Formal Methods in System Design (2022) 61:164–197
https://doi.org/10.1007/s10703-023-00433-y

Enhancing active model learning with equivalence checking using simulation relations

Natasha Yogananda Jeppu[1] · **Tom Melham**[1] · **Daniel Kroening**[2]

Received: 1 April 2022 / Accepted: 19 June 2023 / Published online: 6 August 2023

Abstract
We present a new active model-learning approach to generating abstractions of a system from its execution traces. Given a system and a set of observables to collect execution traces, the abstraction produced by the algorithm is guaranteed to admit all system traces over the set of observables. To achieve this, the approach uses a pluggable model-learning component that can generate a model from a given set of traces. Conditions that encode a certain completeness hypothesis, formulated based on simulation relations, are then extracted from the abstraction under construction and used to evaluate its degree of completeness. The extracted conditions are sufficient to prove model completeness but not necessary. If all conditions are true, the algorithm terminates, returning a system overapproximation. A condition falsification may not necessarily correspond to missing system behaviour in the abstraction. This is resolved by applying model checking to determine whether it corresponds to any concrete system trace. If so, the new concrete trace is used to iteratively learn new abstractions, until all extracted completeness conditions are true. To evaluate the approach, we reverse-engineer a set of publicly available Simulink Stateflow models from their C implementations. Our algorithm generates an equivalent model for 98% of the Stateflow models.

Keywords Active model learning · Execution traces · System abstraction · Equivalence checking · Model checking · Simulation relation

✉ Natasha Yogananda Jeppu
natasha.jeppu@gmail.com

Tom Melham
tom.melham@cs.ox.ac.uk

Daniel Kroening
daniel.kroening@magd.ox.ac.uk

1 University of Oxford, Oxford, UK

2 Amazon, Inc., Seattle, USA

1 Introduction

Modern hardware and software system implementations often have complex behaviour and it is difficult to specify integrated system behaviour, particularly emergent behaviour, ahead of time. Execution traces provide an exact representation of the system behaviours that are exercised when an implementation runs. This can therefore be leveraged to reverse-engineer system abstractions, such as the model in Fig. 1, that are easy to understand, and can be used in place of the actual implementation for simulation and debugging.

Model-learning algorithms are classified into *passive* and *active* algorithms. In passive model learning [8, 27, 36, 40], the behaviours admitted by the generated models are limited to only those manifest in the given traces. So 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 it is not obvious how the components will behave collectively. Random input sampling is a pragmatic choice in this scenario, but it does not guarantee that generated models admit all system behaviour.

By contrast, active learning algorithms can, in principle, generate exact models [4, 5, 11, 62]. 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. The most popular form of active learning is query-based learning [4], where the learning framework poses membership and equivalence queries to an oracle and uses the responses to guide model refinement. But when these algorithms are used in practice, especially to learn symbolic abstractions such as the models in Figs. 1 and 2, they suffer from high query complexity [23, 28, 31]. Consequently, many active model-learning implementations are constrained to learning partial models for large systems.

In this article we present a new active learning approach to derive abstractions of a system implementation from its execution traces. The approach, on termination, is guaranteed to generate an abstraction that is a good overapproximation of the system. As illustrated in Fig. 3, 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 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 generates a model that accepts a given set of system execution traces. The novelty of the approach is the procedure used to evaluate the degree of completeness for a learned candidate abstraction: the structure of the candidate abstraction is used to extract a set of conditions that

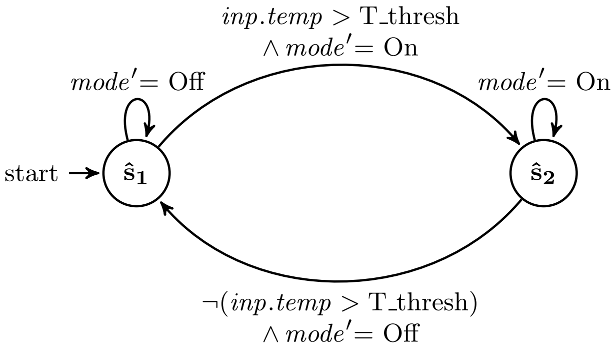

$inp.temp > \mathrm{T\_thresh}$
$\wedge\, mode' = \mathrm{On}$
$mode' = \mathrm{Off}$
$mode' = \mathrm{On}$
start
$\hat{\mathbf{s}}_1$
$\hat{\mathbf{s}}_2$
$\neg(inp.temp > \mathrm{T\_thresh})$
$\wedge\, mode' = \mathrm{Off}$

Fig. 1 Abstraction modelling operation mode switches for a Home Climate-Control Cooler System generated by our algorithm

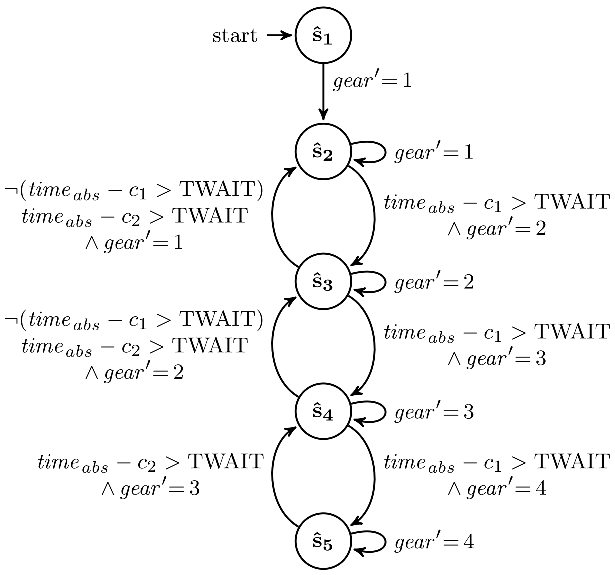

start
$\hat{s}_1$
$gear' = 1$
$\hat{s}_2$
$gear' = 1$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge\, gear' = 1$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge\, gear' = 2$
$\hat{s}_3$
$gear' = 2$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$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

Fig. 2 Abstraction modelling gear-shift logic for an Automatic Transmission Gear System generated by our algorithm

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 admits (at least) all system execution traces.

To verify the hypothesis, the truth value of all extracted conditions is checked using Boolean satisfiability (SAT) solving. The procedure returns the fraction of extracted conditions that hold as a quantitative measure of the degree of completeness for the given model. If all conditions are true, the algorithm terminates, returning the learned system overapproximation. In the event of a condition falsification, the SAT procedure returns a counterexample.

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 [16] to determine if the counterexample for a condition check is spurious. If found to be spurious, the respective extracted completeness condition is strengthened to guide the SAT solver towards a non-spurious counterexample, if any exists.

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.

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:

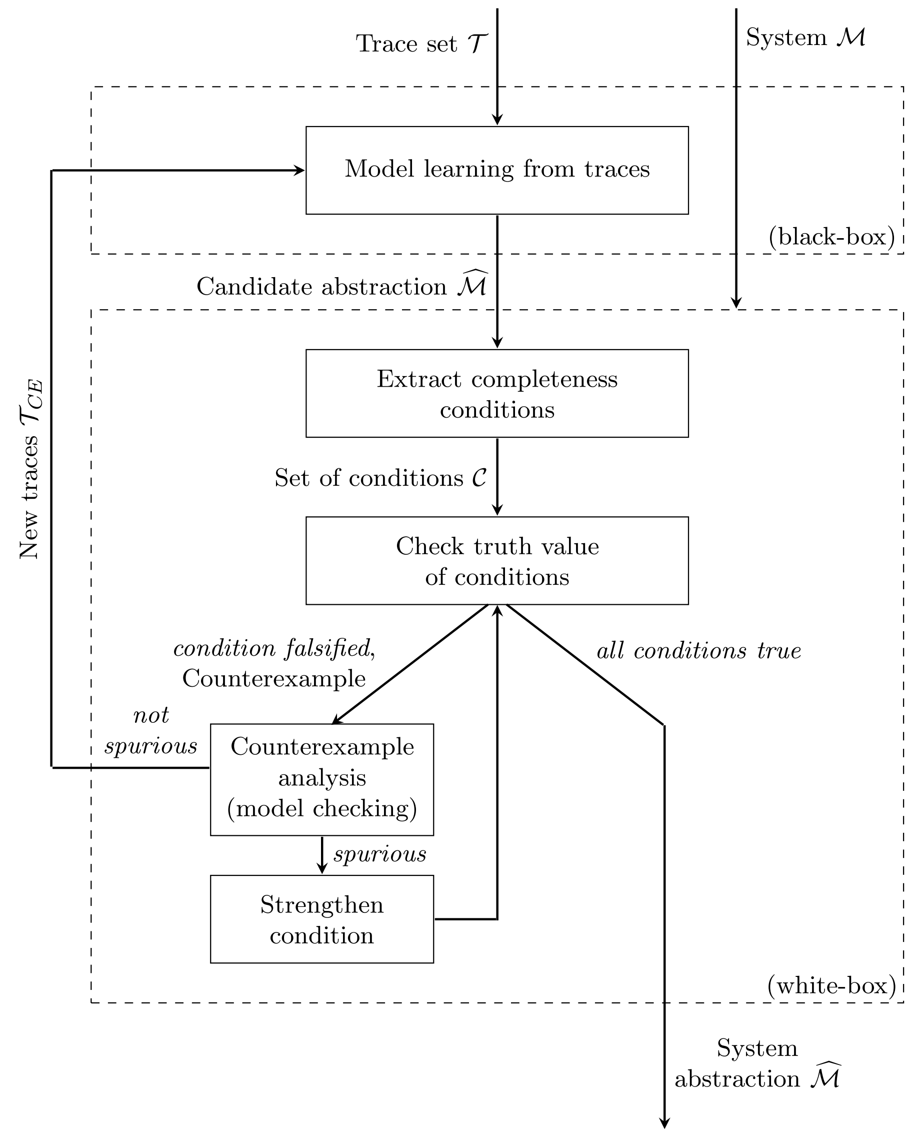

Trace set $\mathcal{T}$
System $\mathcal{M}$
Model learning from traces
(black-box)
Candidate abstraction $\widehat{\mathcal{M}}$
Extract completeness conditions
Set of conditions $\mathcal{C}$
Check truth value of conditions
condition falsified, Counterexample
all conditions true
New traces $\mathcal{T}_{CE}$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

Fig. 3 Overview of the active model-learning algorithm

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

2 Active learning of abstract system models

2.1 Formal model

The system for which we wish to generate an abstraction is represented as a Labelled Transition System (LTS).

Definition 1 (*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* that can be used to collect execution traces. An observation $o \in \Omega$ could be any event that depends on a transition from state s to state s'. A *path* π in $\mathcal{M}$ is a finite sequence $\pi = s_0, o_0, s_1, \dots, o_{n-1}, s_n$ of alternating states and observations such that $(s_i, o_i, s_{i+1}) \in \Delta$ for $0 \le i < n$. The trace of π, denoted $\sigma(\pi)$, is the corresponding sequence of observations $o_0, \dots, o_{n-1}$ 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. A trace σ is *accepted* by $\mathcal{M}$ if $\sigma \in \mathcal{L}(\mathcal{M})$, and is termed an execution trace or *positive* trace. A trace $\sigma \notin \mathcal{L}(\mathcal{M})$ is termed a *negative* trace.

A system abstraction is represented as an LTS $\widehat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{\Omega}, \hat{\Delta}, \hat{s}_0)$. The language of the learned abstraction $\mathcal{L}(\widehat{\mathcal{M}})$ is defined over the alphabet of observations, i.e., $\hat{\Omega} = \Omega$. The abstraction accepts a trace $\sigma = o_0, \dots, o_{n-1}$ if $\sigma \in \mathcal{L}(\widehat{\mathcal{M}})$, i.e., if there exists a sequence $\hat{s}_0, \dots, \hat{s}_n$ of states in $\hat{\mathcal{S}}$ such that $(\hat{s}_i, o_i, \hat{s}_{i+1}) \in \hat{\Delta}$ for $0 \le i < n$. We will show that our active learning algorithm returns an abstraction $\widehat{\mathcal{M}}$ that admits all execution traces of the system $\mathcal{M}$, i.e., $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$.

2.2 Overview of the algorithm

Given a system $\mathcal{M}$, our goal is to learn a system abstraction $\widehat{\mathcal{M}}$ such that $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$. An overview of our approach is provided in Fig. 3. It consists of the following steps:

1. *Generate candidate abstraction from available traces.* The algorithm learns a candidate abstraction $\widehat{\mathcal{M}}$ from an initial set of system traces $\mathcal{T}$ using a pluggable model-learning algorithm. This is discussed in Sect. 2.3.
2. *Extract completeness conditions.* To evaluate the degree of completeness of the candidate abstraction returned by model learning, the structure of $\widehat{\mathcal{M}}$ is used to extract a set of conditions $\mathcal{C}$ that collectively encode a completeness hypothesis. If all conditions are

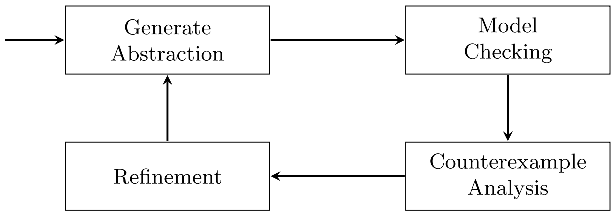


Fig. 4 Counterexample-Guided Abstraction Refinement (CEGAR) loop

true, it implies $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$. The formulation of the completeness hypothesis and a formal proof of the above claim is provided in Sect. 2.4.

3. *Check truth value of extracted conditions.* The algorithm uses a SAT procedure to check the truth value of each condition, and thereby checks the hypothesis. If all conditions are true, the algorithm returns $\widehat{\mathcal{M}}$ as the learned system overapproximation. The extracted conditions are sufficient to prove model completeness, i.e., $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$, 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 $\widehat{\mathcal{M}}$. This is discussed in Sect. 2.5.
4. *Counterexample analysis.* To check if a condition falsification actually indicates missing system behaviour in $\widehat{\mathcal{M}}$, i.e., $\mathcal{L}(\mathcal{M}) \backslash \mathcal{L}(\widehat{\mathcal{M}}) \neq \emptyset$, the algorithm uses model checking to determine whether the counterexample returned by the SAT procedure corresponds to a concrete system trace. 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. Details are provided in Sect. 2.6.
5. *Generate new abstraction.* A set of new traces $\mathcal{T}_{CE}$ is constructed from non-spurious counterexamples that exemplify missing system behaviour in $\widehat{\mathcal{M}}$. 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 Sect. 2.6.

In each iteration i of the algorithm, $\mathcal{T}_{CE_i} \cap \mathcal{L}(\widehat{\mathcal{M}}_{i-1}) = \emptyset$, 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 $\widehat{\mathcal{M}}_{i-1}$ generated in the previous iteration. The new abstraction $\widehat{\mathcal{M}}_i$ 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) [13], illustrated in Fig. 4. 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.

The following sections describe each step of our algorithm in detail.

2.3 Model learning from execution traces

The approach uses a pluggable model-learning component to generate a candidate abstraction from a set of 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 $\widehat{\mathcal{M}}$ that accepts (at least) all traces in $\mathcal{T}$, i.e., $\mathcal{L}(\widehat{\mathcal{M}}) \supseteq \mathcal{T}$.
- 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 [8, 36, 41, 58, 63, 65]. In general, these algorithms operate by employing automaton inference techniques, such as state-merging [8, 40] or SAT [27, 36], to generate a finite state automaton that conforms to a given set of traces.

Among these, the algorithm in [36] 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 states.

It is straightforward to transform a finite state automaton that accepts a prefix-closed language into an LTS abstraction, as defined in Sect. 2.1, by removing the non-accepting states and all transitions that lead into them.

2.4 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, which can be applied to symbolic abstractions such as the model in Fig. 2.

2.4.1 Set-based definition

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

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

$$\exists s \in \mathcal{S} : (s_0, o, s) \in \Delta \implies \exists \hat{s} \in \hat{\mathcal{S}} : (\hat{s}_0, o, \hat{s}) \in \hat{\Delta} \tag{1}$$

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

$$\begin{aligned}&(\exists \hat{s}'' \in \hat{\mathcal{S}} : (\hat{s}'', o', \hat{s}) \in \hat{\Delta} \wedge \\ &\exists s'', s, s' \in \mathcal{S} : (s'', o', s), (s, o, s') \in \Delta) \implies \\ &\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s}, o, \hat{s}') \in \hat{\Delta}\end{aligned} \tag{2}$$

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 $\widehat{\mathcal{M}}$ defined by $\hat{\Delta}$.

In the following section we prove that if the above hypothesis holds, i.e., if the completeness conditions (1) and (2) evaluate to true, then a simulation relation can be constructed between $\mathcal{M}$ and $\widehat{\mathcal{M}}$. We then use the fact that the existence of a simulation relation between $\mathcal{M}$ and $\widehat{\mathcal{M}}$ implies $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$.

Constructing a Simulation Relation

We formally define simulation relations for LTSs as follows:

Definition 2 (*Simulation Relation for LTSs*) Given two LTSs $\mathcal{M} = (\mathcal{S}, \Omega, \Delta, s_0)$ and $\widehat{\mathcal{M}} = (\hat{\mathcal{S}}, \hat{\Omega}, \hat{\Delta}, \hat{s}_0)$ 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 $\mathcal{M} \preceq_{\mathcal{R}} \widehat{\mathcal{M}}$.

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 $\widehat{\mathcal{M}}$, 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 (1) and (2) hold, the relation $\mathcal{R}'$ is constructed as follows:

1. Initialise $\mathcal{R}' = \{(s_0, \hat{s}_0)\}$
2. If the condition (1) holds non-trivially for some observation $o \in \Omega$, i.e., $\exists s \in \mathcal{S} : (s_0, o, s) \in \Delta$ and $\exists \hat{s} \in \hat{\mathcal{S}} : (\hat{s}_0, o, \hat{s}) \in \hat{\Delta}$, then add the state pair $(s, \hat{s})$ to $\mathcal{R}'$.
$$\mathcal{R}' \leftarrow \mathcal{R}' \cup \{(s, \hat{s})\}$$
3. If the condition (2) extracted for a state $\hat{s} \in \hat{\mathcal{S}}$ holds non-trivially for some observations $o', o \in \Omega$, i.e., $\exists s'', s, s' \in \mathcal{S} : (s'', o', s), (s, o, s') \in \Delta$ and $\exists \hat{s}'', \hat{s}' \in \hat{\mathcal{S}} : (\hat{s}'', o', \hat{s}), (\hat{s}, o, \hat{s}') \in \hat{\Delta}$, 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 exist incoming transitions to the states s and $\hat{s}$ with some observation $o' \in \Omega$. That is,

$$\begin{aligned} &\forall (s, \hat{s}) \in \mathcal{R}' \backslash (s_0, \hat{s}_0) \cdot \exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega : \\ &(s'', o', s) \in \Delta \wedge (\hat{s}'', o', \hat{s}) \in \hat{\Delta} \end{aligned} \tag{3}$$

Theorem 1 *The constructed relation $\mathcal{R}'$ forms a simulation, i.e. $\mathcal{M} \preceq_{\mathcal{R}'} \widehat{\mathcal{M}}$*

Proof We use contradiction to prove that when the completeness conditions (1) and (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 $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega : (s, o, s') \in \Delta$ 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 $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega : (s,o,s') \in \Delta$ and $\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \notin \mathcal{R}$

The above clauses (a), (b) and (c) are obtained by negating conditions (1) and (2) in definition 2 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 2 can be written as

$$\begin{aligned}\forall(s,\hat{s}) \in \mathcal{R} \cdot \forall s' \in \mathcal{S} \cdot \forall o \in \Omega \cdot ((s,o,s') \in \Delta \implies \\ (\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \in \mathcal{R}))\end{aligned} \tag{4}$$

On negating the above expression we get

$$\begin{aligned}\neg(\forall(s,\hat{s}) \in \mathcal{R} \cdot \forall s' \in \mathcal{S} \cdot \forall o \in \Omega \cdot ((s,o,s') \in \Delta \implies \\ (\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \in \mathcal{R})))\end{aligned} \tag{5}$$

$$\begin{aligned}\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega : ((s,o,s') \in \Delta \wedge \\ \neg(\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \in \mathcal{R}))\end{aligned} \tag{6}$$

$$\begin{aligned}\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega : ((s,o,s') \in \Delta \wedge \\ \forall \hat{s}' \in \hat{\mathcal{S}} \cdot ((\hat{s},o,\hat{s}') \in \hat{\Delta} \implies (s',\hat{s}') \notin \mathcal{R}))\end{aligned} \tag{7}$$

$$\begin{aligned}\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega : ((s,o,s') \in \Delta \wedge \\ ((\forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta}) \vee \\ (\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \notin \mathcal{R})))\end{aligned} \tag{8}$$

$$\begin{aligned}\implies \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega : ((s,o,s') \in \Delta \wedge \forall \hat{s}' \in \hat{\mathcal{S}} \cdot (\hat{s},o,\hat{s}') \notin \hat{\Delta}) \\ \vee \\ \exists(s,\hat{s}) \in \mathcal{R} \cdot \exists s' \in \mathcal{S} \cdot \exists o \in \Omega : ((s,o,s') \in \Delta \wedge \\ (\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s},o,\hat{s}') \in \hat{\Delta} \wedge (s',\hat{s}') \notin \mathcal{R}))\end{aligned} \tag{9}$$

Here, expression (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 $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega : (s,o,s') \in \Delta$ 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

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

  This violates completeness condition (1), which is a contradiction.

- If $(s, \hat{s}) \in \mathcal{R}' \backslash (s_0, \hat{s}_0)$, then from (3) there exists incoming transitions to s and $\hat{s}$ on some observation $o' \in \Omega$, i.e., $\exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega : (s'', o', s) \in \Delta \wedge (\hat{s}'', o', \hat{s}) \in \hat{\Delta}$. This implies

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

This violates completeness condition (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 $\exists s' \in \mathcal{S} \cdot \exists o \in \Omega : (s, o, s') \in \Delta$ and $\exists \hat{s}' \in \hat{\mathcal{S}} : (\hat{s}, o, \hat{s}') \in \hat{\Delta} \wedge (s', \hat{s}') \notin \mathcal{R}'$. There are two possibilities here:

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

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

This is a case where condition (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 (3) there exists incoming transitions to s and $\hat{s}$ on some observation $o' \in \Omega$, i.e., $\exists s'' \in \mathcal{S} \cdot \exists \hat{s}'' \in \hat{\mathcal{S}} \cdot \exists o' \in \Omega : (s'', o', s) \in \Delta \wedge (\hat{s}'', o', \hat{s}) \in \hat{\Delta}$. This implies

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

This is a case where condition (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., $\mathcal{M} \preceq_{\mathcal{R}'} \widehat{\mathcal{M}}$. □

Note that the satisfaction of the completeness hypothesis is sufficient to guarantee the existence of a simulation relation between $\mathcal{M}$ and $\widehat{\mathcal{M}}$, but not necessary. An example is provided in Fig. 5. Here, the completeness conditions extracted for state $\hat{s}_2$ do not hold:

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

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

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

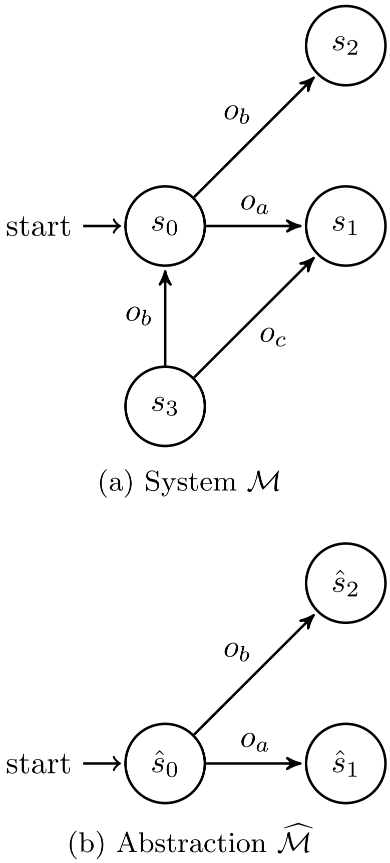


Fig. 5 Example system and its abstraction

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

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

Theorem 2 *If $\mathcal{M} \preceq_{\mathcal{R}} \widehat{\mathcal{M}}$ for LTSs $\mathcal{M}$ and $\widehat{\mathcal{M}}$, then $\mathcal{L}(\mathcal{M}) \subseteq \mathcal{L}(\widehat{\mathcal{M}})$.*

Variants of this theorem are commonplace; Park offers a proof in [47].

By Theorems 1 and 2 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 the 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 Sect. 2.5.

2.4.2 Symbolic definition

Symbolic representations of abstractions have transitions labelled with characteristic functions or predicates for sets of observations, such as the models in Figs. 1 and 2. A single edge in these graphs in fact corresponds to a set of multiple transitions. There are three benefits of this representation:

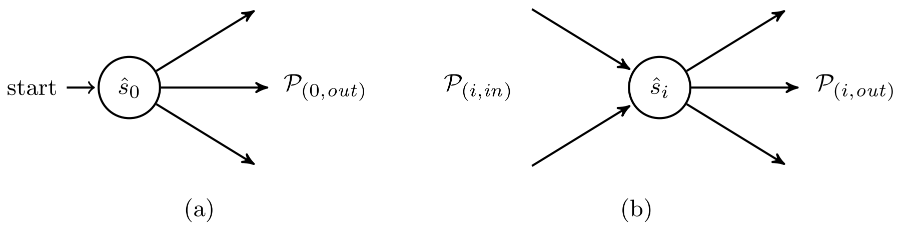


Fig. 6 Symbolic representation of abstract model states and transitions

1. It reduces the computational cost of the method when compared to an explicit representation that enumerates concrete transitions.
2. We hypothesise that human engineers prefer the more succinct symbolic presentation over an explicit list. In lieu of experimental evidence, we remark that popular design tools such as Simulink [54] strongly encourage the use of symbolic transition predicates.
3. The symbolic representation also enables an extension of our method to infinite alphabets, provided the model learning component can infer such models from execution traces.

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. 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:

$$O = \{o \in D^{|X|} \mid o \vDash f_O\} \tag{10}$$

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

As is standard, the function is given by means of a Boolean-valued expression. We refrain from defining a full syntax and semantics for the expressions we use. For sake of exposition, we use a C-like syntax, and semantics that roughly correspond to the bit-vector theory in SMT-LIB 2.

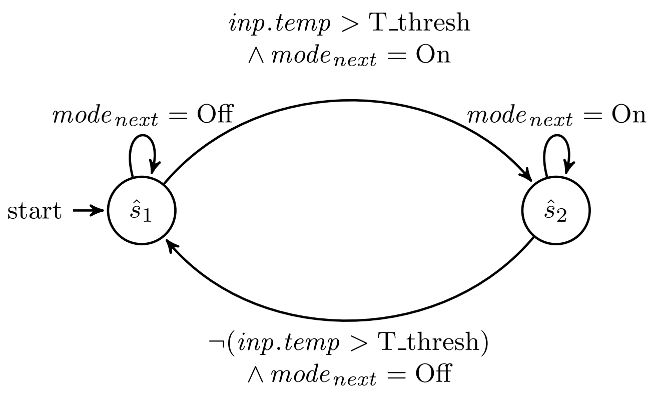
As defined in Sect. 2.1, the transitions of our LTSs comprise

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. 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:

$$\{(\hat{s}_i, o, \hat{s}_{i+1}) \mid o \vDash p\} \tag{11}$$

To derive the completeness conditions (1) and (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 $\mathit{Init} : \Omega \longrightarrow \mathbb{B}$ respectively, defined as follows:


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

(a) Given system abstraction

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

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

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

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

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

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

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

(b) Extracted conditions

Fig. 7 Completeness hypothesis for a symbolic abstraction

$$(o', o) \vDash f_\Delta \iff \exists s'', s, s' \in \mathcal{S},\ (s'', o', s), (s, o, s') \in \Delta \tag{12}$$

$$o \vDash Init \iff \exists s \in \mathcal{S},\ (s_0, o, s) \in \Delta \tag{13}$$

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

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

$$o \models Init \implies o \models \bigvee_{p_{out} \in \mathcal{P}_{(0,out)}} p_{out} \tag{14}$$

where $\mathcal{P}_{(0,out)}$ is the set of predicates for all outgoing transitions from $\hat{s}_0$, as illustrated in Fig. 6a.

And for all states $\hat{s}_i \in \hat{\mathcal{S}}$, $\forall p_{in} \in \mathcal{P}_{(i,in)}$, $\forall o', o \in \Omega$

$$(o' \models p_{in} \wedge (o', o) \models f_\Delta) \implies o \models \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out} \tag{15}$$

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. 6b. We illustrate the formulation of the completeness hypothesis for a symbolic abstraction as described above with an example in Fig. 7.

Note that the conditions (14) and (15) are symbolic representations of the completeness conditions (1) and (2), respectively. For the remainder of the article we use the symbolic representation of the completeness hypothesis as encoded by conditions (14) and (15).

2.5 Checking the truth 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 (14) and (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 (14) and (15), represented as

$$\neg(\omega \models Init \implies \omega \models \bigvee_{p_{out} \in \mathcal{P}_{(0,out)}} p_{out}) \tag{16}$$

and

$$\neg((\omega' \models p_{in} \wedge (\omega', \omega) \models f_\Delta) \implies \omega \models \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out}) \tag{17}$$

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 evaluates to true for all observations in Ω.

As discussed in Sect. 2.4, the satisfaction of the completeness hypothesis is sufficient to guarantee completeness, but not necessary. In the event of a falsification of condition (14), the SAT solver returns a counterexample $\omega = o$, such that $o \models Init$ 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., $\mathcal{L}(\mathcal{M}) \not\subseteq \mathcal{L}(\widehat{\mathcal{M}})$. But, in the event of a falsification of condition (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.

2.6 Counterexample analysis

To check if a counterexample $\omega' = o'$, $\omega = o$ for condition (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 $\pi = s_0, o_0, s_1, o_1, \dots, o_{n-1}, s_n$ 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 spurious.

In the event that the counterexample for condition (15) is spurious, the corresponding input to the SAT solver is strengthened by adding the clause $\omega' \neq o'$ to (17) as follows

$$\neg((\omega' \vDash p_{in} \wedge (\omega', \omega) \vDash f_\Delta) \implies \omega \vDash \bigvee_{p_{out} \in \mathcal{P}_{(i,out)}} p_{out}) \wedge \omega' \neq o' \tag{18}$$

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.

All non-spurious counterexamples 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 (14), we add a trace $\sigma_{CE} = o$ to the set $\mathcal{T}_{CE}$. For each counterexample $\omega' = o', \omega = o$ for condition (15), we find the smallest prefix $\sigma' = o_1, o_2, \dots, o_m$ for all $\sigma \in \mathcal{T}$ such that $o_m \vDash p_{in}$. We then construct a new trace $\sigma_{CE} = o_1, o_2, \dots, o_{m-1}, o', o$ for each prefix σ'. Note that since $o' \vDash p_{in}$, 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.

Example run of the approach

An example run demonstrating the active model-learning algorithm for a Home Climate Control Cooling system is illustrated in Fig. 8 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. 8a. The abstraction models the following sequential system behaviour: the system stays in the Off mode ($\hat{s}_1 \to \hat{s}_1$), or switches from the Off mode to the On mode when *inp.temp* > T_thresh ($\hat{s}_1 \to \hat{s}_2$). The system then switches back to the Off mode and stays in the Off mode indefinitely ($\hat{s}_2 \to \hat{s}_2$).

The structure of the generated abstraction is used to extract the following completeness conditions:

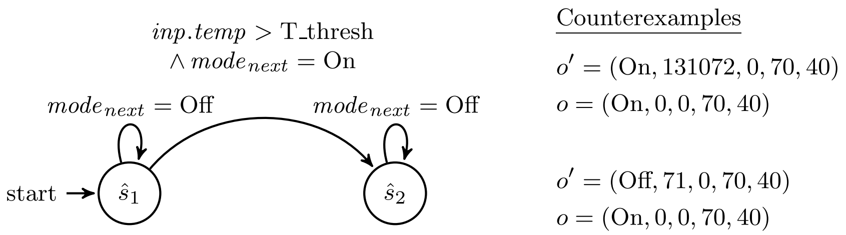
For state $\hat{s}_1$, $\forall o', o \in \Omega$:

$$\begin{aligned} o \vDash \mathit{Init} \implies o \vDash (\mathit{mode}_{next} = \text{Off} \vee \\ (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{next} = \text{On})) \end{aligned} \tag{19}$$

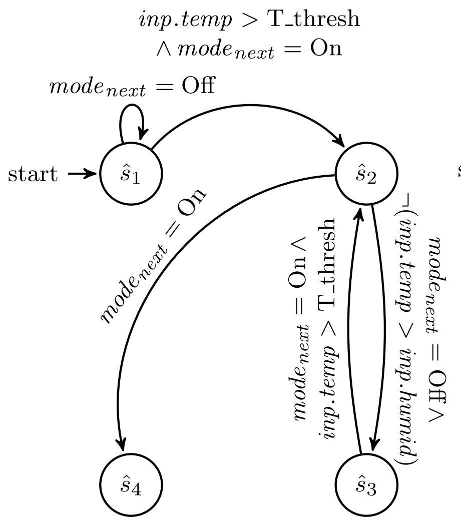
$$\begin{aligned} (o' \vDash \mathit{mode}_{next} = \text{Off} \wedge (o', o) \vDash f_\Delta) \implies \\ o \vDash (\mathit{mode}_{next} = \text{Off} \vee (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{next} = \text{On})) \end{aligned} \tag{20}$$

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

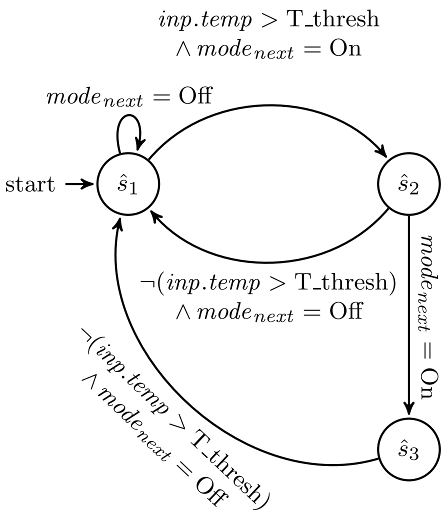
$$\begin{aligned} (o' \vDash (\mathit{inp.temp} > \text{T\_thresh} \wedge \mathit{mode}_{next} = \text{On}) \wedge (o', o) \vDash f_\Delta) \implies \\ o \vDash \mathit{mode}_{next} = \text{Off} \end{aligned} \tag{21}$$


$inp.temp > \text{T\_thresh}$
$\wedge\, mode_{next} = \text{On}$
$mode_{next} = \text{Off}$
$mode_{next} = \text{Off}$
start
$\hat{s}_1$
$\hat{s}_2$
Counterexamples
$o' = (\text{On}, 131072, 0, 70, 40)$
$o = (\text{On}, 0, 0, 70, 40)$
$o' = (\text{Off}, 71, 0, 70, 40)$
$o = (\text{On}, 0, 0, 70, 40)$

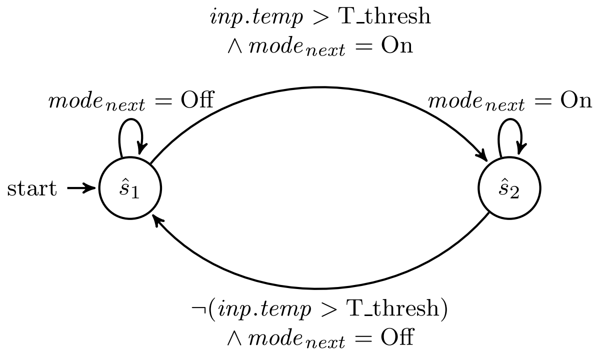
(a) Iteration 0, $\rho = 0.5$


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

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


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

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


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

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

Fig. 8 Example run of the active learning algorithm for a Home Climate Control Cooling system with observable system variables $X = \{mode_{next}, inp.temp, inp.humid, \text{T\_thresh}, \text{H\_thresh}\}$

$$(o' \vDash mode_{next} = \text{Off} \wedge (o', o) \vDash f_{\Delta}) \implies o \vDash mode_{next} = \text{Off} \tag{22}$$

The subsequent completeness hypothesis check yields falsifications for conditions (21) and (22). The SAT procedure returns the counterexamples provided in Fig. 8a for the two conditions respectively. These are found to be not spurious.

The counterexamples exemplify the following system behaviours that are missing from the abstraction in Fig. 8a:

1. The first counterexample corresponding to a falsification of condition (21) 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 (22) 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 model in Fig. 8b. Note that the new model 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 as 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. 8c, d. All conditions extracted from the abstraction in Fig. 8d evaluate to true, i.e., $\rho = 1$. Thus, the algorithm terminates, returning the model in Fig. 8d as the final generated system overapproximation.

3 Evaluation and results

3.1 Implementation

We implement the active learning approach using the Trace2Model (T2M) [33, 36] tool as the model-learning component. T2M generates symbolic finite state automata from traces using a combination of SAT and program synthesis [36].

We use the C Bounded Model Checker (CBMC v5.35) [15] 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 [52] 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 treat 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.

3.2 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 [55], available as part of the Simulink documentation. We select this dataset since 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) [53] to automatically generate a corresponding C code implementation. The generated C code 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 Stateflow examples, Embedded Coder fails to generate code for 7; a total of 13 have no sequential behaviour and 3 implement Recursive State Machines (RSM) [3].[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 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 3 in the Appendix. The algorithm implementation and benchmarks are available online [34].[2]

3.3 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 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 $|\hat{S}|$ 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 system overapproximation by assigning a score d computed as the fraction of state transitions in the target state machine that match corresponding transitions in the

[1] In this work we learn abstractions as FSAs (Sect. 2.3), 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 FSAs, such as Push-Down Automata (PDA) [22], which is outside the scope of this work. In the future, we wish to look at extensions of this work to generate RSMs.

[2] An archive of the sources used for the experiments in this article is available here [35].

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.

3.3.1 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 solver towards non-spurious counterexamples. For the B5 benchmark, the SAT solver 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 problem and the solving runtime.

3.3.2 Accuracy of the generated models

The algorithm terminates when $\rho = 1$ and therefore, by Theorems 1 and 2 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 ($d = 1$).

3.3.3 Impact of the initial sample

As described in Sect. 2.2, in each learning iteration i, $\mathcal{L}(\widehat{\mathcal{M}}_i) \supseteq \mathcal{T}_{CE_i} \cup \mathcal{T}_{CE_{i-1}} \cup \ldots \cup \mathcal{T}_{CE_1} \cup \mathcal{T}$ and $\mathcal{T}_{CE_i} \cap \mathcal{L}(\widehat{\mathcal{M}}_{i-1}) = \emptyset$. The algorithm terminates when $\mathcal{L}(\widehat{\mathcal{M}}_i) \supseteq \mathcal{L}(\mathcal{M})$. The number of learning iterations depends on $|\mathcal{L}(\mathcal{M}) \backslash \mathcal{L}(\widehat{\mathcal{M}}_0)|$, where $\widehat{\mathcal{M}}_0$ 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 $\mathcal{L}(\widehat{\mathcal{M}}_0) = \emptyset$. We observe that on an average the number of iterations increases ≈ 5 times compared to the number of iterations reported in Table 1.

3.4 Comparison with random sampling

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

Table 1 Results of experimental evaluation of the active learning algorithm

	$\|X\|$	Our Algorithm						Random Sampling			
		$\#iter$	$\|\hat{\mathcal{S}}\|$	ρ	d	$T(s)$	$\%T_m$	$\|\hat{\mathcal{S}}\|$	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	13	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		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	6	2	6	**1**	**1**	123.5	13.4	6	0.9	0.8	966
B17		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	11	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	16	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		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

Table 1 (continued)

	$\|X\|$	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

The bold is meant to highlight the benchmarks for which the algorithm was able to generate complete models

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

3.5 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 equivalent model ($d = 1$) 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

Table 2 Summary of related active model-learning implementations

Oracle	Model-learning algorithm		Generated model characteristics	
	State-Merge	L*	Symbolic	Complete
Black-box	[19, 59] [62, 64]	[1, 2, 7, 10, 11, 26] [29, 48, 57, 60]	[1, 2, 7, 10] [11, 29, 60]	
White-box		[9, 12, 17, 20] [23, 24, 30, 56]	[9, 23]	[9]

on the algorithm used as the model-learning component and its ability to consolidate trace information into symbolic abstractions. The procedure to evaluate the degree of completeness for a learned abstraction only formally guarantees the generation of a system overapproximation on algorithm termination.

For complex systems, model checking for counterexample analysis as described in Sect. 2.6 can be computationally expensive in practice. Here, simulation-based techniques [18, 50] could be a pragmatic alternative to explore system paths to check if the observation in the counterexample is reachable.

4 Related work

4.1 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 [8, 27, 40] algorithm and query-based learning [4, 32, 49] are popular choices for the model-learning component. State-merge algorithms reverse-engineer abstractions by constructing a Prefix Tree Acceptor (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.

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 Linear Temporal Logic (LTL) system properties are checked against the generated model to identify wrong behaviours [26, 57, 59, 62]. 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 [18], through a finite number of membership queries. For a white-box oracle implementation, algorithms use techniques such as fuzzing [66] and symbolic execution [38].

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 [6, 14, 25, 39, 44–46, 61]. Among these, the most closely related to our work are the techniques based on SAT and Bounded Model Checking (BMC). These techniques primarily check for input/output equivalence, i.e., assuming the inputs to each implementation are equal, the corresponding outputs are equal.

The SAT based techniques [25, 44] 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 [14, 39] 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 [39] 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 work, 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 expressive overapproximations for a system. A summary of related active learning implementations is provided in Table 2. In the following sections we describe these techniques in detail and compare them in terms of generated model completeness and expressivity.

4.2 Learning system overapproximations

State-merge algorithms are predominantly passive and generated abstractions admit only those system behaviours exemplified by the traces [27, 40, 41, 63, 65]. One of the earliest active algorithms using state-merge is Query-Driven State Merging (QSM) [19], where model refinement is guided by responses to membership queries posed to an end-user. Other active versions of state-merge use model checking [59, 62] and model-based testing [64] to identify spurious behaviours in the generated model. In [59, 62] a priori known LTL system properties are checked against the generated model. Counterexamples for property violations serve as negative traces for automaton refinement. In [64], tests generated from the learned model are used to simulate the system to identify any discrepancies. However, abstractions generated by these algorithms are not guaranteed to accept all system traces.

Query-based learning algorithms, such as Angluin's L* algorithm and its variants [5, 32, 37, 51], 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 over-approximations. 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 [2, 11, 51] on the system generated using techniques such as conformance testing or random walks of the hypothesis model.

An essential pre-requisite to enable black-box 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 [28, 31].

One way to address these challenges is to combine model learning with white-box techniques, such as fuzzing [56], symbolic execution [9, 24, 30] and model checking [17, 20], to extract system information at a lower cost. But, these are not always guaranteed to generate system overapproximations.

In [56], 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 [24, 30], 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* [9] algorithm combines L* with symbolic execution to iteratively learn an over-approximation in parallel to the models learned using L*. The algorithm terminates when the hypothesis model equals the over-approximation, and therefore generates exact system models. In [17, 20], 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.

Very closely related to our work are the algorithms that use L* in combination with black-box testing [48] and model checking [26, 57]. The latter use pre-defined LTL properties, similar to [59, 62], that are model-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 a refinement of the abstraction to remove incorrect behaviours. Black-box testing [48] may be a pragmatic approach to identify missing behaviours for an abstraction by simulating the learned model with a set of system execution traces. However, it is not guaranteed that the model admits all system traces, as this requires a complete set of execution traces.

4.3 Learning symbolic models

An open challenge with query-based active model learning is learning symbolic abstractions. Many practical applications of L* [12, 17] 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 [42, 43] to extend model inference to large alphabets by learning symbolic models where transitions are labelled with partitions of the alphabet.

In [1], 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 [2] 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 [29], CEGAR is used for automated alphabet abstraction refinement to preserve determinism in the generated abstraction. Given a model, the refinement procedure is triggered by counterexamples exposing non-determinism in the current abstraction.

The MAT* algorithm [5] generates symbolic finite automata (SFA), 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, a Binary Decision Diagram (BDD) algebra and SFAs over SFAs that accept a finite sequence 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 [7], an inferred Mealy machine is converted to a symbolic abstraction in a post-processing step. The algorithm, however, is restricted to learning models with simple predicates

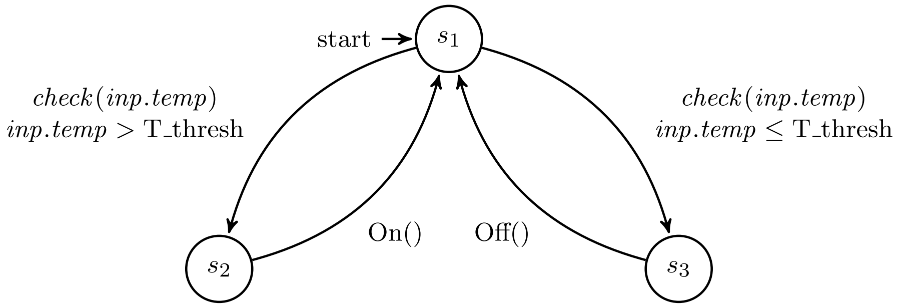
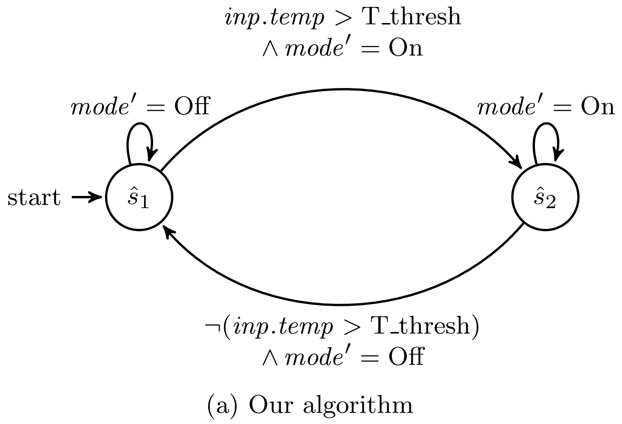


Fig. 9 IORA modelling a Home Climate Control Cooling system (B6)



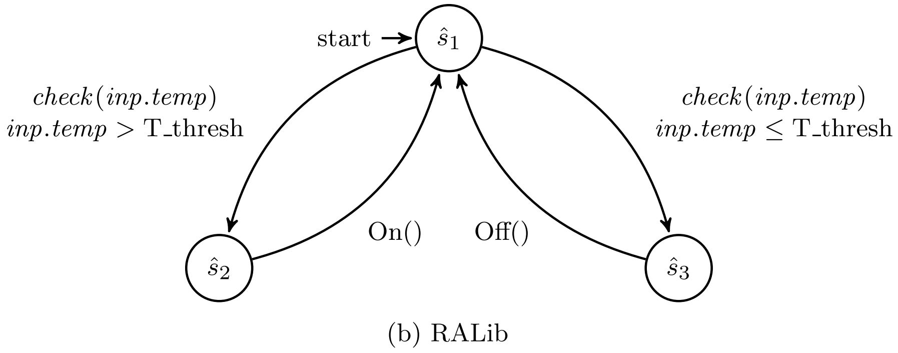


Fig. 10 Abstractions generated for a Home Climate Control Cooling system (B6)

such as equality/inequality relations. The algorithm in [60] is restricted to generating Mealy machines with a single timer. Sigma* [9] 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 abstractions that are complete, as illustrated in Table 2, an implementation of the algorithm is not publicly available for an experimental comparison.

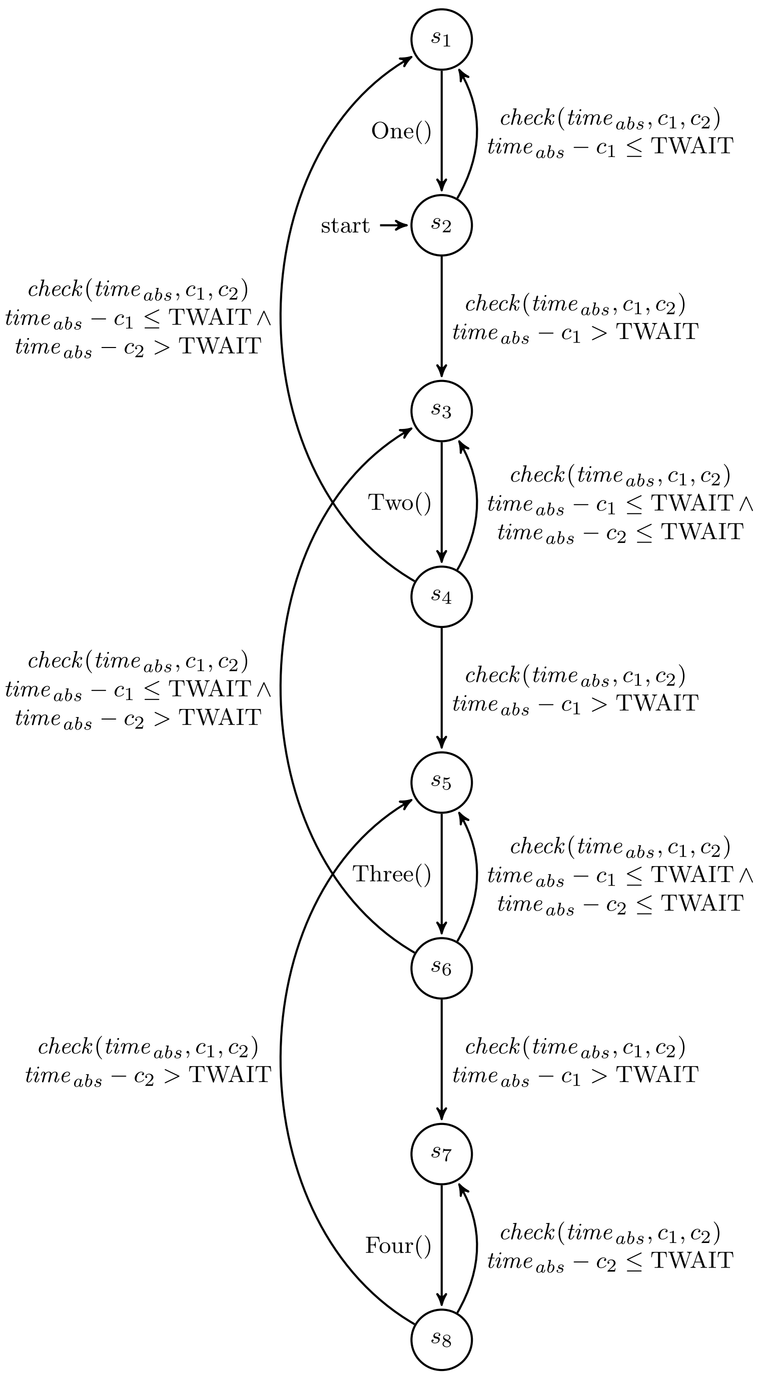

s_1
One()
$check(time_{abs}, c_1, c_2)$ $time_{abs} - c_1 \leq \text{TWAIT}$
start
s_2
$check(time_{abs}, c_1, c_2)$ $time_{abs} - c_1 \leq \text{TWAIT} \wedge$ $time_{abs} - c_2 > \text{TWAIT}$
$check(time_{abs}, c_1, c_2)$ $time_{abs} - c_1 > \text{TWAIT}$
s_3
Two()
$check(time_{abs}, c_1, c_2)$ $time_{abs} - c_1 \leq \text{TWAIT} \wedge$ $time_{abs} - c_2 \leq \text{TWAIT}$
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

Fig. 11 IORA modelling gear-shift logic for an Automatic Transmission Gear system (B1)

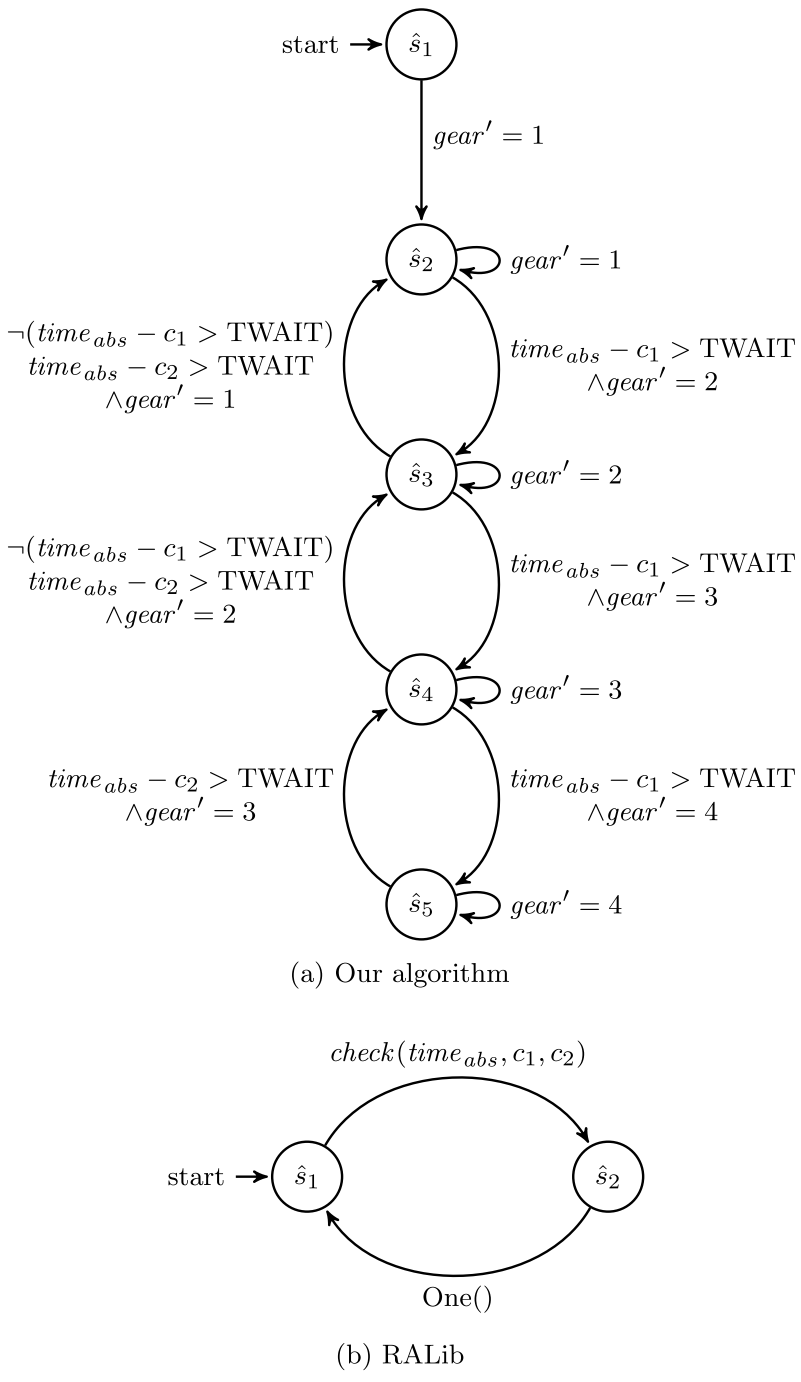

start
$\hat{s}_1$
$gear' = 1$
$\hat{s}_2$
$gear' = 1$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 1$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 2$
$\hat{s}_3$
$gear' = 2$
$\neg(time_{abs} - c_1 > \text{TWAIT})$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 2$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 3$
$\hat{s}_4$
$gear' = 3$
$time_{abs} - c_2 > \text{TWAIT}$
$\wedge gear' = 3$
$time_{abs} - c_1 > \text{TWAIT}$
$\wedge gear' = 4$
$\hat{s}_5$
$gear' = 4$
(a) Our algorithm
$check(time_{abs}, c_1, c_2)$
start
$\hat{s}_1$
$\hat{s}_2$
One()
(b) RALib

Fig. 12 Abstractions modelling gear-shift logic for an Automatic Transmission Gear system (B1)

The SL* algorithm [11] 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. RALib [10] implements the SL* algorithm and supports the inference of Input–Output Register Automaton (IORA). An IORA is a register automaton transducer that generates an output action after each input action.

We attempted to reverse-engineer the Simulink state machine benchmarks modelled as IORA using RALib. We present the results obtained for benchmarks B1 and B6. We modelled state machine B6 of a Home Climate Control Cooling system as an IORA with input action $check(inp.temp)$ that takes a parameter $inp.temp$, and output actions On() and Off() representing the operation modes of the system, as illustrated in Fig. 9. This is fed as the system-under-learning (SUL) to RALib. The models generated by our active learning approach and RALib are provided in Fig. 10. 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. 10b.

As illustrated in Fig. 11, 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. 12. RALib was only able to generate the partial model illustrated in Fig. 12b before timing out at 10 h. Our algorithm, on the other hand, was able to generate a complete model (Fig. 12a) in less than 12 min, as evidenced in Table 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 as TCP protocols [21]. Owing 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 [23] 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.

5 Use-cases and future work

In this article, we have presented a new active model-learning algorithm to learning abstractions of a system from its execution traces. The generated models are guaranteed to admit all system traces defined over a set of observations.

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 the implementation. The approach can also be used to evaluate test coverage for a given test suite and generate new tests to address coverage holes.

In the future, we intend to explore these potential use-cases further. This will drive improvements to reduce runtime, such as ways to guide the condition check procedure towards non-spurious counterexamples. We intend also to investigate extensions of the approach to model recursive state machines.

Appendix 1: List of benchmarks

See Table 3.

Table 3 Mapping of Simulink Stateflow example models to their benchmark number B# used in this article

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
ModelingARedundantSensorPairUsingAtomicSubchart			B21
ModelingASecuritySystem	InAlarm	On	B22
		Overall	B23

Table 3 (continued)

Benchmark Name			B#
ModelingASecuritySystem	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

Acknowledgements This work was done prior to joining Amazon.

Funding This research was funded in part by the Semiconductor Research Corporation, Task 2707.001, and Balliol College, Jason Hu scholarship.

References

1. Aarts F, Jonsson B, Uijen J (2010) Generating models of infinite-state communication protocols using regular inference with abstraction. In: Petrenko A, Simão A, Maldonado JC (eds) Testing software and systems. Springer, pp 188–204. https://doi.org/10.1007/978-3-642-16573-3_14
2. Aarts F, Heidarian F, Kuppens H, et al (2012) Automata learning through counterexample guided abstraction refinement. In: Giannakopoulou D, Méry D (eds) Formal methods. Springer, pp 10–27. https://doi.org/10.1007/978-3-642-32759-9_4
3. Alur R, Benedikt M, Etessami K, et al (2005) Analysis of recursive state machines. In: ACM Trans. Program. Lang. Syst., vol 27. Association for Computing Machinery, pp 786-818. https://doi.org/10.1145/1075382.1075387
4. Angluin D (1987) Learning regular sets from queries and counterexamples. In: Inf. Comput., vol 75. Academic Press, Inc., pp 87–106. https://doi.org/10.1016/0890-5401(87)90052-6
5. Argyros G, D'Antoni L (2018) The learnability of symbolic automata. In: Chockler H, Weissenbacher G (eds) Computer aided verification. Springer, pp 427–445. https://doi.org/10.1007/978-3-319-96145-3_23
6. Ashar P, Ghosh A, Devadas S (1992) Boolean satisfiability and equivalence checking using general binary decision diagrams. In: Integration, pp 1–16. https://doi.org/10.1016/0167-9260(92)90015-Q
7. Berg T, Jonsson B, Raffelt H (2008) Regular inference for state machines using domains with equality tests. In: Fiadeiro JL, Inverardi P (eds) Fundamental approaches to software engineering. Springer, pp 317–331.https://doi.org/10.1007/978-3-540-78743-3_24
8. Biermann AW, Feldman JA (1972) On the synthesis of finite-state machines from samples of their behavior. In: IEEE Trans. Comput., vol 21. IEEE Computer Society, pp 592–597. https://doi.org/10.1109/TC.1972.5009015
9. Botinčan M, Babić D (2013) Sigma*: Symbolic learning of input-output specifications. In: Principles of programming languages. ACM, POPL '13, pp 443–456. https://doi.org/10.1145/2429069.2429123
10. Cassel S, Howar F, Jonsson B (2015) RALib: a LearnLib extension for inferring EFSMs. In: DIFTS
11. Cassel S, Howar F, Jonsson B, et al (2016) Active learning for extended finite state machines. In: Formal aspects of computing, pp 233–263. https://doi.org/10.1007/s00165-016-0355-5
12. Chockler H, Kesseli P, Kroening D, et al (2020) Learning the language of software errors. In: Journal artificial intelligence research, vol 67. Morgan Kaufmann Publishers, Inc., pp 881–903. https://doi.org/10.1613/jair.1.11798
13. Clarke E, Grumberg O, Jha S, et al (2000) Counterexample-guided abstraction refinement. In: Emerson EA, Sistla AP (eds) Computer aided verification. Springer, pp 154–169. https://doi.org/10.1007/10722167_15
14. Clarke E, Kroening D, Yorav K (2003) Behavioral consistency of C and Verilog programs using bounded model checking. In: Design automation conference. Association for Computing Machinery, DAC '03, pp 368–371.https://doi.org/10.1145/775832.775928
15. Clarke E, Kroening D, Lerda F (2004) A tool for checking ANSI-C programs. In: Jensen K, Podelski A (eds) Tools and algorithms for the construction and analysis of systems. Springer, pp 168–176. https://doi.org/10.1007/978-3-540-24730-2_15
16. Clarke EM, Grumberg O, Kroening D et al (2018) Model checking, 2nd edn. MIT Press
17. Cobleigh JM, Giannakopoulou D, Păsăreanu CS (2003) Learning assumptions for compositional verification. In: Garavel H, Hatcliff J (eds) Tools and algorithms for the construction and analysis of systems. Springer, pp 331–346. https://doi.org/10.1007/3-540-36577-X_24

18. Dorofeeva R, El-Fakih K, Maag S, et al (2010) FSM-based conformance testing methods: a survey annotated with experimental evaluation. In: Information and software technology, pp 1286–1297. https://doi.org/10.1016/j.infsof.2010.07.001
19. Dupont P, Lambeau B, Damas C, et al (2008) The QSM algorithm and its application to software behavior model induction. In: Applied artificial intelligence, vol 22. Taylor & Francis, pp 77–115. https://doi.org/10.1080/08839510701853200
20. Feng L, Kwiatkowska M, Parker D (2011) Automated learning of probabilistic assumptions for compositional reasoning. In: Giannakopoulou D, Orejas F (eds) Fundamental approaches to software engineering. Springer, pp 2–17. https://doi.org/10.1007/978-3-642-19811-3_2
21. Fiterău-Broştean P, Howar F (2017) Learning-based testing the sliding window behavior of TCP implementations. In: Petrucci L, Seceleanu C, Cavalcanti A (eds) Critical systems: formal methods and automated verification. Springer, pp 185–200. https://doi.org/10.1007/978-3-319-67113-0_12
22. Gallier J, La Torre S, Mukhopadhyay S (2003) Deterministic finite automata with recursive calls and DPDAs. In: Inf. Process. Lett., pp 187–193. https://doi.org/10.1016/S0020-0190(03)00281-3
23. Garhewal B, Vaandrager F, Howar F, et al (2020) Grey-box learning of register automata. In: Integrated formal methods. Springer, pp 22–40. https://doi.org/10.1007/978-3-030-63461-2_2
24. Giannakopoulou D, Rakamarić Z, Raman V (2012) Symbolic learning of component interfaces. In: Miné A, Schmidt D (eds) Static analysis. Springer, pp 248–264. https://doi.org/10.1007/978-3-642-33125-1_18
25. Goldberg E, Prasad M, Brayton R (2001) Using SAT for combinational equivalence checking. In: Design, automation and test in Europe, pp 114–121. https://doi.org/10.1109/DATE.2001.915010
26. Groce A, Peled D, Yannakakis M (2002) Adaptive model checking. In: Katoen JP, Stevens P (eds) Tools and algorithms for the construction and analysis of systems. Springer, pp 357–370. https://doi.org/10.1007/3-540-46002-0_25
27. Heule MJH, Verwer S (2010) Exact DFA identification using SAT solvers. In: Sempere JM, García P (eds) Grammatical inference: theoretical results and applications. Springer, pp 66–79. https://doi.org/10.1007/978-3-642-15488-1_7
28. Howar F, Steffen B (2018) Active automata learning in practice: an annotated bibliography of the years 2011 to 2016. In: Bennaceur A, Hähnle R, Meinke K (eds) Machine learning for dynamic software analysis, lecture notes in computer science, vol 11026. Springer, pp 123–148. https://doi.org/10.1007/978-3-319-96562-8_5
29. Howar F, Steffen B, Merten M (2011) Automata learning with automated alphabet abstraction refinement. In: Jhala R, Schmidt D (eds) Verification, model checking, and abstract interpretation. Springer, pp 263–277.https://doi.org/10.1007/978-3-642-18275-4_19
30. Howar F, Giannakopoulou D, Rakamarić Z (2013) Hybrid learning: Interface generation through static, dynamic, and symbolic analysis. In: Symposium on software testing and analysis. Association for Computing Machinery, ISSTA, pp 268–279. https://doi.org/10.1145/2483760.2483783
31. Howar F, Jonsson B, Vaandrager F (2019) Combining black-box and white-box techniques for learning register automata. In: Steffen B, Woeginger G (eds) Computing and software science: state of the art and perspectives. Springer, pp 563–588. https://doi.org/10.1007/978-3-319-91908-9_26
32. Isberner M, Howar F, Steffen B (2014) The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour B, Smolka SA (eds) Runtime verification. Springer, pp 307–322. https://doi.org/10.1007/978-3-319-11164-3_26
33. Jeppu NY (2020) Trace2Model Github repository. https://github.com/natasha-jeppu/Trace2Model
34. Jeppu NY (2021) ActiveLearning. https://github.com/natasha-jeppu/ActiveLearning
35. Jeppu NY (2023). Active learning implementation. https://doi.org/10.5287/ora-aownkwvym
36. Jeppu NY, Melham T, Kroening D, et al (2020) Learning concise models from long execution traces. In: 57th ACM/IEEE design automation conference (DAC), pp 1–6. https://doi.org/10.1109/DAC18072.2020.9218613
37. Kearns MJ, Vazirani UV (1994) An introduction to computational learning theory. MIT Press
38. King JC (1976) Symbolic execution and program testing. In: Commun. ACM, vol 19. Association for Computing Machinery, pp 385–394. https://doi.org/10.1145/360248.360252
39. Kroening D, Clarke E (2004) Checking consistency of C and Verilog using predicate abstraction and induction. pp 66–72. https://doi.org/10.1109/ICCAD.2004.1382544
40. Lang KJ, Pearlmutter BA, Price RA (1998) Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar V, Slutzki G (eds) Grammatical inference. Springer, pp 1–12. https://doi.org/10.1007/BFb0054059
41. Lorenzoli D, Mariani L, Pezzè M (2008) Automatic generation of software behavioral models. In: ACM/IEEE 30th international conference on software engineering, pp 501–510. https://doi.org/10.1145/1368088.1368157

42. Maler O, Mens I (2014a) Learning regular languages over large ordered alphabets. In: Logical methods in computer science. https://doi.org/10.2168/LMCS-11(3:13)2015
43. Maler O, Mens IE (2014b) Learning regular languages over large alphabets. In: Ábrahám E, Havelund K (eds) Tools and algorithms for the construction and analysis of systems. Springer, pp 485–499. https://doi.org/10.1007/978-3-642-54862-8_41
44. Marques-Silva J, Glass T (1999) Combinational equivalence checking using satisfiability and recursive learning. In: Design, automation and test in Europe, pp 145–149. https://doi.org/10.1109/DATE.1999.761110
45. Marquez CIC, Strum M, Chau WJ (2013) Formal equivalence checking between high-level and RTL hardware designs. In: Latin American test workshop (LATW), pp 1–6. https://doi.org/10.1109/LATW.2013.6562666
46. Mukherjee R, Kroening D, Melham T, et al (2015) Equivalence checking using trace partitioning. In: VLSI, pp 13–18. https://doi.org/10.1109/ISVLSI.2015.110
47. Park DMR (1981) Concurrency and automata on infinite sequences. In: Theoretical computer science, lecture notes in computer science, vol 104. Springer, pp 167–183. https://doi.org/10.1007/BFb0017309
48. Peled D, Vardi M, Yannakakis M (2002) Black box checking. In: Journal of automata, languages and combinatorics, pp 225–246. https://doi.org/10.1007/978-0-387-35578-8_13
49. Rivest RL, Schapire RE (1989) Inference of finite automata using homing sequences. In: Theory of computing. Association for Computing Machinery, STOC ’89, pp 411–420. https://doi.org/10.1145/73007.73047
50. Ruf J, Hoffmann D, Kropf T, et al (2001) Simulation-guided property checking based on multi-valued AR-automata. In: Design, automation and test in Europe. IEEE, pp 742–748. https://doi.org/10.1109/DATE.2001.915111
51. Shahbaz M, Groz R (2009) Inferring Mealy machines. In: Cavalcanti A, Dams DR (eds) Formal methods. Springer, pp 207–222. https://doi.org/10.1007/978-3-642-05089-3_14
52. Sheeran M, Singh S, Stålmarck G (2000) Checking safety properties using induction and a SAT-solver. In: Hunt WA, Johnson SD (eds) Formal methods in computer-aided design. Springer, pp 127–144. https://doi.org/10.1007/3-540-40922-X_8
53. Simulink (2021) Embedded Coder. https://uk.mathworks.com/products/embedded-coder.html
54. Simulink (2021a) Simulation and Model-Based Design. https://www.mathworks.com/products/simulink.html
55. Simulink (2021b) Stateflow Examples. https://uk.mathworks.com/help/stateflow/examples.html?s_tid=CRUX_topnav
56. Smetsers R, Moerman J, Janssen M, et al (2016) Complementing model learning with mutation-based fuzzing. arXiv:1611.02429
57. Steffen B, Hungar H (2003) Behavior-based model construction. In: Zuck LD, Attie PC, Cortesi A, et al (eds) Verification, model checking, and abstract interpretation. Springer, pp 5–19. https://doi.org/10.1007/s10009-004-0139-8
58. Ulyantsev V, Tsarev F (2011) Extended finite-state machine induction using SAT-solver. In: International conference on machine learning and applications and workshops, pp 346–349. https://doi.org/10.1109/ICMLA.2011.166
59. Ulyantsev V, Buzhinsky I, Shalyto A (2018) Exact finite-state machine identification from scenarios and temporal properties. In: International journal on software tools for technology transfer. https://doi.org/10.1007/s10009-016-0442-1
60. Vaandrager F, Bloem R, Ebrahimi M (2021) Learning Mealy machines with one timer. In: Leporati A, Martín-Vide C, Shapira D, et al (eds) Language and automata theory and applications. Springer, pp 157–170. https://doi.org/10.1007/978-3-030-68195-1_13
61. van Eijk CAJ (2000) Sequential equivalence checking based on structural similarities. In: IEEE Transactions on computer-aided design of integrated circuits and systems, pp 814–819. https://doi.org/10.1109/43.851997
62. Walkinshaw N, Bogdanov K (2008) Inferring finite-state models with temporal constraints. In: Automated software engineering, pp 248–257. https://doi.org/10.1109/ASE.2008.35
63. Walkinshaw N, Hall M (2016) Inferring computational state machine models from program executions. In: International conference on software maintenance and evolution (ICSME). IEEE, pp 122–132. https://doi.org/10.1109/ICSME.2016.74
64. Walkinshaw N, Derrick J, Guo Q (2009) Iterative refinement of reverse-engineered models by model-based testing. In: Cavalcanti A, Dams DR (eds) Formal methods. Springer, pp 305–320. https://doi.org/10.1007/978-3-642-05089-3_20

65. Walkinshaw N, Taylor R, Derrick J (2016) Inferring extended finite state machine models from software executions. In: Empirical software engineering, pp 811–853. https://doi.org/10.1007/s10664-015-9367-7
66. Zalewski M (2013) American Fuzzy Lop (AFL) fuzzer. https://lcamtuf.coredump.cx/afl/