Learning Symbolic Abstractions from System Execution Traces

Natasha Yogananda Jeppu

Balliol College

University of Oxford

A dissertation submitted for the degree of

Doctor of Philosophy

Michaelmas Term 2022

Humbly offering this work at the feet of Amma and Swami
Dedicated to Mummy, Daddy and Adi

Abstract

This dissertation shows that symbolic abstractions for a system can be inferred from a set of system execution traces using a combination of Boolean satisfiability and program synthesis. In addition, the degree of completeness of an inferred abstraction can be evaluated by employing equivalence checking using simulation relations, that can further be used to iteratively infer an overapproximating system abstraction with provable completeness guarantees.

The first part of this dissertation presents a novel algorithm to infer a symbolic abstraction for a system as a finite state automaton from system execution traces. Given a set of execution traces the algorithm uses Boolean satisfiability to learn a finite state automaton that accepts (at least) all traces in the set. To learn a symbolic system abstraction over large and possibly infinite alphabets, the algorithm uses program synthesis to consolidate trace information into syntactic expressions that serve as transition predicates in the learned model.

The system behaviours admitted by the inferred abstraction are limited to only those manifest in the set of execution traces. The abstraction may therefore only be a partial model of the system and may not admit all system behaviours. The second part of this dissertation presents a novel procedure to evaluate the degree of completeness for an inferred system abstraction. The structure of the abstraction is used to extract a set of conditions that collectively encode a completeness hypothesis. The hypothesis is formulated such that the satisfaction of the hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system and the abstraction. Further, the existence of a simulation relation is sufficient to guarantee that the inferred system abstraction is overapproximating. In addition, counterexamples to the hypothesis can be used to construct new traces and iteratively learn new abstractions, until the completeness hypothesis is satisfied and an overapproximating system abstraction is obtained.

Research Hypothesis

This dissertation argues the following thesis.

Symbolic abstractions for a system can be inferred from a set of system execution traces using a combination of Boolean satisfiability and program synthesis. In addition, the degree of completeness of an inferred abstraction can be evaluated by employing equivalence checking using simulation relations, that can further be used to iteratively infer an overapproximating system abstraction with provable completeness guarantees.

We validate this claim with the following evidence.

1. We show that given a set of execution traces of a system, an abstraction can be learned for the system as a finite state automaton that accepts (at least) all execution traces in the set. Specifically, we show that the problem of learning a finite state automaton that accepts a given set of execution traces can be formulated as a Boolean satisfiability problem.

2. We show that symbolic system abstractions over large and possibly infinite alphabets can be learned by inferring transition predicates from traces using program synthesis. We report experiments that demonstrate the approach using execution traces from a variety of sources.

3. We show that the degree of completeness of an inferred system abstraction can be evaluated by checking the truth value of a completeness hypothesis that is formulated by extracting a set of completeness conditions using the structure of the inferred abstraction. We formally prove that the satisfaction of the hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system and the abstraction, and further that the existence of a simulation relation is sufficient to guarantee that the inferred abstraction is an overapproximation.

4. We show that counterexamples to the hypothesis can be used to construct new traces. By means of experiments we show that these traces can be used to iteratively learn new abstractions, until the completeness hypothesis is satisfied and an overapproximating abstraction is obtained.

Publications

The majority of the work presented in this dissertation has been published in the following papers.

1. N. Y. Jeppu, T. Melham, D. Kroening and J. O'Leary, "Learning Concise Models from Long Execution Traces," 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1-6, doi: 10.1109/DAC18072.2020.9218613.

   I designed, developed and implemented the presented algorithm. I also set up and ran experiments to evaluate the algorithm, and interpreted the experimental results. I contributed to the preparation and review of the manuscript in collaboration with my co-authors.

2. N. Y. Jeppu, T. Melham and D. Kroening, "Active Learning of Abstract System Models from Traces using Model Checking," 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE), 2022, pp. 100-103, doi: 10.23919/DATE54114.2022.9774595.

   I designed, developed and implemented the presented algorithm. I also set up and ran experiments to evaluate the algorithm, and interpreted the experimental results. I contributed to the preparation and review of the manuscript in collaboration with my co-authors.

3. N. Y. Jeppu, T. Melham and D. Kroening, "Enhancing Active Model Learning with Equivalence Checking using Simulation Relations"

   This paper is an extended version of publication 2 above and is currently under review in the Formal Methods in System Design (FMSD) journal. My contributions to this work are as described under publication 2.

During the course of my PhD, I also contributed to the following publication outside of the scope of this dissertation.

4. M. Hasanbeig, N. Yogananda Jeppu, A. Abate, T. Melham, and D. Kroening, “DeepSynth: Automata Synthesis for Automatic Task Segmentation in Deep Reinforcement Learning,” AAAI, vol. 35, no. 9, pp. 7647-7656, May 2021.

   This work integrates deep reinforcement learning with model-learning from traces to enable learning in sparse reward settings. My contribution to this work is the model learning algorithm described in publication 1. I participated in implementing the algorithm, and contributed to the preparation and review of the manuscript in collaboration with my co-authors.

Acknowledgments

My journey into verification commenced in 2014 during my undergraduate days with a small experiment to verify if my implementation of a priority logic met the specified requirements. That little experiment ignited in me an interest for the field, and has led me to this moment where I submit a DPhil dissertation in the field. This journey has been fun and exciting with its highs and lows, but it would not have been possible without the support of people whom I would like to sincerely acknowledge here.

I would like to thank my supervisors, Professor Tom Melham and Professor Daniel Kroening, for their continued support and guidance throughout my DPhil studies at the Department of Computer Science, University of Oxford. I am thankful for their advice through these years, and for giving me the confidence to pursue this research.

I am very grateful to the donors of the Jason Hu scholarship, Balliol College and the Semiconductor Research Corporation (SRC) for funding my DPhil studies. I would also like to thank the MCR community at Balliol College for making me feel welcome and for making my graduate life a memorable experience.

I would like to thank my friends and colleagues at Oxford for their constant support and encouragement throughout my DPhil journey. Finally, this dissertation would not have been possible without the unconditional love and support of my parents and my brother, who have always been by my side through the highs and the lows. Thank you all for making this journey memorable. I have learnt a lot along the way, and hope to continue learning moving forward.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the growing demand for automation across sectors ranging from medicine to avionics, there has been a steady rise in the complexity of modern hardware and software systems. As systems grow more and more complex, it becomes progressively challenging to deliver resilient and error-free systems that correctly serve their intended purpose. The emphasis on system verification and validation has therefore increased over the years to guarantee system correctness across all stages of the systems engineering process, from design to implementation and deployment.

Although conventional verification practices such as simulation and testing have proved valuable in identifying bugs in a system, particularly in the early stages of debugging, they do not guarantee the absence of bugs. This has encouraged research into formal verification techniques [12, 18, 31, 55, 64, 74, 75, 90], especially for the verification of hardware and software systems for safety critical applications, where system failure can lead to loss of life or large-scale damage.

Unlike simulation and testing that explore partial trajectories of a system in search of bugs, formal verification techniques, in principle, perform an exhaustive search of all possible system trajectories to mathematically prove or disprove system correctness. But, these formal guarantees of correctness come at the cost of increased resources and time spent on system verification. This has prompted efforts to devise methods to scale formal verification techniques to enable their adoption in practice.

1.1 Abstract System Models

Abstraction [32, 76, 79] is a popular method that intelligently reduces the search space for formal verification while preserving its ability to guarantee system correctness.

Typically, the goal of system verification is to check if a formal model of the system satisfies a property. Here, a property is a formal specification describing correct

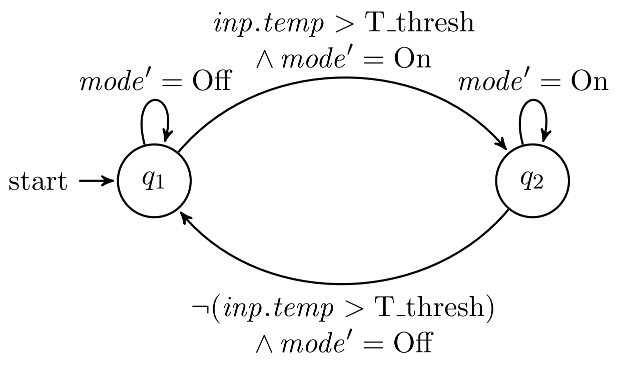

$inp.temp > \text{T\_thresh}$
$\wedge\, mode' = \text{On}$
$mode' = \text{Off}$
$mode' = \text{On}$
start
q_1
q_2
$\neg(inp.temp > \text{T\_thresh})$
$\wedge\, mode' = \text{Off}$

Figure 1.1: Abstraction modelling operation mode switches for a Home Climate-Control Cooler system generated by our algorithm.

system behaviour. An abstract system model is obtained by hiding details of the system that appear irrelevant to the property of interest. To enable verification using abstractions, an overapproximating abstract model is constructed which simulates the original system and is usually much smaller than the system model [31]. As a consequence, it is much easier to verify if the property holds on the abstract model and extrapolate the result to reason about the correctness of the original system.

Conventionally, these abstract models are manually specified [34, 41, 94]. However, modern systems often have complex behaviour with many components—both in hardware and software. The components of a system are often designed by different teams, or even different companies, and integrated only when first silicon or a stable hardware emulation model is available. It is difficult to ascertain in advance how software behaviour is affected by hardware design decisions or how a given hardware IP will fare under a software workload. It is therefore challenging to craft a good abstraction that accurately captures system behaviour of interest.

One way to address this is to have some means of automatically extracting useful information about integrated system behaviour. Execution traces provide an exact representation of the system behaviours that are exercised when an instrumented system implementation runs, and can provide insight into how various components of the system behave as a collective. But raw trace data is inevitably large and unstructured, and this undigested representation is difficult to employ as a high-level view of the system and its requirements. Concise, human-readable models that express high-level system behaviour, such as the model in Fig. 1.1, provide better insight into the working of the system. This has prompted efforts to devise model-learning algorithms to automatically infer system abstractions from traces.

1.2 Learning Abstract System Models from Traces

Model-learning algorithms for inferring system abstractions from traces use automaton inference techniques, such as state-merging or query-based learning, to infer an abstract system model as a finite state automaton that conforms to the traces. Automaton inference from traces is an extensively researched subject dating back to 1960s–70s [16, 47, 48, 87] and many modern model-learning implementations are built on top of this research. A survey of related model-learning algorithms is provided in Chapter 2. Over the years, automaton inference has also been extended to learn extended finite state machines such as register automata, thereby enabling the construction of symbolic system abstractions from traces [8, 15, 17, 25].

It remains largely impractical to apply existing inference techniques to learn abstract system models for system-wide analysis of large and complex systems. There are several reasons for this:

1. A majority of the algorithms require labelled traces—positive and negative traces—to learn concise system models. In the context of learning system abstractions from traces, a positive trace is typically defined as a sequence of observations that can be reproduced by executing the system, while a negative trace cannot be produced by any system execution. While positive traces can be obtained by simply executing an instrumented system implementation, it is a challenge to collect or come by negative traces in practice.

2. Symbolic abstractions generated by existing model-learning implementations are limited in their expressiveness. They cannot generalise and accurately express dependencies and computations involving multiple system observables.

3. Passive model-learning algorithms generate system models from a given set of traces, without a feedback loop for additional trace gathering. As a result, the system behaviours admitted by the generated models are limited to only those manifest in the given traces. Therefore, capturing all system behaviour by the generated system models is conditional on devising a system load that exercises all relevant system behaviours. This can be difficult to achieve in practice, particularly when a system comprises multiple components and it is not obvious how the components will behave as a collective.

4. Active model-learning algorithms can, in principle, generate exact system models. They iteratively refine a hypothesis model by extracting information from

the system or an oracle that has sufficient knowledge of the system, using the hypothesis model as a guide. Conventionally, these algorithms operate by posing membership and equivalence queries to the oracle, and the responses are used for model refinement. When these algorithms are used in practice, however, they suffer from high query complexity, particularly when they are applied to learn symbolic system abstractions. Consequently, many active model-learning implementations are constrained to learning partial models for large systems.

The work presented in this dissertation is motivated by the challenges enumerated above. In this dissertation we describe a methodology to infer concise, accurate and expressive symbolic overapproximations with provable completeness guarantees using only system execution trace data.

To address challenges 1 and 2 enumerated above, the first part of the dissertation presents a novel passive model-learning algorithm to infer a symbolic system abstraction from a given set of system execution traces. The algorithm uses a combination of Boolean satisfiability (SAT) and program synthesis to learn an abstract system model as a finite state automaton that accepts (at least) all execution traces in the set.

We describe a SAT formulation to infer an automaton using only system execution traces or positive traces. Additionally, we integrate algorithmic methods designed to enable SAT-based model-learning to scale to long traces based on trace segmentation and incremental model learning.

We extend SAT-based automaton inference from traces to learn symbolic finite automata over large and possibly infinite alphabets, using program synthesis to consolidate system execution trace data into syntactic expressions that are not explicit in the execution traces. These serve as transition predicates on the automaton edges, thereby generating symbolic system abstractions.

As is the case with passive model-learning, an abstraction generated by the model-learning algorithm from a given set of execution traces is guaranteed to admit only those system behaviours exemplified by the traces in the set. Learned system abstractions may therefore be partial. To address challenges 3 and 4 enumerated above, the second part of the dissertation presents a novel method based on equivalence checking using simulation relations to evaluate the degree of completeness of an abstraction inferred by the model-learning algorithm, and using the information to iteratively learn symbolic overapproximations with provable completeness guarantees.

The structure of an inferred abstraction is used to extract a set of conditions that collectively encode a completeness hypothesis. The hypothesis is formulated based on defining a simulation relation between the system and its learned abstraction, such that the satisfaction of the hypothesis is sufficient to guarantee that the learned abstraction is overapproximating. Additionally, counterexamples to the hypothesis can be used to construct new traces. These traces can further be used to iteratively learn new abstractions, until the completeness hypothesis holds and an overapproximating system abstraction is obtained.

1.3 Contributions

The overall contribution of this dissertation is the development and implementation of a methodology to infer concise, accurate and expressive symbolic system overapproximations with provable completeness guarantees using only system execution trace data. We summarize below the main elements of the contribution.

1. *A novel model-learning algorithm that combines SAT and program synthesis to generate symbolic abstractions for a system from system execution traces.*

   To this end,

   - we describe and implement a SAT formulation for inferring a finite state automaton from a set of available traces, using only system execution trace data or positive traces.
   - we describe and implement a method to consolidate trace information into expressive automaton transition predicates using program synthesis.

   The significance of this approach is that it enables the inference of system abstractions in the absence of labelled traces or additional system information such as temporal properties of the system that are known in advance. Further, it enables the extension of SAT-based automaton inference from traces to learn symbolic abstractions over large and possibly infinite alphabets.

   The model-learning algorithm is described in detail in Chapter 3.

2. *Algorithmic methods for scaling to long traces based on trace segmentation and incremental learning.*

   By means of experiments, we show that these algorithmic optimizations significantly reduce algorithm runtime. Details of the experimental evaluation, including benchmarks, experimental setup and algorithm implementation are described in Chapter 4.

3. *An approach to evaluate the degree of completeness of an abstraction learned from traces based on equivalence checking using simulation relations.*

   We show that the approach can further be used to iteratively learn an overapproximating system abstraction with provable completeness guarantees.

   To this end,

   - we describe the formulation of a completeness hypothesis encoded as a set of completeness conditions that are extracted using the structure of the learned abstraction.
   - we formally prove that the satisfaction of the formulated hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system and the learned abstraction, and further that the existence of a simulation relation is sufficient to guarantee that the abstraction is overapproximating.
   - we implement an active model-learning procedure that combines model-learning from traces with equivalence checking using simulation relations to iteratively learn new abstractions, until the completeness hypothesis holds and an overapproximating abstraction is obtained.

   The significance of this approach is that it enables the generation of overapproximating symbolic system abstractions with provable completeness guarantees. Further, generated abstractions are more expressive than abstractions generated by existing active model-learning implementations.

   The equivalence checking procedure is described in Chapter 5. Details of the active model-learning approach, including an experimental evaluation is provided in Chapter 6.

Chapter 2

Background

In this chapter we present background on related automaton inference algorithms and program synthesis. We also introduce formal preliminaries on equivalence relations between formal models of systems.

2.1 Automaton Inference

Existing algorithms to generate abstract system models typically use automaton inference techniques. We classify these algorithms into three broad categories based on the automaton inference algorithm used: state-merging, query-based learning, and machine learning algorithms that use recurrent neural network structures. These algorithms are discussed in detail in the following sections. We compare our work with these algorithms in Chapters 4 and 6.

2.1.1 State-Merging

State-merging is a prominent approach for learning automata from trace data. State-merge algorithms begin by constructing a Prefix Tree Acceptor (PTA) from the given learning sample, i.e., the trace data. As illustrated in Fig 2.1, a PTA is the smallest tree-like automaton that is strongly consistent with the learning sample, with the automaton states representing all prefixes of the sample.

Subsequently, equivalent state pairs in the PTA are identified and then merged to generate a compact representation. State equivalence is determined using a variety of model inference techniques. The simplest among them is the k-Tails algorithm [16]. Here, two states are said to be *k-equivalent* if they have the same set of strings of length k or shorter that correspond to valid paths in the PTA beginning from that state. The algorithm terminates when there are no k-equivalent states left.

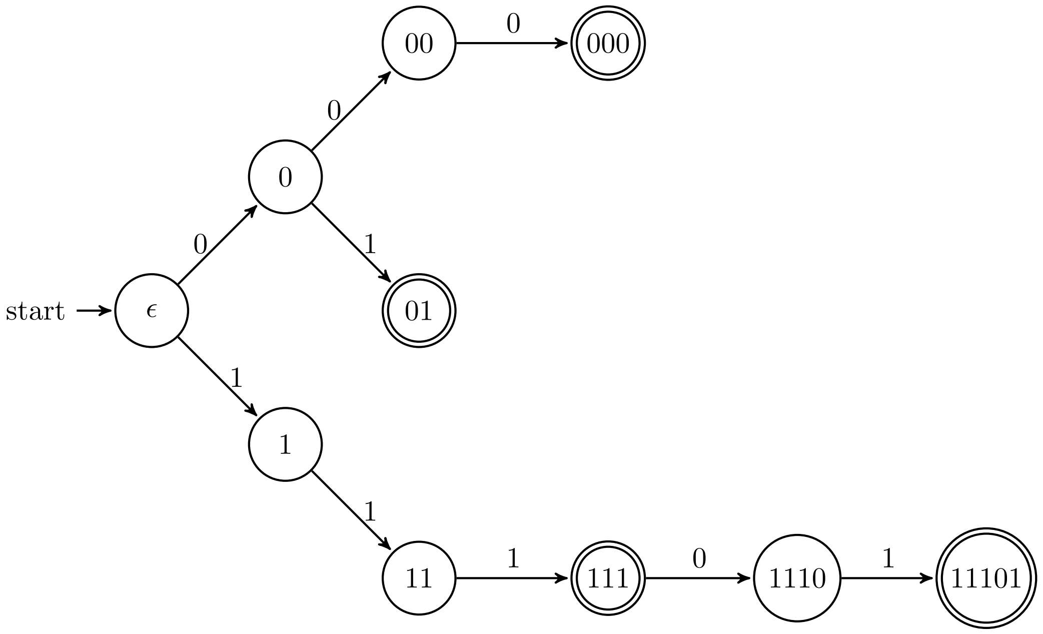


Figure 2.1: PTA for the positive trace sample $\{111, 000, 11101, 01\}$ [89]

The generated models improve in precision and increase in number of states for increasing values of k. One extension of the k-Tails approach [115] uses data classifiers to determine state equivalence; in addition to k-equivalence, states are merged only if the classifier predicts the same next event for these states.

Conventional state-merge algorithms are partial because they fail to model how system variables change during execution. One extension of the state-merge algorithm [114] generates "computational" state machines, where data update functions over transitions are automatically inferred using genetic programming. First, a transition system is inferred from a set of traces. It is then simulated to record variable changes that occur as a result of a transition, which are used as input to genetic programming. The GkTails [80] algorithm integrates Daikon [39] with the k-Tails approach to derive transition guards for Extended Finite State Machines (EFSM).

The k-Tails algorithm and many of its successors generate models using only positive traces, and hence run the risk of overgeneralising. A popular model inference algorithm, Evidence Driven State Merging (EDSM) [77], overcomes this problem by using both positive and negative traces to determine equivalence of states to be merged. The Regular Positive and Negative Inference (RPNI) algorithm [89] also uses labelled data and generates a Deterministic Finite Automaton (DFA) consistent with the data. Moreover, the algorithm can identify any regular language in the limit.

In an extension of the EDSM algorithm [56], finite automaton inference is mapped to a graph-colouring problem based on the red-blue EDSM framework [77]. Models are generated by converting the problem to Boolean satisfiability and using state-of-the-art SAT solvers to get an optimal solution. The approach generates a minimal automaton that is consistent with a set of positive and negative trace samples; the automaton accepts all positive and rejects all negative traces.

An extension of the SAT-based algorithm [57] starts with a few iterations of EDSM, followed by SAT solving once the problem is small enough to be efficiently solved. Another extension [10] learns models incrementally to ensure algorithm scalability for long traces. Variants of the SAT-based approach [106, 107] are used to learn EFSMs from a set of test *scenarios*. Here, a test scenario corresponds to a sequence of triples comprising an event, a transition guard, and a sequence of output actions.

The EDSM algorithm was further extended to incorporate inherent temporal behaviour in the models [106, 111]. Here, the state-merge algorithm is adapted to accept temporal constraints in the form of Linear Temporal Logic (LTL) system specifications, in addition to labelled trace data. Models are checked against LTL properties to validate state merges as they are encountered. Similarly, the work in [106] uses a combination of test scenarios and LTL properties to generate exact EFSMs with SAT. In [21, 22], the requirement of labelled trace data is relaxed to generate plant models from only positive traces, while LTL properties are used to refine the generated model. These approaches actively generate system models where additional negative traces are obtained as a result of checking LTL properties against a hypothesis model. These are used to iteratively refine the generated abstraction.

Other active versions of the state-merge algorithm [37, 112] use manual feedback in the form of responses to queries generated from a hypothesis model that are classified as positive or negative by an end user.

2.1.2 Query-based Automata Learning

Angluin's L* algorithm [7] forms the basis for a majority of query-based automaton inference algorithms. It learns a DFA for an unknown regular language L over a known finite alphabet A. The method assumes the presence of a Minimally Adequate Teacher (MAT) or an oracle that answers two types of queries regarding L: membership and equivalence queries, the results of which are used to generate and refine the DFA. Membership queries are used to determine whether a given string defined over A belongs to the language L. Equivalence queries are used to check if the language defined by the candidate DFA is equal to L. When the DFA's language is not equal to L, the query returns a counterexample in the symmetric difference of the two languages. This is used to refine the generated DFA.

The responses to the queries are organised into structures called observations tables that consist of three components: a non-empty finite prefix-closed set of strings S, a non-empty finite suffix-closed set of strings E and a function T mapping $(S \cup S \cdot A) \cdot E$ to $\{0, 1\}$. The function T is interpreted as follows:

$$T(u) = \begin{cases} 1, & \text{if}\, u \in L \\ 0, & \text{if}\, u \notin L \end{cases}$$

The observation table, denoted (S, E, T), can be represented as a two-dimensional array, as illustrated in Fig. 2.2a, where the rows are labelled with elements of $(S \cup S \cdot A)$ and the columns are labelled with elements of E. The entry in row s and column e corresponds to $T(s \cdot e)$.

Starting with $S = E = \{\epsilon\}$, the L* algorithm augments the observation table with the responses to membership queries, until the table is *closed* and *consistent*. The table is said to be closed if for each element $t \in S \cdot A$, there exists an element $s \in S$ such that $\text{row}(s) = \text{row}(t)$. Here, $\text{row}(s)$ represents a function $f \colon E \to \{0, 1\}$ defined by $f(e) = T(s \cdot e)$. The table is said to be consistent provided that whenever there are elements $s_1, s_2 \in S$ such that $\text{row}(s_1) = \text{row}(s_2)$, for all $a \in A$, $\text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a)$.

When the table (S, E, T) is closed and consistent, the algorithm constructs a DFA (Q, q_0, F, δ, A) from the table over the alphabet A, with state set Q, initial state q_0, accepting states F and transition function δ as follows:

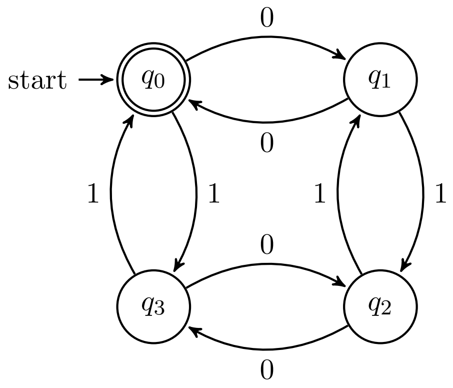
$$\begin{aligned} Q &= \{\text{row}(s) \colon s \in S\} \\ q_0 &= \text{row}(\epsilon) \\ F &= \{\text{row}(s) \colon s \in S \wedge T(s) = 1\} \\ \delta(\text{row}(s), a) &= \text{row}(s \cdot a) \end{aligned}$$

An example of the above construction is provided in Fig. 2.2. Once a candidate DFA is constructed equivalence queries are used to determine if the constructed DFA represents L. The responses are used to further augment the observation table and subsequently refine the automaton, until the generated DFA exactly represents L.

The algorithm was later improved by Rivest and Schapire [92], and many applications of L* now use this improved version. The TTT algorithm [66] is an improvement over L* that achieves optimal space complexity, enabling its application to runtime verification problems. The L* algorithm has also been extended to learn nominal

T	ϵ	0	1
ϵ	1	0	0
0	0	1	0
1	0	0	1
11	1	0	0
01	0	0	0
011	0	1	0
00	1	0	0
10	0	0	0
110	0	1	0
111	0	0	1
010	0	0	1
0110	1	0	0
0111	0	0	0

(a) Observation Table



(b) Constructed DFA

Figure 2.2: DFA constructed using L* over the alphabet $A = \{0, 1\}$ from the observation table (S, E, T) with $S = \{\epsilon, 0, 1, 11, 01, 011\}$ and $E = \{\epsilon, 0, 1\}$ [7].

non-deterministic automata for languages over infinite structured alphabets [85] and Mealy machines [97].

L* and many of its variants have been used to learn system abstractions for a variety of verification applications [26, 27, 33, 51, 63, 91, 103]. In one such application, model checking and model-based testing are used in combination with L* in a collaborative framework [51, 91, 103]. Here, model checking and model-based testing serve as teaching aids for L*. A set of system properties are checked against the generated model and the result is used as feedback for learning. The models learnt using L* in turn provide system abstractions that help scale and guide the verification process.

For compositional verification of large systems, L* has been used to iteratively infer assumptions for assume-guarantee reasoning [26, 33]. In [27], L* is used to learn a DFA describing scenarios under which a program error occurs. One of the main challenges to employing L*-style algorithms in practice is the large number of queries. The approach in [27] therefore employs an optimisation using lazy learning to reduce the number of membership queries. A similar optimisation is discussed in [63], where domain-specific rules are used to filter membership queries.

In all these applications either a complete or a partial model of the system is available, such that membership queries posed by the L* framework can be answered by "querying" the system itself. Also, the alphabet over which the automaton is learned is a set of Boolean events that is known a priori. These may correspond to actual system events, such as function calls, or a known high level abstraction of underlying system behaviour. The absence of an oracle and the inability to ascertain the exact alphabet of system internals beforehand, such as all values that system variables take during execution, often makes it difficult to apply L*-style algorithms for learning symbolic abstractions for complex systems.

Sigma* [17] addresses this by extending the L* algorithm to learn symbolic models of software defined over large alphabet sets. Dynamic symbolic execution is used to find constraints on inputs and expressions generating output to build a symbolic alphabet. In addition, instead of employing an equivalence check between the generated model and system, as in L*, it iteratively learns an overapproximation in parallel with the conjectures learned by L*, and returns a system model when the conjecture equals the overapproximation.

The MAT* algorithm [8] is another extension that generates symbolic models as symbolic finite automata. The transitions of the generated automata carry predicates over a Boolean algebra that is efficiently learnable using membership and equivalence queries, such as the equality algebra. The algorithm takes as input a membership oracle, an equivalence oracle, and a learning algorithm for the Boolean algebra used in the target symbolic automaton.

Query-based learning has also been extended to learn a special class of EFSMs called register automata [25]. Register automata extend finite automata with registers that can store values, and transition guards that compare data parameters to registers. The algorithm uses Symbolic Decision Trees (SDTs) to symbolically represent relations between concrete data values. Additionally, the SDTs also represent how these relations can be used to determine if a sequence of concrete data values is accepting or rejecting, thereby allowing the observation tables of L* to be extended to symbolic models. The algorithm uses individual oracles, called tree oracles, for each type of operation represented in the symbolic model.

2.1.3 Automata Learning with Neural Architectures

Neural networks, particularly Recurrent Neural Networks (RNNs), provide a mechanism to learn sequential information and make predictions about future data. The relationship between RNNs and automata learning has been previously explored [50, 88, 105, 116]. The foundation of this relationship stems from the fact that the internal state of a network trained to predict future events given an input event sequence can be used as a representation of the state of the automaton to be learned.

One of the more recent approaches [50] uses RNNs to generate DFAs for a predefined language using a training data set consisting of pairs (w, ans), where w is a word defined over an alphabet Σ, and *ans* is 1 if the word belongs to the language and 0 if not. During network operation, the computed values from the layer of neurons preceding the output layer of the network are normalised such that they can be interpreted as a probability distribution over states of the DFA to be learnt. The network input is a pair of tensors: one representing the input word, and the second representing the DFA state. The DFA state is initially set to 1 and subsequent iterations use the learned next-state as input using recurrent connections. The approach works well for small automata of up to six states and an alphabet size of up to four.

Long Short Term Memory (LSTM) networks extend RNN functionality by providing a means to capture long-term dependencies [58]. Each LSTM cell consists of three gates: *input*, *forget*, and *output*. All three gates take into account the current input and a representation of previous inputs called the hidden state. The input gate determines what information should be part of the cell state or memory, the forget gate looks at what information is irrelevant and can be ignored, and the output gate generates the new hidden state for the LSTM cell.

LSTMs are often used for sequence prediction, and this has been exploited for a variety of applications. For example, they have been used for anomaly detection [54], trace restorations [104], and mining message flow patterns from trace data [23]. The research reported in [23] closely relates to our problem domain. Here, system-on-chip system specifications in the form of message flow patterns are mined from execution traces. Multiple LSTM networks trained to predict the next event for different input sequence lengths are used to infer patterns of different length. A set of pre-defined ground truth message sequences are used to determine pattern validity; mined patterns that possess a temporal ordering between messages not found in any of the ground truth sequences are termed invalid. Valid patterns are used as system specifications. Experiments show that the approach sometimes generates more invalid than valid patterns, and the number of valid patterns found is low compared to the set of ground truth sequences. Several measures are proposed to address these issues, including additional causality based filtering of generated patterns.

The Apperception Engine [40] is a new unsupervised learning algorithm to construct a symbolic theory to explain sequences of sensor data. The engine iterates over a set of *templates* that build on top of an initial template, each specifying a large and finite set of theories that conform to it. It has been found to outperform neural network baselines. However, when dealing with execution traces tracking system variables with possibly infinite domain, such as a temperature sensor, defining the initial template is not easy. Further, enumerating templates can be time consuming.

2.2 Program Synthesis

In this section we present background on program synthesis. In Chapter 3 we use program synthesis to derive syntactic expressions from system execution traces, that serve as transition predicates for learning symbolic system abstractions.

Program synthesis is the task of automatically finding programs that satisfy a given user intent expressed as some form of logical constraints [53]. It is a second-order search problem where the goal is to identify a function that satisfies a given specification. A program synthesizer can be viewed as a solver for existential second-order logic formula of the form

$$\exists P.Qx.\sigma(x, P) \tag{2.1}$$

where P ranges over functions, Q is either $\exists$ or $\forall$, x ranges over ground terms and σ is a quantifier-free formula. Second-order logic is an extension of first-order logic. While first-order logic only allows quantification over variables that range over ground terms, second-order logic additionally allows quantification over functions, as in (2.1).

Virtually all tools that perform program synthesis implement a form of Counter Example-Guided Inductive Synthesis (CEGIS). In CEGIS, the synthesis problem is split into two components: *search* and *validation*, as illustrated in Fig. 2.3. In the *search* component the solver looks for a candidate program that satisfies a simplified version of the original specification, often obtained as a reduction of the original second-order specification to first-order logic. In the *validation* component a verifier validates that the candidate program satisfies the original second-order requirement

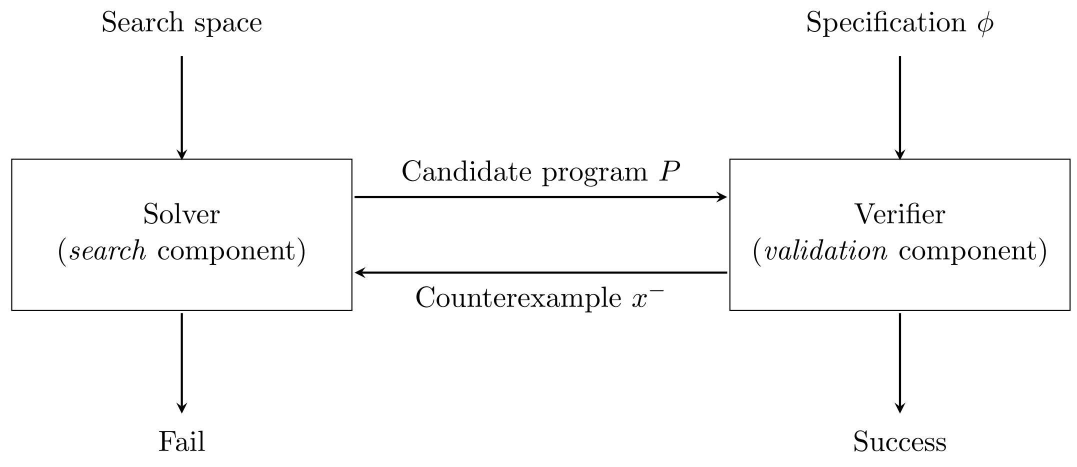


Figure 2.3: Counterexample-Guided Inductive Synthesis (CEGIS) [53].

on the desired program. If this is not the case, then the verifier returns a counterexample, which is returned to the solver. The solver then identifies another program that, additionally, satisfies the counterexample.

This iterative process is repeated until either the candidate program is successfully validated by the verifier, or the solver is unable to find a candidate program satisfying the specifications, i.e., the original synthesis problem is unsolvable.

The program that is generated is usually required to conform to a grammar, which is given as part of the problem description. Tools that require this grammar implement Syntax-Guided Synthesis (SyGuS) [6]. The input to SyGuS typically consists of a background theory T, a logical formula ϕ that specifies the semantic correctness of the desired program and a syntactic template representing the set of candidate programs or the search space for synthesis in the form of a context-free grammar G. The SyGuS problem is defined as finding a program P conforming to the grammar G that satisfies specification ϕ in the theory T.

An example SyGuS formulation for synthesising a function *max2* that returns the maximum of two given values x and y is provided in Fig. 2.4. The background theory for the formulation is set to Linear Integer Arithmetic (LIA), where variables are either integers or Booleans and the vocabulary consists of integer and Boolean constants, addition, relations and conditionals. The grammar specified for the example corresponds to linear expressions over the terminal x, y, 0 and 1, and if-then-else (ite) expressions. The correctness constraints define the logical specification $\phi_1 \land \phi_2$ where

$$\begin{aligned}\phi_1 &\coloneqq \mathit{max2}(x, y) \geq x \land \mathit{max2}(x, y) \geq y\\ \phi_2 &\coloneqq \mathit{max2}(x, y) = x \lor \mathit{max2}(x, y) = y\end{aligned}$$

Given the formulation in Fig. 2.4, one candidate program implementation for the function *max2* could be

$$(\mathrm{ite}\,(>=\ x\ y)\,x\ y)$$

interpreted as

$$\mathrm{if}\, x \geq y \text{ then } x \text{ else } y$$

that conforms to the given grammar and satisfies the specified correctness constraints.

Specifying user intent for the synthesis problem as a logical specification, such as the correctness constraints in the above formulation, can often be tricky particularly when a complete specification is not known. Example-based specifications are generally easier to define in such scenarios. Synthesis from examples [52] is a program

```
; set the background theory to LIA
(set-logic LIA)

; grammar for max2 candidate implementations
(synth-fun max2 ((x Int) (y Int)) Int
  ((Start Int (x y 0 1
           (+ Start Start)
           (- Start Start)
           (ite StartBool Start Start)))
  (StartBool Bool ((and StartBool StartBool)
               (or StartBool StartBool)
               (not StartBool)
               (<= Start Start)
               (= Start Start)
               (>= Start Start)))))

; universally quantified input variables x and y
(declare-var x Int)
(declare-var y Int)

; correctness constraints on the max2 function
(constraint (>= (max2 x y) x))
(constraint (>= (max2 x y) y))
(constraint (or (= x (max2 x y)) (= y (max2 x y))))

; synthesize command
(check-synth)
```

Figure 2.4: SyGuS formulation for max function over two variables [53].

synthesis paradigm where user intent is specified by input-output examples. The synthesis constraints define behaviour of the desired program on a subset of its valid inputs and the synthesised function is a generalisation of the specified behaviour.

2.3 Equivalence Relations

In this section we introduce formal preliminaries on equivalence relations between formal models of systems. We will use these formalisms in Chapter 5 for determining the degree of completeness of system abstractions generated by model-learning from system execution traces. We begin by formalizing transition systems.

Definition 1 (Transition System)**.** *A transition system T is a triple $(\mathcal{S}, \mathcal{S}_0, \Delta)$ where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states and $\Delta \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation.*

A transition system with state labelling is called a Kripke structure.

Definition 2 (Kripke Structure)**.** *A Kripke structure $\mathcal{M}$ is a five-tuple $(\mathcal{S}, \mathcal{S}_0, \Delta, \mathcal{AP}, L)$ where $\mathcal{S}$ is a set of states, $\mathcal{S}_0 \subseteq \mathcal{S}$ is the set of initial states, $\Delta \subseteq \mathcal{S} \times \mathcal{S}$ is a transition relation, $\mathcal{AP}$ is a set of atomic propositions and $L \colon \mathcal{S} \to 2^{\mathcal{AP}}$ is a labelling function that labels each state with a set of atomic propositions that are true in that state.*

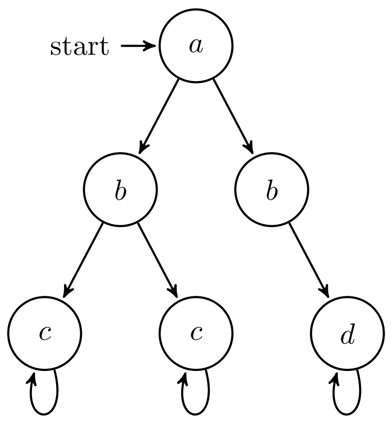
2.3.1 Bisimulation

Definition 3 (Bisimulation)**.** *Consider two Kripke structures $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \Delta, \mathcal{AP}, L)$ and $\mathcal{M}' = (\mathcal{S}', \mathcal{S}'_0, \Delta', \mathcal{AP}, L')$ with the same set of atomic propositions. A binary relation $\mathcal{B} \subseteq \mathcal{S} \times \mathcal{S}'$ is a bisimulation if*

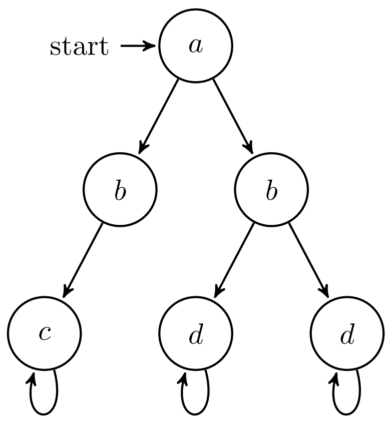
- $\forall s_0 \in \mathcal{S}_0$, $\exists s'_0 \in \mathcal{S}'_0$ *such that* $(s_0, s'_0) \in \mathcal{B}$
- $\forall s'_0 \in \mathcal{S}'_0$, $\exists s_0 \in \mathcal{S}_0$ *such that* $(s_0, s'_0) \in \mathcal{B}$
- $\forall (s, s') \in \mathcal{B}$,
  - $L(s) = L'(s')$
  - *for every* $s_1 \in \mathcal{S}$ *such that* $(s, s_1) \in \Delta$, $\exists s'_1 \in \mathcal{S}'$ *such that* $(s', s'_1) \in \Delta'$ *and* $(s_1, s'_1) \in \mathcal{B}$
  - *for every* $s'_1 \in \mathcal{S}'$ *such that* $(s', s'_1) \in \Delta'$, $\exists s_1 \in \mathcal{S}$ *such that* $(s, s_1) \in \Delta$ *and* $(s_1, s'_1) \in \mathcal{B}$

The structures $\mathcal{M}$ and $\mathcal{M}'$ are *bisimulation equivalent* if there exists a bisimulation relation between the two structures. An example of two bisimulation equivalent structures is provided in Fig. 2.5. The structures in Fig. 2.6, on the other hand, are not bisimulation equivalent. The state labelled b in Fig. 2.6b does not correspond to any state in Fig. 2.6a, since there is no state labelled b in $\mathcal{M}$ with transitions to both a state labelled c and a state labelled d.

A path $\pi = s_0 s_1 \dots$ in $\mathcal{M}$ and a path $\pi' = s'_0 s'_1 \dots$ in $\mathcal{M}'$ *correspond* if $(s_i, s'_i) \in \mathcal{B}$ for every $i \geq 0$. The existence of a bisimulation between $\mathcal{M}$ and $\mathcal{M}'$ guarantees that the two structures are behaviourally equivalent, i.e., for every path in $\mathcal{M}$ from an initial state, there is a *corresponding* path from an initial state in $\mathcal{M}'$, and vice versa. A formal proof is provided in Theorem 1.

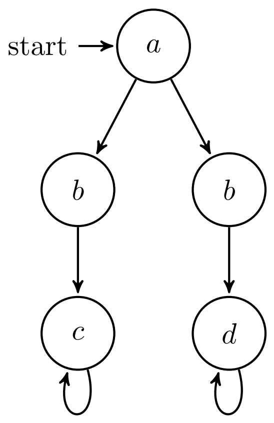


(a) $\mathcal{M}$

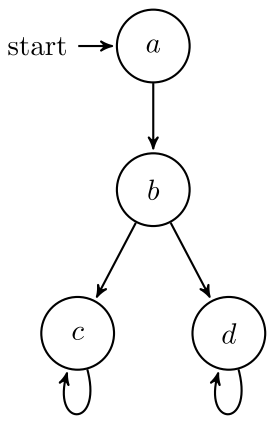


(b) $\mathcal{M}'$

Figure 2.5: Bisimilar structures [31].



(a) $\mathcal{M}$



(b) $\mathcal{M}'$

Figure 2.6: Nonbisimilar structures [31].

Theorem 1. *Let $\mathcal{M}$ and $\mathcal{M}'$ be two structures that are bisimulation equivalent, and $s \in \mathcal{S}$ and $s' \in \mathcal{S}'$ be two states such that $(s, s') \in \mathcal{B}$. Then, for every path starting from s there is a corresponding path starting from s', and for every path starting from s' there is a corresponding path starting from s.*

Proof. Let $(s, s') \in \mathcal{B}$. Let $\pi = s_0 s_1 \ldots$ be a path in $\mathcal{M}$ such that $s_0 = s$. Then we can construct a corresponding path $\pi' = s'_0 s'_1 \ldots$ in $\mathcal{M}'$ by induction as follows: it is clear that $(s_0, s'_0) \in \mathcal{B}$. Assume $(s_i, s'_i) \in \mathcal{B}$ for some i. By definition 3, since $(s_i, s_{i+1}) \in \Delta$, $\exists t' \in \mathcal{S}'$ such that $(s'_i, t') \in \Delta'$ and $(s_{i+1}, t') \in \mathcal{B}$. Choosing $s'_{i+1} = t'$ produces the corresponding path π'.

Similarly, given a path π' from s' in $\mathcal{M}'$, a path π can be constructed from s.

□

2.3.2 Simulation

Definition 4 (Simulation)**.** *Consider two Kripke structures* $\mathcal{M} = (\mathcal{S}, \mathcal{S}_0, \Delta, \mathcal{AP}, L)$ *and* $\mathcal{M}' = (\mathcal{S}', \mathcal{S}'_0, \Delta', \mathcal{AP}, L')$ *with the same set of atomic propositions. A binary relation* $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}'$ *is a simulation if*

- $\forall s_0 \in \mathcal{S}_0$, $\exists s'_0 \in \mathcal{S}'_0$ *such that* $(s_0, s'_0) \in \mathcal{R}$
- $\forall (s, s') \in \mathcal{R}$,
  - $L(s) = L'(s')$
  - *for every* $s_1 \in \mathcal{S}$ *such that* $(s, s_1) \in \Delta$, $\exists s'_1 \in \mathcal{S}'$ *such that* $(s', s'_1) \in \Delta'$ *and* $(s_1, s'_1) \in \mathcal{R}$

Simulation is closely related to bisimulation. While a bisimulation equivalence between structures guarantees that the two structures have the same behaviour, a simulation relation between $\mathcal{M}$ and $\mathcal{M}'$, denoted $\mathcal{M} \preceq \mathcal{M}'$, guarantees that $\mathcal{M}'$ is an *overapproximation* of $\mathcal{M}$, i.e., $\mathcal{M}'$ includes (at least) all behaviours of $\mathcal{M}$. Similar to the proof in Theorem 1, given $(s, s') \in \mathcal{R}$ we can construct a path π' starting from state s' in $\mathcal{M}'$ for every path π starting from state s in $\mathcal{M}$.

We say $\mathcal{M}'$ *simulates* $\mathcal{M}$, if $\mathcal{M} \preceq \mathcal{M}'$. Although the structures in Fig. 2.6 are not bisimulation equivalent as discussed before, $\mathcal{M}'$ simulates $\mathcal{M}$. We can define a simulation relation $\mathcal{R}$ that associates every state in $\mathcal{M}$ to a state with the same label in $\mathcal{M}'$. The above definition also has the property that if states s, s' are associated by $\mathcal{R}$, for every successor of s there is a successor of s' with the same label. On the other hand, $\mathcal{M}$ does not simulate $\mathcal{M}'$ as the state labelled b in $\mathcal{M}'$ does not have a corresponding state in $\mathcal{M}$.

Chapter 3

Learning Symbolic Models from Execution Traces

Abstract system models have applications in design exploration, analysis, testing and verification. However, crafting a good abstraction for system-wide analysis can be challenging. Modern systems often have multiple components and it is difficult to specify integrated system behaviour, particularly emergent behaviour, ahead of time. Here, execution traces can provide valuable information as they exemplify system behaviours that are exercised when an instrumented implementation runs. This is leveraged by model-learning algorithms to automatically infer abstract system models as finite automata from trace data.

It is however challenging to apply existing methods to learn high-level symbolic abstractions for system-wide analysis, as discussed in Chapter 1. They often rely on the presence of labelled traces to construct concise models, and negative traces are difficult to come by in practice. Further, generated abstractions are limited in expressiveness—many model-learning implementations are tailored to generate system models that represent a very specific and often small subset of system behaviours, such as Mealy machines with a single timer [108]. In this chapter we present a novel model-learning algorithm to reverse-engineer concise, accurate and expressive symbolic abstractions from traces using only system execution trace data.

Plan of the chapter

The rest of the chapter is organized as follows. Section 3.1 introduces formal notations used throughout the chapter and provides a formal description of the problem. Section 3.2 describes the model-learning algorithm in detail. Section 3.3 describes the algorithmic methods used for improving algorithm performance and scalability.

Claim of Novelty

Unlike existing techniques, the model-learning algorithm presented in this chapter uses a combination of SAT and program synthesis to infer symbolic system abstractions using only system execution traces. Given a set of execution traces, SAT is used to systematically search for a minimal automaton that accepts (at least) all traces in the set, while program synthesis is used to consolidate trace information into syntactic expressions that serve as transition predicates in the learned abstraction.

In our algorithm the transition predicate inference procedure is agnostic of the automaton construction procedure. The input to automaton construction using SAT is a sequence of Boolean expressions—these could be concrete Boolean observations in a trace or syntactic expressions representing a set of observations. As long as the program synthesis procedure for transition predicate inference can generate such a sequence of Boolean valued expressions as output, the specific implementation details of automaton construction are not relevant to the procedure. This enables the model-learning algorithm to fully utilize the strength of SAT and program synthesis to generate concise, accurate and expressive models.

3.1 Formal Model

We suppose that we can collect execution traces of the system we wish to generate a model for by tracking a finite set of user-defined values represented by variables, $X = \{x_1, \ldots, x_k\}$, ranging over some domain D. We simplify the presentation by assuming all variables have the same domain, but the generalization to multiple domains is routine. The variables in X could stand for concrete values that are directly observable in the system (register values, elements of program state, etc.) or the results of some functions of such values, depending on the user's intent. The set $X' = \{x'_1, \ldots, x'_k\}$ contains corresponding primed variables, also over the domain D. A primed variable x'_i represents the same value as the unprimed variable x_i represents following a single discrete step of system execution.

A *valuation* $v : X \rightarrow D$ maps the variables in X to elements of D. An *observation* at a discrete time step t of system execution is a valuation of the variables at that time, and is denoted by v_t. An *execution trace* is a finite sequence of observations over successive discrete steps of time; we write an execution trace σ with n observations as a sequence of valuations $\sigma = v_0, v_1, \ldots, v_{n-1}$.

Given a set of execution traces $\mathcal{T}$, our task is to generate a system model as a symbolic finite automaton (SFA) that accepts all traces in $\mathcal{T}$. In symbolic automata, transitions carry predicates over a Boolean algebra.

Definition 5 (Boolean Algebra)**.** *A Boolean algebra $\mathcal{A}$ is defined to be a tuple $(\Sigma, \psi, [\![\_]\!], \bot, \top, \wedge, \vee, \neg)$ where Σ is the set of domain elements, ψ is the set of predicates closed under the Boolean connectives $\wedge$, $\vee$ and $\neg$, with $\bot, \top \in \psi$, and the component $[\![\_]\!] : \psi \rightarrow 2^{\Sigma}$ is a denotation function such that (i) $[\![\bot]\!] = \emptyset$, (ii) $[\![\top]\!] = \Sigma$, and (iii) for all $p_i, p_j \in \psi$, $[\![p_i \vee p_j]\!] = [\![p_i]\!] \cup [\![p_j]\!]$, $[\![p_i \wedge p_j]\!] = [\![p_i]\!] \cap [\![p_j]\!]$, and $[\![\neg p_i]\!] = \Sigma \backslash [\![p_i]\!]$.*

Definition 6 (Symbolic Finite Automaton)**.** *An SFA is a tuple $\mathcal{M} = (\mathcal{A}, Q, q^0, F, \Delta)$ where $\mathcal{A}$ is a Boolean algebra, Q is a finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \psi \times Q$ is a finite set of transitions, where ψ is the set of predicates of $\mathcal{A}$.*

An SFA $\mathcal{M}$ is deterministic if, for all transitions $(q, p_1, q_1), (q, p_2, q_2) \in \Delta$ where $p_1, p_2 \in \psi$, if $q_1 \neq q_2$ then $[\![p_1 \wedge p_2]\!] = \emptyset$. In this work we learn non-deterministic symbolic automata. For the rest of the article, we use SFA to mean non-deterministic symbolic finite automaton, unless specified otherwise.

Our algorithm generates SFAs by constructing characteristic functions of sets of observations in a trace (Section 3.2.2). These are Boolean-valued expressions over $(X \cup X')$, that serve as predicates on the transition edges. We define the Boolean algebra for the generated SFA as the smallest set that contains $\top$, $\bot$, and all the predicates constructed by our algorithm, and is closed under conjunction, disjunction and negation.

The domain Σ for the Boolean algebra $\mathcal{A}$ of our SFA will contain elements $a : (X \cup X') \rightarrow D$. That is, a pair of consecutive observations of the system. Given an execution trace $\sigma = v_0, \ldots, v_{n-1}$, the element a_t for $t = 0, \ldots, n-2$ is defined as follows:

$$\begin{aligned} a_t(x) &= v_t(x) \\ a_t(x') &= v_{t+1}(x) \end{aligned} \tag{3.1}$$

i.e., $a_t(x)$ is the valuation of the variable x at the discrete time step t of system execution and $a_t(x')$ is the valuation of the corresponding unprimed variable x in the following time step $(t + 1)$.

The learned SFA accepts a trace $\sigma = v_0, v_1, \ldots, v_{n-1}$ iff for all $0 \leq i < n - 1$ there exists a transition $(q_i, p_i, q_{i+1}) \in \Delta$ such that $q_0 = q^0$, $q_{n-1} \in F$ and $a_i \in [\![p_i]\!]$, where a_i corresponds to the observation pair (v_i, v_{i+1}) as defined in (3.1). In our setting, all states of the automaton are accepting states; i.e., our automaton rejects by running into a ‘dead end’. The language of the automaton, denoted by $\mathcal{L}(\mathcal{M})$, is defined as the set of all traces that are accepted by the SFA.

3.2 Model Learning with SAT and Program Synthesis

An overview of the model-learning algorithm is illustrated in Fig. 3.1. We first describe the model-learning algorithm using a single trace σ as input for the sake of clarity. At the end of this chapter, we discuss how the algorithm can be adapted to generate models from a set of execution traces $\mathcal{T}$.
The algorithm has the following components:

1. *Tracing infrastructure.* This records traces by observing the set of user-defined variables X during system execution.

2. *Transition predicate synthesizer.* This generates a sequence of transition predicates from trace data using program synthesis.

3. *Automaton construction algorithm.* This uses SAT to iteratively construct an automaton from a sequence of predicates obtained from the previous step and checks its compliance with the sequence input.

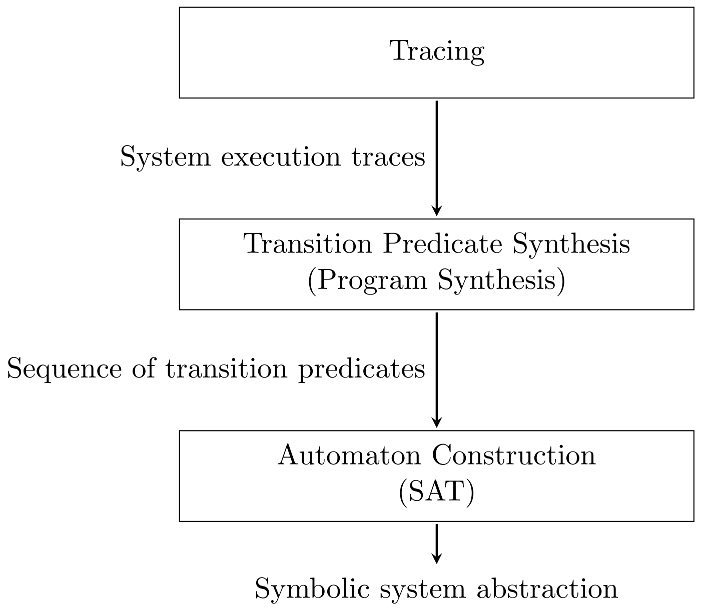

Tracing
System execution traces
Transition Predicate Synthesis
(Program Synthesis)
Sequence of transition predicates
Automaton Construction
(SAT)
Symbolic system abstraction

Figure 3.1: Overview of the model-learning algorithm.

We describe these components in detail in the sections that follow.

3.2.1 Tracing Setup

We use implementations of the system of interest to obtain execution traces. For most of our experiments, execution traces are produced simply by instrumenting source code with print statements. This provides flexibility in getting the required information from the simulation runs. Target components of interest are identified and trace statements added at relevant points in the source code based on the end goal for analysis. Traces can also be produced using any other means, for example inbuilt tracing or logging frameworks.

3.2.2 Transition Predicate Synthesis

Trace data recording system variable valuations over time can be difficult to interpret in their raw, undigested form. We therefore consolidate the trace information into syntactic expressions that will serve as transition predicates in the generated model.

We use program synthesis to infer Boolean valued expressions over $(X \cup X')$ from the trace data. The method used is an instance of *synthesis from examples* [52], where concrete examples demonstrating the behaviour of the function to be synthesised are used to guide synthesis. There are many algorithms that implement this synthesis technique. We discuss choices for the synthesis algorithm in Section 4.2.3, Chapter 4.

The system abstractions generated by our algorithm model the following two aspects of system behaviour:

- updates to a variable $x_i \in X$ during system execution
- transitions between predefined high-level system states or operation modes

In the following sections we describe the methodology used to infer data update functions and system-state transition guards as predicates for the generated abstraction.

3.2.2.1 Data Updates

To model updates to a variable $x_i \in X$ during system execution in the generated model, we infer a data-update function $f_{x_i}(X)$ from trace data as follows:
Given a trace $\sigma = v_0, v_1, \ldots, v_{n-1}$ we define the following synthesis constraint

$$f_{x_i}(v_t) = v_{t+1}(x_i), \forall\, 0 \leq t < n-1 \tag{3.2}$$

This is fed to a program synthesis tool, which in turn returns a syntactic expression for the function f_{x_i} that satisfies the constraint.

For example, consider a system with $X = \{x_1, x_2\}$, and let $v_0 = (1, 0)$, $v_1 = (2, 0)$, $v_2 = (3, 0)$, $v_3 = (4, 0)$ be a trace of that system. To model updates to the variable x_1 we infer the data update function f_{x_1} by defining the following synthesis constraints as in (3.2).

$$\begin{aligned} f_{x_1}(1, 0) &= 2 \\ f_{x_1}(2, 0) &= 3 \\ f_{x_1}(3, 0) &= 4 \end{aligned}$$

From these constraints, the synthesis tool might generate $f_{x_1} = x_1 + 1$.

The synthesised function f_{x_i} provides the value of the variable x_i following a single discrete step of system execution, and is used to define a Boolean expression '$x_i' = f_{x_i}$', that will serve as a transition predicate modelling updates to the variable x_i in the generated abstraction.

To tackle synthesis complexity for long traces, we divide the trace $\sigma = v_0, \ldots, v_{n-1}$ into consecutive segments using a sliding window. We then define synthesis constraints as in (3.2) for each segment in independent instances of program synthesis. The function synthesised in each program synthesis instance is used to construct a sequence of predicates $P = p_0, p_1, \ldots, p_{n-2}$, such that

$$p_t \coloneqq x_i' = f_{x_i}, \forall\, 0 \leq t < n - 1 \tag{3.3}$$

where f_{x_i} is the data update function synthesised using $f_{x_i}(v_t) = v_{t+1}(x_i)$ as a constraint for synthesis.

It is often difficult to determine a suitable sliding-window size a priori. Therefore, we generate multiple predicates by varying the window size from 3 to 10 and use the smallest synthesised predicate. We then slide the window along the trace by the window size corresponding to the smallest predicate and repeat the process. The sequence of predicates P is constructed as described in (3.3).

The sliding window procedure is a general scheme that can break down a large program synthesis instance into multiple smaller instances to tackle synthesis complexity. In practice we often come across scenarios where the nature of updates to a data variable $x_i \in X$ is influenced by system triggers, events, high-level system states or operation modes. For instance, the nature of updates to the length of a queue in a Serial Input-Output (I/O) Port system varies depending on whether there is *read*, *write* or *reset* event on the queue. We employ a more intuitive scheme to tackle synthesis complexity in such scenarios.

We generalise the above scenario to the case where updates to a variable $x_i \in X$ is influenced by a variable $x_j \in X$ with a finite enumerable domain D_{x_j}. To model updates to x_i in the generated abstraction we synthesise a data-update function $f_{x_i|x_j=d}$ for each value $d \in D_{x_j}$ in independent instances of program synthesis. The function $f_{x_i|x_j=d}$ models updates to the variable x_i when $x_j = d$. To synthesise the function $f_{x_i|x_j=d}$ from a trace $\sigma = v_0, \ldots, v_{n-1}$ we define the following synthesis constraint:

$$f_{x_i|x_j=d}(v_t) = v_{t+1}(x_i), \forall\, 0 \leq t < n-1 \wedge v_t(x_j) = d \tag{3.4}$$

The synthesised function is used to define a Boolean expression '$x_j = d \wedge x_i' = f_{x_i|x_j=d}$', that will serve as a transition predicate in the generated abstraction.

Synthesised functions $f_{x_i|x_j=d}$ for all values $d \in D_{x_j}$ from trace $\sigma = v_0, \ldots, v_{n-1}$ are used to construct the sequence of predicates $P = p_0, p_1, \ldots, p_{n-1}$ such that

$$p_t \coloneqq x_j = v_t(x_j) \;\wedge\; x_i' = f_{x_i|x_j=v_t(x_j)}, \forall\, 0 \leq t < n-1 \tag{3.5}$$

$$\begin{aligned}
v_0 &= (0, \mathit{reset})\\
v_1 &= (0, \mathit{write})\\
v_2 &= (1, \mathit{write})\\
v_3 &= (2, \mathit{read})\\
v_4 &= (1, \mathit{read})\\
v_5 &= (0, \mathit{write})\\
v_6 &= (1, \mathit{reset})\\
v_7 &= (0, \mathit{write})\\
v_8 &= (1, \mathit{read})\\
v_9 &= (0, \mathit{read})
\end{aligned}$$

(a) Execution trace σ

$$\begin{aligned}
f_{x_i|x_j=\mathit{reset}}(0, \mathit{reset}) &= 0\\
f_{x_i|x_j=\mathit{reset}}(1, \mathit{reset}) &= 0\\[1ex]
f_{x_i|x_j=\mathit{write}}(0, \mathit{write}) &= 1\\
f_{x_i|x_j=\mathit{write}}(1, \mathit{write}) &= 2\\[1ex]
f_{x_i|x_j=\mathit{read}}(2, \mathit{read}) &= 1\\
f_{x_i|x_j=\mathit{read}}(1, \mathit{read}) &= 0
\end{aligned}$$

(b) Synthesis constraints

$$\begin{aligned}
f_{x_i|x_j=\mathit{reset}} &= 0\\[1ex]
f_{x_i|x_j=\mathit{write}} &= x_i + 1\\[1ex]
f_{x_i|x_j=\mathit{read}} &= x_i - 1
\end{aligned}$$

(c) Synthesised Expressions

$$\begin{aligned}
p_0 &\coloneqq x_j = \mathit{reset} \wedge x_i' = 0\\
p_1 &\coloneqq x_j = \mathit{write} \wedge x_i' = x_i + 1\\
p_2 &\coloneqq x_j = \mathit{write} \wedge x_i' = x_i + 1\\
p_3 &\coloneqq x_j = \mathit{read} \wedge x_i' = x_i - 1\\
p_4 &\coloneqq x_j = \mathit{read} \wedge x_i' = x_i - 1\\
p_5 &\coloneqq x_j = \mathit{write} \wedge x_i' = x_i + 1\\
p_6 &\coloneqq x_j = \mathit{reset} \wedge x_i' = 0\\
p_7 &\coloneqq x_j = \mathit{write} \wedge x_i' = x_i + 1\\
p_8 &\coloneqq x_j = \mathit{read} \wedge x_i' = x_i - 1
\end{aligned}$$

(d) Transition Predicate Sequence P

Figure 3.2: Synthesising predicates for an abstraction of a Serial I/O Port system with $X = \{x_i, x_j\}$, modelling updates to queue length x_i under the influence of event x_j with domain $D_{x_j} = \{\mathit{reset}, \mathit{write}, \mathit{read}\}$.

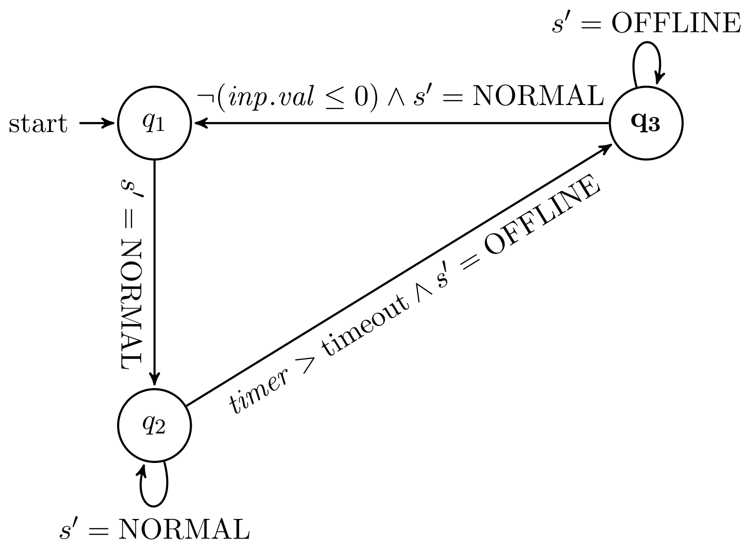
An example run of the above described scheme is illustrated in Fig. 3.2 using the execution traces of a Serial I/O Port system, monitoring queue length x_i and events x_j on the queue.

3.2.2.2 System-state Transition Guards

Inferring system-state transition guards from trace data can be useful when we wish to model how a system switches between predefined high-level system states or operation modes. For instance, the system abstraction of an IoT Heartbeat Monitoring System provided in Fig. 3.3 models how the system switches between its operation modes NORMAL and OFFLINE during execution.

We suppose that the high-level system states or operation modes of interest are observable either as an output of the system or can be tapped from inside the system with relevant instrumentation. We generalise system-state transition guard inference to synthesising a Boolean expression that describes the condition under which a variable $x_i \in X$ with a finite enumerable domain D_{x_i} changes its value during system execution. We synthesise a guard $g_{x_i,d \to d'}$ for each pair $(d, d') \in D_{x_i} \times D_{x_i}$, where $d \neq d'$, in independent instances of program synthesis. The synthesised guard will in turn represent the condition under which x_i changes from d to d'.

Given a trace $\sigma = v_0, v_1, \ldots, v_{n-1}$, we define the following synthesis constraints to infer the transition guard $g_{x_i,d \to d'}$:



$s' = \text{OFFLINE}$
$\neg(inp.val \leq 0) \wedge s' = \text{NORMAL}$
start
q_1
q_3
$s' = \text{NORMAL}$
$timer > \text{timeout} \wedge s' = \text{OFFLINE}$
q_2
$s' = \text{NORMAL}$

Figure 3.3: Abstraction modelling operation mode switches for an IoT Heartbeat Monitoring system generated by our algorithm.

$$g_{x_i,d\to d'}(v_t) = \begin{cases} \mathit{true}, & \text{if } v_{t+1}(x_i) = d' \\ \mathit{false}, & \text{if } v_{t+1}(x_i) \neq d' \end{cases}, \forall 0 \leq t < n-1 \wedge v_t(x_i) = d \tag{3.6}$$

The transition guard $g_{x_i,d\to d'}$ synthesised from these constraints is used to define a Boolean expression '$g_{x_i,d\to d'} \wedge x'_i = d'$', that will serve as a transition predicate in the generated abstraction. Synthesised guards $g_{x_i,d\to d'}$ from trace $\sigma = v_0, \ldots, v_{n-1}$ for all pairs $(d, d') \in D_{x_i} \times D_{x_i}$ where $d \neq d'$ are used to construct the sequence of predicates $P = p_0, p_1, \ldots, p_{n-1}$ such that

$$p_t := \begin{cases} x'_i = v_{t+1}(x_i), & \text{if } v_t(x_i) = v_{t+1}(x_i) \\ g_{x_i,v_t(x_i)\to v_{t+1}(x_i)} \wedge x'_i = v_{t+1}(x_i), & \text{if } v_t(x_i) \neq v_{t+1}(x_i) \end{cases}, \forall 0 \leq t < n-1 \tag{3.7}$$

An example run of the above described method for transition guard inference is illustrated in Fig. 3.4 using the execution trace of an IoT Heartbeat Monitoring system.

$$\begin{aligned} v_0 &= (\mathrm{N}, 1, 0, 5) \\ v_1 &= (\mathrm{N}, 1, 0, 5) \\ v_2 &= (\mathrm{N}, -10, 1, 5) \\ v_3 &= (\mathrm{N}, -10, 2, 5) \\ v_4 &= (\mathrm{N}, -10, 3, 5) \\ v_5 &= (\mathrm{N}, -10, 4, 5) \\ v_6 &= (\mathrm{N}, -10, 5, 5) \\ v_7 &= (\mathrm{N}, -10, 6, 5) \\ v_8 &= (\mathrm{O}, -10, 6, 5) \\ v_9 &= (\mathrm{O}, 1, 6, 5) \end{aligned}$$

(a) Execution trace σ

$$\begin{aligned} g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, 1, 0, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 1, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 2, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 3, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 4, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 5, 5) &= \mathit{false} \\ g_{s,\mathrm{N}\to\mathrm{O}}(\mathrm{N}, -10, 6, 5) &= \mathit{true} \\ \\ g_{s,\mathrm{O}\to\mathrm{N}}(\mathrm{O}, -10, 6, 5) &= \mathit{false} \end{aligned}$$

(b) Synthesis Constraints

$$g_{s,\mathrm{N}\to\mathrm{O}} = \mathit{timer} > \text{timeout}$$

$$g_{s,\mathrm{O}\to\mathrm{N}} = \mathit{false}$$

(c) Synthesised Expressions

$$\begin{aligned} p_0 &:= s' = \mathrm{N} \\ p_1 &:= s' = \mathrm{N} \\ p_2 &:= s' = \mathrm{N} \\ p_3 &:= s' = \mathrm{N} \\ p_4 &:= s' = \mathrm{N} \\ p_5 &:= s' = \mathrm{N} \\ p_6 &:= s' = \mathrm{N} \\ p_7 &:= \mathit{timer} > \text{timeout} \wedge s' = \mathrm{O} \\ p_8 &:= s' = \mathrm{O} \end{aligned}$$

(d) Transition Predicate Sequence P

Figure 3.4: Synthesising predicates for an abstraction of an IoT Heartbeat Monitoring system with $X = \{s, \mathit{inp.val}, \mathit{timer}, \text{timeout}\}$ modelling switches between operation modes NORMAL and OFFLINE represented by $s \in X$. To simplify representation we use N and O here in place of NORMAL and OFFLINE respectively.

Note that each predicate p_t in P constructed from trace σ is satisfied by (v_t, v_{t+1}) for all $0 \le t < n-1$. Thus, the element $a_t \in \Sigma$ corresponding to (v_t, v_{t+1}) as defined in (3.1) satisfies p_t by construction, i.e., $a_t \in [\![p_t]\!]$.

3.2.3 Automaton Construction

Given a predicate sequence $P = p_0, \ldots, p_{n-2}$ constructed from execution trace $\sigma = v_0, \ldots, v_{n-1}$ as described in Section 3.2.2, the automaton construction algorithm learns an SFA that accepts σ. To this end, the algorithm learns an SFA that conforms to P.

Definition 7. *An SFA $\mathcal{M} = (\mathcal{A}, Q, q^0, F, \Delta)$ is said to conform to a sequence of predicates $P = p_0, \ldots, p_{n-2}$ if there exists a finite path $q_0 \xrightarrow{p_0} q_1 \ldots q_{n-2} \xrightarrow{p_{n-2}} q_{n-1}$ in $\mathcal{M}$ such that $q_0 = q^0$ and $q_{n-1} \in F$.*

Theorem 2. *If an SFA $\mathcal{M} = (\mathcal{A}, Q, q^0, F, \Delta)$ conforms to the predicate sequence $P = p_0, \ldots, p_{n-2}$ constructed from a trace $\sigma = v_0, \ldots, v_{n-1}$, then $\mathcal{M}$ accepts σ.*

Proof. Since $\mathcal{M}$ conforms to $P = p_0, \ldots, p_{n-2}$, by definition 7 there exists a finite path $q_0 \xrightarrow{p_0} q_1 \ldots q_{n-2} \xrightarrow{p_{n-2}} q_{n-1}$ in $\mathcal{M}$ such that $q_0 = q^0$ and $q_{n-1} \in F$. As discussed in Section 3.2.2, the element $a_i \in \Sigma$ corresponding to the observation pair (v_i, v_{i+1}) in trace σ satisfies p_i by construction, i.e., $a_i \in [\![p_i]\!]$ for all $0 \le i < n-1$. This implies that for $0 \le i < n-1$ there exists transitions $(q_i, p_i, q_{i+1}) \in \Delta$ such that $q_0 = q^0$, $a_i \in [\![p_i]\!]$ and $q_{n-1} \in F$. The SFA $\mathcal{M}$, therefore, accepts trace σ. □

The automaton construction algorithm is provided in Algorithm 1. The automaton to be constructed is represented as an array $\mathcal{M}$, each element of which encodes a transition (line 2). The i-th element in the array is a triple comprising the following symbolic variables: a state variable q_i for the state from which the transition occurs, a variable p'_i for the corresponding transition predicate and a next state variable q'_i for the state the automaton moves to at the end of the transition. The sequence of predicates P is used to define the following Boolean constraints on array $\mathcal{M}$ such that the learned automaton conforms to P (line 3):

$$\bigwedge_{i=0}^{n-2} p'_i = p_i \tag{3.8}$$

$$\bigwedge_{i=0}^{n-3} q_{i+1} = q'_i \tag{3.9}$$

Algorithm 1 Automaton Construction Algorithm

```
1: procedure GenerateAutomata($P = p_0, \ldots, p_{n-2}$ : sequence of predicates
   constructed from trace $\sigma = v_0, \ldots, v_{n-1}$)
2:     Target automaton $\mathcal{M}$ is represented as
       $\mathcal{M} = (q_0, p'_0, q'_0), \ldots, (q_{n-2}, p'_{n-2}, q'_{n-2})$
3:     $c \leftarrow (\bigwedge_{i=0}^{n-2} p'_i = p_i) \wedge (\bigwedge_{i=0}^{n-3} q_{i+1} = q'_i)$   ▷ Encode predicate sequence as
                                                              transition sequences in $\mathcal{M}$
4:     $N \leftarrow 1$                                        ▷ Number of automaton states
5:     $c \leftarrow c \wedge (\bigwedge_{i=0}^{n-2} 1 \leq q_i, q'_i \leq N)$   ▷ Set number of automaton states
6:     $result$ = Check_SAT($c$)                               ▷ Invoke SAT solver
7:     if $result$ = UNSAT then                                ▷ No $N$-state automaton found
8:         $N \leftarrow (N + 1)$
9:         go to 5
10:    else                                                    ▷ Candidate automaton $\mathcal{M}$ found
11:        $l \leftarrow$ level of precision
12:        $S_l \leftarrow$ set of all transition sequences
               of length $l$ in $\mathcal{M}$
13:        $P_l \leftarrow$ set of all subsequences of $P$
               of length $l$
14:        if $S_l \nsubseteq P_l$ then                        ▷ Subsequence check failed
15:            Encode sequences in $(S_l - P_l)$ as
               blocking constraints $c_b$
16:            $c \leftarrow c \wedge c_b$
17:            go to 6
18:        else                                                ▷ Subsequence check successful
19:            return $\mathcal{M}$
20:        end if
21:    end if
22: end procedure
```

Constraint (3.8) assigns synthesized predicates p_i in P to the symbolic variable p'_i in $\mathcal{M}$. The two constraints together ensure that the predicate sequence P is captured by the automaton, i.e., there exists a finite path $q_0 \xrightarrow{p_0} q_1 \ldots q_{n-2} \xrightarrow{p_{n-2}} q_{n-1}$ in $\mathcal{M}$.

To construct the automaton, we search systematically for an N-state automaton that satisfies the constraints above. The number of automaton states is set to N by restricting all state variables of $\mathcal{M}$ to take values between 1 and N (line 5). The Boolean formula encoding the constraints is fed to a SAT solver, which in turn looks for a satisfying assignment (line 6). If a satisfying assignment is found, the solution is returned as a candidate automaton (line 10). If a satisfying assignment cannot be found, it implies that there is no N-state automaton that meets the constraints; in this case we increment N by 1 and repeat the search (lines 7–9). We begin automaton construction with $N = 1$ (line 4), increasing N by 1 each time an automaton cannot be found. This ensures that we learn the smallest automaton that conforms to P.

Due to the absence of negative traces, a trivial solution to the SAT formulation described above is a single state automaton that conforms to P. We therefore need some means of tackling overgeneralisation when learning abstractions from only positive trace samples. In our algorithm, we introduce a tuning mechanism to address overgeneralisation and control the precision of the learned system model.

Learning models from only positive trace samples

The problem of model-learning from labelled traces is conventionally translated to generating an automaton that accepts all positive and rejects all negative traces. However, in the absence of negative traces defining the exact language of the automaton to be learned is not straightforward. At one extreme, we can produce an overgeneralised, single-state automaton that accepts not only the given traces but also every other sequence of events defined over the alphabet set. At the other extreme, we can generate an exact automaton that exclusively accepts the given traces and nothing else. But such an automaton can be very large and complex to interpret. Identifying the right compromise between these two extremes is challenging.

Previous work on generating models from only positive trace samples [11, 38] try to learn the strictest automaton with at most N states. These algorithms look for an automaton $\mathcal{M}$ with at most N states that is consistent with the given positive trace samples, such that there is no other automaton $\mathcal{M}'$ with at most N states where the language of $\mathcal{M}'$ is a strict subset of the language of $\mathcal{M}$. While the language inclusion constraint ensures that the model generated is not overgeneralised, the number of states N controls the degree of generalisation. However, it is not possible to make an educated guess for the value of N without any domain knowledge. Also, given the generated model there is no means of identifying the sequential behaviours represented by the inferred model that were not exemplified in the input trace set.

In our algorithm we introduce a single tunable parameter l, similar to the parameter k in k-Tails described in Section 2.1.1, Chapter 2, to address overgeneralisation and control the precision of the learned automaton. Once a candidate automaton is generated, we check the compliance of the generated automaton with P by looking for *invalid* transition sequences of length l in $\mathcal{M}$ (lines 10–21). A transition sequence is said to be invalid if it is not a subsequence of P (line 14).

We encode invalid sequences as Boolean formulae as in (3.8) and (3.9), add their negation to the set of existing constraints and repeat the search (lines 14–17). These additional constraints on the automaton, referred to as *blocking constraints* in Algorithm 1, restrict the automaton from encoding invalid transitions. The constraints also provide provable guarantees that all sequential behaviour exemplified by transition sequences of length l in the generated automaton are indeed exemplified by the input trace and therefore correspond to observed system execution behaviour. The algorithm returns the N-state automaton $\mathcal{M}$ if a candidate automaton is found and the compliance check is successful (lines 18–19).

A higher value of l implies tighter constraints of the automaton and hence, generates a more exact representation of the trace. This is analogous to increasing the parameter k in k-Tails that increases the precision, size and complexity of the learned model. While k-Tails is a top-down approach where the parameter k identifies state merges in a PTA, our algorithm is a bottom-up approach where the parameter l identifies state splits in an overgeneralised model. An experimental evaluation demonstrating the correlation between l values, and automaton size and algorithm runtime is provided in Section 4.2.2, Chapter 4.

Although this dissertation focuses on inferring abstractions from system execution traces, the automaton construction algorithm can also be extended to infer automata from labelled traces, i.e., positive and negative traces. Here, the negative traces can be encoded as blocking constraints in Algorithm 1 to restrict the automaton from representing the behaviours exemplified by them.

The automaton construction algorithm also provides an optional functionality to generate automata where no two transitions from a state are marked with the same predicate. To this end, we use standard techniques for NFA to DFA conversion followed by DFA minimisation, treating the set of transition predicates as the automaton alphabet. Note that the resulting automaton need not be a deterministic SFA. For instance, the automaton may have syntactically different predicates p_{i1} and p_{i2} on transitions from some state q_i to states q'_{i1} and q'_{i2} respectively. But, $[\![p_{i1}]\!] \cap [\![p_{i2}]\!]$ may not be empty, thereby introducing the possibility of non-determinism in the automaton.

3.3 Performance Optimizations

The SAT formulation described Section 3.2.3 for automaton construction leads to $\mathcal{O}(|P|)$ Boolean constraints. As a result, when dealing with long execution traces, the Boolean formula encoding constraints on the automaton can become very large, thereby increasing algorithm runtime. To improve algorithm scalability, we introduce two performance optimizations that help break down a large SAT problem into multiple smaller instances with manageable runtime.

3.3.1 Trace Segmentation

Trace segmentation as an optimization leverages the presence of repeating patterns in the predicate sequence to significantly reduce the size of the SAT problem from $\mathcal{O}(|P|)$, and thereby reduce algorithm runtime. In our context, a pattern is any finite sequence of predicates that are consecutive in P. Each pattern can be viewed as a symbolic characterisation of sequential system behaviour. For instance, the sequence of predicates $s' = \text{NORMAL}$, $(\mathit{timer} > \text{timeout}) \wedge (s' = \text{OFFLINE})$, $s' = \text{OFFLINE}$ generated from traces of an IoT Heartbeat Monitoring system is a characterisation of the sequential behaviour where the system switches from NORMAL mode to OFFLINE mode on timeout and remains in the OFFLINE mode.

With trace segmentation, we relax the constraints for automaton construction such that instead of learning an automaton that conforms to the entire predicate sequence, we learn an automaton that captures all sequential system behaviour exemplified by patterns of a specified length w. We use a sliding window to extract the set of all unique patterns of length w from the predicate sequence. This is illustrated in Fig. 3.5 for $w = 3$. We define Boolean constraints 3.8 and 3.9 for each unique pattern and feed the conjunction to the SAT solver.

This leads to $\mathcal{O}(w \cdot |\mathit{Uniq}_w|)$ constraints, where Uniq_w is the set of all unique patterns of length w. For long traces with frequently repeating patterns $(w \cdot |\mathit{Uniq}_w|) \ll |P|$. In such scenarios, trace segmentation produces a significant reduction in algorithm runtime. A detailed discussion on the need for segmentation in practice and its effect on scalability is given in Section 4.4, Chapter 4. We discuss strategies for choosing a suitable window size for trace segmentation in Section 4.2.1, Chapter 4.

Note that only a window size equal to the length of the predicate sequence P guarantees model correctness, i.e., $w = |P|$ guarantees that the generated abstraction conforms to P. For $w < |P|$ the Boolean constraints defined for automaton construction do not suffice to guarantee model correctness. To ensure correctness of generated abstractions irrespective of the window size w we employ a check after automaton construction to verify that the generated abstraction conforms to P. If the check fails we add missing predicate sequences incrementally to the generated automaton, until the automaton conforms to P. This incremental procedure for model-learning is described in Section 3.3.2.

s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
s'=NORMAL
$\mathit{timer}>$timeout $\land$ s'=OFFLINE
s'=OFFLINE
s'=OFFLINE
s'=OFFLINE
s'=OFFLINE
s'=OFFLINE
s'=OFFLINE
s'=OFFLINE

(a) Predicate Sequence P

Pattern 1: s'=NORMAL
s'=NORMAL
s'=NORMAL

Pattern 2: s'=NORMAL
s'=NORMAL
$\mathit{timer}>$timeout $\land$ s'=OFFLINE

Pattern 3: s'=NORMAL
$\mathit{timer}>$timeout $\land$ s'=OFFLINE
s'=OFFLINE

Pattern 4: $\mathit{timer}>$timeout $\land$ s'=OFFLINE
s'=OFFLINE
s'=OFFLINE

Pattern 5: s'=OFFLINE
s'=OFFLINE
s'=OFFLINE

(b) Unique patterns of length 3 in P

Figure 3.5: Trace segmentation with $w = 3$.

The effectiveness of trace segmentation is heavily dependent on the frequency of pattern recurrence and number of such patterns. For predicate sequences with infrequent or no repeating patterns, the difference in the number of constraints with and without segmentation is negligible. In the implementation of our algorithm we preprocess the predicate sequence to compute the number of constraints for automaton construction with and without trace segmentation and select the smaller of the two. Additionally, we introduce a second optimisation, incremental learning, to tackle algorithm scalability where trace segmentation is ineffective.

3.3.2 Incremental Learning

In this optimisation, instead of attempting to construct an automaton that conforms to the entire predicate sequence P, we incrementally construct an automaton that progressively conforms to finite prefixes of P of increasing length. This optimisation is inspired by previous work [10] where a set of traces are processed incrementally to generate a model, and it has been adapted to the SAT encoding used in our algorithm.

We use finite prefixes of P of increasing length as input in each iteration of incremental learning. Starting with the smallest prefix, we construct a candidate abstraction that conforms to the prefix. For each subsequent prefix of increasing length, we check to see if the previously generated automaton conforms to the input sequence. If yes, we move onto the next prefix. If not, we encode the prefix as additional constraints on the previously generated automaton such that the new automaton conforms to the input prefix, and repeat the SAT check. In each iteration i of incremental learning, the construction algorithm generates an automaton $\mathcal{M}_i$ such that $\mathcal{M}_i$ is the smallest automaton that satisfies the specified construction constraints and $\mathcal{L}(\mathcal{M}_{i-1}) \subseteq \mathcal{L}(\mathcal{M}_i)$. The optimisation yields a significant decrease in runtime for long traces. Further details are provided in Section 4.4, Chapter 4.

Generating models from a set of execution traces

The tracing setup may often yield a set of system execution traces rather than a single trace. We assume the system implementation used to collect traces can be reset and executed again to collect more traces. In this section we describe how the model-learning algorithm can be adapted to learn symbolic abstractions from a set of system execution traces thus obtained.

Given a set of system execution traces $\mathcal{T}$, we start by constructing a single long trace $\hat{\sigma}$ obtained by concatenating all traces in $\mathcal{T}$. Each trace is separated by a delimiter in $\hat{\sigma}$. The transition predicate synthesis step constructs a sequence of predicates as described in Section 3.2.2 from trace $\hat{\sigma}$. The delimiters that separate each trace in $\hat{\sigma}$ are used to divide the constructed predicate sequence into consecutive segments, one corresponding to each trace in $\mathcal{T}$. This gives rise to a set of transition predicate sequences that serve as input to the automaton construction step.

For automaton construction, we process each predicate sequence in the set one at a time in incremental calls to the construction algorithm. In the algorithm described in Section 3.2.3, we do not explicitly specify an initial automaton state; the initial state would correspond to whatever value is assigned to the symbolic state variable q_0 by the SAT solver. But for constructing an automaton from a set of traces, since each predicate sequence in the set corresponds to an execution trace obtained after system reset, we need to ensure that each predicate sequence is captured in the automaton as a finite path starting from an initial automaton state. To this end, we explicitly set the initial automaton state to be represented by the value 1 by extending the constraints 3.8 and 3.9 to include the constraint $q_0 = 1$. The remaining symbolic state variables can take any value between 1 and N.

In each iteration of the automaton construction algorithm we employ both performance optimizations—trace segmentation and incremental learning—to enable the algorithm to scale. The final generated system abstraction conforms to all predicate sequences in the set and therefore accepts all traces in $\mathcal{T}$.

This approach can also be applied in a setting where given a system abstraction $\mathcal{M}$ inferred from a trace set $\mathcal{T}$, and a set of additional system execution traces $\mathcal{T}'$, we wish to learn an abstraction that accepts all traces in $\mathcal{T} \cup \mathcal{T}'$. Here, rather than repeating the model generation process with $\mathcal{T} \cup \mathcal{T}'$ as input to the model-learning algorithm, the system behaviours exemplified by the traces in $\mathcal{T}'$ can be incrementally added to the inferred abstraction $\mathcal{M}$. To this end, we start by constructing a set of transition predicate sequences from $\mathcal{T}'$, and then process each predicate sequence in the set in incremental calls to the construction algorithm as described above. As an additional optimisation step, we can check which traces in $\mathcal{T}'$ are already accepted by $\mathcal{M}$ and process only the non-accepting traces for incremental learning.

Chapter 4

Experimental Evaluation

In this chapter we present an empirical evaluation of the model-learning algorithm described in Chapter 3. For our experiments we use three sets of benchmarks from different domains, including system-on-chip (SoC) and internet-of-things (IoT). We report the results of a series of ablation tests to evaluate algorithm performance, with and without each of the algorithmic optimizations described in Section 3.3, Chapter 3, to expose their individual contribution to improving algorithm runtime.

We also report on experiments that compare the algorithm with the most prominent existing methods, state-merge and query-based learning, as well as a machine learning approach using contemporary recurrent neural network technology. A detailed account of these algorithms is provided in Chapter 2. Our experiments compare the algorithms on their ability to generate symbolic abstractions, as well as algorithm runtime, the size of automaton produced, and the accuracy of the system behaviours captured by the generated system abstractions.

Plan of the chapter

The rest of the chapter is organized as follows. Section 4.1 provides a detailed description of the benchmarks used for our evaluation. Section 4.2 describes the implementation of the algorithm and the experimental setup used for the evaluation. Section 4.3 provides an overview of the generated models and the experimental results. In Section 4.4 we discuss the benefits of the optimizations introduced in our algorithm using empirical data. Section 4.5 details the results of our comparison experiments with state-merge, query-based learning and machine learning algorithms.

4.1 Benchmarks

We evaluate the model-learning algorithm using three sets of benchmarks:

SoC: This is a set of five benchmarks from the SoC domain. Four of the five benchmarks use the QEMU[1] virtual platform [14] to emulate an x86 system that includes hardware components. The virtual platform runs a full CentOS Linux distribution and is given an application to exercise system behaviours of interest. These benchmarks are used to evaluate the capability of the model-learning algorithm in a realistic setting. The last example is artificial and enables us to benchmark particular aspects of the method. Details of all five benchmarks are provided below:

USB xHCI Slot State Machine The Extensible Host Controller Interface (xHCI) [65] specifies a controller's register-level operations for USB 2.0 and above. Here, we look specifically at slot-level operations that take place when we access a virtual USB storage device as implemented in QEMU x86 platform emulation.

QEMU USB Interface Emulation We use the same setup as above but record all interface events that take place when a virtual USB storage device is attached to the virtual platform. The resulting trace records the series of ring fetch and ring write operations between the command ring and event ring of the xHCI protocol.

QEMU Serial I/O Port We used QEMU's x86 emulation of a serial I/O port and recorded changes in queue length. The trace contains (numerical) queue length data and the queue size 'limit', along with (Boolean) *read*, *reset* and *write* events on the queue. In our experiments, we aim to generate an abstraction of the serial I/O port modelling updates to the queue length under the influence of the events on the queue.

Real Time Linux Kernel We generated an automaton describing the behaviour of threads in the Linux PREEMPT_RT kernel on a single core system. This is motivated by work in [34, 35] where hand-drawn models of the kernel are used as monitors for runtime kernel verification. For this example, we used the built-in Linux tracing infrastructure *ftrace* to trace scheduler-related calls made by the thread under analysis, as described in [35]. We used the Linux PREEMPT_RT kernel version 5.0.7-rt5 on a single core QEMU emulated x86 machine for our experiments.

Integrator Control applications frequently track an integral of an input signal *ip*. We implement an anti-windup integrator where the computed output *op* is saturated at

[1]https://www.qemu.org/

predefined thresholds, ‘upper_limit’ and ‘lower_limit’. The trace contains valuations of (ip, op) pairs at discrete steps of system execution.

Log Data: The second set of benchmarks is obtained from [19]. Here, trace data is generated from 10 finite-state machine models using a trace generator [78]. The traces range from log data pertaining to SSH and TCP protocols to Java API logs, and are comprised of sequences of Boolean events. The complete set of benchmarks is publicly available in [20]. These benchmarks are used to evaluate algorithm scalability.

AWS IoT: The third benchmark set is from the IoT domain. Amazon Web Services (AWS) IoT uses manually specified *detector models* [96] to monitor customer IoT systems and raise alarms. Each detector model defines a set of operation modes. In our experiments we aim to generate the detector models automatically using trace data by modelling how the system switches between these operation modes. We use C implementations of 5 publicly available detector models described below to generate traces and attempt to reverse-engineer the models from these traces. These benchmarks are used to evaluate the predicate synthesis component and the ability of the algorithm to generate expressive symbolic abstractions.

Heartbeat Monitoring This detector model switches between two modes of operation: NORMAL and OFFLINE. When an input less than 0 is received, the system starts a timer. If the timer expires before receiving an input signal greater than 0, the model moves to the OFFLINE mode and raises an alert. The model switches back to the NORMAL mode on receiving an input greater than 0.

HVAC Temperature Control The temperature control detector model for Heating, Ventilation and Air Conditioning (HVAC) receives temperature data from sensors and the user can set a desired temperature. The model switches between the operation modes IDLE, COOLING and HEATING depending on the temperature reading. To protect the HVAC system from damage an additional pair of timers and corresponding timeouts are defined for the cooling and heating operations.

IoT Sensor This detector model receives inputs from both a sensor device and a software application. It has three operation modes: DEVICE_IDLE, DEVICE_IN_USE and DEVICE_EXCEPTION. The device provides a GPS position and a device ID as input to the model, which in turn checks the values against corresponding inputs received from the software application to trigger relevant actions.

Simple Alarm This detector model is used to detect if a monitored device is in an alarming state. It has three operation modes: NORMAL, ALARMING and SNOOZE. The model raises an alarm when the input signal exceeds a predefined threshold.

ISA Alarm This detector model is used to detect if a monitored device is in an alarming state in accordance to ISA 18.2. It has 7 operation modes: NORMAL, UNACK, ACK, RTN_UNACK, OOS, SHEL, SUPR. Unlike *Simple Alarm* described above, the model checks the input against predefined upper and lower thresholds.

4.2 Implementation Details and Experimental Setup

The model-learning algorithm is implemented as a tool Trace2Model (T2M). We use CVC4 v1.9 [13] implementing SyGuS as the program synthesis tool for transition predicate synthesis. For automaton construction we define the automaton and encode the construction constraints in a C program and use the C Bounded Model Checker (CBMC v5.11) [29] as our solver. The codebase for the T2M tool and the benchmarks used in this work can be found in [68].

In the following sections we discuss specific aspects of the T2M tool and the experimental setup.

4.2.1 Selecting Window Size for Trace Segmentation

For model learning we would ideally like to choose a value for w that results in the smallest number of constraints for automaton construction. It is however challenging to determine an optimal value for w a priori. Although choosing $w = 1$ will result in the smallest number of constraints, equal to the number of unique predicates appearing in P, patterns of length 1 do not characterise any sequential system behaviour. The corresponding constraints for automaton construction only ensure that there is at least one transition in the automaton labelled with each predicate appearing in P.

One strategy is to approximate the number of constraints for windows of different sizes by preprocessing the predicate sequence. We can then choose the window size that produces the smallest number of constraints. But for very long sequences this may cause unnecessary overheads. The strategy we adopt for our experiments is to fix the window size $w = 3$, which is small enough to capture interesting sequential behaviour and produce a reasonably small number of constraints.

4.2.2 Tuning Precision of Generated Models

As described in Section 3.2.3, Chapter 3, we use a tunable parameter l to control the precision of the generated abstraction during automaton construction. To better

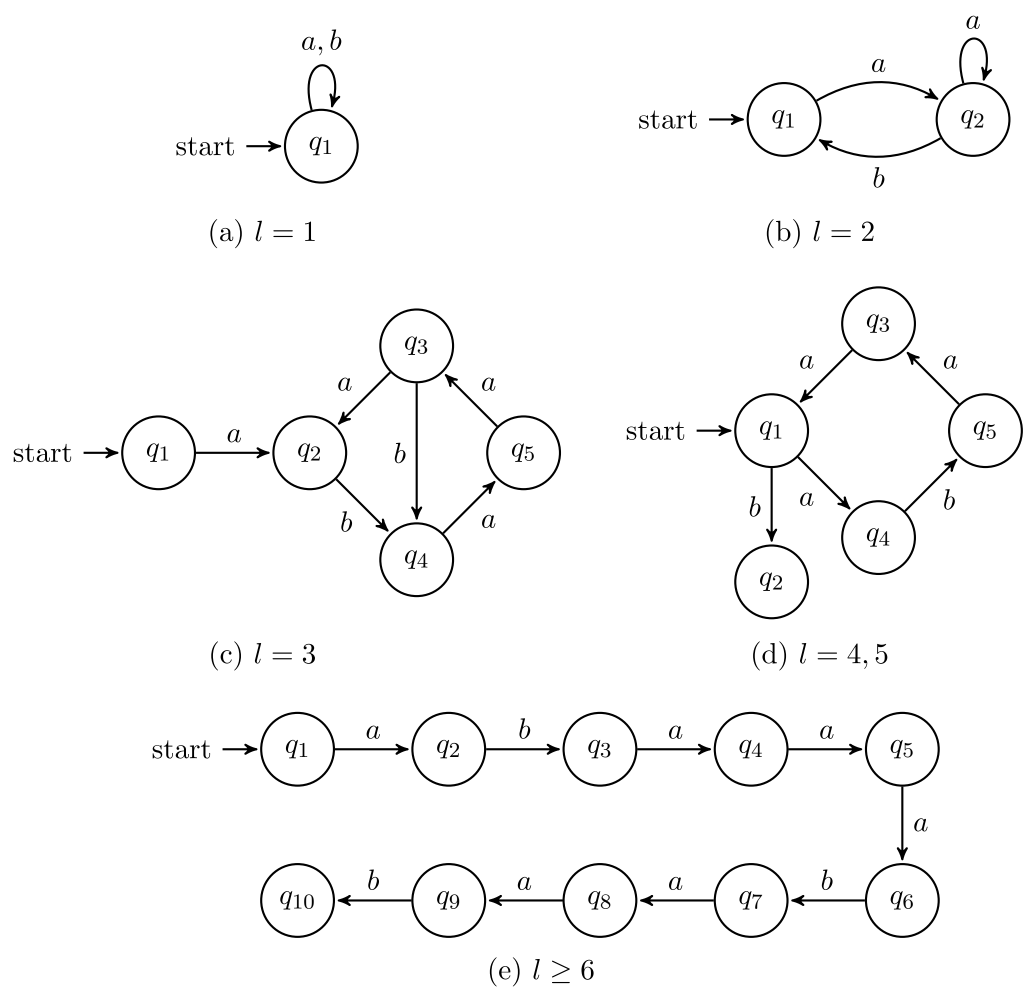


Figure 4.1: Models generated for different l values for trace $\sigma = a, b, a, a, a, b, a, a, b$

understand the effect of l on automaton construction, we performed a series of experiments where we generated abstractions for a simple trace $\sigma = a, b, a, a, a, b, a, a, b$ with increasing values for l starting with $l = 1$. We set a timeout of 5 h for our experiments. We recorded the algorithm runtime and automaton size. The generated abstractions are illustrated in Fig. 4.1.

As is evident from the runtime plot in Fig. 4.2, algorithm runtime increases almost exponentially for increasing values of l. This is because a higher l value implies tighter constraints on the automaton to be constructed, thereby increasing the complexity of the SAT problem. For $l \geq 8$ the algorithm timed out before generating an abstraction.

A higher value for l generates a more exact representation of the trace as is evident from Fig. 4.1. The generated abstractions become progressively more precise with increasing l, similar to the parameter k in k-Tails. For model learning using k-Tails, it is common practice to use a k value between 2 and 4 [80]. For our experiments we

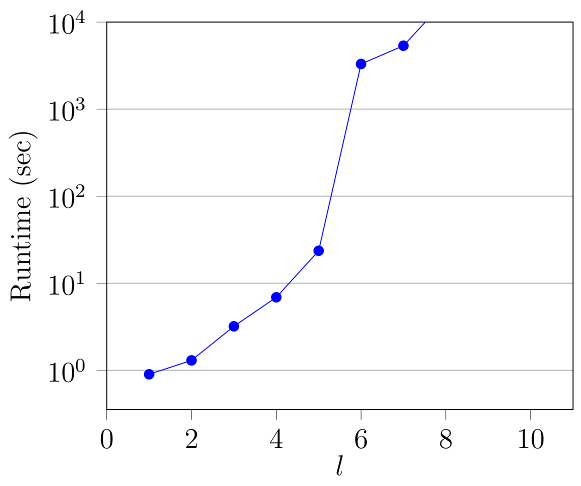


Figure 4.2: Runtime plot (log scale) demonstrating the effect of increasing l values on algorithm runtime for generating an abstraction from trace $\sigma = a, b, a, a, a, b, a, a, b$.

use $l = 2$ to ensure that the problem is not too complex to solve but at the same time the learned automaton is not overgeneralised to fit the trace.

4.2.3 Discussion of Program Synthesis Engines

We experimented with two program synthesis tools for generating transition predicates for the automaton: CVC4 version 1.9 [13], which by default employs SyGuS with CEGIS, and fastsynth, which is based on the work done in [4].

The SyGuS-based approach requires a grammar for the program. The key effort is to determine the constants that are required; often they have to be adjusted manually for every model. An alternative is to automatically sample valuations from trace data to be used as constants in the grammar using data sampling techniques. Fastsynth also implements CEGIS but does not rely on a user-specified grammar to restrict the search space. Fastsynth ignores any grammar given as part of the problem and produces the smallest function that satisfies the constraints. Any constants that may be required are generated automatically.

CVC4 also implements an alternative algorithm that does not require syntax guidance; however, that produces trivial solutions. For example, given the trace sequence $\sigma = 1, 2, 4, 8$ for $X = \{x\}$, CVC4 generates the expression

$$(\text{ite}\,(=x\,4)\,8\,(\text{ite}\,(\text{not}\,(=x\,2))\,2\,4)) \tag{4.1}$$

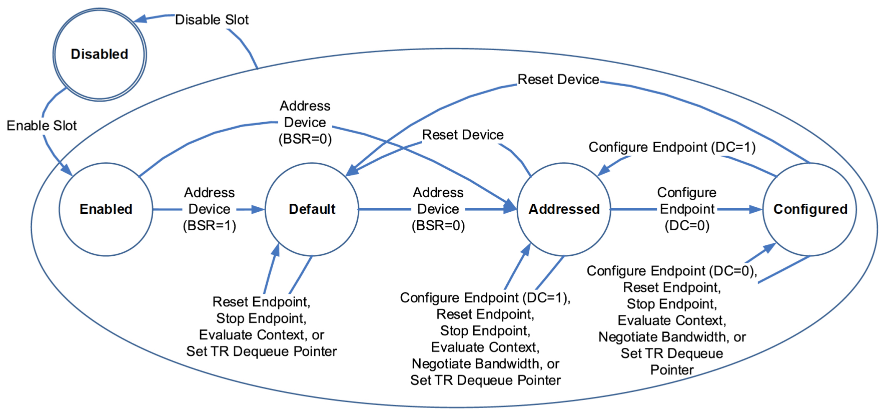
for the data update function f_x, whereas fastsynth produces the expression

$$x + x \tag{4.2}$$

which is a better fit for our problem. The type of transition predicates synthesised and as a consequence the expressiveness of generated models depends on the ability of the program synthesis tool and the approach to synthesis. A suitable synthesis tool can be chosen based on the target models we wish to obtain and the application domain.

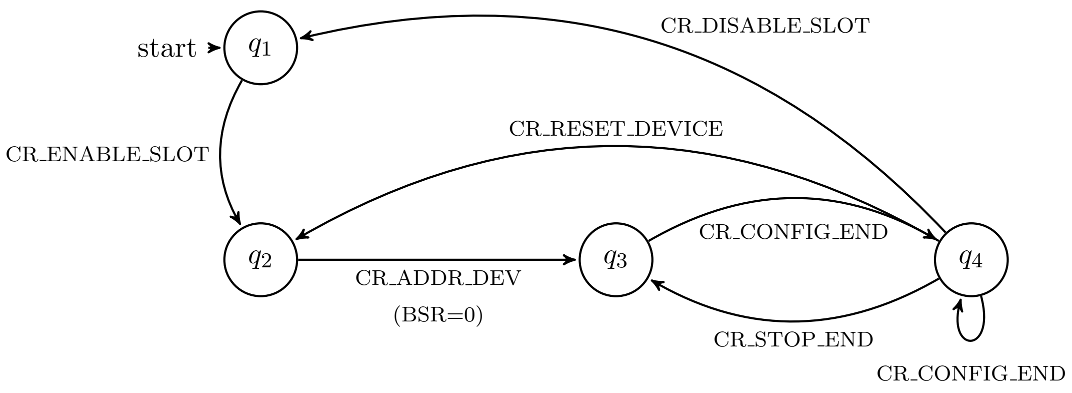
4.3 Results

SoC Benchmark Set The system models generated by T2M for the SoC benchmark set are provided in Fig. 4.3–4.7. For *USB xHCI Slot State Machine*, our algorithm learns the automaton provided in Fig. 4.3b, resembling the Intel datasheet diagram in Fig. 4.3a. It is worth noting that the model-learning algorithm is able to generate an accurate representation of system behaviours that are exercised under a given application load. Some transitions in Fig. 4.3a do not appear in the learned



Disabled
Disable Slot
Enable Slot
Enabled
Address Device (BSR=0)
Address Device (BSR=1)
Default
Reset Device
Reset Device
Address Device (BSR=0)
Addressed
Configure Endpoint (DC=1)
Configure Endpoint (DC=0)
Configured
Reset Endpoint, Stop Endpoint, Evaluate Context, or Set TR Dequeue Pointer
Configure Endpoint (DC=1), Reset Endpoint, Stop Endpoint, Evaluate Context, Negotiate Bandwidth, or Set TR Dequeue Pointer
Configure Endpoint (DC=0), Reset Endpoint, Stop Endpoint, Evaluate Context, Negotiate Bandwidth, or Set TR Dequeue Pointer

(a)



(b)

Figure 4.3: USB Slot state machine provided in (a) Intel datasheet [65] and (b) automaton learnt by our approach.

model because either QEMU does not implement those behaviours or the application load does not drive the system into those scenarios. The abstractions generated thus provide valuable coverage information.

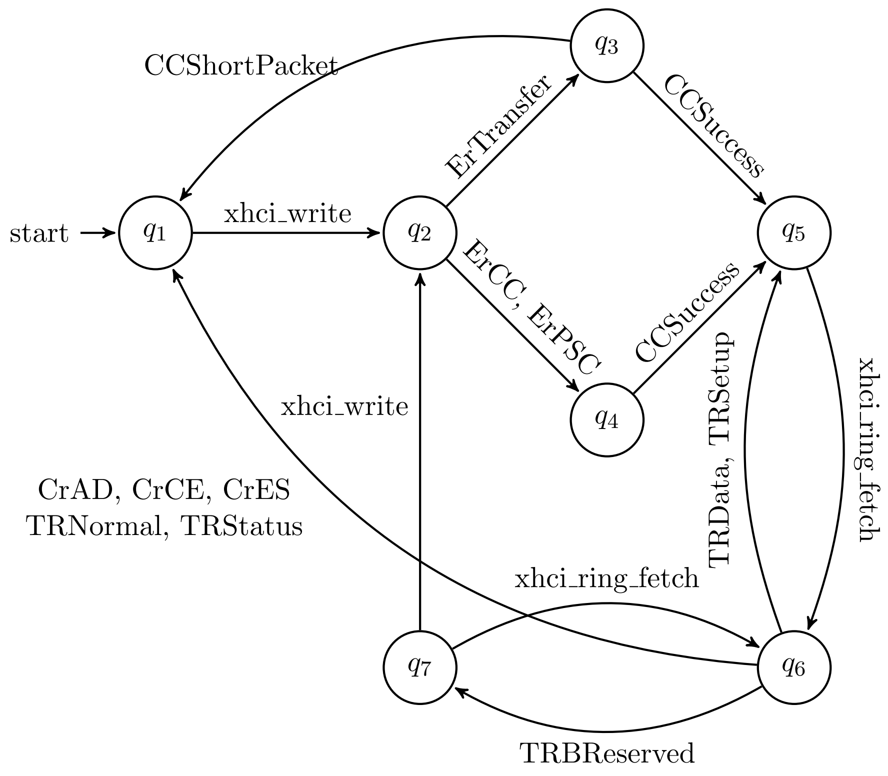


Figure 4.4: Model of USB interface for attach operation.

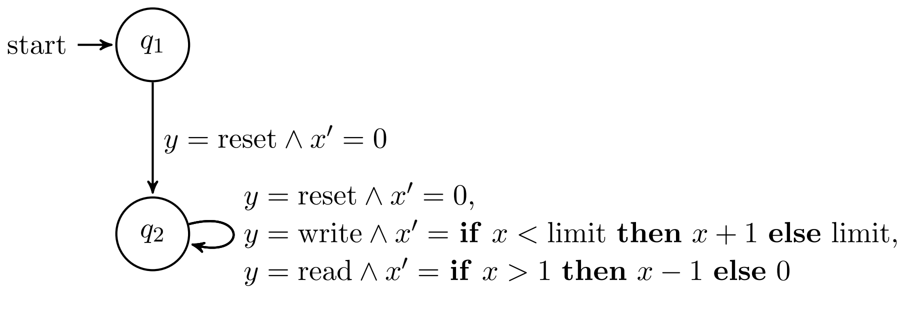

start
q_1
$y = \text{reset} \wedge x' = 0$
q_2
$y = \text{reset} \wedge x' = 0,$
$y = \text{write} \wedge x' = \textbf{if } x < \text{limit } \textbf{then } x + 1 \textbf{ else } \text{limit},$
$y = \text{read} \wedge x' = \textbf{if } x > 1 \textbf{ then } x - 1 \textbf{ else } 0$

Figure 4.5: Model of QEMU Serial I/O Port.

For *QEMU Serial I/O Port*, our algorithm was able to infer data update expressions that accurately represent how the queue length x changes under the influence of the *read*, *write* and *reset* events on the queue, as illustrated in Fig. 4.5.

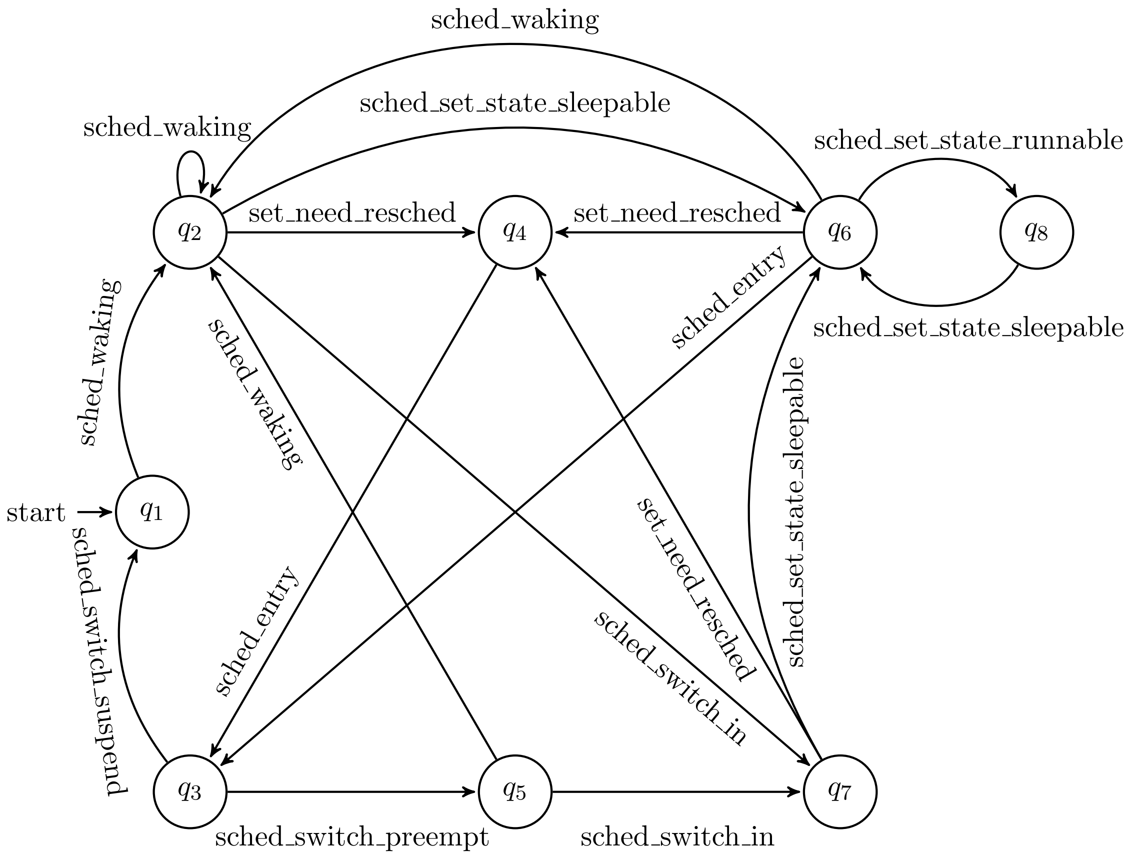


Figure 4.6: Model of RT-Linux Kernel Thread Scheduling.

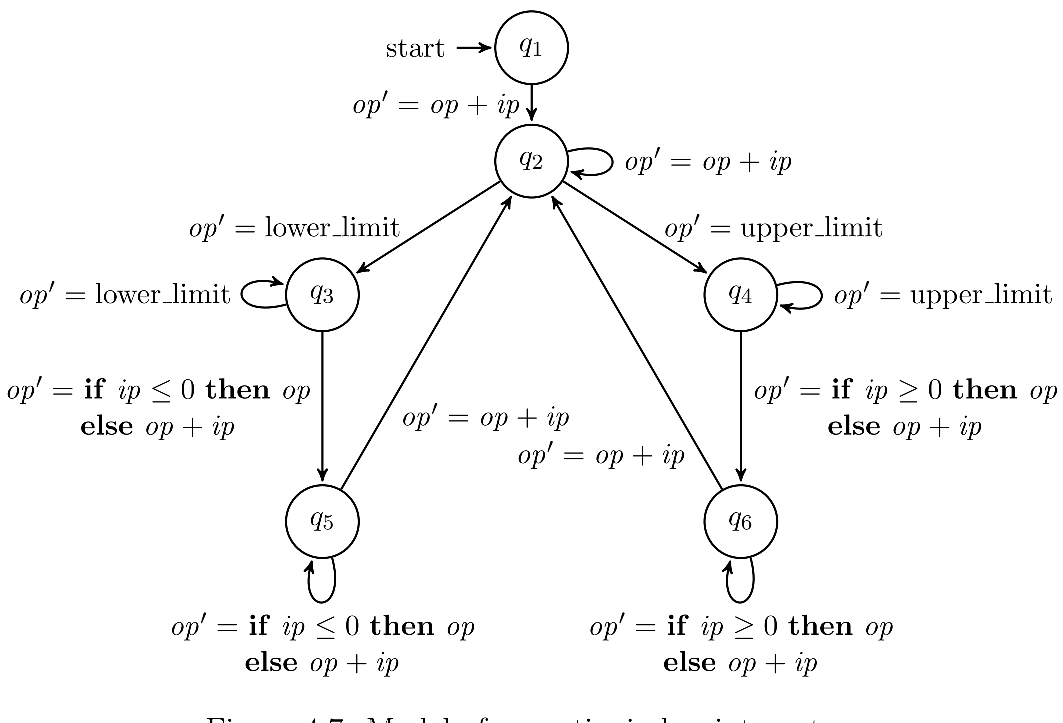

start
q_1
$op' = op + ip$
q_2
$op' = op + ip$
$op' =$ lower_limit
$op' =$ upper_limit
$op' =$ lower_limit
q_3
q_4
$op' =$ upper_limit
$op' =$ **if** $ip \leq 0$ **then** op **else** $op + ip$
$op' = op + ip$
$op' = op + ip$
$op' =$ **if** $ip \geq 0$ **then** op **else** $op + ip$
q_5
q_6
$op' =$ **if** $ip \leq 0$ **then** op **else** $op + ip$
$op' =$ **if** $ip \geq 0$ **then** op **else** $op + ip$

Figure 4.7: Model of an anti-windup integrator.

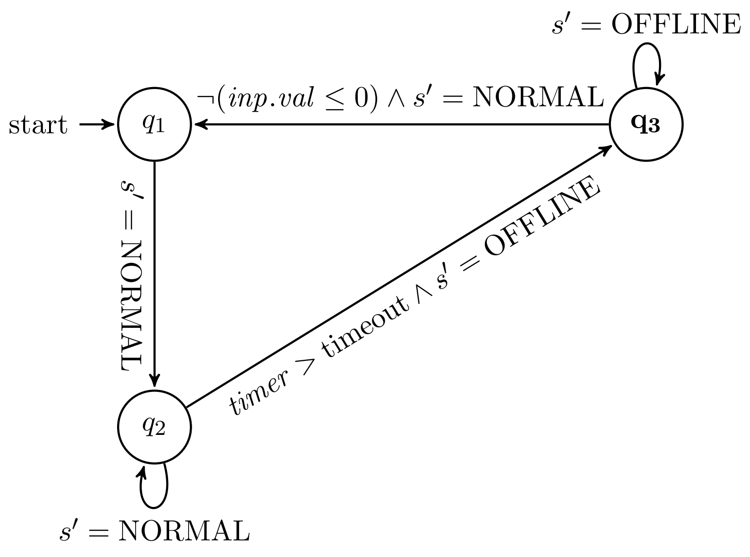
Initial attempts at modelling RT-Linux Kernel thread behaviour with our algorithm, using the *pi_stress* tests from the *rt-tests* suite as system load, revealed that some states in the hand-drawn model provided in [35] are not covered by the given load. On running an additional kernel module to cover these corner cases, we obtain the automaton in Fig. 4.6. This experiment provides evidence in support of potentially using the models learned by our algorithm for functional test coverage analysis.

For the *Integrator* benchmark, the algorithm generates the system model provided in Fig. 4.7. The transitions labelled with predicate $op' = op + ip$ encode integrator behaviour outside of saturation. In the learned abstraction, the finite paths $q_2 \to q_3 \to q_5$ and $q_2 \to q_4 \to q_6$ accurately capture saturation behaviour at the lower and upper thresholds respectively.

Log Data Benchmark Set These benchmarks are used to evaluate algorithm scalability. This is discussed in detail in Sections 4.4 and 4.5.

Using a combination of trace segmentation and incremental learning, T2M is able to generate concise models in less than 1 h for 9 out of 10 benchmarks. For the *java.net.DatagramSocket* benchmark, T2M takes a little over an hour to generate a model with 64 states.

AWS IoT Benchmark Set For the *AWS IoT* benchmarks, T2M is able to generalise over multiple system variables of different datatypes to generate transition predicates that accurately capture system behaviour, such as the model learned for the



$s' = \text{OFFLINE}$
$\neg(inp.val \leq 0) \wedge s' = \text{NORMAL}$
start
q_1
$\mathbf{q_3}$
$s' = \text{NORMAL}$
$timer > \text{timeout} \wedge s' = \text{OFFLINE}$
q_2
$s' = \text{NORMAL}$

Figure 4.8: Model of an IoT Heartbeat Monitoring system

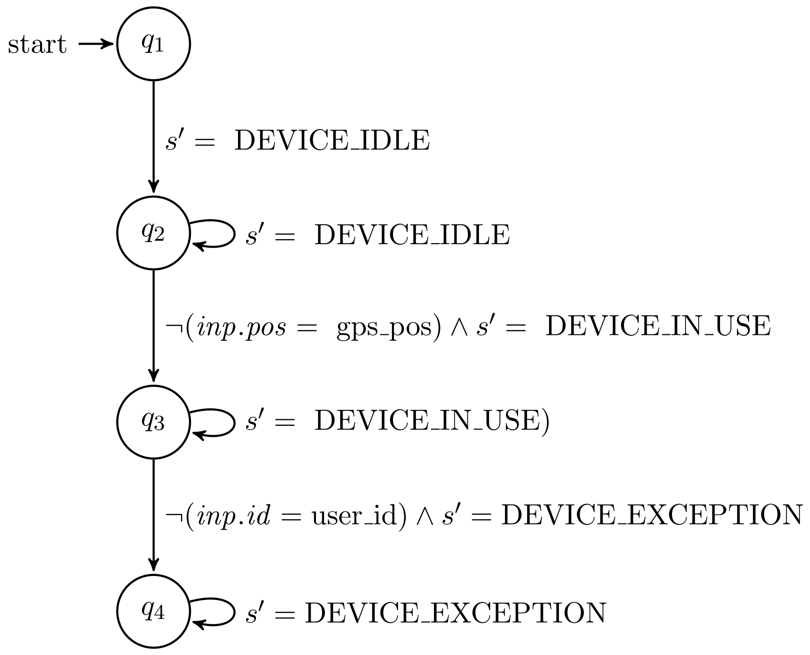

start
q_1
$s' =$ DEVICE_IDLE
q_2
$s' =$ DEVICE_IDLE
$\neg(inp.pos =$ gps_pos$) \wedge s' =$ DEVICE_IN_USE
q_3
$s' =$ DEVICE_IN_USE)
$\neg(inp.id =$ user_id$) \wedge s' =$ DEVICE_EXCEPTION
q_4
$s' =$ DEVICE_EXCEPTION

Figure 4.9: Model of IoT Sensor system.

IoT Sensor benchmark in Fig. 4.9. The models generated for 4 out of 5 benchmarks are provided in Fig. 4.8–4.11. The model generated for the *ISA Alarm* benchmark has 19 states. We have therefore provided only the inferred transition guards in Table. 4.1 describing switching behaviour between operation modes.

The IoT benchmarks are unique in that the detector models we attempt to reverse-engineer from traces data feature priorities on the transition edges. T2M is able to automatically infer transition priority from trace data and incorporate it into the transition predicates for the generated abstraction.

For instance, in the detector model implementation for *ISA Alarm*, transitions to operation modes UNACK and NORMAL from operation mode ACK have the following transition guards with priorities 1 and 2 respectively.

$$\mathit{snooze\_time} > \text{snooze\_timeout} \tag{4.3}$$

$$\mathit{inp.cmd} = \text{RESET} \tag{4.4}$$

The corresponding guards inferred by T2M from trace data are (Table 4.1)

$$g_{s,\text{ACK}\rightarrow\text{UNACK}} \coloneqq \mathit{snooze\_time} > \text{snooze\_timeout} \tag{4.5}$$

$$g_{s,\text{ACK}\rightarrow\text{NORMAL}} \coloneqq \neg(\mathit{snooze\_time} > \text{snooze\_timeout}) \wedge \mathit{inp.cmd} = \text{RESET} \tag{4.6}$$

Here, $g_{s,\text{ACK}\rightarrow\text{NORMAL}}$ evaluates to *true* only if $g_{s,\text{ACK}\rightarrow\text{UNACK}}$ is not satisfied, thereby accurately capturing transition priority order.

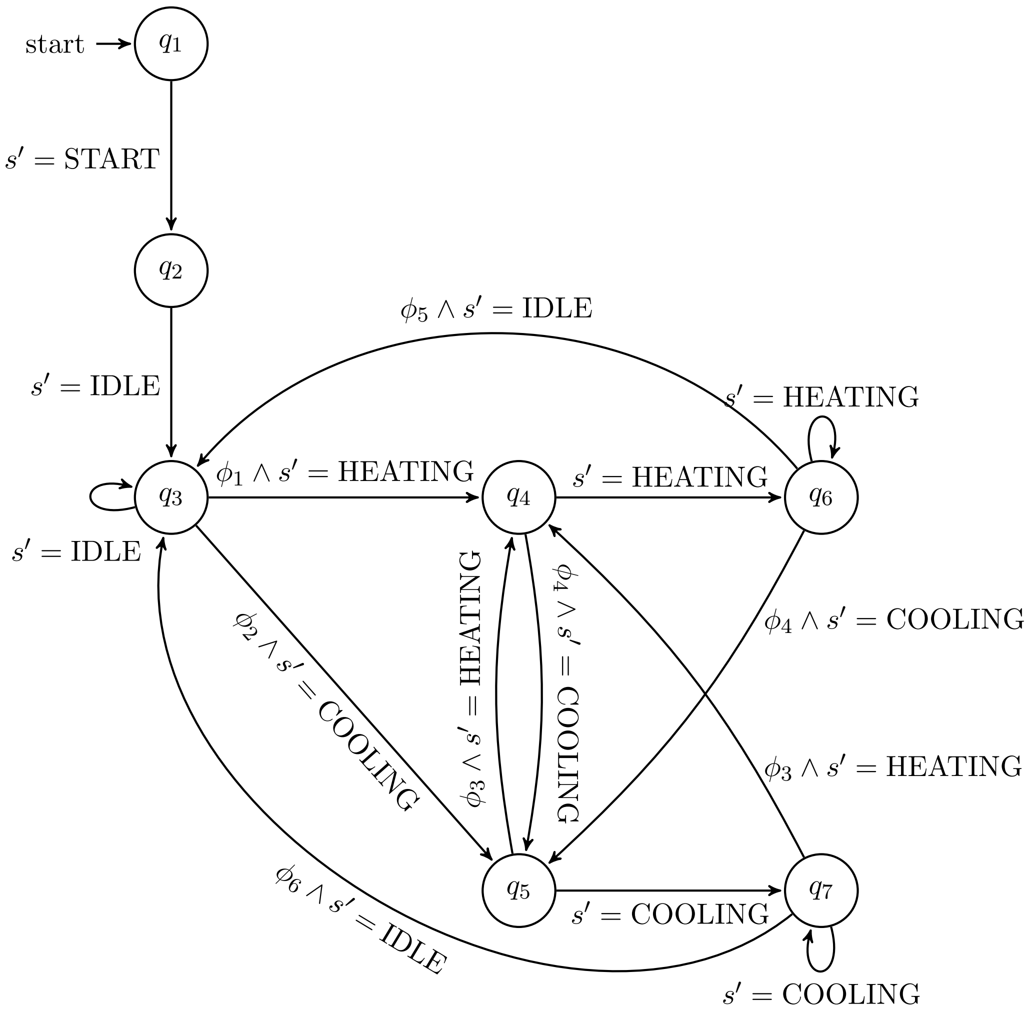
Table 4.1: Transition guards inferred from traces for the IoT ISA Alarm system using program synthesis.

Mode	Next Mode	Inferred Transition Guard
NORMAL	UNACK	$\mathit{inp.val} > \text{high\_thresh} \lor (\neg(\mathit{inp.val} > \text{high\_thresh}) \land \neg(\mathit{inp.val} \geq \text{low\_thresh}))$
	SUPR	$\mathit{inp.cmd} = \text{c\_SUPR} \land \neg(\mathit{inp.val} > \text{high\_thresh}) \land \mathit{inp.val} \geq \text{low\_thresh}$
	OOS	$\mathit{inp.cmd} = \text{REMOVE} \land \neg(\mathit{inp.val} > \text{high\_thresh}) \land \mathit{inp.val} \geq \text{low\_thresh}$
	SHEL	$\mathit{inp.cmd} = \text{c\_SHEL} \land \neg(\mathit{inp.val} > \text{high\_thresh}) \land \mathit{inp.val} \geq \text{low\_thresh}$
UNACK	RTN_UNACK	$\mathit{inp.val} \geq \text{low\_thresh} \land \neg(\mathit{inp.val} > \text{high\_thresh}) \land \neg(\mathit{inp.cmd} = \text{c\_ACK})$
	ACK	$\mathit{inp.cmd} = \text{c\_ACK}$
	SUPR	$(\mathit{inp.cmd} = \text{c\_SUPR} \land \mathit{inp.val} \geq \text{low\_thresh} \land \mathit{inp.val} > \text{high\_thresh}) \lor$ $(\mathit{inp.cmd} = \text{c\_SUPR} \land \neg(\mathit{inp.val} \geq \text{low\_thresh}))$
	OOS	$(\mathit{inp.cmd} = \text{REMOVE} \land \mathit{inp.val} \geq \text{low\_thresh} \land \mathit{inp.val} > \text{high\_thresh}) \lor$ $(\mathit{inp.cmd} = \text{REMOVE} \land \neg(\mathit{inp.val} \geq \text{low\_thresh}))$
	SHEL	$(\mathit{inp.cmd} = \text{c\_SHEL} \land \mathit{inp.val} \geq \text{low\_thresh} \land \mathit{inp.val} > \text{high\_thresh}) \lor$ $(\mathit{inp.cmd} = \text{c\_SHEL} \land \neg(\mathit{inp.val} \geq \text{low\_thresh}))$
ACK	UNACK	$\mathit{snooze\_timer} > \text{snooze\_timeout}$
	NORMAL	$\mathit{inp.cmd} = \text{RESET} \land \neg(\mathit{snooze\_timer} > \text{snooze\_timeout})$
	SUPR	$\mathit{inp.cmd} = \text{c\_SUPR} \land \neg(\mathit{snooze\_timer} > \text{snooze\_timeout})$
	OOS	$\mathit{inp.cmd} = \text{REMOVE} \land \neg(\mathit{snooze\_timer} > \text{snooze\_timeout})$
	SHEL	$\mathit{inp.cmd} = \text{c\_SHEL} \land \neg(\mathit{snooze\_timer} > \text{snooze\_timeout})$
RTN_UNACK	UNACK	$\mathit{inp.val} > \text{high\_thresh} \lor (\neg(\mathit{inp.val} > \text{high\_thresh}) \land \neg(\mathit{inp.val} \geq \text{low\_thresh}))$
	NORMAL	$\mathit{inp.cmd} = \text{c\_ACK} \land \neg(\mathit{inp.val} > \text{high\_thresh}) \land \mathit{inp.val} \geq \text{low\_thresh}$
	SUPR	$\mathit{inp.cmd} = \text{c\_SUPR} \land \mathit{inp.val} \geq \text{low\_thresh} \land \neg(\mathit{inp.val} > \text{high\_thresh})$

Continued on next page

Table 4.1 – *Continued from previous page*

Mode	Next Mode	Inferred Transition Guard
RTN_UNACK	OOS	$inp.cmd = \text{REMOVE} \wedge inp.val \geq \text{low\_thresh} \wedge \neg(inp.val > \text{high\_thresh})$
	SHEL	$inp.cmd = \text{c\_SHEL} \wedge inp.val \geq \text{low\_thresh} \wedge \neg(inp.val > \text{high\_thresh})$
SUPR	NORMAL	$inp.cmd = \text{UNSUPR} \wedge prev\_s = \text{NORMAL}$
	UNACK	$inp.cmd = \text{UNSUPR} \wedge prev\_s = \text{UNACK}$
	RTN_UNACK	$inp.cmd = \text{UNSUPR} \wedge prev\_s = \text{RTN\_UNACK}$
	ACK	$inp.cmd = \text{UNSUPR} \wedge \neg(prev\_s = \text{UNACK}) \wedge \neg(prev\_s = \text{RTN\_UNACK}) \wedge \neg(prev\_s = \text{NORMAL})$
OOS	NORMAL	$inp.cmd = \text{ADD} \wedge prev\_s = \text{NORMAL}$
	UNACK	$inp.cmd = \text{ADD} \wedge prev\_s = \text{UNACK}$
	RTN_UNACK	$inp.cmd = \text{ADD} \wedge prev\_s = \text{RTN\_UNACK}$
	ACK	$inp.cmd = \text{ADD} \wedge \neg(prev\_s = \text{UNACK}) \wedge \neg(prev\_s = \text{RTN\_UNACK}) \wedge \neg(prev\_s = \text{NORMAL})$
SHEL	NORMAL	$inp.cmd = \text{UNSHEL} \wedge prev\_s = \text{NORMAL}$
	UNACK	$inp.cmd = \text{UNSHEL} \wedge prev\_s = \text{UNACK}$
	RTN_UNACK	$inp.cmd = \text{UNSHEL} \wedge prev\_s = \text{RTN\_UNACK}$
	ACK	$inp.cmd = \text{UNSHEL} \wedge \neg(prev\_s = \text{UNACK}) \wedge \neg(prev\_s = \text{RTN\_UNACK}) \wedge \neg(prev\_s = \text{NORMAL})$

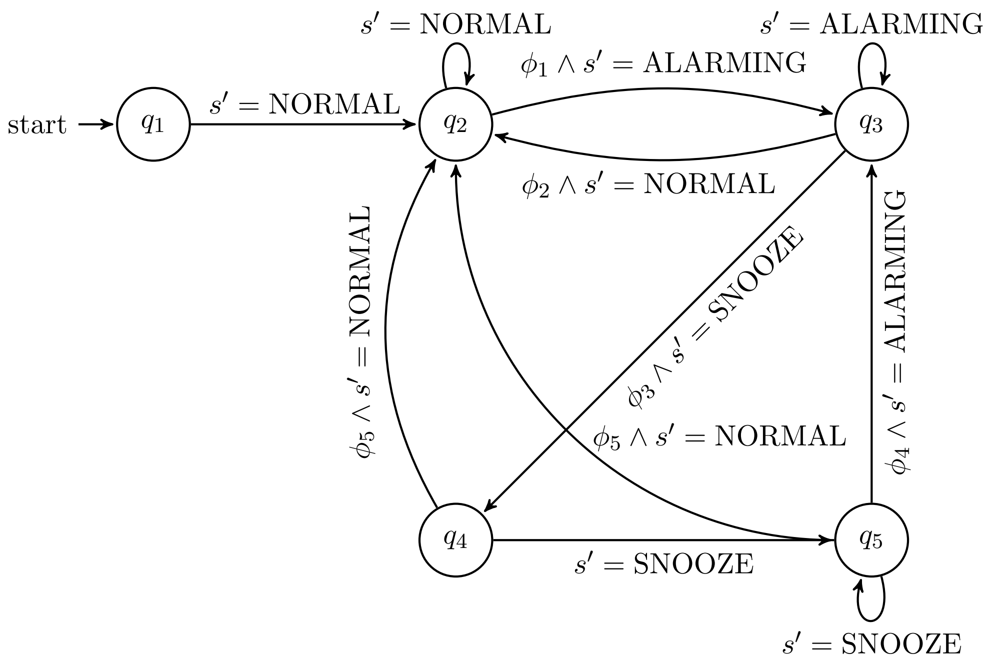


start
q_1
$s' = \text{START}$
q_2
$s' = \text{IDLE}$
$\phi_5 \wedge s' = \text{IDLE}$
$s' = \text{HEATING}$
q_3
$\phi_1 \wedge s' = \text{HEATING}$
q_4
$s' = \text{HEATING}$
q_6
$s' = \text{IDLE}$
$\phi_2 \wedge s' = \text{COOLING}$
$\phi_3 \wedge s' = \text{HEATING}$
$\phi_4 \wedge s' = \text{COOLING}$
$\phi_4 \wedge s' = \text{COOLING}$
$\phi_3 \wedge s' = \text{HEATING}$
$\phi_6 \wedge s' = \text{IDLE}$
q_5
$s' = \text{COOLING}$
q_7
$s' = \text{COOLING}$

$\phi_1 = g_{s,\text{IDLE}\to\text{HEATING}} \coloneqq \neg(\mathit{inp.temp} \geq \text{desiredTemp} - \text{allowedError})$
$\phi_2 = g_{s,\text{IDLE}\to\text{COOLING}} \coloneqq \mathit{inp.temp} > \text{desiredTemp} + \text{allowedError}$
$\phi_3 = g_{s,\text{COOLING}\to\text{HEATING}} \coloneqq \neg(\mathit{inp.temp} \geq \text{rangeLow})$
$\phi_4 = g_{s,\text{HEATING}\to\text{COOLING}} \coloneqq \mathit{inp.temp} > \text{rangeHigh}$
$\phi_5 = g_{s,\text{HEATING}\to\text{IDLE}} \coloneqq \mathit{heatingTimer} > \text{heatingTimeout} \wedge$
$\mathit{inp.temp} \geq \text{desiredTemp} + \text{allowedError} \wedge \neg(\mathit{inp.temp} > \text{rangeHigh})$
$\phi_6 = g_{s,\text{COOLING}\to\text{IDLE}} \coloneqq \mathit{coolingTimer} > \text{coolingTimeout} \wedge$
$\neg(\mathit{inp.temp} > \text{desiredTemp} - \text{allowedError}) \wedge \neg(\text{rangeLow} > \mathit{inp.temp})$

Figure 4.10: Model of IoT HVAC Temperature Control system.

Similar guards are generated for the *HVAC Temperature Control* and *Simple Alarm* benchmarks as well. For the *HVAC Temperature Control* benchmark a transition to the HEATING mode is prioritised over a transition to the IDLE mode from the COOLING mode. Similarly, a transition to the COOLING mode is prioritised over


start
q_1
q_2
q_3
q_4
q_5
$s' = \text{NORMAL}$
$s' = \text{NORMAL}$
$s' = \text{ALARMING}$
$\phi_1 \wedge s' = \text{ALARMING}$
$\phi_2 \wedge s' = \text{NORMAL}$
$\phi_5 \wedge s' = \text{NORMAL}$
$\phi_3 \wedge s' = \text{SNOOZE}$
$\phi_4 \wedge s' = \text{ALARMING}$
$\phi_5 \wedge s' = \text{NORMAL}$
$s' = \text{SNOOZE}$
$s' = \text{SNOOZE}$

$\phi_1 = g_{s,\text{NORMAL}\to\text{ALARMING}} \coloneqq \mathit{inp.val} > \text{thresh}$
$\phi_2 = g_{s,\text{ALARMING}\to\text{NORMAL}} \coloneqq \mathit{inp.cmd} = \text{RESET}$
$\phi_3 = g_{s,\text{ALARMING}\to\text{SNOOZE}} \coloneqq \mathit{inp.cmd} = \text{ACK}$
$\phi_4 = g_{s,\text{SNOOZE}\to\text{ALARMING}} \coloneqq \mathit{snooze\_time} > \text{snooze\_timeout}$
$\phi_5 = g_{s,\text{SNOOZE}\to\text{NORMAL}} \coloneqq \mathit{inp.cmd} = \text{RESET} \wedge$
$\neg(\mathit{snooze\_time} > \text{snooze\_timeout})$

Figure 4.11: Model of IoT Simple Alarm system.

a transition to the IDLE mode from the HEATING mode. This is captured by the transition guards $g_{s,\text{COOLING}\to\text{HEATING}}$ and $g_{s,\text{COOLING}\to\text{IDLE}}$, and $g_{s,\text{HEATING}\to\text{COOLING}}$ and $g_{s,\text{HEATING}\to\text{IDLE}}$ in Fig. 4.10, respectively.

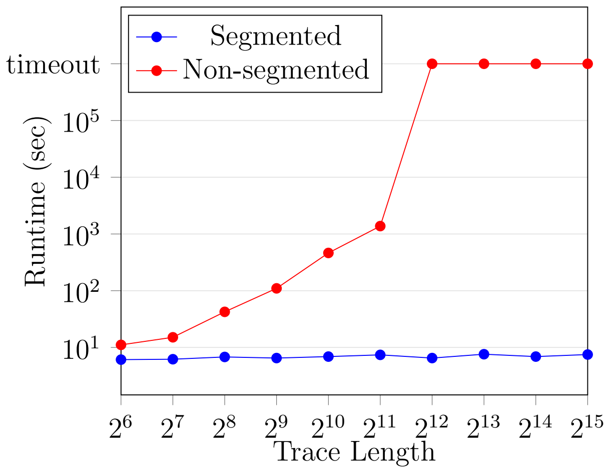
For the *Simple Alarm* benchmark a transition to the ALARMING mode is prioritised over a transition to the NORMAL mode from the SNOOZE mode. This is captured by the transition guards $g_{s,\text{SNOOZE}\to\text{ALARMIMG}}$ and $g_{s,\text{SNOOZE}\to\text{NORMAL}}$ in Fig. 4.11.

4.4 The Benefit of Trace Segmentation and Incremental Learning

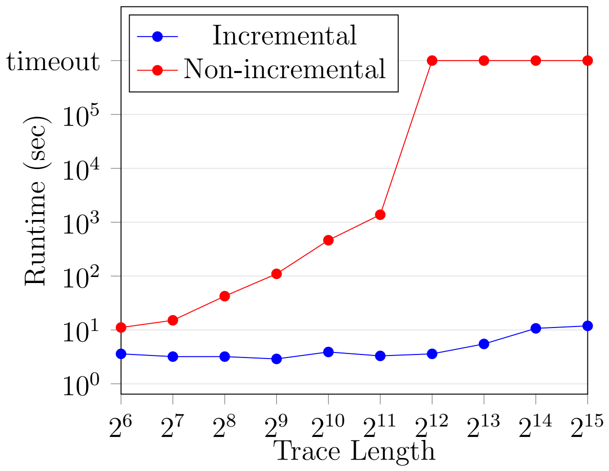
To learn models from execution traces we require efficient and scalable mechanisms for mining useful information from large amounts of trace data. The two algorithmic optimizations used in our model-learning algorithm—trace segmentation and incremental learning—as discussed in Section 3.3, Chapter 3, break down a large SAT problem into multiple small instances that have manageable runtime.

To evaluate the benefit of these optimizations, we give the results of a runtime comparison of the algorithm with and without each optimization: Fig. 4.12a is a plot of the runtime against trace length for segmented and non-segmented inputs, Fig. 4.12b is a runtime plot for incremental and non-incremental learning. For each comparison we use execution traces of the *Integrator* benchmark with exponentially increasing trace length. Both optimizations independently produce a significant reduction in algorithm runtime, with the incremental learning optimization producing a larger runtime reduction than trace segmentation.

We performed similar comparison experiments on all benchmarks. We set a time-



(a) Trace Segmentation



(b) Incremental Learning

Figure 4.12: Graph plot (log–log plot) comparing runtime for T2M with and without the optimizations for the *Integrator* benchmark.

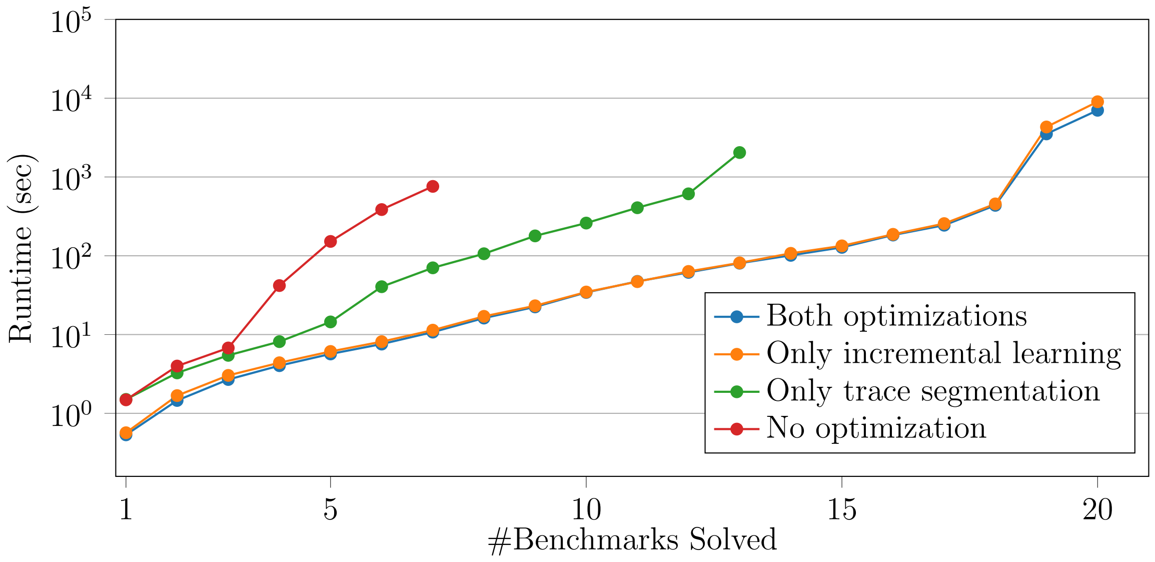


Figure 4.13: Cactus plot (log scale) for the T2M algorithm with and without optimizations for all benchmarks.

out of ≈ 3 h for each benchmark. The cactus plot representing algorithm performance with and without the optimizations is provided in Fig. 4.13. A combination of both optimizations produces the best results, with the algorithm generating system models for more benchmarks in a given time frame as compared to other algorithm settings.

We observe that algorithm performance with only the incremental learning optimization is very close to the performance obtained using both optimizations for the benchmarks used in our evaluation. But, this is not always true. For instance, when we attempt to learn an abstraction from an execution trace of length $\approx 2^{17}$ for the *Integrator* benchmark we observe that the algorithm takes 71.6 s using only incremental learning and 18.1 s using the combination of both optimizations. Here, trace segmentation produces an additional 75% decrease in algorithm runtime.

4.5 Comparison with Related Work

We experimentally compare our model-learning algorithm with relevant related work: state-merge, query-based learning using L* and an LSTM-based approach.

4.5.1 Learning Symbolic Abstractions

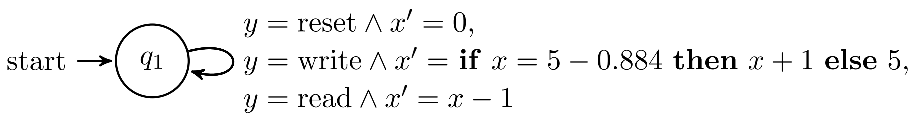
The predicate synthesis component of our model-learning algorithm enables T2M to generalise over multiple system variables of different types to learn symbolic system models. In this section we qualitatively compare the symbolic abstractions generated by T2M with the models generated by state-merge and query-based learning. The LSTM-based approach does not support inference of symbolic models.

Out of the benchmarks described in Section 4.1, we use the *QEMU Serial I/O Port*, *Integrator* and *AWS IoT* benchmarks for our comparison. System traces for the remaining benchmarks are sequences of only Boolean events. Since we wish to compare the algorithms on their ability to infer symbolic models, these benchmarks are not relevant. We use the complete set of benchmarks to quantitatively compare the algorithms on model size and accuracy, and algorithm runtime in Sections 4.5.2–4.5.4.

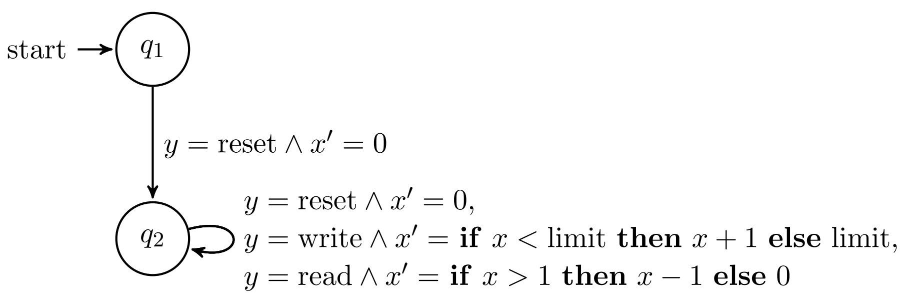
State-merge: Conventional state-merge algorithms, such as k-Tails, do not generate symbolic abstractions. They often fail to model how system variables change during execution. But over the years these algorithms have been extended to infer EFSMs [80, 106, 107, 114, 115]. A detailed account is provided in Chapter 2.

We use the MINT (Model Inference Technique) tool [110] to compare state-merge algorithms for EFSM inference with T2M. In addition to the k-Tails algorithm, the tool provides support for the GkTails algorithm that uses Daikon to infer transition guards for EFSMs, and the algorithm in [114] that generates data update functions using genetic programming. We use both these approaches for our comparison.

The abstractions generated by T2M and MINT for the *QEMU Serial I/O Port*, *Heartbeat Monitoring* and *Sensor* benchmarks, using the same set of system execution traces as input, are provided in Fig. 4.14–4.16.

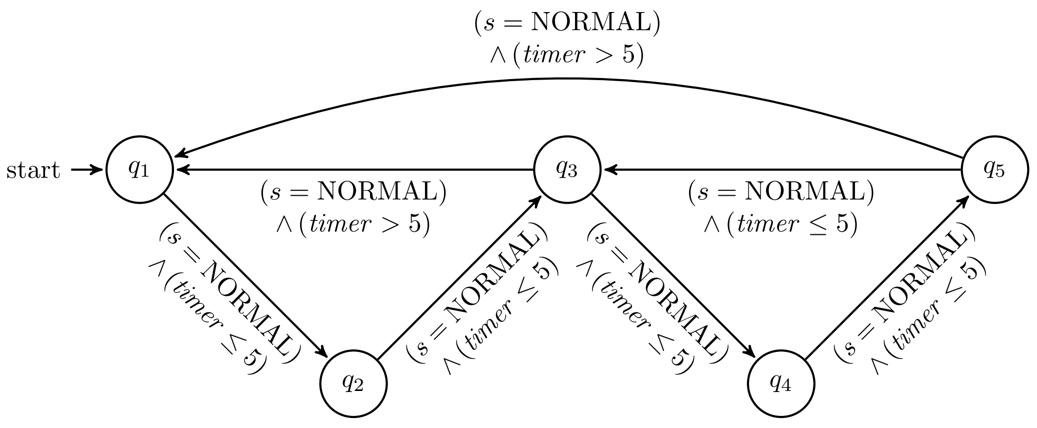

start
q_1
$y = \text{reset} \wedge x' = 0$,
$y = \text{write} \wedge x' = \textbf{if } x = 5 - 0.884 \textbf{ then } x + 1 \textbf{ else } 5$,
$y = \text{read} \wedge x' = x - 1$

(a) MINT


start
q_1
$y = \text{reset} \wedge x' = 0$
q_2
$y = \text{reset} \wedge x' = 0$,
$y = \text{write} \wedge x' = \textbf{if } x < \text{limit} \textbf{ then } x + 1 \textbf{ else } \text{limit}$,
$y = \text{read} \wedge x' = \textbf{if } x > 1 \textbf{ then } x - 1 \textbf{ else } 0$

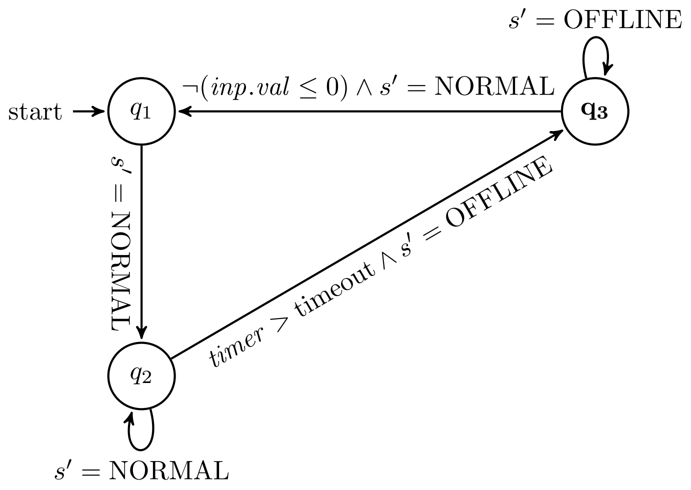
(b) T2M

Figure 4.14: Models learnt for a Serial I/O Port system using T2M and MINT. Here, 'limit' represents the total queue size, which was set to 5.



start
q_1
q_2
q_3
q_4
q_5
$(s = \text{NORMAL}) \wedge (timer > 5)$
$(s = \text{NORMAL}) \wedge (timer > 5)$
$(s = \text{NORMAL}) \wedge (timer \leq 5)$
$(s = \text{NORMAL}) \wedge (timer \leq 5)$
$(s = \text{NORMAL}) \wedge (timer \leq 5)$
$(s = \text{NORMAL}) \wedge (timer \leq 5)$
$(s = \text{NORMAL}) \wedge (timer \leq 5)$

(a) MINT



start
q_1
q_2
$\mathbf{q_3}$
$s' = \text{OFFLINE}$
$\neg(inp.val \leq 0) \wedge s' = \text{NORMAL}$
$s' = \text{NORMAL}$
$timer > \text{timeout} \wedge s' = \text{OFFLINE}$
$s' = \text{NORMAL}$

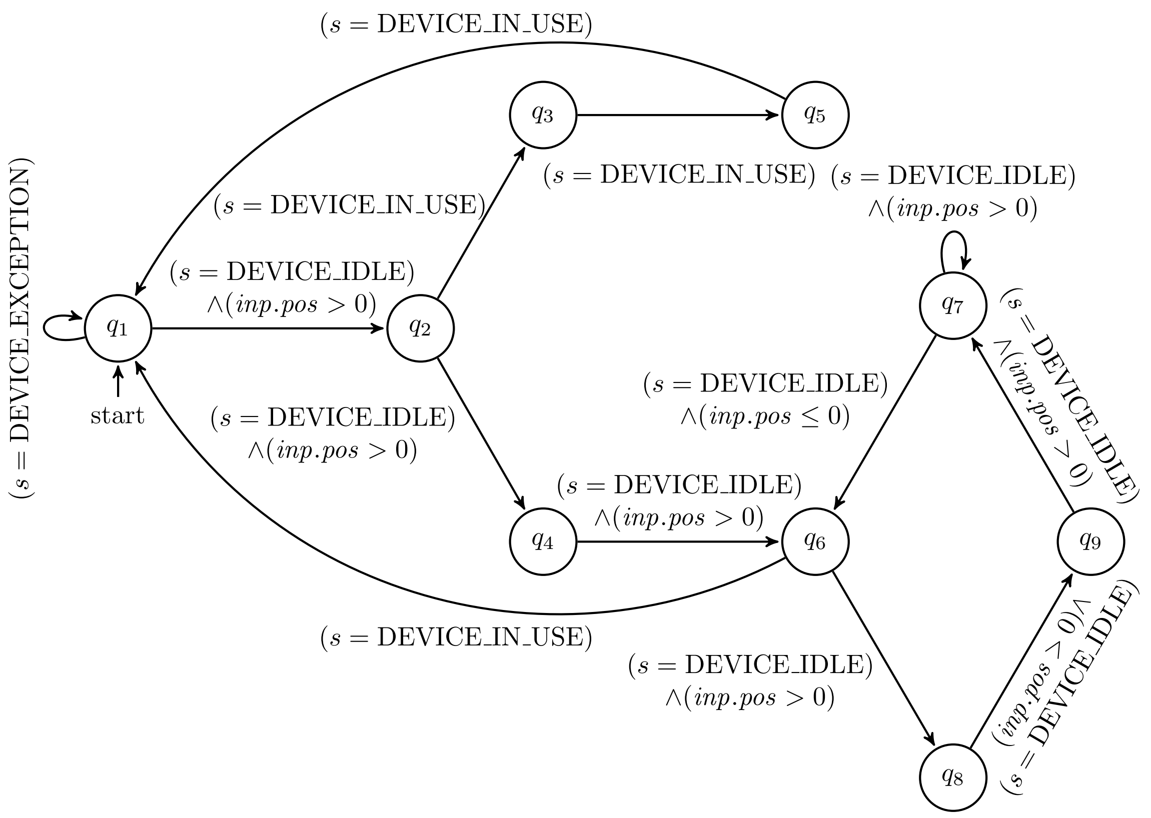
(b) T2M

Figure 4.15: Models learnt for a Heartbeat Monitoring system using T2M and MINT.

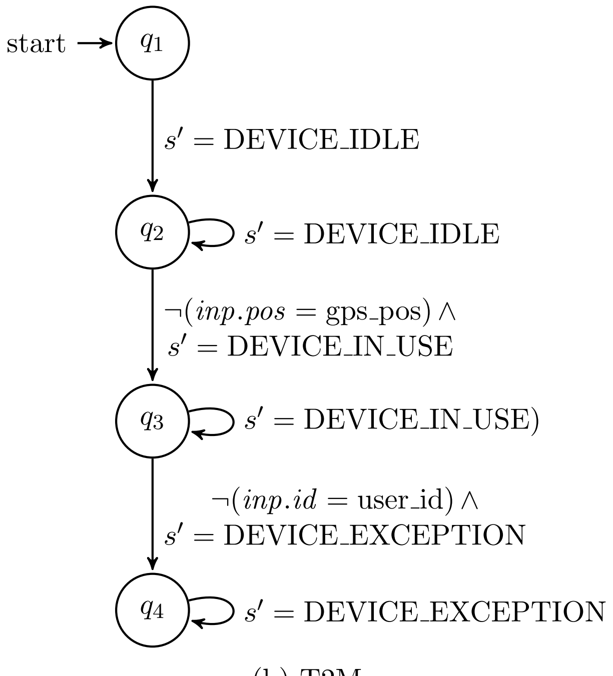
For the *QEMU Serial I/O Port* benchmark the MINT tool is able to infer expressive data update functions, such as the expression

$$y = \text{write} \wedge x' = \textbf{if } x = 5 - 0.884 \textbf{ then } x + 1 \textbf{ else } 5$$

as illustrated in Fig. 4.14a. The inferred functions are however partial. For instance, the tool infers the expression $x' = x - 1$ modelling updates to queue length x when there is a *read* event on the queue. This is however not the case when the queue is empty, i.e., $x = 0$ and therefore not accurate. The abstraction reverse-engineered by T2M in Fig. 4.14b is able to accurately model this behaviour. For the other benchmarks MINT is found to generate models that are less expressive than T2M, as illustrated in Fig. 4.15 and 4.16; Here, MINT fails to generalise and accurately express dependencies and computations involving multiple system variables.


(s = DEVICE_IN_USE)
(s = DEVICE_EXCEPTION)
(s = DEVICE_IN_USE)
(s = DEVICE_IDLE) ∧(inp.pos > 0)
start
(s = DEVICE_IDLE) ∧(inp.pos > 0)
(s = DEVICE_IN_USE)
(s = DEVICE_IDLE) ∧(inp.pos > 0)
(s = DEVICE_IDLE) ∧(inp.pos ≤ 0)
(s = DEVICE_IDLE) ∧(inp.pos > 0)
(s = DEVICE_IDLE) ∧(inp.pos > 0)
(s = DEVICE_IN_USE)
(s = DEVICE_IDLE) ∧(inp.pos > 0)
(inp.pos > 0)∧ (s = DEVICE_IDLE)
q1
q2
q3
q4
q5
q6
q7
q8
q9

(a) MINT



(b) T2M

Figure 4.16: Models learnt for an IoT Sensor system using T2M and MINT.

MINT infers transition guards and data update expressions on a per-transition basis, and therefore requires a large amount of trace data to generate a good enough abstraction; each transition needs to be executed with a broad range of data values to infer accurate predicates. As a result, for the same input set of execution traces, MINT is found to generate models that are less expressive than T2M.

Query-based learning: Although the classic L* algorithm does not support symbolic abstraction inference, several extensions of the algorithm have been developed to generate symbolic models such as SFAs and register automata [8, 17, 25]. In practice, however, implementations of these algorithms are often tailored to generate a specific and often small subset of symbolic models due to high query complexity [60, 61]. For instance, Maler and Mens developed a symbolic version of the L* algorithm to learn models where transitions are labelled with elements of a finite partition of the alphabet [81, 82]. Another extension [108] learns a specific class of Mealy machines with a single timer. The symbolic abstractions generated by T2M can feature more complex expressions, such as linear arithmetic, involving multiple variables.

The MAT* algorithm [8] learns SFAs over Boolean algebras that are efficiently learnable using membership and equivalence queries, such as the equality algebra. In addition to membership and equivalence oracles the algorithm takes as input a learning algorithm for the Boolean algebra used in the target SFA. Although the algorithm is able to generate SFAs over the equality algebra and SFAs with transitions labelled with Binary Decision Diagrams (BDDs), designing and implementing oracles for richer models such as SFAs over the theory of linear integer arithmetic is not straightforward. This is discussed further in Section 6.3.3, Chapter 6.

The Sigma* algorithm [17] uses symbolic execution to find constraints on inputs and expressions generating output to build a symbolic alphabet. Unlike T2M, Sigma* does not infer these expressions from system execution traces. Since an implementation of the algorithm is not available for public use, we were unable to experimentally compare Sigma* with T2M.

The SL* algorithm [25] extends query-based learning to infer register automata that model both control flow and data flow. Register automata have registers that can store input characters, and allow comparisons with existing values that are already stored in registers, making them inherently more expressive that SFAs. To evaluate the SL* algorithm and compare it with T2M we convert the SFAs generated by T2M into register automata, and attempt to reverse engineer the models using RALib [24]. RALib implements the SL* algorithm and supports the inference of Input-Output

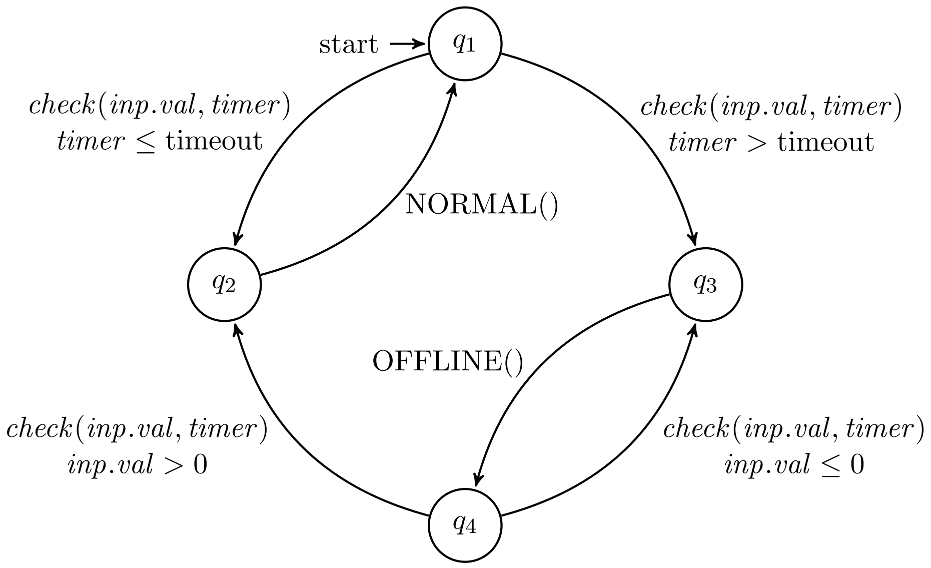


Figure 4.17: IORA modelling a Heartbeat Monitoring system.

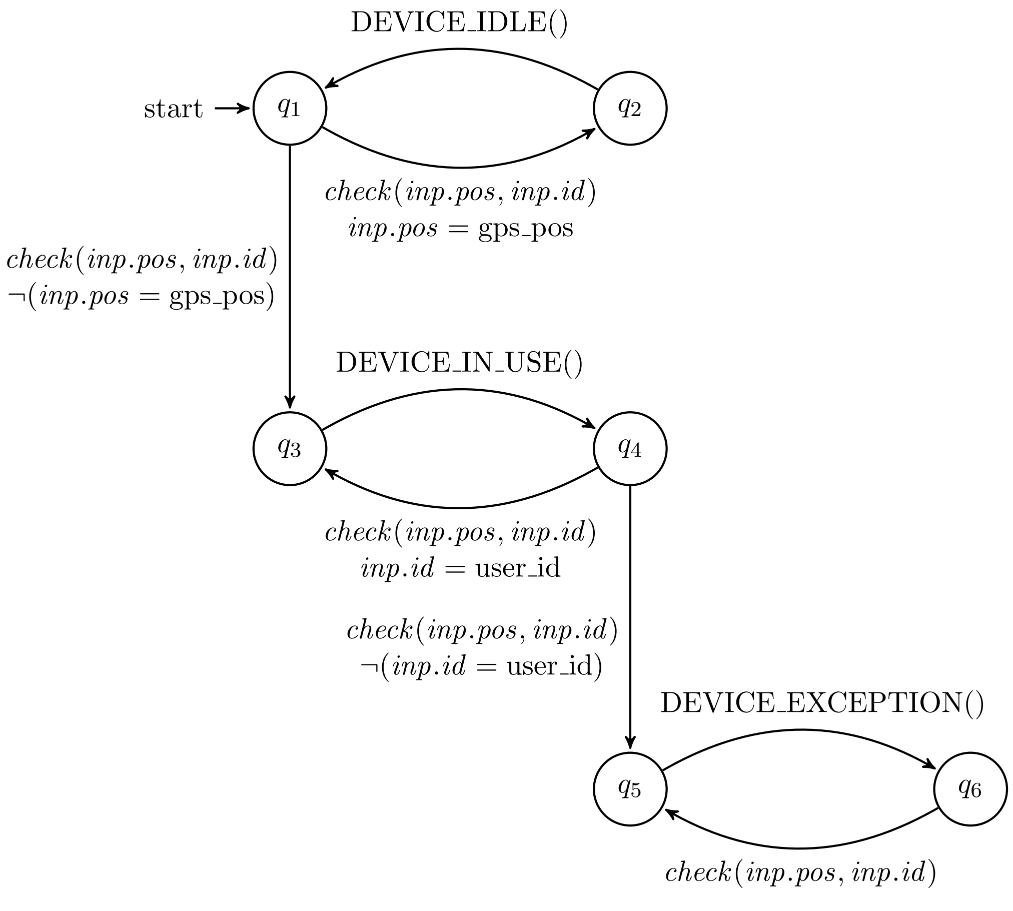


Figure 4.18: IORA modelling an IoT Sensor system.

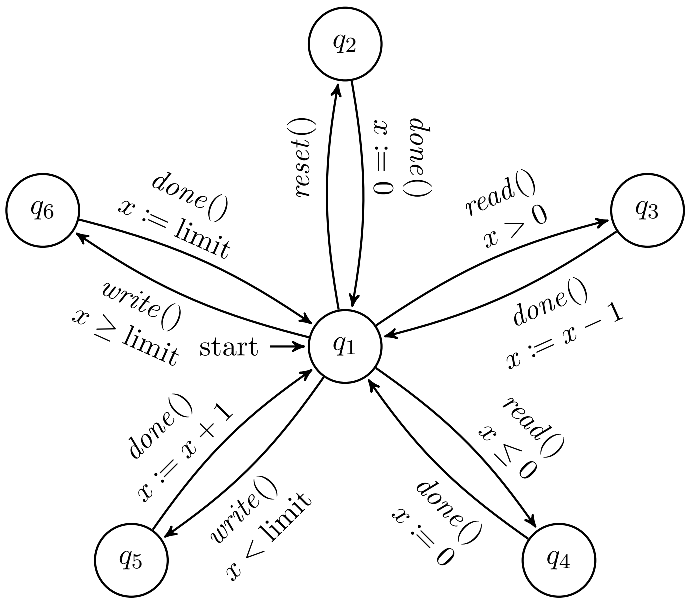


Figure 4.19: IORA modelling a Serial I/O Port system.

Register Automaton (IORA). An IORA is a register automaton transducer that generates an output action after each input action. Here, we discuss the results obtained for the *QEMU Serial I/O Port*, *Heartbeat Monitoring* and *Sensor* benchmarks.

We model the *Heartbeat Monitoring* benchmark as an IORA with input action $check(inp.val, timer)$ that takes parameters $inp.val$ and $timer$, and output actions NORMAL() and OFFLINE() corresponding to the system operation modes, as illustrated in Fig. 4.17. We use a similar modelling formalism to construct an IORA for the *Sensor* benchmark, provided in Fig. 4.18. RALib is able to reverse-engineer these models exactly for both benchmarks.

We model the *QEMU Serial I/O Port* benchmark as an IORA with inputs actions $read()$, $write()$ and $reset()$, and output action $done()$, as illustrated in Fig. 4.19. RALib, however, does not currently support assignments involving symbolic expressions on the right-hand side, and therefore the tool was unable to generate a model for this benchmark. We further discuss RALib and its ability to generate symbolic abstractions on benchmarks from other domains in Section 6.3.3, Chapter 6.

In the following sections the algorithms are compared on runtime, generated model size and correctness. The results are summarised in Tables 4.2 and 4.3. For the comparison experiments, we use the set of predicate sequences constructed from the execution trace set as described in Chapter 3 as the algorithm input for all benchmarks. In the following sections we refer to the set of predicates sequences as the trace input to the algorithms, unless specified otherwise.

Table 4.2: Runtime and model size comparison for automata generated using T2M, state-merge and L*. Here, '-na-' indicates 'not applicable'. The corresponding benchmark traces consist of only Boolean events, that themselves serve as transition predicates in the learned SFA and therefore do not require predicate synthesis.

Benchmark Set	Benchmark	Transition Predicate Synthesis				Automaton Construction					
		Predicate Sequence Stats.			Runtime(s)	Runtime(s)			Number of States		
		Total Length	Avg. Length	#Uniq. Pred.		T2M	State Merge	L*	T2M	State Merge	L*
Soc	USB Slot	39	39	6	-na-	**1.4**	13.4	205.3	**4**	7	25
	USB Attach	259	259	15	-na-	**6.9**	19	>2h	**7**	92	>45
	Serial I/O Port	2076	2076	3	0.4	**1.1**	18.8	>2h	**2**	2	>114
	Linux Kernel	20165	10082.5	8	-na-	**52.2**	>2h	>2h	**8**	-	>94
	Integrator	32768	32768	5	1775.2	**10.4**	>2h	>2h	**6**	-	>56
Log Data	cvs.net.txt	31977	8	15	-na-	**26**	113.6	422.9	**14**	142	22
	java.net. DatagramSocket	1518	15.2	28	-na-	3739.1	**33.3**	>2h	64	**22**	>96
	java.net. MultiCastSocket	12801	12.8	15	-na-	**18.6**	4340.6	>2h	**6**	93	>230
	java.net.Socket	49724	21.6	41	-na-	**1854.8**	>2h	>2h	**16**	-	>719
	java.net.URL	21845	21.8	16	-na-	**11.5**	>2h	27.2	**4**	113	4
	java.util. Formatter	7476	7.5	7	-na-	**1.9**	469.9	8.7	**4**	36	4
	java.util. StringTokenizer	5139	5.1	6	-na-	**1.6**	249.3	>2h	**3**	39	>286

Continued on next page

Table 4.2 – *Continued from previous page*

Benchmark Set	Benchmark	Transition Predicate Synthesis				Automaton Construction					
		Predicate Sequence Stats.			Runtime(s)	Runtime(s)			Number of States		
		Total Length	Avg. Length	#Uniq. Pred.		T2M	State Merge	L*	T2M	State Merge	L*
Log Data	RapidMiner	25440	18.9	18	-na-	**57.7**	1016.6	>2h	**19**	25	>138
	ssh.net	68436	18.9	9	-na-	**11.5**	>2h	>2h	**10**	100	>106
	tcpip.net	40966	17.4	10	-na-	**17.5**	>2h	>2h	**9**	80	>154
AWS IoT	Heart Beat	44	11	4	0.5	**0.6**	13	27.8	**3**	22	14
	HVACTemp	176	11	10	2.3	**5.7**	10.6	1355.6	**7**	85	59
	ISA Alarm	1364	11	35	3	**193.8**	247.3	>2h	**19**	670	>46
	Sensor	50	10	5	0.2	**1.4**	10.8	44	**4**	29	16
	Simple Alarm	187	11	8	0.3	**3.3**	11.9	983.4	**5**	93	57

4.5.2 Comparison with State-Merge

To compare T2M with state-merge we use the MINT tool. Most state-merge algorithms do not focus on producing the most succinct models but rather produce a good enough approximation that is consistent with the trace input. Algorithms that do generate minimal automata often require labelled traces or additional system information in the form of predefined LTL properties.

As is evident from Table 4.2, for majority of the benchmarks the state-merge algorithm is slower than T2M and it often generates large automata that are difficult to comprehend. By contrast, T2M learns models that are succinct in reasonable runtime. The MINT tool was unable to produce models for the *Linux Kernel*, *Integrator* and *java.net.Socket* benchmarks.

Since the state-merge approach generates automata by merging states in the PTA, the generated automaton is guaranteed to admit the trace input. Model correctness is therefore guaranteed, as is the case for T2M.

4.5.3 Comparison with L*

To compare T2M with L* we implement simple procedures that approximate oracles for membership and equivalence queries on the basis of the trace input only. Membership queries are answered by checking if the queried sequence is a finite prefix of any sequence in the trace input. Equivalence queries are handled by checking if the generated model admits all sequences in the trace input. If not, the procedure returns the smallest finite prefix of the sequence that is not accepted by the candidate model as a counterexample for model refinement.

As discussed above, the trace input is the only available information and therefore must serve as a proxy for the oracles used in L*. Thus, as evidenced in Table 4.2, L* learns exact representations of the trace input resulting in large automata that are hard to interpret. To illustrate this, consider the trace $\sigma = a, b, c, a, b, c, c, \ldots, a, b, \{c\}^n$.

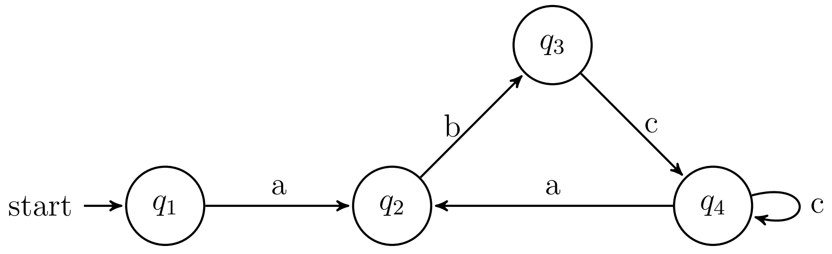
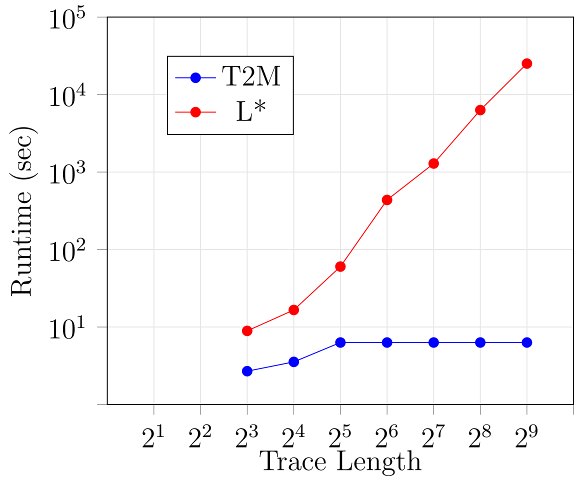
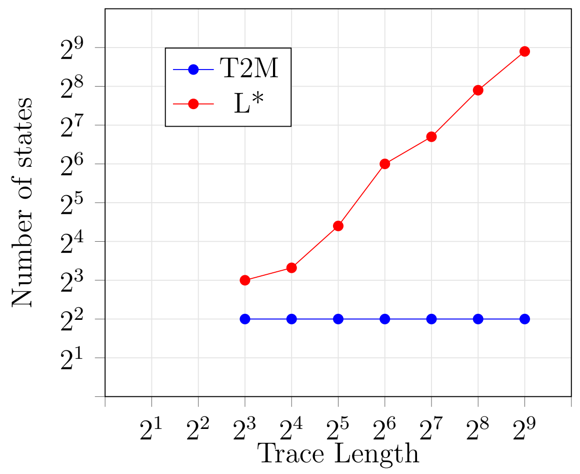


Figure 4.20: Automaton generated by T2M for trace $\sigma = a, b, c, a, b, c, c, \ldots, a, b, \{c\}^n$.



(a) Runtime



(b) Number of States

Figure 4.21: Log-log plot comparing runtime and number of states for L* and T2M for exponentially increasing trace length.

We use finite prefixes of σ of exponentially increasing length as input to L* and T2M in separate calls to the algorithms, and record the runtime and generated model size. T2M generates the 4-state automaton in Fig. 4.20 for all input sequences. Note that any finite prefix of σ is accepted by this automaton. On the other hand, L* generates automata with exponentially increasing number of states, as illustrated in Fig. 4.21b. We also observe an exponential increase in runtime (Fig. 4.21a). Unlike T2M, the lack of additional system information makes it difficult for L* to generalise and learn small system models using only the trace data.

4.5.4 Comparison with the LSTM-based Approach

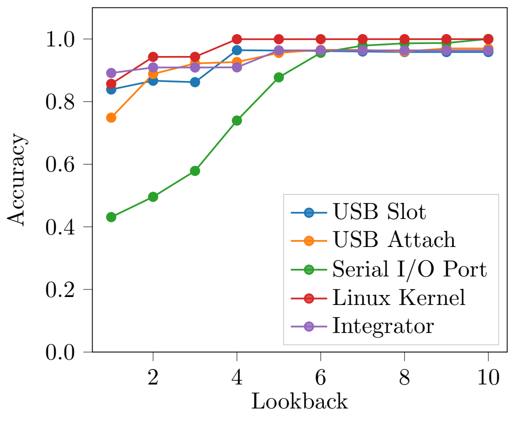
To compare T2M with LSTM-based learning, we look at two LSTM architectures: a single layer LSTM network (LSTM1) and a multi-layer LSTM network (LSTM2) comprising two layers of stacked LSTM cells. These are standard architectures used in relevant literature for learning temporal information from trace data [23, 104].

For training, each event in the input sequence is converted into the corresponding one-hot tensor encoding. The training set is generated using the *TimeseriesGenerator* module in Tensorflow Keras [3]. This produces batches of temporal data for some predefined *lookback*—which determines the amount of history taken into account to make a prediction. For each benchmark, we train multiple networks with different lookback; starting with all values between 1 and 10, followed by 20, 40, 60, 80, 100, 150 and 200 in order to cover a broad range of values. We compare training curves and test accuracy to identify the network with minimum lookback that gives the best performance for each benchmark. Once trained, we can present a sequence of events of length equal to the lookback as input to the network, which in return will produce a likelihood measure for possible next-event values as the network output.

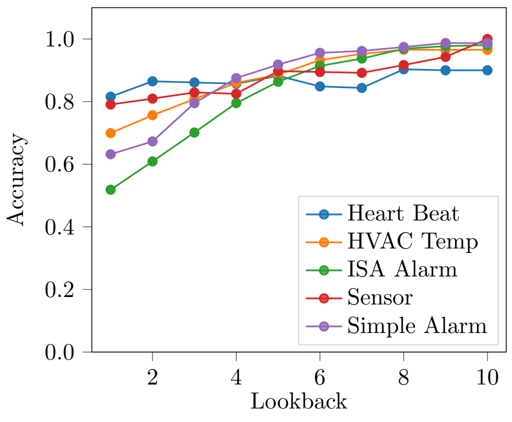
In our experiments, for a network trained with lookback lb, we begin by feeding the trace prefix of length lb as input to the network. For each next-event value in the network output with likelihood greater than a pre-defined threshold $\theta = 0.1$, we append the next-event value to the current network input and use the last lb events as the next input to the network. This is done for 1000 iterations. Starting with the initial input to the network, we track all next-event values with likelihood greater than θ for each iteration, in effect, creating a trie [72] like structure where each next-event value corresponds to a new branch in the trie. Every path in the trie from root to leaves represents a sequence of next-event values produced by the network, and is preceded by the event sequence that forms the initial input to the network. This corresponds to the sequential behaviour learnt by the LSTM network.

Effect of lookback on training: For most of the benchmarks we observe smooth training curves for lookback values up to 10 and training gets progressively erratic beyond a lookback value of 40. This is a common observation with LSTM networks [95].

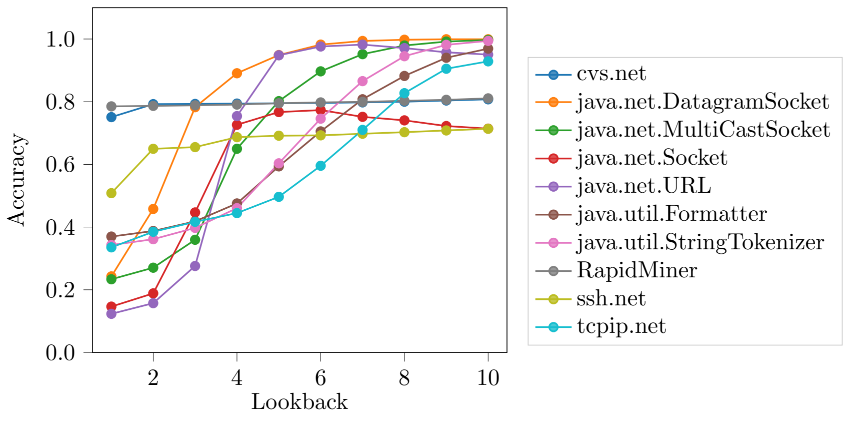
Fig. 4.22 provides graph plots for training accuracy vs. lookback for the three benchmark sets. For the *SoC* and *AWS IoT* benchmarks we see similar plots for LSTM1 and LSTM2 so we provide results for only LSTM1 in Fig. 4.22. For the *java.net.Socket* benchmark, LSTM2 (Fig. 4.22d) exhibits higher accuracy than LSTM1 (Fig. 4.22c). Table 4.3 provides the value of minimum lookback that provides best performance for each benchmark. From Table 4.3 it is evident that we cannot pick a single value for lookback that works well for all benchmarks, but a value between 4 and 10 is best for consistent training results and good network performance.



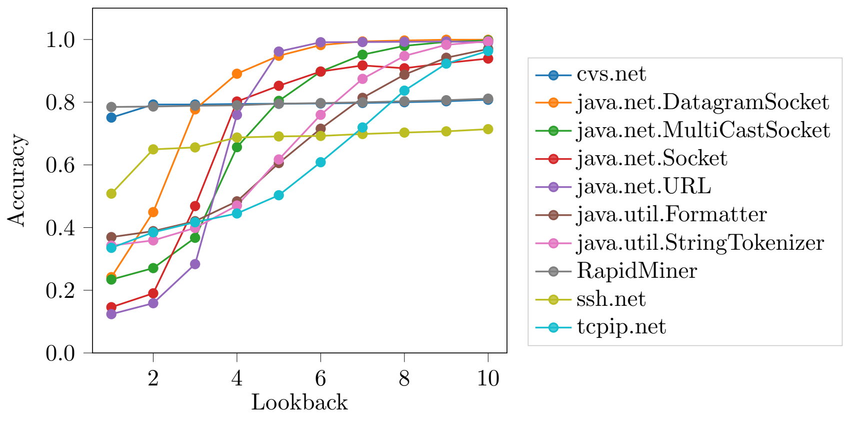
(a) SoC



(b) AWS



(c) Log Data (LSTM1)



(d) Log Data (LSTM2)

Figure 4.22: Lookback vs. Accuracy plots for all benchmark sets.

Table 4.3: Accuracy of the captured behaviour for the network with minimum lookback that produces consistent training results with maximum training accuracy.

Benchmark Set	Benchmark	LSTM1				LSTM2			
		Total Runtime(s)	Look back	%Miss	%Wrong	Total Runtime(s)	Look back	%Miss	%Wrong
Soc	USB Slot	358.5	4	0	62.9	488.2	4	0	71.4
	USB Attach	1619.1	6	0	79.2	2348.6	9	0	0
	Serial I/O Port	≈3h	10	0	0	≈4h	10	0	0
	Linux Kernel	≈29h	4	61.1	0	≈41h	4	61.1	0
	Integrator	≈47h	5	41.7	0	≈63h	5	41.7	0
Log Data	cvs.net	≈51h	6	0	0	≈72h	5	0	0
	java.net.DatagramSocket	9157.9	8	0.3	47.2	≈3h	8	72.8	0
	java.net.MultiCastSocket	≈20h	10	59.2	0	≈28h	10	59.2	0
	java.net.Socket	≈63h	5	0	26.3	≈103h	10	3.6	16.5
	java.net.URL	≈35h	6	0	12.1	≈46h	6	0	11.5
	java.util.Formatter	≈13h	10	0	5.3	≈17h	10	0	34.5
	java.util.StringTokenizer	≈9h	10	3.4	0	≈12h	10	3.4	0
	RapidMiner	≈38h	5	0	0	≈51h	6	0	0
	ssh.net	≈108h	7	0	0	≈138h	7	0	0
	tcpip.net	≈64h	10	0	43.9	≈86h	10	0	42.9
AWS IoT	Heart Beat	432.5	8	12.5	41.7	554.3	8	12.5	0
	HVACTemp	1300.5	8	4	71.4	1664	8	8	68

Continued on next page

Table 4.3 – *Continued from previous page*

Benchmark Set	Benchmark	LSTM1				LSTM2			
		Total Runtime(s)	Look back	%Miss	%Wrong	Total Runtime(s)	Look back	%Miss	%Wrong
AWS IoT	ISA Alarm	8150.6	9	73.1	0	10497.4	9	73.1	0
	Sensor	525.9	10	0	70	662.2	10	0	68.9
	Simple Alarm	1340.9	9	4.8	0	1700.4	9	4.8	64.3

Model correctness and runtime: To quantitatively compare the accuracy of behaviours captured by the LSTM approach and T2M, we introduce two parameters: '%Miss' represents the fraction of predicate sequences captured by T2M but not by the LSTM networks, and '%Wrong' represents the fraction of predicate sequences captured by the LSTM networks but not seen in the trace input. The LSTM approach does not always capture all sequential behaviour exemplified by the trace input, as is evident from Table. 4.3. Moreover, for many benchmarks we see that more than 50% of the behaviours captured by the LSTM approach are invalid. Similar results are observed in related literature [23], as already discussed in Section 2.1.3, Chapter 2.

Since networks trained on different lookback values capture information with varying levels of accuracy for different benchmarks, we need to train multiple networks with different lookback in order to get the best outcome. This process can be time-consuming, as is evident from the 'Total Runtime' column in Table. 4.3. This captures the training time summed over network lookback values between 1 and 10.

Unlike T2M, the LSTM approach does not always guarantee that all sequential behaviour exemplified by the input traces are captured. For all benchmarks, T2M generates models that admit all input traces with provable guarantees, and also significantly faster compared to the training time for LSTM networks.

Chapter 5

Equivalence Checking using Simulation Relations

Model-learning algorithms are broadly classified into passive and active algorithms. While passive learning algorithms generate a model from a given set of traces [16, 56, 67, 77], active learning algorithms [7, 8, 25, 111] iteratively refine a hypothesis model by extracting information from the system or an oracle that has sufficient knowledge of the system, using the hypothesis model as a guide.

The model-learning algorithm described in Chapter 3 is an instance of passive learning. As discussed in Chapter 1, the behaviours admitted by models generated by passive learning are limited to only those manifest in the given traces. Therefore, capturing all system behaviour by the generated system models is conditional on devising a software load that exercises all relevant system behaviours. This can be difficult to achieve in practice, especially when a system comprises multiple components and specifying emergent behaviour can be tricky. Random input sampling is a pragmatic choice in this scenario, but it does not guarantee that generated abstractions admit all system behaviour. This is discussed further in Section 6.2.4, Chapter 6.

On the other hand, active model-learning algorithms can, in principle, generate exact system models. The most popular form of active learning is query-based learning, where the learning framework poses membership and equivalence queries to an oracle and uses the responses to guide model generation and refinement. But, when these algorithms are used in practice, particularly to learn symbolic abstractions over large and possibly infinite alphabets, such as the model in Fig. 5.1, they suffer from high query complexity [45, 60, 61]. As a result, many active model-learning implementations are constrained to generating partial models for large systems.

In the following two chapters we describe an active learning approach to derive overapproximating abstractions of a system, with provable completeness guarantees.

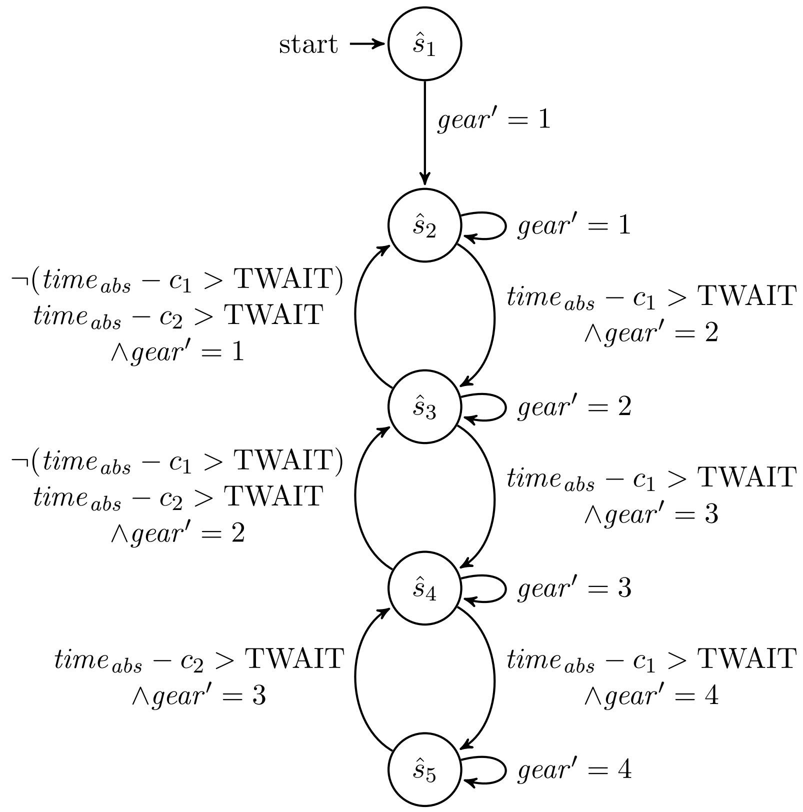

start
$\hat{s}_1$
$gear' = 1$
$\hat{s}_2$
$gear' = 1$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 1$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 2$
$\hat{s}_3$
$gear' = 2$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 2$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 3$
$\hat{s}_4$
$gear' = 3$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 3$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 4$
$\hat{s}_5$
$gear' = 4$

Figure 5.1: Abstraction modelling gear-shift logic for an Automatic Transmission Gear system generated by our algorithm.

In this chapter we describe an equivalence checking procedure using simulation relations to evaluate the degree of completeness of a given abstraction for a system. The structure of the abstraction is used to extract a set of conditions that collectively encode a completeness hypothesis. The hypothesis is formulated such that the satisfaction of the hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system and the given abstraction. Further, the existence of a simulation relation is sufficient to guarantee that the given abstraction is overapproximating, i.e., it accepts (at least) all system execution traces.

To verify if the hypothesis holds, we check the truth value of all extracted completeness conditions using SAT solving. The procedure returns the fraction of extracted conditions that hold as a quantitative measure of the degree of completeness for the given abstraction. In the event of a condition falsification, the SAT procedure returns a counterexample to the condition.

The satisfaction of the hypothesis is sufficient to guarantee that a given abstraction is overapproximating, but not necessary. Counterexamples to the hypothesis may therefore be spurious, i.e., a condition falsification may not actually correspond to missing system behaviour in the abstraction. This is resolved by model checking [31] to verify if the counterexample for a condition is spurious. Non-spurious counterexamples provide useful insight to identify missing behaviours in the given abstraction. In Chapter 6, we describe how this information can be used to iteratively learn an overapproximating abstraction for a system.

Plan of the chapter

The rest of the chapter is organized as follows. Section 5.1 provides a formal overview of the terminology, definitions and notations used in this chapter. Section 5.2 describes the formulation of the completeness hypothesis and provides a formal proof for the claim: satisfaction of the completeness hypothesis is sufficient to guarantee the existence of a simulation relation between the system and the given abstraction, which further guarantees that the given system abstraction is overapproximating. Section 5.3 describes the procedure used to check the truth value of extracted conditions, and generate counterexamples for any condition falsifications. Further, it describes a method to verify if a counterexample to the hypothesis is spurious.

Claim of Novelty

Unlike query-based learning, our procedure to evaluate the degree of completeness for a given abstraction operates at the level of the abstraction and not concrete system traces:

- The scope of each extracted completeness condition is the set of incoming and outgoing transitions to a state rather than a finite path in the system or its abstraction.

- The completeness hypothesis can be represented symbolically, incorporating characteristic functions for sets of observations in a system trace, therefore eliminating the need for explicit enumeration of concrete transitions.

This enables the procedure to be applied to symbolic abstractions over large alphabets. Further, the procedure is agnostic of the algorithm used to learn the abstraction. This enables the approach to be easily integrated with model-learning algorithms that generate symbolic abstractions from traces, to iteratively learn expressive system overapproximations with provable completeness guarantees.

5.1 Formal Model

We represent the system as a Labelled Transition System (LTS) $\mathcal{M}$.

Definition 8 (Labelled Transition System). *An LTS $\mathcal{M}$ is a quadruple $(\mathcal{S}, \Omega, \Delta, s_0)$ where $\mathcal{S}$ is a set of states, Ω is a set of labels, $\Delta \subseteq \mathcal{S} \times \Omega \times \mathcal{S}$ is the transition relation and $s_0 \in \mathcal{S}$ is the initial state.*

The set of labels Ω is a set of system observations. An observation $o \in \Omega$ could be any event that depends on a transition from some state s to state s'. A path π in $\mathcal{M}$ is a finite sequence $\pi = s_0, o_0, s_1, \ldots, o_{n-1}, s_n$ of alternating states and observations such that $(s_i, o_i, s_{i+1}) \in \Delta$ for $0 \leq i < n$. The trace of π, denoted $\sigma(\pi)$, is the corresponding sequence of observations $o_0, \ldots, o_{n-1}$ along π. The set of all traces of paths in $\mathcal{M}$ is called the language of $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$, defined over the alphabet of observations Ω.

The learned system abstraction is represented as an LTS $\hat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{\Omega}, \hat{\Delta}, \hat{s}_0)$. The language of the abstraction $\mathcal{L}(\hat{\mathcal{M}})$ is defined over the alphabet of observations, i.e., $\hat{\Omega} = \Omega$. The abstraction accepts a trace $\sigma = o_0, \ldots, o_{n-1}$ if $\sigma \in \mathcal{L}(\hat{\mathcal{M}})$, i.e., if there exists a sequence $\hat{s}_0, \ldots, \hat{s}_n$ of states in $\hat{\mathcal{S}}$ such that $(\hat{s}_i, o_i, \hat{s}_{i+1}) \in \hat{\Delta}$ for $0 \leq i < n$.

With the equivalence checking procedure described in this chapter we evaluate if a given abstraction $\hat{\mathcal{M}}$ for a system $\mathcal{M}$ is complete, i.e., $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$.

5.2 Completeness Conditions for a Candidate Abstraction

We first give an explicit-state, set-based definition of our completeness criterion, for the sake of clarity. We subsequently describe a symbolic representation of the completeness conditions using characteristic functions, that can be applied to symbolic abstractions such as the model in Fig. 5.1.

5.2.1 Set-based Definition

To determine whether a given abstraction $\hat{\mathcal{M}}$ for the system $\mathcal{M}$ is complete, we use the structure of the abstraction to extract the following completeness conditions:

For initial state $\hat{s}_0 \in \hat{\mathcal{S}}$, $\forall o \in \Omega$:

$$\exists s \in \mathcal{S} \colon (s_0, o, s) \in \Delta \implies \exists \hat{s} \in \hat{\mathcal{S}} \colon (\hat{s}_0, o, \hat{s}) \in \hat{\Delta} \tag{5.1}$$

And for all states $\hat{s} \in \hat{\mathcal{S}}$, $\forall o', o \in \Omega$:

$$\begin{aligned}
&(\exists \hat{s}'' \in \hat{\mathcal{S}} \colon (\hat{s}'', o', \hat{s}) \in \hat{\Delta} \wedge \\
&\ \exists s'', s, s' \in \mathcal{S} \colon (s'', o', s), (s, o, s') \in \Delta) \implies \\
&\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}, o, \hat{s}') \in \hat{\Delta}
\end{aligned} \tag{5.2}$$

These conditions collectively encode the following completeness hypothesis: for any transition available in the system $\mathcal{M}$, defined by the transition relation Δ, there is a corresponding transition in $\hat{\mathcal{M}}$ defined by $\hat{\Delta}$.

In the following section we prove that if the above hypothesis holds, i.e., if the completeness conditions (5.1) and (5.2) evaluate to true, then a simulation relation can be constructed between $\mathcal{M}$ and $\hat{\mathcal{M}}$. Further, we formally prove that the existence of a simulation relation between $\mathcal{M}$ and $\hat{\mathcal{M}}$ implies $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$.

5.2.1.1 Constructing a Simulation Relation

In Section 2.3, Chapter 2 we introduced formal preliminaries on equivalence relations between formal system models, including simulation relations for Kripke structures. Here, we formally define simulation relations specifically for LTSs.

Definition 9 (Simulation Relation for LTSs)**.** *Given two LTSs $\mathcal{M} = (\mathcal{S}, \Omega, \Delta, s_0)$ and $\hat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{\Omega}, \hat{\Delta}, \hat{s}_0)$ with $\hat{\Omega} = \Omega$, we define a binary relation $\mathcal{R} \subseteq \mathcal{S} \times \hat{\mathcal{S}}$ to be a simulation if*

1. *$(s_0, \hat{s}_0) \in \mathcal{R}$, and*
2. *$\forall (s, \hat{s}) \in \mathcal{R}$, for every $s' \in \mathcal{S}$ and $o \in \Omega$ such that $(s, o, s') \in \Delta$, $\exists \hat{s}' \in \hat{\mathcal{S}}$ such that $(\hat{s}, o, \hat{s}') \in \hat{\Delta}$ and $(s', \hat{s}') \in \mathcal{R}$.*

If such a relation $\mathcal{R}$ exists, we write $\mathcal{M} \preceq_{\mathcal{R}} \hat{\mathcal{M}}$.

To support our claim that the satisfaction of the completeness hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system $\mathcal{M}$ and the given abstraction $\hat{\mathcal{M}}$, we first describe a method to construct a binary relation $\mathcal{R}' \subseteq \mathcal{S} \times \hat{\mathcal{S}}$ when all extracted completeness conditions hold, and later formally prove that $\mathcal{R}'$ is indeed a simulation.

Assuming the completeness conditions (5.1) and (5.2) hold, the relation $\mathcal{R}'$ is constructed as follows:

1. Initialise $\mathcal{R}' = \{(s_0, \hat{s}_0)\}$

2. If the condition (5.1) holds non-trivially for some observation $o \in \Omega$, i.e., $\exists s \in \mathcal{S} \colon (s_0, o, s) \in \Delta$ and $\exists \hat{s} \in \hat{\mathcal{S}} \colon (\hat{s}_0, o, \hat{s}) \in \hat{\Delta}$, then add the state pair $(s, \hat{s})$ to $\mathcal{R}'$.

$$\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(s, \hat{s})\}$$

3. If the condition (5.2) extracted for a state $\hat{s} \in \hat{\mathcal{S}}$ holds non-trivially for some observations $o', o \in \Omega$, i.e., $\exists s'', s, s' \in \mathcal{S} \colon (s'', o', s), (s, o, s') \in \Delta$ and $\exists \hat{s}'', \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}'', o', \hat{s}), (\hat{s}, o, \hat{s}') \in \hat{\Delta}$, then add the state pairs $(s, \hat{s})$ and $(s', \hat{s}')$ to $\mathcal{R}'$.

$$\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(s, \hat{s}), (s', \hat{s}')\}$$

Note that in the above construction, for every state pair $(s, \hat{s}) \in \mathcal{R}' \backslash (s_0, \hat{s}_0)$, there exists incoming transitions to the states s and $\hat{s}$ on some observation $o' \in \Omega$. That is,

$$\forall (s, \hat{s}) \in \mathcal{R}' \backslash (s_0, \hat{s}_0) \cdot \exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega \colon (s'', o', s) \in \Delta \wedge (\hat{s}'', o', \hat{s}) \in \hat{\Delta} \tag{5.3}$$

Theorem 3. *The constructed relation $\mathcal{R}'$ forms a simulation, i.e. $\mathcal{M} \preceq_{\mathcal{R}'} \hat{\mathcal{M}}$*

Proof. We use contradiction to prove that when the completeness conditions (5.1) and (5.2) hold, the constructed relation $\mathcal{R}'$ forms a simulation. Let us assume $\mathcal{R}'$ is not a simulation. A binary relation $\mathcal{R} \subseteq \mathcal{S} \times \hat{\mathcal{S}}$ is not a simulation if either

(a) $(s_0, \hat{s}_0) \notin \mathcal{R}$ or

(b) $\exists (s, \hat{s}) \in \mathcal{R}$ such that $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon (s, o, s') \in \Delta$ and $\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s}, o, \hat{s}') \notin \hat{\Delta}$ or

(c) $\exists (s, \hat{s}) \in \mathcal{R}$ such that $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon (s, o, s') \in \Delta$ and $\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}, o, \hat{s}') \in \hat{\Delta} \wedge (s', \hat{s}') \notin \mathcal{R}$

The above clauses (a), (b) and (c) are obtained by negating conditions (1) and (2) in definition 9 for a simulation relation; while clause (a) is obtained by negating condition (1), clauses (b) and (c) are obtained by negating condition (2) as follows: Condition (2) in definition 9 can be written as

$$\begin{aligned} &\forall (s, \hat{s}) \in \mathcal{R} \cdot \forall s' \in \mathcal{S} \cdot \forall o \in \Omega \cdot ((s, o, s') \in \Delta \implies \\ &\qquad\qquad (\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}, o, \hat{s}') \in \hat{\Delta} \wedge (s', \hat{s}') \in \mathcal{R})) \end{aligned} \tag{5.4}$$

On negating the above expression we get

$$\neg(\forall(s,\hat{s}) \in \mathcal{R} \cdot \forall s' \in \mathcal{S} \cdot \forall o \in \Omega \cdot ((s,o,s') \in \Delta \implies (\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \in \mathcal{R}))) \tag{5.5}$$

$$\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon ((s,o,s') \in \Delta \wedge \neg(\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \in \mathcal{R})) \tag{5.6}$$

$$\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon ((s,o,s') \in \Delta \wedge \forall \hat{s}' \in \hat{\mathcal{S}} \cdot ((\hat{s},o,\hat{s}') \in \hat{\Delta} \implies (s',\hat{s}') \notin \mathcal{R})) \tag{5.7}$$

$$\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon ((s,o,s') \in \Delta \wedge ((\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta}) \vee (\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \notin \mathcal{R}))) \tag{5.8}$$

$$\begin{aligned} &\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon ((s,o,s') \in \Delta \wedge \forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta}) \\ &\qquad \vee \\ &\exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon ((s,o,s') \in \Delta \wedge (\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \notin \mathcal{R})) \end{aligned} \tag{5.9}$$

Here, expression (5.9) corresponds to (clause (b) $\vee$ clause (c)).

Assume clause (a) holds. Then, $(s_0,\hat{s}_0) \notin \mathcal{R}'$. But, $(s_0,\hat{s}_0) \in \mathcal{R}'$ by construction. This is a contradiction and therefore, clause (a) does not hold.

Assume clause (b) holds. Then, $\exists(s,\hat{s}) \in \mathcal{R}'$ such that $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon (s,o,s') \in \Delta$ and $\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta}$. There are two possibilities here:

- If $s = s_0$ and $\hat{s} = \hat{s}_0$, then

$$\begin{aligned} &\exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon (s_0,o,s') \in \Delta \wedge \\ &\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s}_0,o,\hat{s}') \notin \hat{\Delta} \end{aligned}$$

  This violates completeness condition (5.1), which is a contradiction.

- If $(s,\hat{s}) \in \mathcal{R}' \backslash (s_0,\hat{s}_0)$, then from (5.3) there exists incoming transitions to s and $\hat{s}$ on some observation $o' \in \Omega$, i.e., $\exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega \colon (s'',o',s) \in \Delta \wedge (\hat{s}'',o',\hat{s}) \in \hat{\Delta}$. This implies

$$\begin{aligned} &\exists \hat{s}'' \in \hat{\mathcal{S}} \colon (\hat{s}'',o',\hat{s}) \in \hat{\Delta} \wedge \\ &\exists s'',s,s' \in \mathcal{S} \colon (s'',o',s),(s,o,s') \in \Delta \wedge \\ &\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta} \end{aligned}$$

  This violates completeness condition (5.2), which is a contradiction.

Therefore, clause (b) does not hold.

Assume clause (c) holds. Then, $\exists(s, \hat{s}) \in \mathcal{R}'$ such that $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega \colon (s, o, s') \in \Delta$ and $\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}, o, \hat{s}') \in \hat{\Delta} \wedge (s', \hat{s}') \notin \mathcal{R}'$. There are two possibilities here:

- If $s = s_0$ and $\hat{s} = \hat{s}_0$, then

  $$\begin{aligned} &\exists s' \in \mathcal{S} \colon (s_0, o, s') \in \Delta \wedge \\ &\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}_0, o, \hat{s}') \in \hat{\Delta} \end{aligned}$$

  This is a case where condition (5.1) holds non-trivially, and therefore $(s', \hat{s}') \in \mathcal{R}'$ by construction. This contradicts our assumption that clause (c) holds.

- If $(s, \hat{s}) \in \mathcal{R}' \backslash (s_0, \hat{s}_0)$, then from (5.3) there exists incoming transitions to s and $\hat{s}$ on some observation $o' \in \Omega$, i.e., $\exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega \colon (s'', o', s) \in \Delta \wedge (\hat{s}'', o', \hat{s}) \in \hat{\Delta}$. This implies

  $$\begin{aligned} &\exists \hat{s}'' \in \hat{\mathcal{S}} \colon (\hat{s}'', o', \hat{s}) \in \hat{\Delta} \wedge \\ &\exists s'', s, s' \in \mathcal{S} \colon (s'', o', s), (s, o, s') \in \Delta \wedge \\ &\exists \hat{s}' \in \hat{\mathcal{S}} \colon (\hat{s}, o, \hat{s}') \in \hat{\Delta} \end{aligned}$$

  This is a case where condition (5.2) holds non-trivially, and therefore $(s', \hat{s}') \in \mathcal{R}'$ by construction. This contradicts our assumption that clause (c) holds.

Therefore, clause (c) does not hold.

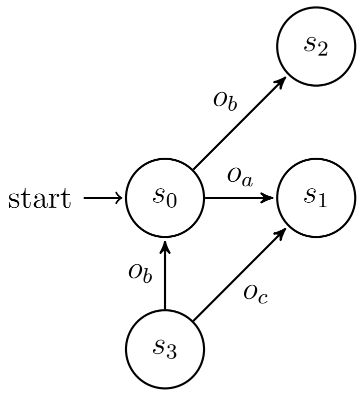
As none of the clauses (a), (b) or (c) hold, the constructed relation $\mathcal{R}'$ is a simulation by contradiction, i.e., $\mathcal{M} \preceq_{\mathcal{R}'} \hat{\mathcal{M}}$. □

Note that the satisfaction of the completeness hypothesis is sufficient to guarantee the existence of a simulation relation between $\mathcal{M}$ and $\hat{\mathcal{M}}$, but not necessary. An example is provided in Fig. 5.2. Here, the completeness conditions extracted for state $\hat{s}_2$ do not hold:

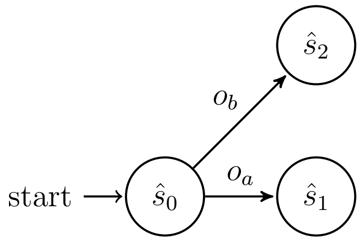
$$\begin{aligned} &(\exists \hat{s}_0 \in \hat{\mathcal{S}} \colon (\hat{s}_0, o_b, \hat{s}_2) \in \hat{\Delta} \wedge \\ &\exists s_3, s_0, s_1 \in \mathcal{S} \colon (s_3, o_b, s_0), (s_0, o_a, s_1) \in \Delta) \implies \\ &\exists \hat{s} \in \hat{\mathcal{S}} \colon (\hat{s}_2, o_a, \hat{s}) \in \hat{\Delta} \end{aligned}$$

does not hold as $\forall \hat{s} \in \hat{\mathcal{S}} \cdot (\hat{s}_2, o_a, \hat{s}) \notin \hat{\Delta}$. Similarly,

$$\begin{aligned} &(\exists \hat{s}_0 \in \hat{\mathcal{S}} \colon (\hat{s}_0, o_b, \hat{s}_2) \in \hat{\Delta} \wedge \\ &\exists s_3, s_0, s_2 \in \mathcal{S} \colon (s_3, o_b, s_0), (s_0, o_b, s_2) \in \Delta) \implies \\ &\exists \hat{s} \in \hat{\mathcal{S}} \colon (\hat{s}_2, o_b, \hat{s}) \in \hat{\Delta} \end{aligned}$$



(a) System $\mathcal{M}$



(b) Abstraction $\hat{\mathcal{M}}$

Figure 5.2: Example system and its abstraction.

does not hold as $\forall \hat{s} \in \hat{\mathcal{S}} \cdot (\hat{s}_2, o_b, \hat{s}) \notin \hat{\Delta}$.

However, $\mathcal{M} \preceq_{\mathcal{R}} \hat{\mathcal{M}}$ with $\mathcal{R} = \{(s_0, \hat{s}_0), (s_1, \hat{s}_1), (s_2, \hat{s}_2)\}$.

Theorem 4. *If $\mathcal{M} \preceq_{\mathcal{R}} \hat{\mathcal{M}}$ for LTSs $\mathcal{M}$ and $\hat{\mathcal{M}}$ then, $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$.*

Proof. Let $\pi = s_0, o_0, s_1, \ldots, o_{n-1}, s_n$ be the path in $\mathcal{M}$ corresponding to a trace $\sigma = o_0, \ldots, o_{n-1} \in \mathcal{L}(\mathcal{M})$, such that $(s_i, o_i, s_{i+1}) \in \Delta$ for $0 \leq i < n$. Then, we can construct a sequence $\hat{s}_0, \ldots, \hat{s}_n$ of states in $\hat{\mathcal{S}}$ such that $(\hat{s}_i, o_i, \hat{s}_{i+1}) \in \hat{\Delta}$ for $0 \leq i < n$ by induction as follows: it is clear that $(s_0, \hat{s}_0) \in \mathcal{R}$. Assume $(s_i, \hat{s}_i) \in \mathcal{R}$ for some i. By Definition 9, since $(s_i, o_i, s_{i+1}) \in \Delta$, $\exists \hat{t} \in \hat{\mathcal{S}}$ such that $(\hat{s}_i, o_i, \hat{t}) \in \hat{\Delta}$ and $(s_{i+1}, \hat{t}) \in \mathcal{R}$. Choosing $\hat{s}_{i+1} = \hat{t}$ produces the required sequence of states. We now have a path $\hat{\pi} = \hat{s}_0, o_0, \hat{s}_1, \ldots, o_{n-1}, \hat{s}_n$ in $\hat{\mathcal{M}}$ corresponding to the trace σ, and therefore $\sigma \in \mathcal{L}(\hat{\mathcal{M}})$. $\square$

By Theorems 3 and 4 it is guaranteed that if all completeness conditions extracted for a given system abstraction are true, then the abstraction is an overapproximation accepting (at least) all system execution traces. We compute the fraction of completeness conditions that are true, denoted by ρ, as a quantitative measure of the degree of completeness of the given system abstraction. The procedure used to check the truth value of the extracted completeness conditions is described in Section 5.3.

5.2.2 Symbolic Definition

Symbolic representations of abstractions have transitions labelled with characteristic functions or predicates for sets of observations, such as the abstractions generated in Chapter 3. A single edge in these graphs in fact corresponds to a set of multiple transitions. There are three benefits of this representation:

1. It reduces the computational cost of the method in the case of very large state spaces when compared to an explicit representation that enumerates concrete transitions.

2. We hypothesize that human engineers prefer the more succinct symbolic presentation over an explicit list. In lieu of evidence, we remark that popular design tools such as Simulink [100] strongly encourage the use of symbolic transition predicates.

3. The symbolic representation also enables an extension of the equivalence checking method to symbolic abstractions over large alphabets.

The standard means to represent sets or relations symbolically is to use characteristic functions. We expect that a single observation $o \in \Omega$ can be described as a valuation of a set of system variables X that range over some domain D, as defined in Chapter 3. We can therefore give a description of a subset $O \subseteq \Omega$ as a characteristic function $f_O : D^{|X|} \longrightarrow \mathbb{B}$, where $\mathbb{B}$ is the set of Boolean truth values. The subset O then corresponds to:

$$O = \{o \in D^{|X|} \mid o \models f_O\} \tag{5.10}$$

where, $o \models f_O \iff f_O(o) = \mathit{true}$.

As defined in Section 5.1, the transitions of our LTSs comprise

1. a source automaton state $\hat{s}_i \in \hat{\mathcal{S}}$,

2. a destination automaton state $\hat{s}_{i+1} \in \hat{\mathcal{S}}$,

3. and an observation $o \in \Omega$.

Of these three components, we represent only the observation symbolically as a transition predicate p. Both automaton states are represented explicitly. Hence, a symbolic transition is a triple $(\hat{s}_i, p, \hat{s}_{i+1})$, which corresponds to the following set of concrete transitions:

$$\{(\hat{s}_i, o, \hat{s}_{i+1}) \mid o \models p\} \tag{5.11}$$

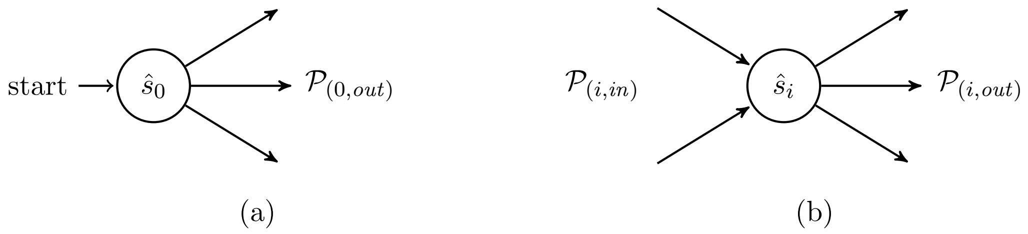


Figure 5.3: Symbolic representation of abstract model states and transitions.

To derive the completeness conditions (5.1) and (5.2) for a symbolic abstraction, we represent the transition relation Δ and the initial state s_0 symbolically as characteristic functions $f_\Delta : \Omega \times \Omega \longrightarrow \mathbb{B}$ and $\mathit{Init}\colon \Omega \longrightarrow \mathbb{B}$ respectively, defined as follows:

$$(o', o) \models f_\Delta \iff \exists s'', s, s' \in \mathcal{S},\ (s'', o', s), (s, o, s') \in \Delta \tag{5.12}$$

$$o \models \mathit{Init} \iff \exists s \in \mathcal{S},\ (s_0, o, s) \in \Delta \tag{5.13}$$

For a given symbolic abstraction, we extract the following conditions encoding a symbolic representation of the completeness hypothesis:

For initial state $\hat{s}_o \in \hat{\mathcal{S}}$, $\forall o \in \Omega$

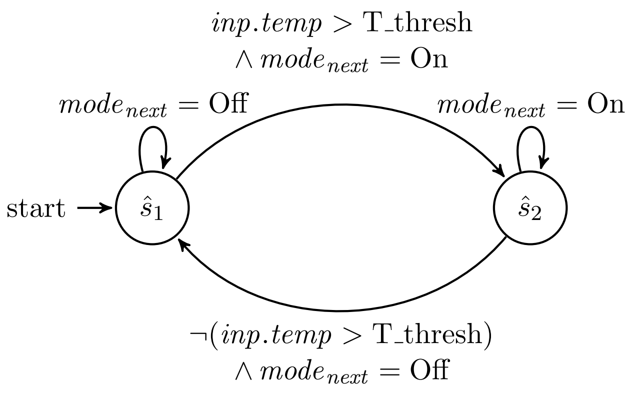
$$o \models \mathit{Init} \implies o \models \bigvee_{p_{out} \in \mathcal{P}_{(0,out)}} p_{out} \tag{5.14}$$

where $\mathcal{P}_{(0,out)}$ is the set of predicates for all outgoing transitions from $\hat{s}_0$, as illustrated in Fig. 5.3a.
And for all states $\hat{s}_i \in \hat{\mathcal{S}}$, $\forall p_{in} \in \mathcal{P}_{(i,in)}$, $\forall o', o \in \Omega$

$$(o' \models p_{in} \wedge (o', o) \models f_\Delta) \implies o \models \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out} \tag{5.15}$$

where $\mathcal{P}_{(i,in)}$ is the set of predicates on the incoming transitions to state $\hat{s}_i$ and $\mathcal{P}_{(i,out)}$ is the set of predicates on outgoing transitions from $\hat{s}_i$, as illustrated in Fig. 5.3b. We illustrate the formulation of the completeness hypothesis for a symbolic abstraction as described above with an example in Fig. 5.4.

Note that the conditions (5.14) and (5.15) are symbolic representations of the completeness conditions (5.1) and (5.2), respectively. For the remainder of the dissertation we use the symbolic representation of the completeness hypothesis as encoded by conditions (5.14) and (5.15).



(a) Given system abstraction

For state $\hat{s}_1$, $\forall o', o \in \Omega$:

$o \models Init \implies o \models (mode_{next} = \text{Off} \vee (inp.temp > \text{T\_thresh} \wedge mode_{next} = \text{On}))$

$(o' \models mode_{next} = \text{Off} \wedge (o', o) \models f_\Delta) \implies$
$\quad o \models (mode_{next} = \text{Off} \vee (inp.temp > \text{T\_thresh} \wedge mode_{next} = \text{On}))$

$(o' \models (\neg(inp.temp > \text{T\_thresh}) \wedge mode_{next} = \text{Off}) \wedge (o', o) \models f_\Delta) \implies$
$\quad o \models (mode_{next} = \text{Off} \vee (inp.temp > \text{T\_thresh} \wedge mode_{next} = \text{On}))$

For state $\hat{s}_2$, $\forall o', o \in \Omega$:

$(o' \models (inp.temp > \text{T\_thresh} \wedge mode_{next} = \text{On}) \wedge (o', o) \models f_\Delta) \implies$
$\quad o \models (mode_{next} = \text{On} \vee (\neg(inp.temp > \text{T\_thresh}) \wedge mode_{next} = \text{Off}))$

$(o' \models mode_{next} = \text{On} \wedge (o', o) \models f_\Delta) \implies$
$\quad o \models (mode_{next} = \text{On} \vee (\neg(inp.temp > \text{T\_thresh}) \wedge mode_{next} = \text{Off}))$

(b) Extracted conditions

Figure 5.4: Completeness hypothesis for a symbolic abstraction.

5.3 Checking the Truth Value of Extracted Conditions

To verify if the completeness conditions evaluate to true for all observations in Ω we use symbolic variables ω', ω to represent the observations o', o in (5.14) and (5.15) respectively, and use a SAT solver to check if there exists an assignment of values in Ω to ω', ω that satisfies the negation of the completeness conditions.

To this end, the negation of the conditions (5.14) and (5.15), represented as

$$\neg(\omega \models Init \implies \omega \models \bigvee_{p_{out} \in \mathcal{P}_{(0,out)}} p_{out}) \tag{5.16}$$

and

$$\neg((\omega' \models p_{in} \wedge (\omega', \omega) \models f_{\Delta}) \implies \omega \models \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out}) \tag{5.17}$$

respectively, is fed to a SAT solver. A satisfying assignment indicates a falsification of the corresponding completeness condition, and serves as a counterexample for the condition. In the event that a satisfying assignment cannot be found, we conclude that the corresponding completeness condition always evaluates to true.

As discussed in Section 5.2, the satisfaction of the completeness hypothesis is sufficient to guarantee completeness, but not necessary. In the event of a falsification of condition (5.14), the SAT solver returns a counterexample $\omega = o$, such that $o \models \mathit{Init}$ and $o \not\models p_{out}, \forall p_{out} \in \mathcal{P}_{(0,out)}$. Since $o \in \mathcal{L}(\mathcal{M})$, this is a non-spurious counterexample indicating missing system behaviour in the learned abstraction, i.e., $\mathcal{L}(\mathcal{M}) \nsubseteq \mathcal{L}(\hat{\mathcal{M}})$. But, in the event of a falsification of condition (5.15), the SAT solver returns a counterexample $\omega' = o'$, $\omega = o$ such that $o' \models p_{in}$, $(o', o) \models f_{\Delta}$ and $o \not\models p_{out}, \forall p_{out} \in \mathcal{P}_{(i,out)}$. Here, it is not guaranteed that the observation o' lies on a system path from the initial system state $s_0 \in \mathcal{S}$. The counterexample may therefore be spurious and may not actually correspond to any missing system behaviour in the abstraction.

5.3.1 Counterexample Analysis

To check if a counterexample $\omega' = o'$, $\omega = o$ for condition (5.15) is spurious, i.e., it does not correspond to any concrete system trace, we use model checking to verify if the observation o' is reachable from s_0. That is, the algorithm checks if there exists a path $\pi = s_0, o_0, s_1, o_1, \ldots, o_{n-1}, s_n$ in $\mathcal{M}$ such that $\bigvee_{i=0}^{n-1}(o_i = o')$ is true. If such a path does not exist, the counterexample is guaranteed to be spurious.

In the event that the counterexample for condition (5.15) is spurious, the corresponding input to the SAT solver is strengthened by adding the clause $\omega' \neq o'$ to (5.17) as follows

$$\neg((\omega' \models p_{in} \wedge (\omega', \omega) \models f_{\Delta}) \implies \omega \models \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out}) \wedge \omega' \neq o' \tag{5.18}$$

The conjunction of $\omega' \neq o'$ guides the SAT solver away from the spurious counterexample $\omega' = o'$, and towards a non-spurious counterexample, if any.

In Chapter 6 we describe how traces exemplifying missing system behaviour can be constructed from non-spurious counterexamples, and used to iteratively learn abstractions until an overapproximating system abstraction is obtained.

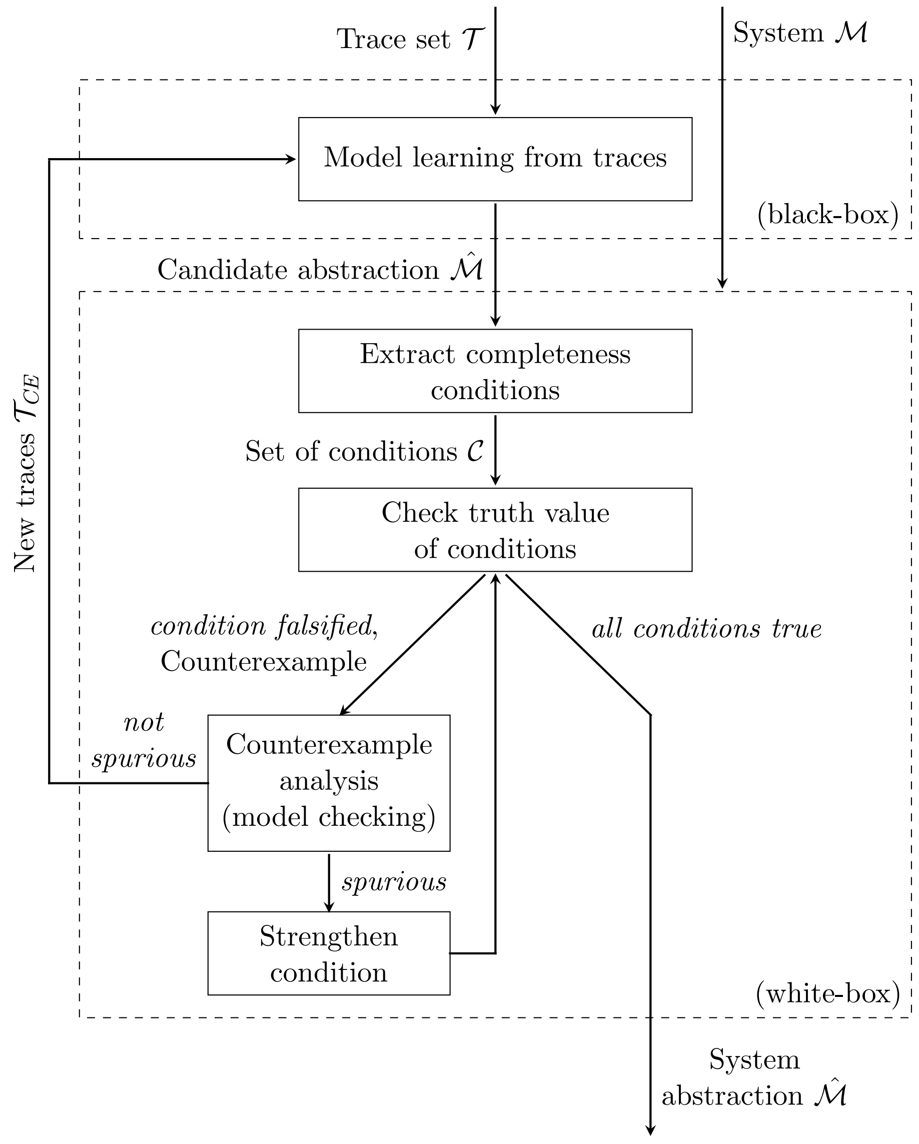
Chapter 6

Active Learning Symbolic Overapproximations

As discussed in Chapter 5, query-based active model-learning implementations often suffer from high query complexity, particularly when applied to learn symbolic abstractions, and are therefore limited to generating partial models for large systems. In this chapter we describe a new active learning approach that integrates equivalence checking using simulation relations described in Chapter 5, with model-learning from traces described in Chapter 3 to generate symbolic system overapproximations with provable completeness guarantees.

As illustrated in Fig 6.1, the approach is a grey-box algorithm. It combines a black-box analysis in the form of model-learning from traces, with a white-box analysis in the form of equivalence checking using simulation relations, that is used to evaluate the degree of completeness for a candidate system model returned by model learning. The model-learning component can be any algorithm that infers an abstraction from a given set of system execution traces. The equivalence checking procedure uses the structure of the learned candidate abstraction to extract conditions that encode a completeness hypothesis.

The algorithm terminates if all conditions are true, returning the learned system overapproximation. In the event of a condition falsification, the counterexample for the condition is checked for spuriousness using model checking. Non-spurious counterexamples are used to construct a set of new traces that exemplify system behaviours identified to be missing from the model. New traces are used to augment the input trace set for model learning and iteratively generate new abstractions until all conditions are true. By Theorems 3 and 4 in Chapter 5, the abstraction returned on algorithm termination is guaranteed to be overapproximating, i.e., the model is guaranteed to accept (at least) all system execution traces.



Trace set $\mathcal{T}$
System $\mathcal{M}$
Model learning from traces
(black-box)
Candidate abstraction $\hat{\mathcal{M}}$
Extract completeness conditions
Set of conditions $\mathcal{C}$
Check truth value of conditions
New traces $\mathcal{T}_{CE}$
condition falsified, Counterexample
all conditions true
not spurious
Counterexample analysis (model checking)
spurious
Strengthen condition
(white-box)
System abstraction $\hat{\mathcal{M}}$

Figure 6.1: Overview of the active model-learning algorithm.

Plan of the chapter

The rest of the chapter is organized as follows. Section 6.1 provides an overview of the active learning approach. Section 6.2 describes the experimental setup, implementation details and the benchmarks used to evaluate the algorithm. It also includes a discussion on the experimental results obtained. Section 6.3 provides a comparison with relevant related work, focusing on two aspects of active model-learning: generating complete models and generating expressive symbolic abstractions.

6.1 Overview

Throughout this chapter we use the notations introduced in Section 5.1, Chapter 5. Given a system $\mathcal{M}$, the goal of our active model-learning approach is to learn an overapproximating system abstraction $\hat{\mathcal{M}}$ such that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$.

An overview of our approach is provided in Fig. 6.1. It consists of the following steps:

1. *Generate candidate abstraction from available traces*: The algorithm learns a candidate abstraction $\hat{\mathcal{M}}$ from an initial set of system execution traces $\mathcal{T}$ using a pluggable model learning algorithm. This is discussed in Section 6.1.1.

2. *Extract completeness conditions*: To evaluate the degree of completeness of the candidate abstraction returned by model learning, the structure of $\hat{\mathcal{M}}$ is used to extract a set of conditions $\mathcal{C}$ that collectively encode a completeness hypothesis. If all conditions are true, it implies $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$. The formulation of the completeness hypothesis and a formal proof of the above claim was provided in Section 5.2, Chapter 5.

3. *Check truth value of extracted conditions*: The algorithm uses a SAT procedure to check the truth value of each condition, and thereby check the hypothesis, as described in Section 5.3, Chapter 5. If all conditions are true, the algorithm returns $\hat{\mathcal{M}}$ as the learned system overapproximation. The extracted conditions are sufficient to prove model completeness, i.e., $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\hat{\mathcal{M}})$, but not necessary. In the event that a condition is falsified, the procedure returns a counterexample. However, a condition falsification may not necessarily indicate missing system behaviour in $\hat{\mathcal{M}}$.

4. *Counterexample analysis*: To check if a condition falsification actually indicates missing system behaviour in $\hat{\mathcal{M}}$, i.e., $\mathcal{L}(\mathcal{M}) \backslash \mathcal{L}(\hat{\mathcal{M}}) \neq \emptyset$, the algorithm uses model checking to determine whether the counterexample returned by the SAT procedure corresponds to a concrete system trace, as described in Section 5.3.1, Chapter 5. If found to be spurious, the condition is strengthened to guide the SAT procedure towards non-spurious counterexamples, if any. The algorithm then repeats step 3.

5. *Generate new abstraction*: A set of new traces $\mathcal{T}_{CE}$ is constructed from non-spurious counterexamples that exemplify missing system behaviour in $\hat{\mathcal{M}}$. These are used as additional trace inputs to the model-learning component in step 1,

which learns a new abstraction admitting $\mathcal{T} \cup \mathcal{T}_{CE}$. Construction of new traces exemplifying missing system behaviour in the model is discussed in Section 6.1.2.

In each iteration i of the algorithm, $\mathcal{T}_{CE_i} \cap \mathcal{L}(\hat{\mathcal{M}}_{i-1}) = \emptyset$, where $\mathcal{T}_{CE_i}$ is the set of new traces constructed by the algorithm in iteration i after evaluating the degree of completeness for the abstraction $\mathcal{L}(\hat{\mathcal{M}}_{i-1})$ generated in the previous iteration. The new abstraction $\hat{\mathcal{M}}_i$ is generated using the set all new traces $\mathcal{T}_{CE_1} \cup \mathcal{T}_{CE_2} \cup \ldots \cup \mathcal{T}_{CE_i}$ and the initial trace set $\mathcal{T}$ as input to model learning.

The methodology described above is similar to Counterexample-Guided Abstraction Refinement (CEGAR) [28], illustrated in Fig. 6.2. The key difference is that CEGAR is a top-down approach that begins by generating an overapproximation, which is progressively pruned to obtain a finer overapproximation. Our algorithm, on the other hand, is a bottom-up approach that progressively extends a candidate abstraction until an overapproximation is obtained.

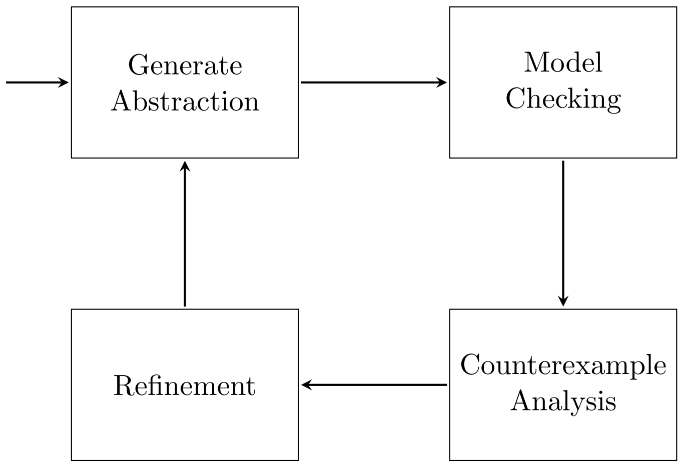


Figure 6.2: Counterexample-Guided Abstraction Refinement (CEGAR) loop.

In the following sections we describe steps 1 and 5 of the algorithm in detail.

6.1.1 Model Learning from Execution Traces

The approach uses a pluggable model-learning component to generate a candidate abstraction from a set of available system execution traces. We impose two requirements on this component:

- Given a set of execution traces $\mathcal{T}$, the model-learning component must return a model $\hat{\mathcal{M}}$ that accepts (at least) all traces in $\mathcal{T}$, i.e., $\mathcal{L}(\hat{\mathcal{M}}) \supseteq \mathcal{T}$.

- The language accepted by the model must be prefix-closed, i.e., if the model accepts a trace σ, then it must also accept all finite prefixes of σ.

There are many model-learning algorithms that satisfy the first requirement [16, 67, 80, 107, 114, 115]. In general, these algorithms operate by employing automaton inference techniques, such as state-merging [16, 77] or SAT [56, 67], to generate a finite state automaton that conforms to a given set of traces.

Among these, our model-learning algorithm [67] described in Chapter 3 satisfies both requirements. To use the other algorithms, simple pre-processing of the input trace set to include all prefixes $\mathit{pref}(\sigma)$ for each trace $\sigma \in \mathcal{T}$, i.e. $\mathcal{T} \leftarrow \bigcup_{\sigma\in\mathcal{T}}\{\sigma' \mid \sigma' \in \mathit{pref}(\sigma)\}$ can be applied. Although this technique enables the generation of prefix-closed automata for conventional state-merging algorithms, it may not always guarantee prefix-closure for models returned by other learning algorithms. A more reliable technique is to convert all non-accepting states that appear on paths to accepting states in the generated finite state automaton to accepting.

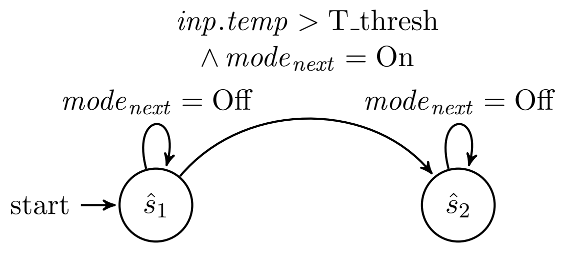
It is straightforward to transform a finite state automaton that accepts a prefix-closed language into an LTS abstraction, as defined in Section 5.1, Chapter 5, by removing the non-accepting states and all transitions that lead into them.

6.1.2 Traces Exemplifying Missing System Behaviour

All non-spurious counterexamples obtained from step 4 of the active learning algorithm are used to construct a set of new traces $\mathcal{T}_{CE}$ that exemplify system behaviours found to be missing from the candidate abstraction.

For each counterexample $\omega = o$ for condition (5.14), we add a trace $\sigma_{CE} = o$ to the set $\mathcal{T}_{CE}$. For each counterexample $\omega' = o', \omega = o$ for condition (5.15), we find the smallest prefix $\sigma' = o_1, o_2, \ldots, o_m$ for all $\sigma \in \mathcal{T}$ such that $o_m \models p_{in}$. We then construct a new trace $\sigma_{CE} = o_1, o_2, \ldots, o_{m-1}, o', o$ for each prefix σ'. Note that since $o' \models p_{in}$, the new trace σ_{CE} does not change the system behaviour exemplified by σ' but merely augments it to include the missing behaviour. The set of new traces $\mathcal{T}_{CE}$ thus generated is used as an additional input to the model-learning component, which in turn generates an abstraction that admits the missing behaviour.

An example run demonstrating the active model-learning algorithm for a Home Climate Control Cooling system is illustrated in Fig 6.3 and described below. First a candidate abstraction is learned from an initial set of system execution traces $\mathcal{T}$. The generated abstraction is provided in Fig. 6.3a. The abstraction models the following sequential system behaviour: the system stays in the Off mode ($\hat{s}_1 \rightarrow \hat{s}_1$), or switches from the Off mode to the On mode when $inp.temp >$ T_thresh ($\hat{s}_1 \rightarrow \hat{s}_2$). The system then switches back to the Off mode and stays in the Off mode indefinitely ($\hat{s}_2 \rightarrow \hat{s}_2$).

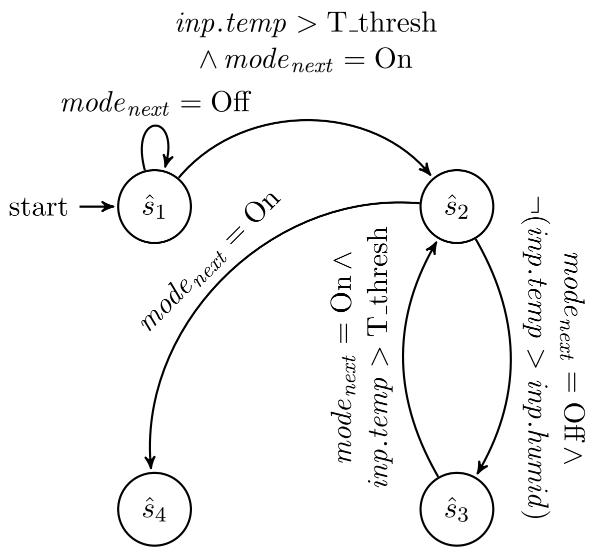


Counterexamples

$o' = (\text{On}, 131072, 0, 70, 40)$
$o = (\text{On}, 0, 0, 70, 40)$

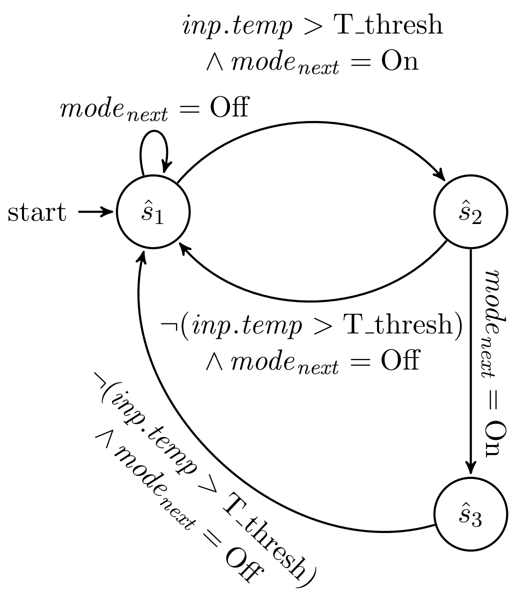
$o' = (\text{Off}, 71, 0, 70, 40)$
$o = (\text{On}, 0, 0, 70, 40)$

(a) Iteration 0, $\rho = 0.5$



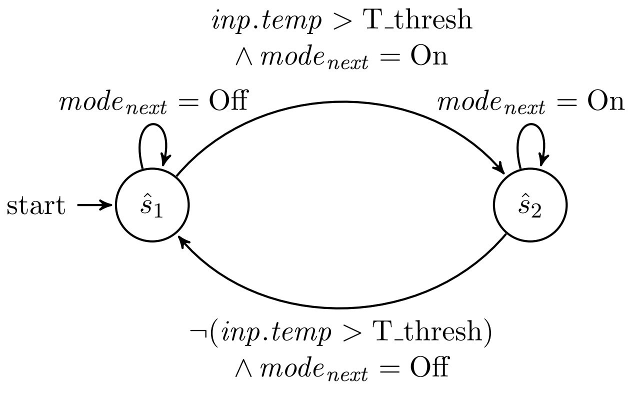
$inp.temp >$ T_thresh $\wedge\ mode_{next} =$ On
$mode_{next} =$ Off
start
$\hat{s}_1$
$\hat{s}_2$
$mode_{next} =$ On
$mode_{next} =$ On $\wedge$ $inp.temp >$ T_thresh
$mode_{next} =$ Off $\wedge$ $\neg(inp.temp > inp.humid)$
$\hat{s}_4$
$\hat{s}_3$

(b) Iteration 1, $\rho = 0.4$



$inp.temp >$ T_thresh $\wedge\ mode_{next} =$ On
$mode_{next} =$ Off
start
$\hat{s}_1$
$\hat{s}_2$
$\neg(inp.temp >$ T_thresh$)$ $\wedge\ mode_{next} =$ Off
$mode_{next} =$ On
$\neg(inp.temp >$ T_thresh$)$ $\wedge\ mode_{next} =$ Off
$\hat{s}_3$

(c) Iteration 2, $\rho = 0.8$



$inp.temp >$ T_thresh $\wedge\ mode_{next} =$ On
$mode_{next} =$ Off
$mode_{next} =$ On
start
$\hat{s}_1$
$\hat{s}_2$
$\neg(inp.temp >$ T_thresh$)$ $\wedge\ mode_{next} =$ Off

(d) Iteration 3, $\rho = 1$

Figure 6.3: Example run of the active learning algorithm for a Home Climate Control Cooling system with $X = \{mode_{next}, inp.temp, inp.humid, \text{T\_thresh}, \text{H\_thresh}\}$.

The structure of the generated abstraction is used to extract the following completeness conditions
For state $\hat{s}_1$, $\forall o', o \in \Omega$:

$$\begin{align} & o \models \mathit{Init} \implies o \models (\mathit{mode}_{\mathit{next}} = \text{Off} \vee \\ & \quad (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{\mathit{next}} = \text{On})) \tag{6.1} \\ & (o' \models \mathit{mode}_{\mathit{next}} = \text{Off} \wedge (o', o) \models f_\Delta) \implies \\ & \quad o \models (\mathit{mode}_{\mathit{next}} = \text{Off} \vee (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{\mathit{next}} = \text{On})) \tag{6.2} \end{align}$$

For state $\hat{s}_2$, $\forall o', o \in \Omega$:

$$\begin{align} & (o' \models (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{\mathit{next}} = \text{On}) \wedge (o', o) \models f_\Delta) \implies \\ & \quad o \models \mathit{mode}_{\mathit{next}} = \text{Off} \tag{6.3} \\ & (o' \models \mathit{mode}_{\mathit{next}} = \text{Off} \wedge (o', o) \models f_\Delta) \implies o \models \mathit{mode}_{\mathit{next}} = \text{Off} \tag{6.4} \end{align}$$

The subsequent completeness hypothesis check indicates falsifications for conditions (6.3) and (6.4). The SAT procedure returns the counterexamples provided in Fig. 6.3a for the two conditions, respectively, and these are found to be not spurious.

The counterexamples exemplify the following system behaviours that are missing from the abstraction in Fig. 6.3a:

1. The first counterexample corresponding to a falsification of condition (6.3) indicates that after the system switches from the Off mode to the On mode ($\hat{s}_1 \rightarrow \hat{s}_2$), the system may remain in the On mode.

2. The second counterexample corresponding to a falsification of condition (6.4) indicates that when the system is in the Off mode after switching from the On mode ($\hat{s}_2 \rightarrow \hat{s}_2$), the system may switch back to the On mode.

The counterexamples are used to construct new traces that serve as additional inputs to the model-learning component, which in turn generates the abstraction in Fig. 6.3b. Note that the new abstraction now captures the system behaviours identified to the missing from the old model: $\hat{s}_1 \rightarrow \hat{s}_2 \rightarrow \hat{s}_4$ captures missing behaviour 1 enumerated above and $\hat{s}_2 \rightarrow \hat{s}_3 \rightarrow \hat{s}_2$ captures missing behaviour 2.

The abstractions generated for subsequent iterations of active learning are provided in Fig. 6.3c–6.3d. All conditions extracted from the abstraction in Fig. 6.3d evaluate to true, i.e., $\rho = 1$. Thus, the algorithm terminates, returning the model in Fig. 6.3d as the final generated system overapproximation.

6.2 Evaluation and Results

6.2.1 Implementation

We implement the active learning approach using the T2M tool as the model learning component. We use CBMC v5.35 [29] to implement the procedure that evaluates degree of completeness for a learned model. The SAT solver in CBMC is used to check the truth value of each extracted condition.

CBMC is used to perform k-induction [98] to verify if the counterexample for a condition check is spurious. This is done by asserting that there does not exist a concrete system path corresponding to the counterexample. If both the base case and step case for k-induction hold, it is guaranteed that the counterexample is spurious, while a violation in the base case indicates otherwise. However, in the event of a violation only in the step case, there is no conclusive evidence for the validity of the counterexample. Since we are interested in generating a system overapproximation, we treat such a counterexample as we would a non-spurious counterexample.

We use a constant value of $k = 10$ for our experiments. Note that we only discard those counterexamples that k-induction guarantees to be spurious. This ensures that, irrespective of the value used for k, all non-spurious counterexamples are used for subsequent model-learning iterations.

For complex systems, model-checking for counterexample analysis as described in Section 5.3.1, Chapter 5 can be computationally expensive in practice. Here, simulation-based techniques [36, 93] could be a pragmatic alternative to explore system paths to check if the observation in the counterexample is reachable.

6.2.2 Benchmarks

To evaluate the algorithm, we attempt to reverse-engineer a set of LTSs from their respective C implementations. For this purpose, we use the dataset of Simulink Stateflow example models [101], available as part of the Simulink documentation. We select this dataset as these example models are state machines that can serve as ground-truth for our evaluation.

For each Stateflow example, we use Embedded Coder (MATLAB 2018b) [99] to automatically generate a corresponding C code implementation. The generated C implementation is used as the system $\mathcal{M}$ in our experiments. To collect traces, we instrument the implementation to observe a set of program variables X. The set of observations Ω is the set of valuations for all variables $x \in X$.

The dataset of Stateflow example models comprises 51 examples that are available in MATLAB 2018b. Out of the 51 examples, Embedded Coder fails to generate code for 7; a total of 13 have no sequential behaviour and 3 implement Recursive State Machines (RSM) [5][1]. We use the remaining 28 examples for our evaluation.

The majority of the Stateflow example models feature predicates on the transition edges. Some of the Stateflow example models are implemented as multiple parallel and hierarchical state machines. Our goal is to reproduce each of these state machines from execution traces, and we therefore obtain a total of 45 target state machines from the 28 Stateflow examples. These serve as our benchmarks for evaluation. A mapping of each Stateflow example model to its set of target state machine benchmarks is provided in Table A.1 in Appendix A. The algorithm implementation and benchmarks are available online [69].

6.2.3 Experiments and Results

For each benchmark, we generate an initial set of 50 traces, each of length 50, by executing the C implementation with randomly sampled inputs. This set of traces and the C implementation are fed as input to the algorithm, which in turn attempts to learn an abstraction overapproximating system behaviours. The results are summarised in Table 6.1.

Each entry in the table from B1 to B45 corresponds to a target state machine that we wish to reverse-engineer. These are grouped by the Stateflow example that they belong to. We record the number of model learning iterations $\#iter$, the number of states $|\hat{\mathcal{S}}|$ and degree of completeness ρ for the final abstraction, the total runtime in seconds $T(s)$ and the percentage of the total runtime attributed to model learning, denoted by $\%T_m$. We also record the cardinality of the set X (the number of variables) for every Stateflow model. We set a timeout of 24 h for our experiments. For benchmarks that time out, we present the results for the candidate abstraction generated right before timeout.

Since the algorithm is designed to generate overapproximating system abstractions, the inferred model for a target state machine could admit traces that are outside the language of target machine, and therefore may not be accurate. We assess the accuracy of the final generated abstraction by assigning a score d computed as

[1]In this work we learn abstractions as finite state automata (Section 6.1.1), which are known to represent exactly the class of regular languages. Reverse-engineering an RSM from traces requires a modelling formalism that is more expressive than finite state automata, such as Push-Down Automata (PDA) [44], which is outside the scope of this work. In the future, we wish to look at extensions of this work to generate RSMs.

the fraction of state transitions in the target state machine that match corresponding transitions in the abstraction we generate. This is done by semantically comparing the corresponding transition predicates in the target state machine and the abstraction using CBMC. For hierarchical Stateflow models, we flatten the hierarchy and compare the abstraction with the flattened state machine.

Table 6.1: Results of experimental evaluation of the active learning algorithm.

	$\lvert X\rvert$	Our Algorithm						Random Sampling			
		$\#iter$	$\lvert\hat{\mathcal{S}}\rvert$	ρ	d	$T(s)$	$\%T_m$	$\lvert\hat{\mathcal{S}}\rvert$	ρ	d	$T(s)$
B1	4	8	5	**1**	**1**	684.1	35.3	4	0.2	0.1	38.2
B2	5	7	3	**1**	**1**	54.3	44.3	3	0.8	0.7	64
B3		6	5	**1**	**1**	90.1	50.8	5	0.9	0.6	89.5
B4	3	2	3	**1**	**1**	11.5	44.3	4	**1**	**1**	56.5
B5	3	1	1	0.5	0.2	——timeout——		2	0.5	0.4	31
B6	7	1	2	**1**	**1**	4.9	17.7	2	**1**	**1**	72.3
B7	5	2	3	**1**	**1**	17	21.3	3	**1**	**1**	33.9
B8		3	3	**1**	**1**	360.9	1.4	3	**1**	**1**	35.2
B9	3	9	4	**1**	**1**	162.1	64.6	3	0	0.2	52.9
B10	2	1	4	**1**	**1**	8.8	47.9	4	**1**	**1**	67
B11		19	6	**1**	**1**	≈20.8 h	2.1	12	0.3	0.8	953.7
B12		9	5	**1**	**1**	≈4.4 h	0.5	7	**1**	**1**	282.8
B13	13[b]	1	4	**1**	**1**	10.9	33.9	4	**1**	**1**	416.1
B14		8	5	**1**	**1**	695.6	49.2	5	**1**	**1**	876.9
B15		1	2	**1**	**1**	4.6	19.4	2	**1**	**1**	138
B16		2	6	**1**	**1**	123.5	13.4	6	0.9	0.8	966
B17	6	1	4	**1**	**1**	7.8	41.1	4	**1**	**1**	70.6
B18		4	5	**1**	**1**	45.7	33.6	5	0.4	0.3	107.8
B19	11	3	5	**1**	**1**	49.1	48.4	8	0.8	0.8	105.6
B20		4	4	**1**	**1**	39	43.2	6	**1**	**1**	81.8
B21	6	5	4	**1**	**1**	71.1	33.8	5	0.6	0.8	72.4
B22		8	4	**1**	**1**	260.4	37.4	4	0.6	0.6	793.1
B23		1	3	**1**	**1**	7.3	22.1	3	**1**	**1**	503.3
B24	16	1	3	**1**	**1**	7.1	16.4	3	**1**	**1**	691.5
B25		14	4	**1**	**1**	386.3	38.4	4	0.9	0.8	1248.3
B26		1	3	**1**	**1**	7.4	16.7	3	**1**	**1**	1188.3

Continued on next page

[b]The dataset includes another implementation of this system with similar results. We present the results for only one of them.

Table 6.1 – *Continued from previous page*

	$\|X\|$	Our Algorithm						Random Sampling			
		$\#iter$	$\|\hat{\mathcal{S}}\|$	ρ	d	$T(s)$	$\%T_m$	$\|\hat{\mathcal{S}}\|$	ρ	d	$T(s)$
B27		1	3	**1**	**1**	7.8	16.3	3	**1**	**1**	1974.2
B28	2	1	2	**1**	**1**	2.9	29.8	2	**1**	**1**	29.6
B29	3	3	7	**1**	**1**	52.8	66.8	9	**1**	**1**	124
B30	2	1	3	**1**	**1**	5.9	28.2	3	**1**	**1**	52.8
B31	3	5	3	**1**	**1**	34.6	27.7	4	**1**	**1**	35.9
B32	2	6	5	**1**	**1**	62.8	53.3	5	**1**	**1**	88.2
B33	4	2	3	**1**	**1**	10.7	40.4	4	0.6	0.7	99.3
B34	2	1	4	**1**	**1**	4.5	38.1	4	**1**	**1**	32.1
B35	3	4	5	**1**	**1**	47.4	56	7	**1**	**1**	89.1
B36	1	1	1	**1**	**1**	142	0.4	1	**1**	**1**	21.8
B37		1	3	**1**	**1**	141.5	0.7	3	**1**	**1**	25.5
B38	2	6	4	**1**	**1**	78.9	15.4	4	**1**	**1**	36
B39	3	1	4	**1**	**1**	7.8	32	4	**1**	**1**	41.5
B40	4	2	3	**1**	**1**	116.7	3.4	4	**1**	**1**	38.9
B41		2	5	**1**	**1**	157.8	4.5	6	**1**	**1**	47.4
B42	2	1	4	**1**	**1**	4.2	41.2	4	**1**	**1**	34.8
B43	8	2	4	**1**	**1**	20.6	56.8	4	**1**	**1**	60.1
B44		2	4	**1**	**1**	19.9	48	4	**1**	**1**	71.9
B45		1	3	**1**	**1**	4.2	24	3	**1**	**1**	43

6.2.3.1 Runtime

The active learning algorithm is able to generate overapproximations in under 12 min for the majority of the benchmarks. For the benchmarks that take more than 1 h to terminate, namely B11 and B12, we see that the model checker tends to go through a large number of spurious counterexamples before arriving at a non-spurious counterexample for a condition falsification. This is because, depending on the size of the domain for the variables $x \in X$, there can be a large number of possible valuations that falsify an extracted condition, of which very few may actually correspond to a concrete system trace. In such cases, runtime can be improved by strengthening the conditions with domain knowledge to guide the SAT procedure towards non-spurious counterexamples. For the B5 benchmark, the SAT procedure takes a long time to check each condition. This is because the implementation features several operations, such as memory access and array operations, that especially increase the complexity of the SAT check, and the solving runtime as a consequence.

6.2.3.2 Accuracy of the Generated Models

The algorithm terminates when $\rho = 1$ and therefore, by Theorems 3 and 4 in Chapter 5 the algorithm is guaranteed to generate an overapproximation on termination. For the benchmarks that terminate, the generated abstraction is found to accurately capture the behaviour of the corresponding state machine ($d = 1$).

6.2.3.3 Number of Learning Iterations

As described in Section 6.1, in each learning iteration i, $\mathcal{L}(\hat{\mathcal{M}}_i) \supseteq \mathcal{T}_{CE_i} \cup \mathcal{T}_{CE_{i-1}} \cup \ldots \cup \mathcal{T}_{CE_1} \cup \mathcal{T}$ and $\mathcal{T}_{CE_i} \cap \mathcal{L}(\hat{\mathcal{M}}_{i-1}) = \emptyset$. The algorithm terminates when $\mathcal{L}(\hat{\mathcal{M}}_i) \supseteq \mathcal{L}(\mathcal{M})$. The number of learning iterations depends on $|\mathcal{L}(\mathcal{M}) \setminus \mathcal{L}(\hat{\mathcal{M}}_0)|$, where $\hat{\mathcal{M}}_0$ is the abstraction generated from the initial trace set $\mathcal{T}$.

To evaluate the impact of the seed traces on the number of iterations that the algorithm requires, we have run our experiments without any seed traces, i.e., using $\mathcal{L}(\hat{\mathcal{M}}_0) = \emptyset$. We observe that on an average the number of iterations increases ≈ 5 times compared to the number of iterations reported in Table 6.1.

6.2.4 Comparison with Random Sampling

We performed a set of experiments to check if random sampling is sufficient to learn complete models, i.e., abstractions that accept all system execution traces. A million randomly sampled inputs are used to execute each benchmark. Generated traces are fed to the T2M tool to passively learn a model. For $\approx 29\%$ of the benchmarks, random sampling fails to produce a model that accepts all system traces ($\rho < 1$).

6.2.5 Threats to Validity

The key threat to the validity of our experimental claim is benchmark bias. We have attempted to limit this bias by using a set of benchmarks that was curated by others. Further, we use C implementations of Simulink Stateflow models that are auto-generated using a specific code generator. Although there is diversity among these benchmarks, our algorithm may not generalise to software that is not generated from Simulink models, or software generated using a different code generator.

While the active-learning implementation used for our experiments produces an accurate model, i.e., $d = 1$ for the benchmarks that terminate, there is no formal guarantee that the algorithm delivers this in all cases. The accuracy of generated models may vary depending on the algorithm used as the model-learning component and its ability to consolidate trace information into symbolic abstractions. The procedure to evaluate degree of completeness only formally guarantees the generation of a system overapproximation on algorithm termination.

6.3 Comparison With Related Work

6.3.1 Overview

Active model-learning implementations largely consist of two components: a model-learning algorithm that generates a model from a set of traces, and an oracle that evaluates the learned model to identify missing and/or wrong behaviours.

The state-merge [16, 56, 77] algorithm and query-based learning [7, 66, 92] are popular choices for the model-learning component. State-merge algorithms reverse-engineer models by constructing a PTA from the traces and identifying equivalent states to be merged in the PTA. The L* algorithm forms the basis of query-based active learning, where the learning algorithm poses equivalence and membership queries to an oracle. The responses to the queries are recorded into an observation table, that is eventually used to construct an automaton. These algorithms have been discussed in detail in Section 2.1, Chapter 2.

The oracle for active-learning can be implemented as a black-box or a white-box procedure. One such black-box oracle implementation uses model checking, where pre-defined LTL system properties are checked against the generated model to identify wrong behaviours [51, 103, 106, 111]. For query-based learning in a black-box setting, membership queries are implemented as tests on the system. Equivalence queries are often approximated using techniques such as conformance testing [36], through a finite number of membership queries. For a white-box oracle implementation, algorithms use techniques such as fuzzing [102] and symbolic execution [71].

In the broader literature of equivalence checking, particularly in the field of Electronic Design Automation (EDA), several techniques are used to prove if two representations or implementations of a system exhibit the same behaviour [9, 30, 49, 73, 83, 84, 86, 109]. Among these, the most closely related to our work are the techniques based on SAT and Bounded Model Checking (BMC) [31]. These techniques primarily

Table 6.2: Summary of related active model-learning implementations.

Oracle	Model-learning algorithm		Generated Model Characteristics	
	State-Merge	L*	Symbolic	Complete
Black-box	[37, 106] [111, 113]	[1, 2, 15, 24, 25, 51] [62, 91, 103, 108]	[1, 2, 15, 25] [24, 62, 108]	
White-box		[17, 27, 33, 42] [45, 46, 59, 102]	[17, 45]	[17]

check for input/output equivalence, i.e., assuming the inputs to each implementation are equal, the corresponding outputs are equal.

The SAT based techniques [49, 83] generally operate by representing the output for each implementation as a Boolean expression over the inputs. The clause obtained by an XOR of these expressions is fed to a SAT solver. If a satisfying assignment is found, it implies that the outputs are not equal and thus the two implementations are not equivalent. In BMC-based equivalence checking [30, 73] the two implementations are unwound a finite number of times, and translated into a formula representing behavioural equivalence that is fed to a SAT solver. In [73] input/output equivalence is verified on abstract overapproximations of the implementations. Equivalence is modelled as a safety property that is checked using CEGAR on the product of the abstract models. Counterexample analysis for CEGAR is performed by simulating the abstract counterexample on the concrete model using BMC.

In this section, we will primarily focus on equivalence checking in the context of active model learning. There are many active learning techniques that use various combinations of model-learning algorithms and oracle implementations discussed above. In this chapter, we described an algorithm that uses a white-box oracle implemented using SAT solving and model checking, that when combined with a symbolic-model learning algorithm can learn symbolic overapproximations for a system. A summary of related active learning implementations is provided in Table 6.2. In the following sections we describe these techniques in detail and compare them in terms of generated model completeness and expressivity.

6.3.2 Learning System Overapproximations

State-merge algorithms are predominantly passive and generated abstractions admit only those system behaviours exemplified by the traces [56, 77, 80, 114, 115]. One of the earliest active algorithms using state-merge is Query-Driven State Merging

(QSM) [37], where model refinement is guided by responses to membership queries posed to an end-user. Other active versions of state-merge use model checking [106, 111] and model-based testing [113] to identify spurious behaviours in the generated model. In [106, 111] a priori known LTL system properties are checked against the generated model. Counterexamples for property violations serve as negative traces for automaton refinement. In [113], tests generated from the learned model are used to simulate the system to identify any discrepancies. However, abstractions generated by these algorithms are not always guaranteed to accept all system traces.

Query-based learning algorithms, such as Angluin's L* algorithm and its variants [8, 66, 70, 97], can in principle generate exact system models. But the absence of an equivalence oracle, in practice, often restricts their ability to generate exact models or even system overapproximations. In a black-box setting, membership queries are posed as tests on the system. The elicited response to a test is used to classify the corresponding query as accepting or rejecting. Equivalence queries are often approximated through a finite number of membership queries [1, 25, 97] on the system. The membership queries are generated using techniques such as conformance testing or random walks of the hypothesis model.

An essential pre-requisite to enable black-box query-based model learning is that the system can be simulated with an input sequence to elicit a response or output, such as systems modelled as Mealy machines or register automata. Moreover, obtaining an adequate approximation of an equivalence oracle may require a large number of membership queries, that is exponential in the number of states in the system. The resulting high query complexity constrains these algorithms to learning only partial models for large systems [60, 61].

One way to address these challenges is to combine model learning with white-box techniques, such as fuzzing [102], symbolic execution [17, 46, 59] and model checking [33, 42], to extract system information at a lower cost. But, these are not always guaranteed to generate system overapproximations.

In [102], model learning is combined with mutation based testing that is guided by code coverage. This proves to be more effective than conformance testing, but the approach does not always produce complete models. In [46, 59], symbolic execution is used to answer membership queries and generate component interface abstractions modelling safe orderings of component method calls. Sequences of method calls in a query are symbolically executed to check if they reach an a priori known unsafe state. However, learned models may be partial as unsafe method call orderings that are unknown to the end-user due to insufficient domain knowledge are missed by the approach. The Sigma* [17] algorithm combines L* with symbolic execution to iteratively learn an overapproximation in parallel to the models learned using L*. The algorithm terminates when the hypothesis model equals the overapproximation, and therefore generates exact system models. In [33, 42], model checking is used in combination with model learning for assume guarantee reasoning. The primary goal of the approach is not to generate an abstract model of a component and may therefore terminate before generating a complete model.

Closely related to our work are the algorithms that use L* in combination with black-box testing [91] and model checking [51, 103]. The latter use predefined LTL properties, similar to [106, 111], that are checked against the generated abstraction. Any counterexamples are checked with the system. This either results in the conclusion that the system does not satisfy the property, or leads to a refinement of the abstraction to remove incorrect behaviours. Black-box testing [91] may be a pragmatic approach to identify missing behaviours for an abstraction by simulating the learned model with a set of system traces. However, it is not guaranteed that the model admits all system traces, as this requires a complete set of execution traces.

6.3.3 Learning Symbolic Models

An open challenge with query-based active model learning is learning symbolic models. Many practical applications of L* [27, 33] and its variants are limited to learning system models defined over an a priori known finite alphabet consisting of Boolean events, such as function calls. Maler and Mens developed a symbolic version of the L* algorithm [81, 82] to extend model inference to large alphabets by learning symbolic models where transitions are labelled with partitions of the alphabet.

In [2], manually constructed mappers abstract concrete values into a finite symbolic alphabet. However, different applications would require different mappers to be manually specified, which can be a laborious and error prone process. The authors in [1] propose a CEGAR-based method to automatically construct mappers for a restricted class of Mealy machines that test for equality of data parameters, but do not allow any data operations. In [62], CEGAR is used for automated alphabet abstraction refinement to preserve determinism in the generated abstraction. Given an abstraction, the refinement procedure is triggered by counterexamples exposing non-determinism in the current abstraction.

The MAT* algorithm [8] generates SFAs, where the transitions carry predicates over a Boolean algebra that can be efficiently learned using membership and equivalence queries. The input to the algorithm is a membership oracle, an equivalence oracle and a learning algorithm to learn the Boolean algebra of the target SFA. The algorithm has been used to learn SFAs over Boolean algebras with finite domain, the equality algebra, BDD algebra and SFAs over SFAs that accept string of strings. But, designing and implementing oracles for richer models such as SFAs over the theory of linear integer arithmetic is not straightforward, as it would require answering queries comprising valuations of multiple variables, some of which could have large and possibly infinite domains.

In [15], an inferred Mealy machine is converted to a symbolic model in a post-processing step. The algorithm, however, is restricted to learning models with simple predicates such as equality/inequality relations. The algorithm in [108] is restricted to generating Mealy machines with a single timer. Sigma* [17] extends the L* algorithm to learn symbolic models of software. Dynamic symbolic execution is used to find constraints on inputs and expressions generating output to build a symbolic alphabet. But, behaviours modelled by the generated abstraction are limited to input-output steps of a software. Although the algorithm generates symbolic models that are complete, as illustrated in Table 6.2, an implementation of the algorithm is not publicly available for an experimental comparison.

The SL* algorithm [25] extends query-based learning to infer register automata. We compared our model-learning implementation T2M with the SL* implementation RALib in Chapter 4. In this chapter we attempt to reverse-engineer the Simulink state machine benchmarks modelled as IORA using RALib. We present the results obtained for benchmarks B1 and B6 below.

We modelled state machine B6 of a Home Climate Control Cooling system as an IORA with input action $\mathit{check}(\mathit{inp.temp})$ that takes a parameter $\mathit{inp.temp}$, and output actions On() and Off() representing the operation modes of the system, as illustrated in Fig. 6.4. This is fed as the system-under-learning (SUL) to RALib. The

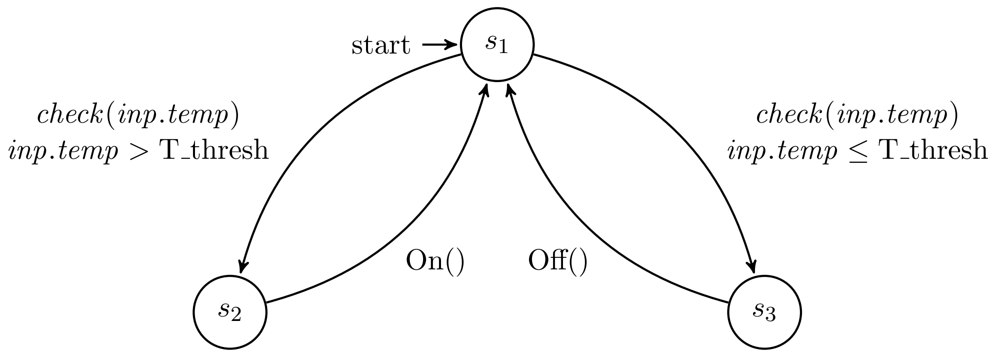
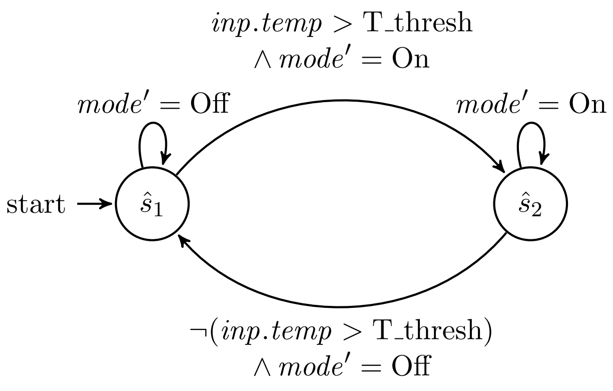
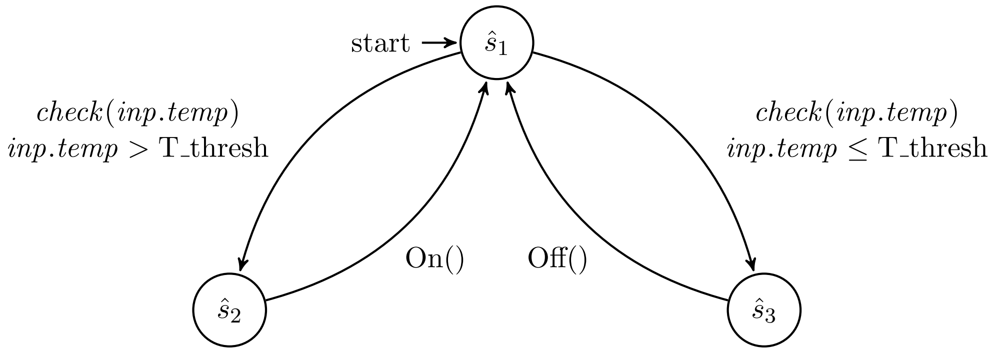

start
s_1
$\mathit{check}(\mathit{inp.temp})$
$\mathit{inp.temp} > \text{T\_thresh}$
$\mathit{check}(\mathit{inp.temp})$
$\mathit{inp.temp} \leq \text{T\_thresh}$
On()
Off()
s_2
s_3

Figure 6.4: IORA modelling a Home Climate Control Cooling system (B6).



$inp.temp > \text{T\_thresh}$
$\wedge\, mode' = \text{On}$
$mode' = \text{Off}$
$mode' = \text{On}$
start
$\hat{s}_1$
$\hat{s}_2$
$\neg(inp.temp > \text{T\_thresh})$
$\wedge\, mode' = \text{Off}$

(a) Our algorithm



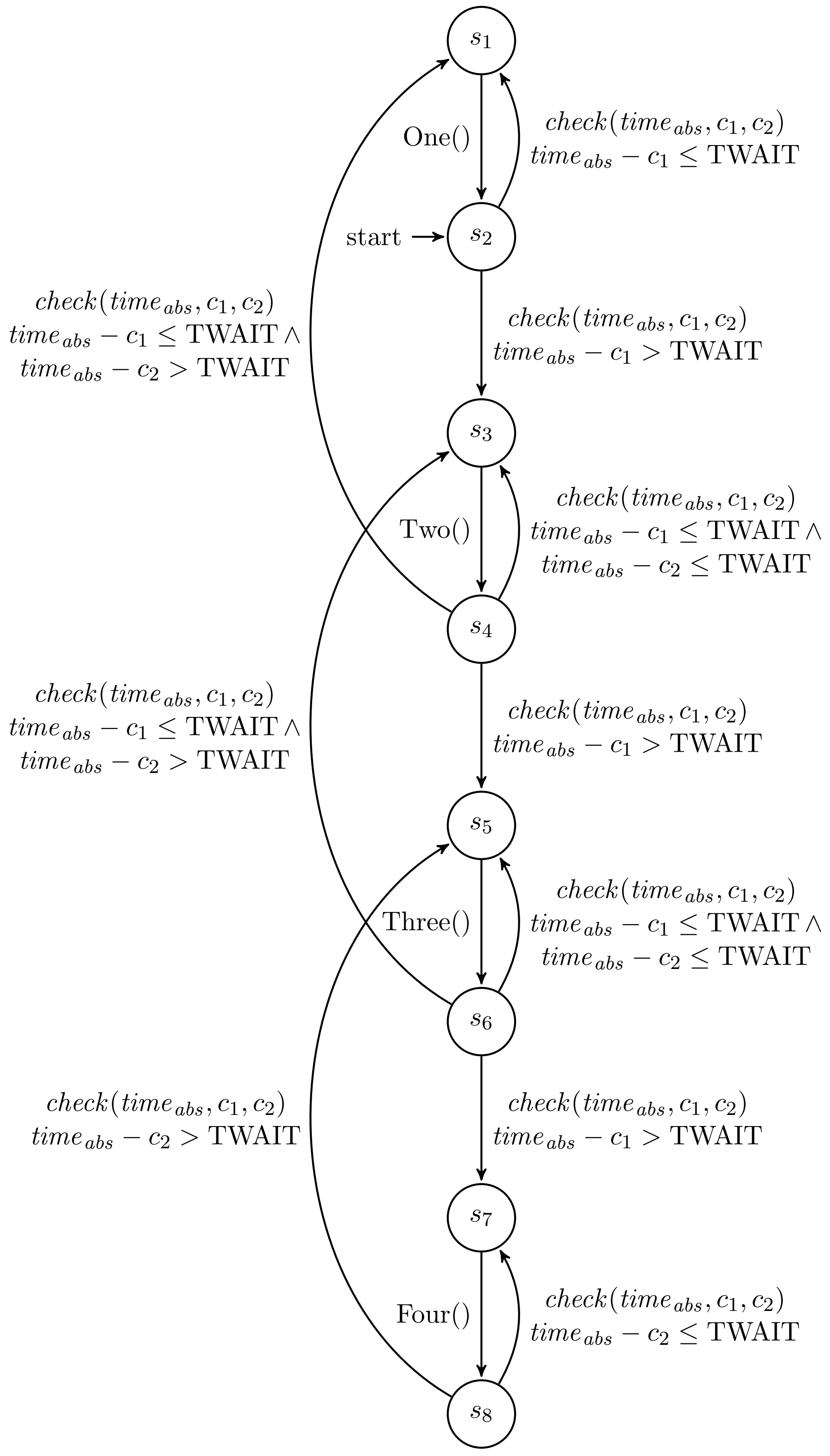
(b) RALib

Figure 6.5: Abstractions generated for a Home Climate Control Cooling system (B6).

models generated by our active learning approach and RALib are provided in Fig. 6.5. Similar to our algorithm, RALib was able to accurately capture the system behaviours and generate an exact representation of the SUL, as is evident from Fig. 6.5b.

As illustrated in Fig. 6.6, we modelled state machine B1 of an Automatic Transmission Gear system as an IORA with input action $check(time_{abs}, c_1, c_2)$ that takes parameters $time_{abs}$, c_1 and c_2, and output actions One(), Two(), Three() and Four() representing the four gears in the system. This is fed to RALib as the SUL. The abstractions generated by our algorithm and RALib are provided in Fig. 6.7. RALib was only able to generate the partial model illustrated in Fig. 6.7b before timing out at 10 h. Our algorithm, on the other hand, was able to generate a complete model (Fig. 6.7a) in less than 12 minutes, as evidenced in Table 6.1.

The basic tool implementation of RALib currently supports predicates featuring equality over integers and inequality over real numbers. In addition to equality/inequality relations, automaton transitions may also feature simple arithmetic expressions such as increment by 1 and *sum*. However, these are still in the development stage and only partially supported, often tailored to specific domains such

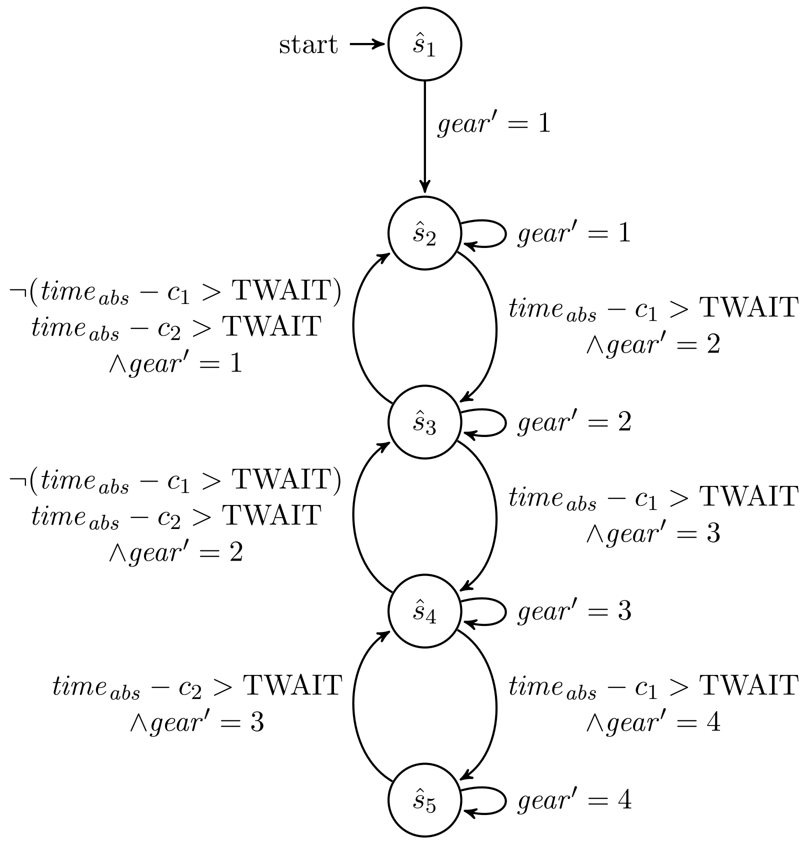


s_1
One()
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 \leq \text{TWAIT}$
start
s_2
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 \leq \text{TWAIT} \wedge$
$time_{abs} - c_2 > \text{TWAIT}$
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 > \text{TWAIT}$
s_3
Two()
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 \leq \text{TWAIT} \wedge$
$time_{abs} - c_2 \leq \text{TWAIT}$
s_4
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 \leq \text{TWAIT} \wedge$
$time_{abs} - c_2 > \text{TWAIT}$
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 > \text{TWAIT}$
s_5
Three()
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 \leq \text{TWAIT} \wedge$
$time_{abs} - c_2 \leq \text{TWAIT}$
s_6
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_2 > \text{TWAIT}$
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_1 > \text{TWAIT}$
s_7
Four()
$check(time_{abs}, c_1, c_2)$
$time_{abs} - c_2 \leq \text{TWAIT}$
s_8

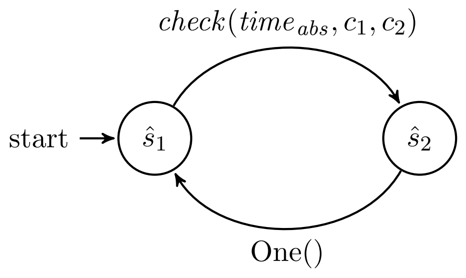
Figure 6.6: IORA modelling gear-shift logic for an Automatic Transmission Gear system (B1).

as TCP protocols [43]. Due to the high query complexity it is not obvious how the approach can be generalised to efficiently learn symbolic models over richer theories.

An extension of the SL* algorithm [45] uses taint analysis to improve performance by extracting constraints on input and output parameters. However, it currently does not allow the analysis of multiple or more involved operations on data values.


start
$\hat{s}_1$
$gear' = 1$
$\hat{s}_2$
$gear' = 1$
$\neg(time_{abs} - c_1 > \text{TWAIT})$ $time_{abs} - c_2 > \text{TWAIT}$ $\wedge gear' = 1$
$time_{abs} - c_1 > \text{TWAIT}$ $\wedge gear' = 2$
$\hat{s}_3$
$gear' = 2$
$\neg(time_{abs} - c_1 > \text{TWAIT})$ $time_{abs} - c_2 > \text{TWAIT}$ $\wedge gear' = 2$
$time_{abs} - c_1 > \text{TWAIT}$ $\wedge gear' = 3$
$\hat{s}_4$
$gear' = 3$
$time_{abs} - c_2 > \text{TWAIT}$ $\wedge gear' = 3$
$time_{abs} - c_1 > \text{TWAIT}$ $\wedge gear' = 4$
$\hat{s}_5$
$gear' = 4$

(a) Our algorithm



(b) RALib

Figure 6.7: Abstractions modelling gear-shift logic for an Automatic Transmission Gear system (B1).

Chapter 7

Conclusions and Future Work

In this chapter we summarize the key contributions of the work reported in this dissertation and suggest some potential directions for future research.

7.1 Summary

In this dissertation we presented a methodology to infer concise, accurate and expressive symbolic system overapproximations with provable completeness guarantees using only system execution trace data.

In Chapter 1 we began by briefly describing the need for abstraction and enumerating some of challenges encountered when applying existing inference techniques to learn abstract system models for system-wide analysis of large and complex system. In Chapter 2, we covered an extensive survey of related model-learning methodologies, and introduced formal preliminaries on program synthesis and equivalence relations.

We described a novel approach to generate symbolic abstractions for a system using only system trace data in Chapter 3. Abstractions are inferred as symbolic finite automata from system execution traces using a combination of SAT and program synthesis. We presented a SAT formulation for constructing a minimal finite state automaton from only positive traces. Additionally, we integrated algorithmic methods designed to enable SAT-based model-learning to scale to long traces, based on trace segmentation and incremental learning. We also extended SAT-based automaton inference from traces to learn symbolic finite automata over large and possibly infinite alphabets, using program synthesis to consolidate system execution trace data into syntactic expressions that are not explicit in the execution traces.

We presented an experimental evaluation of the model-learning algorithm in Chapter 4, on benchmarks from a variety of domains including system-on-chip and internet-of-things. By means of experiments we showed that the algorithmic optimizations introduced in Chapter 3 produce a significant reduction in algorithm runtime. Our comparison experiments demonstrate that the model-learning algorithm generates abstractions that are concise, accurate and often more expressive than models generated by other algorithms in relevant literature.

Although the model-learning algorithm described in Chapter 3 generates abstractions that are guaranteed to accept (at least) all execution traces in the given learning sample, the learned models may be partial. To address this, we developed a method based on equivalence checking using simulation relations to evaluate the degree of completeness of an abstraction inferred from execution traces.

In Chapter 5 we described the formulation of a completeness hypothesis to evaluate the degree of completeness for a given system abstraction. We formally proved that the satisfaction of the formulated hypothesis is sufficient to guarantee that a simulation relation can be constructed between the system and the learned abstraction, and further that the existence of a simulation relation is sufficient to guarantee that the abstraction is overapproximating.

In Chapter 6 we described an active model-learning approach that integrates model-learning from traces with equivalence checking using simulation relations to generate overapproximating system abstractions with provable completeness guarantees. We also presented an experimental evaluation of the active learning approach, and showed that the generated abstractions are more expressive that abstractions learned by existing active learning implementations.

7.2 Future Work

The work reported in this dissertation partially addresses some of the challenges for learning expressive symbolic system abstractions enumerated in Chapter 1. In this section, we identify few avenues for future research.

Increasing Scalability

There are two potential avenues to explore for increasing algorithm scalability

- One of the main bottlenecks for scalability in our model-learning approach is automaton construction using SAT. The use of SAT provides provable guarantees of correctness and is therefore favourable. It however comes at the cost of increased runtime. Although the optimizations we introduce significantly reduce runtime, the algorithm may not scale as well to very large traces, such as log data from industrial applications. A more efficient SAT encoding for automaton construction from positive traces could be a promising step forward.

- As discussed in Section 6.2.3.1, Chapter 6, the active learning approach often goes through a large number of spurious counterexamples before arriving at a non-spurious counterexample, if any, which increases algorithm runtime. Exploring ways to guide the condition check procedure towards non-spurious counterexamples is a potential avenue for future research.

Improving Transition Predicate Inference

Our model-learning algorithm generates symbolic finite automata where the transitions are labelled with predicates that are inferred from traces using program synthesis. The scope of the inferred predicates is therefore limited to the type of expressions that program synthesis can generate. For instance, synthesis from examples does not support the inference of relations. This may however be useful to model non-deterministic system behaviour, such as same system inputs producing different system outputs. Depending on the application and the system for which we wish to learn an abstraction, we may need to explore alternate techniques or enhancements to program synthesis to infer more expressive transition predicates from trace data.

Extension to Richer Models

An interesting direction for future research is exploring extensions of the learning algorithm to richer modelling formalisms, such as register automata and recursive state machines. This will enable the generation of system abstractions modelling a wider range of system behaviours and, as a consequence, aid in the verification and validation of more complex systems.

Potential Use-Cases

The model-learning algorithm generates automata with strong guarantees for model correctness which enables their application to problems in the verification domain:

- They can summarise which aspects of system behaviour have been covered by a suite of tests
- They provide starting points for model-based test generation. The equivalence checking procedure using simulation relations could be used to identify test inputs to cover coverage holes.

- They could be used as candidate invariants in the inductive invariant refinement loop. For instance, the authors of [41] propose user-guided invariant inference for infinite state systems such as distributed protocols. The user specifies a candidate invariant as an automaton, called a phase structure, that captures the logical phases of a protocol. The model-learning algorithm could be used to automatically learn these phase structures.

The active model-learning algorithm generates abstractions that are guaranteed to admit all system traces. This can be particularly useful when system specifications are incomplete, and so any implementation errors outside the scope of defined requirements cannot be flagged. This is a common risk when essential domain knowledge gets progressively pruned as it is passed on from one team to another during the development life cycle. In such scenarios, manual inspection of the learned models can help identify errors in implementation.

Appendix A

List of benchmarks

Table A.1: Mapping of Simulink Stateflow example models to their benchmark number B# used in Table 6.1.

Benchmark Name			B#
AutomaticTransmissionUsingDurationOperator			B1
BangBangControlUsingTemporalLogic	InHeater		B2
	InOn		B3
CountEvents			B4
FrameSyncController			B5
HomeClimateControlUsingTheTruthtableBlock			B6
KarplusStrongAlgorithmUsingStateflow	DelayLine		B7
	MovingAverage		B8
LadderLogicScheduler			B9
MealyVendingMachine			B10
ModelingACdPlayerradio UsingEnumeratedDataType	CdPlayer BehaviourModel	DiscPresent	B11
		Overall	B12
	CdPlayer ModeManager	ModeManager	B13
		On	B14
		Overall	B15
ModelingALaunchAbortSystem	Abort	AbortLogic	B16
		Overall	B17
	ModeLogic		B18
ModelingAnIntersectionOfTwo 1wayStreetsUsingStateflow	InRed		B19
	Overall		B20

Continued on next page

Table A.1 – *Continued from previous page*

Benchmark Name			B#
ModelingARedundantSensorPairUsingAtomicSubchart			B21
ModelingASecuritySystem	InAlarm	On	B22
		Overall	B23
	InDoor		B24
	InMotion	Active	B25
		Overall	B26
	InWin		B27
MonitorTestPointsInStateflowChart			B28
MooreTrafficLight			B29
ReuseStatesByUsingAtomicSubcharts			B30
SchedulingSimulinkAlgorithmsUsingStateflow			B31
SequenceRecognitionUsingMealyAndMooreChart			B32
ServerQueueingSystem			B33
StatesWhenEnabling			B34
StateTransitionMatrixViewForStateTransitionTable			B35
Superstep	With Super Step		B36
	Without Super Step		B37
TemporalLogicScheduler			B38
UsingSimulinkFunctionsToDesignSwitchingControllers			B39
VarSize	SizeBasedProcessing		B40
	VarSizeSignalSource		B41
ViewDifferencesBetweenMessagesEventsAndData			B42
YoYoControlOfSatellite	InActive	ReelMoving	B43
		Overall	B44
	Overall		B45

References

[1] Fides Aarts, Faranak Heidarian, Harco Kuppens, Petur Olsen, and Frits Vaandrager. Automata learning through counterexample guided abstraction refinement. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 10–27. Springer, 2012. doi:10.1007/978-3-642-32759-9_4.

[2] Fides Aarts, Bengt Jonsson, and Johan Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In Alexandre Petrenko, Adenilso Simão, and José Carlos Maldonado, editors, *Testing Software and Systems*, pages 188–204. Springer, 2010. doi:10.1007/978-3-642-16573-3_14.

[3] Martín Abadi and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL: https://www.tensorflow.org/.

[4] Alessandro Abate, Cristina David, Pascal Kesseli, Daniel Kroening, and Elizabeth Polgreen. Counterexample guided inductive synthesis modulo theories. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, LNCS, pages 270–288. Springer, 2018. doi:10.1007/978-3-319-96145-3_15.

[5] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas Reps, and Mihalis Yannakakis. Analysis of recursive state machines. In *ACM Trans. Program. Lang. Syst.*, volume 27, pages 786–818. Association for Computing Machinery, 2005. doi:10.1145/1075382.1075387.

[6] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013. doi:10.1109/FMCAD.2013.6679385.

[7] Dana Angluin. Learning regular sets from queries and counterexamples. In *Information and Computation*, volume 75, pages 87–106. Academic Press, Inc., 1987. doi:10.1016/0890-5401(87)90052-6.

[8] George Argyros and Loris D'Antoni. The learnability of symbolic automata. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 427–445. Springer International Publishing, 2018. doi:10.1007/978-3-319-96145-3_23.

[9] Pranav Ashar, Abhijit Ghosh, and Srinivas Devadas. Boolean satisfiability and equivalence checking using general binary decision diagrams. In *Integration*, volume 13, pages 1–16, 1992. doi:10.1016/0167-9260(92)90015-Q.

[10] Florent Avellaneda and Alexandre Petrenko. FSM inference from long traces. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 93–109. Springer International Publishing, 2018. doi:10.1007/978-3-319-95582-7_6.

[11] Florent Avellaneda and Alexandre Petrenko. Inferring DFA without negative examples. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of The 14th International Conference on Grammatical Inference 2018*, volume 93, pages 17–29. Proceedings of Machine Learning Research (PMLR), 2019.

[12] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* MIT press, 2008.

[13] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.

[14] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41. USENIX Association, 2005.

[15] Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines using domains with equality tests. In José Luiz Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering*, pages 317–331. Springer, 2008. `doi:10.1007/978-3-540-78743-3_24`.

[16] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. In *IEEE Trans. Comput.*, volume 21, pages 592–597. IEEE Computer Society, 1972. `doi:10.1109/TC.1972.5009015`.

[17] Matko Botinčan and Domagoj Babić. Sigma*: Symbolic learning of input-output specifications. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 443–456. Association for Computing Machinery, 2013. `doi:10.1145/2429069.2429123`.

[18] Aaron Bradley and Zohar Manna. *The calculus of computation: Decision procedures with applications to verification.* Springer Science & Business Media, 2007.

[19] Nimrod Busany, Shahar Maoz, and Yehonatan Yulazari. Size and accuracy in model inference. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, page 887–898. IEEE Press, 2019. `doi:10.1109/ASE.2019.00087`.

[20] Nimrod Busany, Shahar Maoz, and Yehonatan Yulazari. *Supporting material for 'Size and Accuracy in Model Inference'*, 2019. URL: `http://smlab.cs.tau.ac.il/xlog/#ASE19a`.

[21] Igor Buzhinsky and Valeriy Vyatkin. Automatic inference of finite-state plant models from traces and temporal properties. In *IEEE Transactions on Industrial Informatics*, volume 13, pages 1521–1530, 2017. `doi:10.1109/TII.2017.2670146`.

[22] Igor Buzhinsky and Valeriy Vyatkin. Modular plant model synthesis from behavior traces and temporal properties. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–7, 2017. `doi:10.1109/ETFA.2017.8247578`.

[23] Yuting Cao, Parijat Mukherjee, M. Ketkar, J. Yang, and Hao Zheng. Mining message flows using recurrent neural networks for system-on-chip designs. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 389–394, 2020. doi:10.1109/ISQED48828.2020.9137001.

[24] Sofia Cassel, Falk Howar, and Bengt Jonsson. RALib: A LearnLib extension for inferring EFSMs. In *Design and Implementation of Formal Tools and Systems (DIFTS)*, volume 5, 2015.

[25] Sofia Cassel, Falk Howar, Bengt Jonsson, and Bernhard Steffen. Active learning for extended finite state machines. In *Formal Aspects of Computing*, volume 28, pages 233–263, 2016. doi:10.1007/s00165-016-0355-5.

[26] Sagar Chaki and Ofer Strichman. Optimized L*-based assume-guarantee reasoning. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 276–291. Springer, 2007. doi:10.1007/978-3-540-71209-1_22.

[27] Hana Chockler, Pascal Kesseli, Daniel Kroening, and Ofer Strichman. Learning the language of software errors. In *Journal Artificial Intelligence Research*, volume 67, pages 881–903. Morgan Kaufmann Publishers, Inc., 2020. doi:10.1613/jair.1.11798.

[28] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.

[29] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

[30] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 368–371. Association for Computing Machinery, 2003. doi:10.1145/775832.775928.

[31] Edmund M. Clarke, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking*. MIT Press, second edition, 2018.

[32] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *ACM Trans. Program. Lang. Syst.*, volume 16, pages 1512–1542. Association for Computing Machinery, 1994. doi:10.1145/186025.186051.

[33] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003. doi:10.1007/3-540-36577-X_24.

[34] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Efficient formal verification for the Linux kernel. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods*, pages 315–332. Springer, 2019. doi:10.1007/978-3-030-30446-1_17.

[35] Daniel Bristot de Oliveira, Tommaso Cucinotta, and Rômulo Silva de Oliveira. Modeling the behavior of threads in the PREEMPT RT Linux kernel using automata. In *SIGBED Rev.*, volume 16, pages 63–68. Association for Computing Machinery, 2019. doi:10.1145/3373400.3373410.

[36] Rita Dorofeeva, Khaled El-Fakih, Stephane Maag, Ana R. Cavalli, and Nina Yevtushenko. FSM-based conformance testing methods: A survey annotated with experimental evaluation. In *Information and Software Technology*, volume 52, pages 1286–1297. Butterworth-Heinemann, 2010. doi:10.1016/j.infsof.2010.07.001.

[37] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM algorithm and its application to software behavior model induction. In *Applied Artificial Intelligence*, volume 22, pages 77–115. Taylor & Francis, 2008. doi:10.1080/08839510701853200.

[38] Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. Learning properties in LTL $\cap$ ACTL from positive examples only. *2020 Formal Methods in Computer Aided Design (FMCAD)*, pages 104–112, 2020. doi:10.34727/2020/isbn.978-3-85448-042-6_17.

[39] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering (ICSE)*, pages 213–224. Association for Computing Machinery, 1999. doi:10.1145/302405.302467.

[40] Richard Evans, José Hernández-Orallo, Johannes Welbl, Pushmeet Kohli, and Marek J. Sergot. Evaluating the apperception engine. In *ArXiv*, volume abs/2007.05367, 2020.

[41] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 405–425. Springer, 2019. doi:10.1007/978-3-030-25543-5_23.

[42] Lu Feng, Marta Kwiatkowska, and David Parker. Automated learning of probabilistic assumptions for compositional reasoning. In Dimitra Giannakopoulou and Fernando Orejas, editors, *Fundamental Approaches to Software Engineering*, pages 2–17. Springer, 2011. doi:10.1007/978-3-642-19811-3_2.

[43] Paul Fiterău-Broştean and Falk Howar. Learning-based testing the sliding window behavior of TCP implementations. In Laure Petrucci, Cristina Seceleanu, and Ana Cavalcanti, editors, *Critical Systems: Formal Methods and Automated Verification*, pages 185–200. Springer International Publishing, 2017. doi:10.1007/978-3-319-67113-0_12.

[44] Jean H. Gallier, Salvatore La Torre, and Supratik Mukhopadhyay. Deterministic finite automata with recursive calls and DPDAs. In *Information Processing Letters*, volume 87, pages 187–193, 2003. doi:10.1016/S0020-0190(03)00281-3.

[45] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. Grey-box learning of register automata. In *Integrated Formal Methods: 16th International Conference, IFM 2020*, pages 22–40. Springer-Verlag, 2020. doi:10.1007/978-3-030-63461-2_2.

[46] Dimitra Giannakopoulou, Zvonimir Rakamarić, and Vishwanath Raman. Symbolic learning of component interfaces. In Antoine Miné and David Schmidt, editors, *Static Analysis*, pages 248–264. Springer, 2012. doi:10.1007/978-3-642-33125-1_18.

[47] E. Mark Gold. System identification via state characterization. In *Automatica*, volume 8, pages 621–636, 1972. doi:10.1016/0005-1098(72)90033-7.

[48] E. Mark Gold. Complexity of automaton identification from given data. In *Information and Control*, volume 37, pages 302–320. Elsevier, 1978. doi:10.1016/S0019-9958(78)90562-4.

[49] E.I. Goldberg, M.R. Prasad, and R.K. Brayton. Using SAT for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition*, pages 114–121, 2001. `doi:10.1109/DATE.2001.915010`.

[50] Petr Grachev, Igor Lobanov, Ivan Smetannikov, and Andrey Filchenkov. Neural network for synthesizing deterministic finite automata. In *Procedia Computer Science*, volume 119, pages 73–82, 2017. 6th International Young Scientist Conference on Computational Science, YSC 2017. `doi:10.1016/j.procs.2017.11.162`.

[51] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In Joost-Pieter Katoen and Perdita Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370. Springer, 2002. `doi:10.1007/3-540-46002-0_25`.

[52] Sumit Gulwani. Synthesis from examples. In *3rd Workshop on Advances in Model-Based Software Engineering (WAMBSE)*, 2012.

[53] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. Program synthesis. In *Foundations and Trends in Programming Languages*, volume 4, pages 1–119, 2017. `doi:10.1561/2500000010`.

[54] L. Gunn, P. Smet, E. Arbon, and M. D. McDonnell. Anomaly detection in satellite communications systems using LSTM networks. In *2018 Military Communications and Information Systems Conference (MilCIS)*, pages 1–6, 2018. `doi:10.1109/MilCIS.2018.8574109`.

[55] John Harrison. *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press, 2009.

[56] Marijn J. H. Heule and Sicco Verwer. Exact DFA identification using SAT solvers. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications*, pages 66–79. Springer, 2010. `doi:10.1007/978-3-642-15488-1_7`.

[57] Marijn J. H. Heule and Sicco Verwer. Software model synthesis using satisfiability solvers. In *Empirical Software Engineering*, volume 18, pages 825–856, 2013. `doi:10.1007/s10664-012-9222-z`.

[58] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. volume 9, pages 1735–1780. MIT Press, 1997. doi:10.1162/neco.1997.9.8.1735.

[59] Falk Howar, Dimitra Giannakopoulou, and Zvonimir Rakamarić. Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA, pages 268–279. Association for Computing Machinery, 2013. doi:10.1145/2483760.2483783.

[60] Falk Howar, Bengt Jonsson, and Frits Vaandrager. Combining black-box and white-box techniques for learning register automata. In Bernhard Steffen and Gerhard Woeginger, editors, *Computing and Software Science: State of the Art and Perspectives*, pages 563–588. Springer International Publishing, 2019. doi:10.1007/978-3-319-91908-9_26.

[61] Falk Howar and Bernhard Steffen. Active automata learning in practice. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits*, pages 123–148. Springer International Publishing, 2018. doi:10.1007/978-3-319-96562-8_5.

[62] Falk Howar, Bernhard Steffen, and Maik Merten. Automata learning with automated alphabet abstraction refinement. In Ranjit Jhala and David Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 263–277. Springer, 2011. doi:10.1007/978-3-642-18275-4_19.

[63] Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In Warren A. Hunt and Fabio Somenzi, editors, *Computer Aided Verification*, pages 315–327. Springer, 2003. doi:10.1007/978-3-540-45069-6_31.

[64] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2004.

[65] Intel. *eXtensible Host Controller Interface for Universal Serial Bus (xHCI)*, 2017. URL: https://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/extensible-host-controler-interface-usb-xhci.pdf.

[66] Malte Isberner, Falk Howar, and Bernhard Steffen. The TTT algorithm: A redundancy-free approach to active automata learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 307–322. Springer International Publishing, 2014. doi:10.1007/978-3-319-11164-3_26.

[67] N. Y. Jeppu, T. Melham, D. Kroening, and J. O'Leary. Learning concise models from long execution traces. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218613.

[68] Natasha Yogananda Jeppu. *Trace2Model Github repository*, 2020. URL: https://github.com/natasha-jeppu/Trace2Model.

[69] Natasha Yogananda Jeppu. *ActiveLearning*, 2021. URL: https://github.com/natasha-jeppu/ActiveLearning.

[70] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[71] James C. King. Symbolic execution and program testing. In *Commun. ACM*, volume 19, pages 385–394. Association for Computing Machinery, 1976. doi:10.1145/360248.360252.

[72] Donald E. Knuth. Chapter 6.3: Digital searching. In *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching.* Addison Wesley Longman Publishing Co., Inc., 1998.

[73] Daniel Kroening and Edmund Clarke. Checking consistency of C and Verilog using predicate abstraction and induction. In *IEEE/ACM International Conference on Computer Aided Design, ICCAD-2004.*, pages 66–72, 2004. doi:10.1109/ICCAD.2004.1382544.

[74] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. doi:10.1007/978-3-540-74105-3.

[75] Thomas Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.

[76] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 2014. doi:doi:10.1515/9781400864041.

[77] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In Vasant Honavar and Giora Slutzki, editors, *Grammatical Inference*, pages 1–12. Springer, 1998. `doi:10.1007/BFb0054059`.

[78] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended FSA from software: An empirical assessment. In *Journal of Systems and Software*, volume 85, pages 2063–2076, 2012. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011). `doi:10.1016/j.jss.2012.04.001`.

[79] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, S. Bensalem, and David Probst. Property preserving abstractions for the verification of concurrent systems. In *Formal Methods in System Design*, volume 6, pages 11–44, 1995. `doi:10.1007/BF01384313`.

[80] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510. Association for Computing Machinery, 2008. `doi:10.1145/1368088.1368157`.

[81] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large alphabets. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 485–499. Springer, 2014. `doi:10.1007/978-3-642-54862-8_41`.

[82] Oded Maler and Irini-Eleftheria Mens. Learning regular languages over large ordered alphabets. In *Logical Methods in Computer Science*, volume 11, 2014. `doi:10.2168/LMCS-11(3:13)2015`.

[83] J. Marques-Silva and T. Glass. Combinational equivalence checking using satisfiability and recursive learning. In *Design, Automation and Test in Europe Conference and Exhibition, 1999. Proceedings (Cat. No. PR00078)*, pages 145–149, 1999. `doi:10.1109/DATE.1999.761110`.

[84] Carlos Ivan Castro Marquez, Marius Strum, and Wang Jiang Chau. Formal equivalence checking between high-level and RTL hardware designs. In *2013 14th Latin American Test Workshop - LATW*, pages 1–6, 2013. `doi:10.1109/LATW.2013.6562666`.

[85] Joshua Moerman, Matteo Sammartino, Alexandra Silva, Bartek Klin, and Michał Szynwelski. Learning nominal automata. In *SIGPLAN Notices*, volume 52, pages 613–625. Association for Computing Machinery, 2017. doi:10.1145/3093333.3009879.

[86] Rajdeep Mukherjee, Daniel Kroening, Tom Melham, and Mandayam Srivas. Equivalence checking using trace partitioning. In *2015 IEEE Computer Society Annual Symposium on VLSI*, pages 13–18, 2015. doi:10.1109/ISVLSI.2015.110.

[87] A. Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544. American Mathematical Society, 1958. doi:10.2307/2033204.

[88] Christian W. Omlin and C. Lee Giles. Constructing deterministic finite-state automata in recurrent neural networks. In *Journal of ACM*, volume 43, pages 937–972. Association for Computing Machinery, 1996. doi:10.1145/235809.235811.

[89] Jose Oncina and Pedro García. Inferring regular languages in polynomial update time. In *World Scientific*, 1992. doi:10.1142/9789812797902_0004.

[90] Doron A. Peled. *Software Reliability Methods.* Texts in Computer Science. Springer, 2013.

[91] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In Jianping Wu, Samuel T. Chanson, and Qiang Gao, editors, *Formal Methods for Protocol Engineering and Distributed Systems: FORTE XII / PSTV XIX'99 IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX) October 5–8, 1999*, pages 225–240. Springer, 1999. doi:10.1007/978-0-387-35578-8_13.

[92] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *Proceedings of the Twenty-First Annual ACM Symposium on Theory of Computing*, STOC '89, pages 411–420. Association for Computing Machinery, 1989. doi:10.1145/73007.73047.

[93] J. Ruf, D.W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued AR-automata. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pages 742–748, 2001. doi:10.1109/DATE.2001.915111.

[94] W. Said, J. Quante, and R. Koschke. Reflexion models for state machine extraction and verification. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 149–159, 2018. doi:10.1109/ICSME.2018.00025.

[95] Arjun Singh Saud and Subarna Shakya. Analysis of look back period for stock price prediction with RNN variants: A case study on banking sector of NEPSE. In *Procedia Computer Science*, volume 167, pages 788–798, 2020. International Conference on Computational Intelligence and Data Science. doi:10.1016/j.procs.2020.03.419.

[96] Amazon Web Services. *AWS IoT Events*, 2021. URL: https://docs.aws.amazon.com/iotevents/latest/developerguide/what-is-iotevents.html.

[97] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In Ana Cavalcanti and Dennis R. Dams, editors, *Formal Methods*, pages 207–222. Springer, 2009. doi:10.1007/978-3-642-05089-3_14.

[98] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144. Springer, 2000. doi:10.1007/3-540-40922-X_8.

[99] Simulink. *Embedded Coder*, 2021. URL: https://uk.mathworks.com/products/embedded-coder.html.

[100] Simulink. *Simulation and Model-Based Design*, 2021. URL: https://www.mathworks.com/products/simulink.html.

[101] Simulink. *Stateflow Examples*, 2021. URL: https://uk.mathworks.com/help/stateflow/examples.html?s_tid=CRUX_topnav.

[102] Rick Smetsers, Joshua Moerman, Mark Janssen, and Sicco Verwer. Complementing model learning with mutation-based fuzzing. In *ArXiv*, volume abs/1611.02429, 2016.

[103] Bernhard Steffen and Hardi Hungar. Behavior-based model construction. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 5–19. Springer, 2003. doi:10.1007/s10009-004-0139-8.

[104] Ilia Sucholutsky, Apurva Narayan, Matthias Schonlau, and Sebastian Fischmeister. Deep learning for system trace restoration. In *International Joint Conference on Neural Networks, IJCNN 2019*, pages 1–8. IEEE, 2019. doi:10.1109/IJCNN.2019.8852116.

[105] Peter Tiňo, Bill G. Horne, C. Lee Giles, and Pete C. Collingwood. Chapter 6 - finite state machines and recurrent neural networks — automata and dynamical systems approaches. In Omid Omidvar and Judith Dayhoff, editors, *Neural Networks and Pattern Recognition*, pages 171 – 219. Academic Press, 1998. doi:10.1016/B978-012526420-4/50007-0.

[106] Vladimir Ulyantsev, Igor Buzhinsky, and Anatoly Shalyto. Exact finite-state machine identification from scenarios and temporal properties. In *International Journal on Software Tools for Technology Transfer*, volume 20, pages 35–55, 2018. doi:10.1007/s10009-016-0442-1.

[107] Vladimir Ulyantsev and Fedor Tsarev. Extended finite-state machine induction using SAT-solver. In *International Conference on Machine Learning and Applications and Workshops*, pages 346–349, 2011. doi:10.1109/ICMLA.2011.166.

[108] Frits Vaandrager, Roderick Bloem, and Masoud Ebrahimi. Learning Mealy machines with one timer. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications*, pages 157–170. Springer International Publishing, 2021. doi:10.1007/978-3-030-68195-1_13.

[109] C A J van Eijk. Sequential equivalence checking based on structural similarities. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 19, pages 814–819, 2000. doi:10.1109/43.851997.

[110] Neil Walkinshaw. *MINT framework Github repository*, 2018. URL: https://github.com/neilwalkinshaw/mintframework.

[111] Neil Walkinshaw and Kirill Bogdanov. Inferring finite-state models with temporal constraints. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 248–257, 2008. doi:10.1109/ASE.2008.35.

[112] Neil Walkinshaw, Kirill Bogdanov, Mike Holcombe, and Sarah Salahuddin. Reverse engineering state machines by interactive grammar inference. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 209–218. IEEE Computer Society, 2007. doi:10.1109/WCRE.2007.45.

[113] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In Ana Cavalcanti and Dennis R. Dams, editors, *Formal Methods*, pages 305–320. Springer, 2009. doi:10.1007/978-3-642-05089-3_20.

[114] Neil Walkinshaw and Mathew Hall. Inferring computational state machine models from program executions. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 122–132, 2016. doi:10.1109/ICSME.2016.74.

[115] Neil Walkinshaw, Ramsay Taylor, and John Derrick. Inferring extended finite state machine models from software executions. In *Empirical Software Engineering*, volume 21, pages 811–853, 2016. doi:10.1007/s10664-015-9367-7.

[116] Raymond L. Watrous and Gary M. Kuhn. Induction of finite-state automata using second-order recurrent networks. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 309–317. Morgan-Kaufmann, 1992.