big)$	LS_1	v	v	✓	✓	✓	v
	LS_2	v	✓	✓	✓	✓	✓
	LS_3	✓	✓	✓	✓	✓	✓
	LS_4	v	v	✓	✓	v	✓
$G\big(? \wedge \text{UP} \wedge \text{BP} \Rightarrow X(\text{CP} \vee \text{BP})\big)$	LS_1	v	v	✓	✓	v	v
	LS_2	v	✓	✓	✓	v	✓
	LS_3	v	✓	v	✓	✓	v
	LS_4	✓	v	✓	v	v	✓
$G\big(? \wedge \text{UP} \wedge \text{CF} \Rightarrow X(\text{BF} \vee \text{CF})\big)$	LS_1	v	v	✓	✓	v	v
	LS_2	✓	v	✓	✓	v	✓
	LS_3	✓	✓	✓	v	v	✓
	LS_4	v	v	✓	v	v	v

b Results for temporal queries derived from R2

	Learning set	$? = \mathbf{P}_L\mathbf{T}_L$	$? = \mathbf{P}_M\mathbf{T}_L$	$? = \mathbf{P}_L\mathbf{T}_M$	$? = \mathbf{P}_M\mathbf{T}_M$	$? = \mathbf{P}_L\mathbf{T}_H$	$? = \mathbf{P}_M\mathbf{T}_H$
$G\big(? \wedge \text{DOWN} \wedge \text{BF} \Rightarrow X(\text{CF} \vee \text{BF})\big)$	LS_1	v	v	✓	v	✓	v
	LS_2	v	v	✓	✓	✓	✓
	LS_3	v	v	✓	✓	✓	v
	LS_4	✓	v	v	v	✓	✓
$G\big(? \wedge \text{DOWN} \wedge \text{BP} \Rightarrow X(\text{BF} \vee \text{BP})\big)$	LS_1	v	v	v	v	✓	v
	LS_2	✓	✓	✓	✓	✓	✓
	LS_3	✓	v	✓	✓	✓	✓
	LS_4	v	v	v	v	✓	✓
$G\big(? \wedge \text{DOWN} \wedge \text{CP} \Rightarrow X(\text{BP} \vee \text{CP})\big)$	LS_1	v	v	v	v	✓	v
	LS_2	✓	✓	✓	v	✓	✓
	LS_3	✓	✓	✓	✓	✓	✓
	LS_4	v	✓	v	v	✓	✓

Table 5.5: Query-checking results under Feedforward Next-Step for the temporal queries derived from system requirements R_1 and R_2.

a Results for temporal queries derived from R1

	Learning set	$? = \mathbf{P}_M\mathbf{T}_M\mathbf{A}_H$	$? = \mathbf{P}_H\mathbf{T}_M\mathbf{A}_H$	$? = \mathbf{P}_M\mathbf{T}_H\mathbf{A}_H$	$? = \mathbf{P}_H\mathbf{T}_H\mathbf{A}_H$	$? = \mathbf{P}_H\mathbf{T}_L\mathbf{A}_H$	$? = \mathbf{P}_M\mathbf{T}_L\mathbf{A}_H$
$G(? \wedge \text{CF} \Rightarrow X(\text{BF}))$	LS_1	v	v	✓	✓	v	v
	LS_2	✗	v	✓	✓	v	✓
	LS_3	✓	✗	✓	v	v	✓
	LS_4	v	v	✗	v	v	v
$G(? \wedge \text{BF} \Rightarrow X(\text{BP}))$	LS_1	v	v	✓	✓	✗	v
	LS_2	v	✓	✓	✓	✗	✗
	LS_3	✗	✗	✗	✓	✓	✗
	LS_4	v	v	✓	✗	v	✗
$G(? \wedge \text{BP} \Rightarrow X(\text{CP}))$	LS_1	v	v	✗	✓	v	v
	LS_2	v	✗	✗	✓	v	✓
	LS_3	v	✗	v	✗	✗	v
	LS_4	✗	v	✗	v	v	✗

b Results for temporal queries derived from R2

	Learning set	$? = \mathbf{P}_L\mathbf{T}_L\mathbf{A}_L$	$? = \mathbf{P}_M\mathbf{T}_L\mathbf{A}_L$	$? = \mathbf{P}_L\mathbf{T}_M\mathbf{A}_L$	$? = \mathbf{P}_M\mathbf{T}_M\mathbf{A}_L$	$? = \mathbf{P}_L\mathbf{T}_H\mathbf{A}_L$	$? = \mathbf{P}_M\mathbf{T}_H\mathbf{A}_L$
$G(? \wedge \text{CP} \Rightarrow X(\text{BP}))$	LS_1	v	v	v	v	✓	v
	LS_2	✓	✓	✓	v	✓	✓
	LS_3	✓	✗	✓	✓	✓	✓
	LS_4	v	✗	v	v	✓	v
$G(? \wedge \text{BF} \Rightarrow X(\text{CF}))$	LS_1	v	v	✓	v	✓	v
	LS_2	v	v	✗	✗	✓	✗
	LS_3	v	v	✓	✗	✓	v
	LS_4	✓	v	v	v	✓	✓
$G(? \wedge \text{BP} \Rightarrow X(\text{BF}))$	LS_1	v	v	v	v	✓	v
	LS_2	✗	✗	✓	✓	✓	✓
	LS_3	✗	v	✓	✓	✓	✗
	LS_4	v	v	v	v	✓	✓

5.3.1 Query Checking

Tables 5.4 and 5.5 report results of query checking for all twenty-four learned Autopilot models under two evaluation modes. Table 5.4 evaluates what we call the *Recurrent Next-Step* condition, which allows the system either to remain in its current state or to advance by one step in the commanded direction on the next step. Table 5.5 evaluates the *Feedforward Next-Step* condition, which is stricter and requires an advance by one step in the commanded direction on the next step, without allowing the system to remain in

its current state. Intuitively, the Recurrent Next-Step condition admits self-loops, while the Feedforward Next-Step condition requires immediate progression. When we evaluate a temporal query, we treat "?" in the query as a placeholder for a specific control setting. For each control setting, we substitute that setting for "?" and check whether the resulting property holds. For each model and configuration, a query is recorded as pass (✓) if it holds, fail (✗) if it is violated, and vacuous (v) if the antecedent of the query is never observed in the traces used to build that model. In the vacuous case, the specified starting state under the corresponding command never occurs, so the query cannot be instantiated. The antecedent refers to the part of the query before the $\Rightarrow$ operator. When evaluating the temporal queries, we found that both the control setting and the choice of learning set significantly affected the results.

For example, the query $G(? \wedge \text{Down} \wedge \text{CP} \Rightarrow X(\text{BP} \vee \text{CP}))$ is satisfied under the Recurrent Next-Step condition in all Learning sets where the control input uses low pitch-wheel and high throttle. Under this condition, when the system is in Certain Pass and a descent command is active, the next step either remains in Certain Pass or moves one step down to Boundary Pass, and both outcomes satisfy the query. The corresponding Feedforward Next-Step query $G(? \wedge \text{Down} \wedge \text{CP} \Rightarrow X(\text{BP}))$ is also satisfied for the same control configuration. In this stricter form, only a one step decrease from Certain Pass to Boundary Pass on the next step is permitted; remaining in Certain Pass is not permitted. By contrast, when pitch-wheel is low and throttle is only medium, the Feedforward Next-Step query fails in one of the learning sets. This shows that medium throttle is not always sufficient to produce the immediate step down from Certain Pass to Boundary Pass under descent. High throttle is required to obtain that immediate decrease consistently across

all learning sets when pitch-wheel is low. When the query $G(? \wedge \text{Down} \wedge \text{CP} \Rightarrow X(\text{BP}))$ is reported as vacuous under the Feedforward Next-Step condition for the setting with low pitch-wheel and medium throttle, this means that the antecedent $\text{Down} \wedge \text{CP}$ never appears for that control setting. In other words, in that learning set we never observe the system in Certain Pass under a descent command with low pitch-wheel and medium throttle, so the query cannot be instantiated and is marked as vacuous.

5.3.2 Analysis

The results presented in Tables 5.4 and 5.5 indicate that some temporal queries are violated and several queries are vacuous. Below, we discuss the reasons for these violations and vacuities and assess whether the results obtained from different state machines are consistent. The pass and fail results in Tables 5.4 and 5.5 show a consistent pattern in how the learned Autopilot models respond under climb and descent commands. Under descent, when pitch-wheel is low and throttle is high, the Autopilot reliably moves downward through the states in the expected order: from Certain Pass toward Boundary Pass, then toward Boundary Fail and Certain Fail. Under these settings, both the Recurrent Next-Step queries (which allow remaining in the same state or moving one step) and the Feedforward Next-Step queries (which require moving one step) are satisfied across all learning sets. This indicates that the controller follows a structured, monotonic descent behavior under low pitch and high throttle.

Do violations (✗) imply defects? The pass and fail results in Tables 5.4 and 5.5 show a consistent pattern in how the learned system models respond under climb and descent

commands. As noted earlier, under descent, the model moves monotonically through the states without violations and satisfies both the Recurrent and Feedforward Next-Step properties across all learning sets.

For climb, high pitch and high throttle drive upward progression from Certain Fail toward Boundary Fail, then toward Boundary Pass and Certain Pass. Under this setting, all Recurrent Next-Step queries are satisfied. In these queries, each entry is either reported as pass or as vacuous. A vacuous result indicates that the relevant starting condition, such as being in Boundary Fail while a climb command is active, never appears in the traces. When the starting condition does appear, the learned automata either remains in the current state or advances exactly one level upward on the next observed step. This is consistent with the intended staged climb behaviour.

Some Feedforward Next-Step queries are reported as failures in the learning sets LS_3 and LS_4. These learning sets correspond to cases where the commanded target altitude is highly above the initial altitude. A failure under Feedforward Next-Step means that, although a climb command is active and the aircraft is in a lower state such as Boundary Fail, the very next recorded step does not immediately advance to the next higher state such as Boundary Pass. Instead, the model is still in the same state at that sampling point, even though the trajectory is already improving in the intended direction.

This behaviour follows directly from how those learning sets were constructed. In these learning sets, many traces begin in an early segment of the run, while the aircraft is still highly below the requested altitude and has only just begun to climb. Reaching a substantially higher altitude reference is not instantaneous. Even under high pitch and high

throttle, the controller must first stop any initial loss of altitude and then accumulate climb authority over time. By contrast, descent responds almost immediately, which explains why the Feedforward Next-Step checks are satisfied under low pitch and high throttle in the descent setting. The Feedforward Next-Step failures for climb therefore do not indicate that the controller skips required intermediate states or behaves incorrectly. They indicate that, for highly above altitude references, an immediate one level improvement at the next sampled step is physically unattainable at the start of the run. With additional time under the same control setting, the model continues to progress upward as required. To ensure correct interpretation, each reported violation was examined by tracing the corresponding counterexample path back to the original simulation logs. This allowed us to determine whether the violation reflected a genuine requirement issue in the underlying system or a limitation of the abstraction and sampling scheme. In the observed cases, the reported failures were consistent with sampling granularity and physical response delay rather than incorrect controller behaviour.

The query violations reported in Tables 5.4 and 5.5 do not indicate defects in the learned system models. Rather, they show that the climb requirement (R1) needs to be stated with an explicit allowance for delay when the target altitude is highly above from the initial altitude, since the next state improvement may require additional time to appear even though the progression is correct.

Do vacuities (v) show gaps? The vacuous cases in Tables 5.4 and 5.5 arise from how the learning data were constructed rather than from missing behaviour in the controller. For each learning set, we did not retain every full run. Instead, we sampled a

representative subset of traces to cover the distinct observed transitions, with the goal of keeping the learned automata compact. This means that if a particular configuration occurred rarely, it was not necessarily included in the final learning set for that model. If a given combination of configuration and starting state appeared only rarely, or only under a narrow control setting, it was not guaranteed to be preserved in the final learning set for that automaton. When a query is reported as vacuous, it is because its antecedent assumes one of these unseen situations. The condition simply never occurs in the traces that were selected to train that automaton. This follows from the way traces were sampled. After the main transition patterns were included, the remaining cases were either very rare or appeared only under narrow control settings. Bringing those in would have added marginal behaviours that almost never occur, which would increase the size of the learned automata without giving a meaningful gain in how well they describe the system. Thus, vacuity in our results reflects controlled trace selection for learnability and interpretability, not an absence of the underlying behaviour in the system.

The observed vacuities are not due to missing behaviour in the controller. They arise from how the training data were formed. The learning sets were built from randomly sampled traces rather than from all possible runs. As a result, some configurations and the corresponding query antecedents at those starting conditions never appear, while the remaining cases are observed only rarely or under narrow control settings. The vacuities in Tables 5.4 and 5.5 therefore reflect this trace selection and not a behavioural gap in the system.

5.4 Threats to Validity and Limitations

As with any empirical study, our results are subject to threats to validity. In this section, we discuss internal validity, external validity, and practical limitations that point to opportunities for future work.

5.4.1 Internal Validity

We took several steps to reduce the influence of uncontrolled factors in our evaluation. First, all traces were generated using a closed-loop Simulink model of the aircraft autopilot, with inputs drawn at random rather than manually configured. This reduces operator bias and ensures that the input-generation procedure is reproducible. Second, the simulation environment is deterministic given fixed inputs. Apart from the randomized selection of input levels and altitude references, the plant and controller behaviour are deterministic, which helps attribute differences in learned automata to abstraction and learning choices rather than stochastic effects. Third, for each learning set we produced a large pool of runs and then sampled traces from that pool. This mitigates the impact of atypical runs and reduces incidental variation in early climb or descent behaviour.

Despite these precautions, several choices may still affect the learned models and verification outcomes. In particular, defining the outcome state categories, selecting which signals to log, choosing the sampling rate and segmentation strategy, and formulating temporal queries can influence both model structure and query results. To reduce this risk, we relied on iterative inspection of simulation logs during refinement of the abstraction

and queries, and we repeated experiments across multiple learning sets and projection configurations.

5.4.2 External Validity

Our evaluation is based on one Simulink model in a specific closed-loop control scenario. The results therefore reflect that model structure and the operating conditions exercised by our simulation campaign. While the overall pipeline is applicable to other settings where time-series logs can be collected from simulation or testing, we have not yet evaluated MELA on additional Simulink controllers or on industrial case studies. Broader generalization would require repeating the study on different models, with varying architectures and operating regimes, and may require domain knowledge to define meaningful state categories and requirements.

5.4.3 Limitations

The first limitation is that the learned Moore machines are derived from a finite set of sampled simulation traces under a fixed discretization. Consequently, the models should not be interpreted as complete representations of the underlying continuous system.

The second limitation concerns the interpretation of next-step properties under delayed control response. In early climb phases, an immediate improvement at the next sampled step may be unrealistic when the target altitude is far from the current altitude. Since the tracking requirement studied here does not impose an explicit time bound, next-step

formulations can be overly strict. Future work can address this by using bounded or time-aware variants of the properties.

The third limitation is that abstraction can sometimes create conflicts in the data, even though the Simulink simulation is deterministic for fixed inputs. In particular, two simulation runs can produce the same abstract trace, but then lead to different next states. Because the learner expects a single next observation for each trace, we must resolve these conflicts before learning, which can affect which transitions appear in the learned state machine. As future work, we will study whether using deeper decision trees to create finer partitions reduces these conflicts and how this changes model size and model checking.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we studied how automata learning can be used to obtain interpretable finite-state behavioural models of a Simulink aircraft autopilot from time-series simulation data.

Below, we reflect on the lessons learned from this work. To begin with, for systems with time-series inputs and outputs, automata learning produces effective, interpretable behavioural models. While interpretable statistical learning is effective at deriving static abstractions from data, it is not as effective at capturing temporal behaviours encoded in time-series data. Interpretable ML methods, such as decision trees and decision rules, are effective in identifying predicates that explain the relationship between system signals and abstract states or modes. However, they are inadequate in capturing temporal relationships between states and in understanding how changes in system signals might trigger changes

of state. Our research shows that: (1) automata learning, due to its ability to capture temporal behaviours effectively, proves useful for mitigating the shortcomings of statistical learning, and (2) to overcome the limitations of automata learning in abstracting data, one can increase the level of abstraction in traces first before attempting to learn automata.

In addition, this work proves that Automata learning is useful for analyzing and gaining a better understanding of Simulink-based control systems such as an aircraft autopilot. Our work highlights some important contextual factors related to the construction of behavioural models for these systems, notably the numeric and time-series nature of their inputs and outputs, the cost of running large numbers of simulations, and the limited amenability to active learning techniques due to the difficulty of building efficient query-and response loops. An important lesson learned from our work is the feasibility of automata learning for Simulink controllers through an explicit treatment of time-series numeric data and the use of passive learning.

This work targets engineers who verify and analyze a system under learning treated as a black box using input/output observations. In this thesis, the SUL is a closed-loop Simulink autopilot model, but the same idea applies whenever time-series logs can be collected from simulation or testing. The learned state machines provide a compact behavioural summary that can support requirement checking and the investigation of unexpected behaviours by linking verification outcomes back to concrete logged runs.

Increasing the simulation length (run duration) directly increases the number of sampled time steps per run and therefore produces longer traces. Longer traces can improve behavioural coverage by allowing the closed-loop controller more time to react, which is

particularly relevant for delayed responses (e.g., early climb phases under highly above reference targets). As a result, longer simulations may reveal additional state progressions that are not observable within a shorter window, potentially reducing vacuous query instances and improving the ability of learned models to capture late-occurring transitions. However, longer runs also increase the amount of redundant steady-state behaviour (repeated symbols), which can amplify self-loops and introduce additional transition occurrences that may enlarge the learned automata and increase learning and verification cost without necessarily adding new behavioural modes. In this thesis, we fix the run length to 25 s to balance (i) sufficient time for observing reference-tracking progress and (ii) tractable trace length and model size under passive learning. A longer run length is expected to benefit cases where progression requires more time to manifest (especially climb toward far targets), but it should be paired with appropriate sampling/segmentation or trace summarization to avoid unnecessary model growth.

6.2 Future Work

While the proposed framework shows promise on the considered autopilot model, there are several directions for future work. A next step is to apply the same pipeline to other Simulink control models to assess how well the learned automata and our query-based analyses generalize. It would also be useful to refine the abstraction and learning settings based on feedback from additional case studies and to explore lightweight tool support that makes the approach easier to use in everyday modelling workflows.

References

[1] Fides Aarts, Harco Kuppens, Jan Tretmans, Frits Vaandrager, and Sicco Verwer. Learning and testing the bounded retransmission protocol. In *Proceedings of the 11th International Conference on Grammatical Inference (ICGI 2012)*, volume 21 of *Proceedings of Machine Learning Research*, pages 4–18, College Park, MD, USA, 2012. PMLR.

[2] Mohammed Al Achhab, Ahmed Hammad, and Hassan Mountassir. Verifying ltl properties on hierarchical systems: Application to aircraft autopilot. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 28–35, 2006.

[3] Bernhard K. Aichernig, Edi Muskardin, and Andrea Pferscher. Active vs. passive: A comparison of automata learning paradigms for network protocols. In Matt Luckcuck and Marie Farrell, editors, *Proceedings Fourth International Workshop on Formal Methods for Autonomous Systems (FMAS) and Fourth International Workshop on Automated and verifiable Software sYstem DEvelopment (ASYDE), FMAS/ASYDE@SEFM 2022, and Fourth International Workshop on Automated and*

verifiable Software sYstem DEvelopment (ASYDE)Berlin, Germany, 26th and 27th of September 2022, volume 371 of *EPTCS*, pages 1–19, Open Access Platform, 2022. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[4] Shahbaz Ali, Hailong Sun, and Yongwang Zhao. Model learning: A survey of foundations, tools and applications. *Frontiers of Computer Science*, 15(5):155210, 2021.

[5] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, 1987.

[6] Aitor Arrieta, Shuai Wang, Urtzi Markiegi, Ainhoa Arruabarrena, Leire Etxeberria, and Goiuria Sagardui. Pareto efficient multi-objective black-box test case selection for simulation-based testing. *Information and Software Technology*, 114:137–154, 2019.

[7] Negin Ayoughi, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. Enhancing automata learning with statistical machine learning: A network security case study. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems (MODELS 2024)*, pages 172–182, Linz, Austria, 2024. ACM. Preprint available as CoRR abs/2405.11141.

[8] Ezio Bartocci, Niveditha Manjunath, Leonardo Mariani, Cristinel Mateis, Dejan Ničković, and Fabrizio Pastore. Cpsdebug: a tool for explanation of failures in cyber-physical systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 569–572, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Gérard Biau and Erwan Scornet. A random forest guided tour. *TEST*, 25(2):197–227, 2016.

[10] Kelvyn Bladen and D. Richard Cutler. Assessing agreement between permutation and dropout variable importance methods for regression and random forest models. *Electronic Research Archive*, 32(7):4495–4514, 2024.

[11] Hendrik Blockeel, Laurens Devos, Benoît Frénay, Géraldin Nanfack, and Siegfried Nijssen. Decision trees: from efficient prediction to responsible ai. *Frontiers in Artificial Intelligence*, Volume 6 - 2023, 2023.

[12] Véronique Bruyère, Bharat Garhewal, Guillermo A. Pérez, Gaëtan Staquet, and Frits W. Vaandrager. Active learning of mealy machines with timers. In Pavithra Prabhakar and Andrea Vandin, editors, *Quantitative Evaluation of Systems and Formal Modeling and Analysis of Timed Systems*, pages 42–61, Cham, 2026. Springer Nature Switzerland.

[13] Jair Cervantes, Farid Garcia-Lamont, Lisbeth Rodríguez-Mazahua, and Asdrubal Lopez. A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing*, 408:189–215, 2020.

[14] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron A. Peled, and Helmut Veith. *Model checking, 2nd Edition*. MIT Press, Cambridge, MA, USA, 2018.

[15] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of tls implementations. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, page 193–206, USA, 2015. USENIX Association.

[16] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. Fsm-based conformance testing methods: A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12):1286–1297, 2010.

[17] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 454–471, Cham, 2016. Springer International Publishing.

[18] Khouloud Gaaloul, Claudio Menghi, Shiva Nejati, Lionel C. Briand, and David Wolfe. Mining assumptions for software components using machine learning. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 159–171, New York, NY, USA, 2020. ACM.

[19] Mathew Hall. Complexity metrics for hierarchical state machines. In Myra B. Cohen and Mel Ó Cinnéide, editors, *Search Based Software Engineering - Third International Symposium, SSBSE 2011, Szeged, Hungary, September 10-12, 2011. Proceedings*, volume 6956 of *Lecture Notes in Computer Science*, pages 76–81, Berlin, Germany, 2011. Springer.

[20] Markus Holzer and Martin Kutrib. Descriptional and computational complexity of finite automata—a survey. *Inf. Comput.*, 209(3):456–470, March 2011.

[21] Falk Howar and Bernhard Steffen. *Active Automata Learning in Practice*, pages 123–148. Springer International Publishing, Cham, 2018.

[22] Nemanja Hranisavljevic, Tom Westermann, Swantje Plambeck, Henrik Sebastian Steude, Gesa Benndorf, and Oliver Niggemann. A model learning perspective on the complexity of cyber-physical systems. In *Machine Learning for Cyber Physical Systems: Proceedings of the Conference ML4CPS 2025*, pages 59–68. Universitätsbibliothek der HSU/UniBw H, 2025.

[23] Baharin Aliashrafi Jodat, Abhishek Chandar, Shiva Nejati, and Mehrdad Sabetzadeh. Test generation strategies for building failure models and explaining spurious failures. *CoRR*, abs/2312.05631(4):93:1–93:32, 2023.

[24] Baharin Aliashrafi Jodat, Shiva Nejati, Mehrdad Sabetzadeh, and Patricio Saavedra. Learning non-robustness using simulation-based testing: a network traffic-shaping case study. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*, pages 386–397, Piscataway, NJ, USA, 2023. IEEE.

[25] Bernard Lambeau, Christophe Damas, and Pierre Dupont. State-merging dfa induction algorithms with mandatory merge constraints. In Alexander Clark, François Coste, and Laurent Miclet, editors, *Grammatical Inference: Algorithms and Applications*, pages 139–153, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[27] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'13, page 431–439, Red Hook, NY, USA, 2013. Curran Associates Inc.

[28] Maher Maalouf. Logistic regression in data analysis: an overview. *Int. J. Data Anal. Tech. Strateg.*, 3(3):281–299, July 2011.

[29] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016.

[30] MathWorks. *Simulink: Simulation and Model-Based Design*. The MathWorks, Inc., Natick, MA, USA, 2024. Version R2024b. Available at: https://www.mathworks.com/products/simulink.html (accessed November 2025).

[31] Edi Muskardin, Bernhard K. Aichernig, Ingo Pill, Andrea Pferscher, and Martin Tappler. Aalpy: An active automata learning library. In Zhe Hou and Vijay Ganesh, editors, *Automated Technology for Verification and Analysis - 19th International Symposium, ATVA 2021, Gold Coast, QLD, Australia, October 18-22, 2021, Proceedings*, volume 12971 of *Lecture Notes in Computer Science*, pages 67–73, Berlin, Germany, 2021. Springer.

[32] Shiva Nejati, Khouloud Gaaloul, Claudio Menghi, Lionel C. Briand, Stephen Foster, and David Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *Proceedings of the 2019 27th ACM Joint*

Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2019, page 1015–1025, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Shiva Nejati, Molka Gaaloul, Claudio Menghi, Lionel C. Briand, Michael Foster, and Daniel Wolfe. Evaluating model testing and model checking for finding requirements violations in simulink models. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Tallinn, Estonia, 2019. Includes the Simulink benchmark models and their textual requirements (with formalizations) as a replication package.

[34] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, Piscataway, NJ, USA, 1977. IEEE Computer Society.

[35] Bill Potter. Autopilot demo for ARP4754A, DO-178C and DO-331. MATLAB Central File Exchange, 2025. Available at: https://www.mathworks.com/matlabcentral/fileexchange/41490-autopilot-demo-for-arp4754a-do-178c-and-do-331 (accessed November 2025).

[36] Gopi Krishnan Rajbahadur, Shaowei Wang, Gustavo A. Oliva, Yasutaka Kamei, and Ahmed E. Hassan. The impact of feature importance methods on the interpretation of defect classifiers. *IEEE Transactions on Software Engineering*, 48(7):2245–2261, 2022.

[37] L. Rokach and O. Maimon. Top-down induction of decision trees classifiers - a survey. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 35(4):476–487, 2005.

[38] Dominik Rotter, Florian Liebgott, Daniel Kessler, Annika Liebgott, and Bin Yang. Machine learning-based identification of root causes for defective units in manufacturing processes. In *Advances in Automotive Production Technology – Towards Software-Defined Manufacturing and Resilient Supply Chains (Proc. of SCAP 2022)*, ARENA2036, pages 168–178, Cham, 2023. Springer.

[39] Lutz Schammer, Swantje Plambeck, Fin Hendrik Bahnsen, and Goerschwin Fey. Learning models of cyber-physical systems using automata learning. In *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1224–1229. IEEE, 2021.

[40] Yuvaraj Selvaraj, Ashfaq Farooqui, Ghazaleh Panahandeh, Wolfgang Ahrendt, and Martin Fabian. Automatically learning formal models from autonomous driving software. *Electronics*, 11(4):643, 2022.

[41] Wouter Smeenk, Joshua Moerman, Frits Vaandrager, and David N. Jansen. Applying automata learning to embedded control software. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering*, pages 67–83, Cham, 2015. Springer International Publishing.

[42] Adil Soubki and Jeffrey Heinz. Benchmarking state-merging algorithms for learning regular languages. In François Coste, Faissal Ouardi, and Guillaume Rabusseau,

editors, *Proceedings of the 16th International Conference on Grammatical Inference*, volume 217 of *Proceedings of Machine Learning Research*, pages 181–198. PMLR, 10–13 Jul 2023.

[43] Bernhard Steffen, Falk Howar, and Maik Merten. *Introduction to Active Automata Learning from a Practical Perspective*, pages 256–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[44] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles*, 5(2):265–280, 2019.

[45] Frits Vaandrager. Model learning. *Commun. ACM*, 60(2):86–95, January 2017.

[46] Frits Vaandrager, Masoud Ebrahimi, and Roderick Bloem. Learning mealy machines with one timer. *Information and Computation*, 295:105013, 2023. Special Issue: Selected papers of the 15th International Conference on Language and Automata Theory and Applications, LATA 2021.

[47] Masaki Waga and Ichiro Hasuo. Moore-machine filtering for timed and untimed pattern matching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2649–2660, 2018.