Learning a Behavior Model of Hybrid Systems Through Combining Model-Based Testing and Machine Learning

Bernhard K. Aichernig, Roderick Bloem, Masoud Ebrahimi, Martin Horn, Franz Pernkopf, Wolfgang Roth, Astrid Rupp, Martin Tappler, and Markus Tranninger

Graz University of Technology, Graz, Austria
aichernig@ist.tugraz.at, roderick.bloem@iaik.tugraz.at, masoud.ebrahimi@iaik.tugraz.at, martin.horn@tugraz.at, pernkopf@tugraz.at, roth@tugraz.at, astrid.rupp@fprimezero.com, martin.tappler@ist.tugraz.at, markus.tranninger@tugraz.at

Abstract. Models play an essential role in the design process of cyber-physical systems. They form the basis for simulation and analysis and help in identifying design problems as early as possible. However, the construction of models that comprise physical and digital behavior is challenging. Therefore, there is considerable interest in learning such hybrid behavior by means of machine learning which requires sufficient and representative training data covering the behavior of the physical system adequately. In this work, we exploit a combination of automata learning and model-based testing to generate sufficient training data fully automatically.
Experimental results on a platooning scenario show that recurrent neural networks learned with this data achieved significantly better results compared to models learned from randomly generated data. In particular, the classification error for crash detection is reduced by a factor of five and a similar F1-score is obtained with up to three orders of magnitude fewer training samples.

Keywords: Hybrid Systems · Behavior Modeling · Automata Learning · Model-Based Testing · Machine Learning · Autonomous Vehicle · Platooning

1 Introduction

In Cyber Physical Systems (CPSs), embedded computers and networks control physical processes. Most often, CPSs interact with their surroundings based on the context and the (history of) external events through an analog interface. We use the term hybrid system to refer to such reactive systems that intermix discrete and continuous components [23]. Since hybrid systems are dominating safety-critical areas, safety assurances are of utmost importance. However, we know that most verification problems for hybrid systems are undecidable [14].

Therefore, models and model-based simulation play an essential role in the design process of such systems. They help in identifying design problems as early as possible and facilitate integration testing with model-in-the-loop techniques. However, the construction of hybrid models that comprise physical and digital behavior is challenging. Modeling such systems with reasonable fidelity requires expertise in several areas, including control engineering, software engineering and sensor networks [9].

Therefore, we see a growing interest in learning such cyber-physical behavior with the help of machine learning. Examples include helicopter dynamics [29], the physical layer of communication protocols [26], standard continuous control problems [11], and industrial process control [35].

However, in general, machine learning requires a large and representative set of training data. Moreover, for the simulation of safety-critical features, rare side-conditions need to be sufficiently covered. Given the large state-space of hybrid systems, it is difficult to gather a good training set that captures all critical behavior. Neither nominal samples from operation nor randomly generated data will be sufficient. Here, advanced test-case generation methods can help to derive a well-designed training set with adequate coverage.

In this paper, we combine automata learning and Model-Based Testing (MBT) to derive an adequate training set, and then use machine learning to learn a behavior model from a black-box hybrid system. We can use the learned behavior model for multiple purposes such as monitoring runtime behavior. Furthermore, it could be used as a surrogate of a complex and heavy-weight simulation model to efficiently analyze safety-critical behavior offline [33]. Figure 1 depicts the overall execution flow of our proposed setting. Given a black-box hybrid system, we learn automata as discrete abstractions of the system. Next, we investigate the learned automata for critical behaviors. Once behaviors of interest are discovered, we use MBT to drive the hybrid system towards these behaviors and determine its observable actions in a continuous domain. This process results in a behavioral dataset with high coverage of the hybrid system's behavior including rare conditions. Finally, we train a Recurrent Neural Network (RNN) model that generalizes the behavioral dataset. For evaluation, we compared four different testing approaches, by generating datasets via testing, learning RNN models from the data and computing various performance measures for detecting critical behaviors in unforeseen situations. Experimental results show that RNNs learned with data generated via MBT achieved signifi-

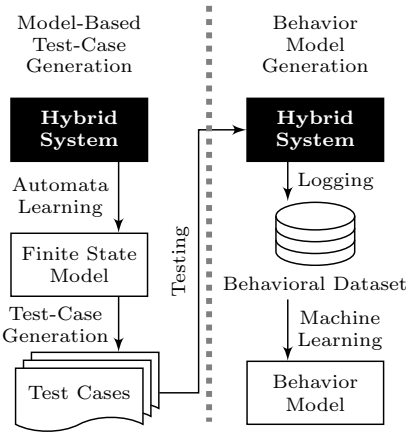


Fig. 1: Learning a behavior model of a black-box hybrid system.

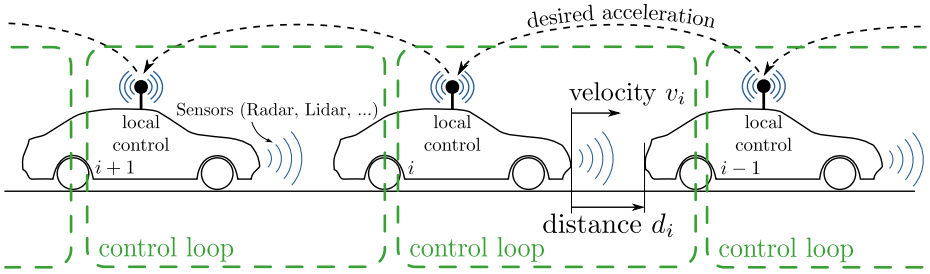


Fig. 2: Platooning as distributed control scenario. Adapted from a figure in [10].

cantly better performance compared to models learned from randomly generated data. In particular, the classification error is reduced by a factor of five and a similar F1-score is accomplished with up to three orders of magnitude fewer training samples.

Motivating Example. Throughout the paper we illustrate our approach utilizing a platooning scenario, implemented in a testbed in the Automated Driving Lab at Graz University of Technology (see also `https://www.tugraz.at/institute/irt/research/automated-driving-lab/`). Platooning of vehicles is a complex distributed control scenario, see Fig. 2. Local control algorithms of each participant are responsible for reliable velocity and distance control. The vehicles continuously sense their environments, e.g. the distance to the vehicle ahead and may use discrete, i.e. event triggered, communication to communicate desired accelerations along the platoon [10]. Besides individual vehicle stability, the most crucial goal in controller design is to guarantee so-called string stability of the platoon [28]. This stability concept basically demands that errors in position or velocity do not propagate along the vehicle string which otherwise might cause accidents or traffic jams upstream. Controllers for different platooning scenarios and spacing policies are available, e. g., constant time headway spacing [28] or constant distance spacing with or without communication [31].

Available controller designs are legitimated by rigorous mathematical stability proofs as an important theoretical foundation. In real applications it is often hard to fulfill every single modeling assumption of the underlying proofs, e. g., perfect sensing or communication. However, these additional uncertainties can often be captured in fine-grained simulation models. This motivates MBT of vehicle platooning control algorithms by the approach presented in this paper. Also, the learned behavior model can be used to detect undesired behavior during run-time. In [10], a hybrid system formulation of a platooning scenario is presented based on control theoretic considerations. In this contribution we aim to determine targeted behavior of such models with as few assumptions as possible by combining MBT and machine learning. As a first step, we consider two vehicles of the platoon, the leader and its first follower, in this paper, but the general approach can be extended to more vehicles.

Outline. This paper has the following structure. Section 2 summarizes automata learning and MBT. Section 3 explains how to learn an automaton from a black-box hybrid system, then use it to target interesting behavior of the hybrid system such that we create a behavioral dataset that can be used for machine learning purposes. Section 4 discusses the results gained by applying our approach to a real-world platooning scenario. Section 5 covers related work. Section 6 concludes and discusses future research directions.

2 Preliminaries

Definition 1 (Mealy Machine). *A Mealy machine is a tuple* $\langle \mathrm{I}, \mathrm{O}, Q, q_0, \delta, \lambda \rangle$ *where* Q *is a nonempty set of states,* q_0 *is the initial state,* $\delta : Q \times \mathrm{I} \to Q$ *is a state-transition function and* $\lambda : Q \times \mathrm{I} \to \mathrm{O}$ *is an output function.*

We write $q \xrightarrow{i/o} q'$ if $q' = \delta(q, i)$ and $o = \lambda(q, i)$. We extend δ as usual to δ^* for input sequences π_i, i. e., $\delta^*(q, \pi_i)$ is the state reached after executing π_i in q.

Definition 2 (Observation). *An observation* π *over input/output alphabet* I *and* O *is a pair* $\langle \pi_i, \pi_o \rangle \in I^* \times O^*$ *s.t.* $|\pi_i| = |\pi_o|$. *Given a Mealy machine* $\mathcal{M}$*, the set of observations of* $\mathcal{M}$ *from state* q *denoted by* $obs_{\mathcal{M}}(q)$ *are* $obs_{\mathcal{M}}(q) = \left\{ \langle \pi_i, \pi_o \rangle \in \mathrm{I}^* \times \mathrm{O}^* \;\middle|\; \exists\, q' : q \xrightarrow{\pi_i/\pi_o}{}^* q' \right\}$*, where* $\xrightarrow{\pi_i/\pi_o}{}^*$ *is the transitive and reflexive closure of the combined transition-and-output function to sequences which implies* $|\pi_i| = |\pi_o|$. *From this point forward,* $obs_{\mathcal{M}} = obs_{\mathcal{M}}(q_0)$. *Two Mealy machines* $\mathcal{M}_1$ *and* $\mathcal{M}_2$ *are observation equivalent, denoted* $\mathcal{M}_1 \approx \mathcal{M}_2$*, if* $obs_{\mathcal{M}_1} = obs_{\mathcal{M}_2}$.

2.1 Active Automata Learning

In her semimal paper, Angluin [5] presented L*, an algorithm for learning a deterministic finite automaton (DFA) accepting an unknown regular language L from a minimally adequate teacher (MAT). Many other active learning algorithms also use the MAT model [16]. An MAT generally needs to be able to answer two types of queries: *membership* and *equivalence* queries. In DFA learning, the learner asks membership queries, checking inclusion of words in the language L. Once gained enough information, the learner builds a hypothesis automaton $\mathcal{H}$ and asks an equivalence query, checking whether $\mathcal{H}$ accepts exactly L. The MAT either responds with *yes*, meaning that learning was successful. Otherwise it responds with a counterexample to equivalence, i. e., a word in the symmetric difference between L and the language accepted by $\mathcal{H}$. If provided with a counterexample, the learner integrates it into its knowledge and starts a new round of learning by issuing membership queries, which is again concluded by a new equivalence query. L* is adapted to learn Mealy machines by Shahbaz and Groz [32]. The basic principle remains the same, but *output queries* replace membership queries asking for outputs produced in response to input sequences. The goal in this adapted L* algorithm, is to learn a Mealy machine that is observation equivalent to a black-box system under learning (SUL).

Abstraction. L* is only affordable for small alphabets; hence, Aarts et al. [1] suggested to abstract away the concrete domain of the data, by forming equivalence classes in the alphabets. This is usually done by a mapper placed in between the learner and the SUL; see Fig. 3. Practically, mappers are state-full components transducing symbols back and forth between abstract and concrete alphabets using constraints defined over different ranges of concrete values. Since the input and output space of control systems is generally large or of unbounded size, we also apply abstraction by using a mapper.

The mapper communicates with the SUL via the concrete alphabet and with the learner via the abstract alphabet. In the setting shown in Fig. 3, the learner behaves like the L* algorithm by Shahbaz and Groz [32], but the teacher answers to the queries by interacting with the SUL through the mapper.

Learning and Model-Based Testing. Teachers are usually implemented via testing to learn models of black-box systems, The teacher in Fig. 3 wraps the SUL, uses a mapper for abstraction and includes a Model-Based Testing (MBT) component. Output queries typically reset the SUL, execute a sequence of inputs and collect the produced outputs, i. e., they perform a single test of the SUL. Equivalence queries are often approximated via MBT [3]. For that, an MBT component derives test cases (test queries) from the hypothesis model, which are executed to find discrepancies between the SUL and the learned hypothesis, i. e., to find counterexamples to equivalence.

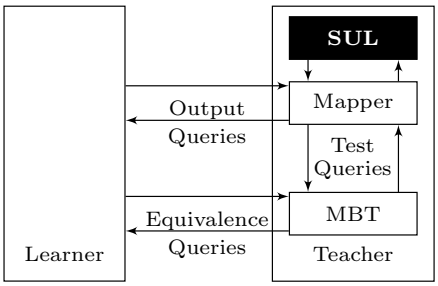


Fig. 3: Abstract automata learning [37].

Various MBT techniques have been applied in active automata learning, like the W-Method [8, 38], or the partial W-Method [13], which are also implemented in LearnLib [18]. These techniques attempt to prove conformance relative to some bound on the SUL states. However, these approaches require a large number of tests. Given the limited testing time available in practice, it is usually necessary to aim at "finding counterexamples fast" [17]. Therefore, randomized testing has recently shown to be successful in the context of automata learning, such as a randomized conformance testing technique [34] and fault-coverage-based testing [4]. We apply a variation of the latter, which combines transition coverage as test selection criterion with randomization.

While active automata learning relies on MBT to implement equivalence queries, it also enables MBT, by learning models that serve as basis for testing [3, 16]. Automata learning can be seen as collecting and incrementally refining information about a SUL through testing. This process is often combined with formal verification of requirements, both at runtime and also offline using learned models. This combination has been pioneered by Peled et al. [27] and called black-box checking. More generally, approaches that use automata learning for testing are also referred to as Learning-Based Testing (LBT) [25].

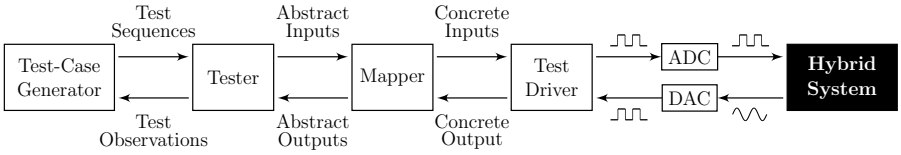


Fig. 4: Components involved in the testing process

3 Methodology

Our goal is to learn a behavior model capturing the targeted behavior of a hybrid SUL. The model's response to a trajectory of input variables, (e. g., sensor information), shall conform to the SUL's response with high accuracy and precision. As in discrete systems, purely random generation of input trajectories is unlikely to exercise the SUL's state space adequately. Consequently, models learned from random traces cannot accurately capture the SUL's behavior. Therefore, we propose to apply automata learning followed by MBT to collect system traces while using a machine learning method (i. e., Recurrent Neural Networks) for model learning. Figure 1 shows a generalized version of our approach.

Our trace-generation approach does not require any knowledge, like random sampling, but may benefit from domain knowledge and specified requirements. For instance, we do not explore states any further, which already violate safety requirements. In the following, we will first discuss the testing process. This includes interaction with the SUL, abstraction, automata learning and test-case generation. Then, we discuss learning a behavior model in the form of a Recurrent Neural Network with training data collected by executing tests.

Back to Motivating Example. We learn a behavior model for our platooning scenario in three steps: (1) automata learning exploring a discretized platooning control system to capture the state space structure in learned models, (2) MBT exploring the state space of the learned model directed towards targeted behavior while collecting non-discrete system traces. In step (3), we generalize from those trace by learning a Recurrent Neural Network.

3.1 Testing Process

We apply various test-case generation methods, with the same underlying abstraction and execution framework. Figure 4 depicts the components implementing the testing process.

- **Test-Case Generator:** the test-case generator creates abstract test cases. These test-cases are generated offline as sequences of abstract inputs.
- **Tester:** the tester takes an input sequence and passes it to the mapper. Feedback from test-case execution is forwarded to the test-case generator.

- **Mapper:** the mapper maps each abstract input to a concrete input variable valuation and a duration, defining how long the input should be applied. Concrete output variable valuations observed during testing are mapped to abstract outputs. Each test sequence produces an abstract output sequence which is returned to the tester.
- **Test Driver & Hybrid System:** The test driver interacts with the hybrid system by setting input variables and sampling output variable values.

3.1.1 System Interface and Sampling. We assume a system interface comprising two sets of real-valued variables: input variables U and observable output variables Y, with U further partitioned into controllable variables U_C and uncontrollable, observable input variables U_E affected by the environment. We denote all observable variables by $Obs = Y \cup U_E$. Additionally, we assume the ability to *reset* the SUL, as all test runs for trace generation need to start from a unique initial state. During testing, we change the controllable variables U_C and observe the evolution of variable valuations at fixed sampling intervals of length t_s.

Back to Motivating Example. We implemented our platooning SUL in MathWorks Simulink®. The implementation actually models a platoon of remote-controlled trucks used in our testbed at the Automated Driving Lab, therefore the acceleration values and distance have been downsized. The SUL interface comprises: $U_C = \{acc\}$, $Y = \{d, v_l, v_f\}$, and $U_E = \{\Delta\}$. The leader acceleration 'acc' is the single controllable input with values ranging from $-1.5m/s^2$ to $1.5m/s^2$, the distance between leader and first follower is 'd', and 'v_l' and 'v_f' are the velocities of the leader and the follower, respectively; finally 'Δ' denotes the angle between the leader and the x-axis in a fixed coordinate systems given in radians, i. e., it represents the orientation of the leader that changes while driving along the road. We sampled values of these variables at fixed discrete time steps, which are $t_s = 250$ milliseconds apart.

3.1.2 Abstraction. We discretize variable valuations for testing via the mapper. With that we effectively abstract the hybrid system such that a Mealy machine over an abstract alphabet can model it. Each abstract input is mapped to a concrete valuation for U_C and a duration specifying how long the valuation shall be applied, thus U_C only takes values from a finite set. As abstract inputs are mapped to uniquely defined concrete inputs, this form of abstraction does not introduce non-determinism. In contrast, values of observable variables *Obs* are not restricted to a finite set. Therefore, we group concrete valuations of *Obs* and assign an abstract output label to each group.

The mapper also defines a set of labels *Violations* containing abstract outputs that signal violations of assumptions or safety requirements. In the abstraction to a Mealy machine, these outputs lead to trap states from which the model does not transit away. Such a policy prunes the abstract state space.

A mapper has five components: (1) an abstract input alphabet I, (2) a corresponding concretization function γ, (3) an abstraction function α mapping

concrete output values to (4) an abstract output alphabet O, and (5) the set *Violations*. During testing, it performs the following two actions:

- **Input Concretization:** the mapper maps an abstract symbol $i \in I$ to a pair $\gamma(i) = (\nu, d)$, where ν is a valuation of U_C and $d \in \mathbb{N}$ defining time steps, for how long U_C is set according to ν. This pair is passed to the test driver.
- **Output Abstraction:** the mapper receives concrete valuations ν of *Obs* from the test driver and maps them to an abstract output symbol $o = \alpha(\nu)$ in O that is passed to the tester. If $o \in$ *Violations*, then the mapper stores o in its state and maps all subsequent concrete outputs to o until it is reset.

The mapper state needs to be reset before every test-case execution. Repeating the same symbol $o \in$ *Violations*, if we have seen it once, creates trap states to prune the abstract state space. Furthermore, the implementation of the mapper contains a cache, returning abstract output sequences without SUL interaction.

Back to Motivating Example. We tested the SUL with an alphabet I of six abstract inputs: *fast-acc*, *slow-acc*, *const*, *const$_l$*, *brake* and *hard-brake*, concretized by $\gamma(\textit{fast-acc}) = (acc \mapsto 1.5m/s^2, 2)$, $\gamma(\textit{slow-acc}) = (acc \mapsto 0.7m/s^2, 2)$, $\gamma(\textit{const}) = (acc \mapsto 0m/s^2, 2)$, $\gamma(\textit{const}_l) = (acc \mapsto 0m/s^2, 8)$, $\gamma(\textit{brake}) = (acc \mapsto -0.7m/s^2, 2)$, and $\gamma(\textit{hard-brake}) = (acc \mapsto -1.5m/s^2, 2)$.Thus, each input takes two time steps, except for *const$_l$*, which represents prolonged driving at constant speed.

The output abstraction depends on the distance d and the leader velocity v_l. If v_l is negative, we map to the abstract output *reverse*. Otherwise, we partition d into 7 ranges with one abstract output per range, e. g., the range $(-\infty, 0.43m)$ (length of a remote-controlled truck) is mapped to *crash*. We assume that platoons do not drive in reverse. Therefore, we include *reverse* in *Violations*, such that once we observe *reverse*, we ignore the subsequent behavior. We also added *crash* to *Violations*, as we are only interested in the behavior leading to a crash.

3.1.3 Test-Case Execution. The concrete test execution is implemented by a test driver. It basically generates step-function-shaped inputs signals for input variables and samples output variable values. For each concrete input (ν_j, d_j) applied at time t_j (starting at $t_1 = 0ms$), the test driver sets U_C according to ν_j for $d_j \cdot t_s$ milliseconds and samples the values ν'_j of observable variables $Y \cup U_E$ at time $t_j + d_j \cdot t_s - {}^{t_s}/_2$. It then proceeds to time $t_{j+1} = t_j + d_j \cdot t_s$ to perform the next input if there is any. In that way, the test driver creates a sequence of sampled output variable values ν'_j, one for each concrete input. This sequence is passed to the mapper for output abstraction.

3.1.4 Viewing Hybrid Systems as Mealy Machines. Our test-case execution samples exactly one output value for each input, ${}^{t_s}/_2$ milliseconds before the next input, which ensures that there is an output for each input, such that

input and output sequences have the same length. Given an abstract input sequence π_i our test-case execution produces an output sequence π_o of the same length. In slight abuse of notation, we denote this relationship by $\lambda_h(\pi_i) = \pi_o$. Hence, we view the hybrid system under test on an abstract level as a Mealy machine $\mathcal{H}_m$ with $obs_{\mathcal{H}_m} = \{\langle \pi_i, \lambda_h(\pi_i)\rangle | \pi_i \in I^*\}$.

3.1.5 Learning Automata of Motivating Example. We applied the active automata learning algorithm by Kearns and Vazirani (KV) [19], implemented by LearnLib [18], in combination with the *transition-coverage* testing strategy described in previous work [4]. We have chosen the KV algorithm, as it requires fewer output queries to generate a new hypothesis model than, e. g., L* [5], such that more equivalence queries are performed. As a result, we can guide testing during equivalence queries more often. The Transition-Coverage Based Testing (TCBT) strategy is discussed below in Section 3.1.6.

Here, our goal is not to learn an accurate model, but to explore the SUL's state space systematically through automata learning. The learned hypothesis models basically keep track of what has already been tested. Automata learning operates in rounds, alternating between series of output queries and equivalence queries. We stop this process once we performed the maximum number of tests N_{autl}, which includes both output queries and test queries implementing equivalence queries. Due to the large state space of the analyzed platooning SUL, it was infeasable to learn a complete model, hence we stopped learning when reaching the bound N_{autl}, even though further tests could have revealed discrepancies.

Back to Motivating Example. The learned automata also provided insights into the behavior of the platooning SUL. A manual analysis revealed that collisions are more likely to occur, if trucks drive at constant speed for several time steps. Since we aimed at testing and analyzing the SUL with respect to dangerous situations, we created the additional abstract $const_l$ input, which initially was not part of the set of abstract inputs.

During active automata learning we executed approximately $N_{\text{autl}}260000$ concrete tests on the platooning SUL in 841 learning rounds, producing 2841 collisions. In the last round, we generated a hypothesis Mealy machine with 6011 states that we use for model-based testing. Generally, N_{autl} should be chosen as large as possible given the available time budget for testing, as a larger N_{autl} leads to more accurate abstract models.

3.1.6 Test-Case Generation. In the following, we describe random test-case generation for Mealy machines, which serves as a baseline. Then, we describe three different approaches to model-based test-case generation. Note that our testing goal is to explore the system's state space and to generate system traces with high coverage, with the intention of learning a neural network. Therefore, we generate a fixed number of test cases N_{train} and do not impose conditions on outputs other than those defined by the set *Violations* in the mapper.

3.1.6.1 Random Testing. Our random testing strategy generates input sequences with a length chosen uniformly at random between 1 and the maximum length $l_{\max}$. Inputs in the sequence are also chosen uniformly at random from I.

3.1.6.2 Learning-Based Testing. The LBT strategy performs automata learning as described in Section 3.1.5. It produces exactly those tests executed during automata learning and therefore sets N_{autl} to N_{train}. While this strategy systematically explores the abstract state space of the SUL, it also generates very simple tests during the early rounds of learning, which are not helpful for learning a behavior model in Section 3.2.

3.1.6.3 Transition-Coverage Based Testing. The Transition-Coverage Based Testing (TCBT) strategy uses a learned model of the SUL as basis. Basically, we learn a model, fix that model and then generate N_{train} test sequences with the *transition-coverage* testing strategy discussed in [4]. We use it, as it performed well in automata learning and it scales to large automata. The intuition behind it is that the combination of variability through randomization and coverage-guided testing is well-suited in a black-box setting as in automata learning.

Test-case generation from a Mealy machine $\mathcal{M}$ with this strategy is split into two phases, a generation phase and a selection phase. The generation phase generates a large number of tests by performing random walks through $\mathcal{M}$. In the selection phase, n tests are selected to optimize the coverage of the transitions of $\mathcal{M}$. Since the n required to cover all transitions may be much lower than N_{train}, we performed several rounds, alternating between generation and selection until we selected and executed N_{train} test cases.

3.1.6.4 Output-Directed Testing. Our Output-Directed Testing strategy also combines random walks with coverage-guided testing, but aims at covering a given abstract output '*label*'. Therefore, it is based on a learned Mealy machine of the SUL. A set consisting of N_{train} tests is generated by Algorithm 1. All tests consist of a random '*prefix*' that leads to a random source state q_r, an '*interfix*' leading to a randomly chosen destination state q'_r and a '*suffix*' from q'_r to the '*label*'. The suffix explicitly targets a specific output, the interfix aims to increase the coverall SUL coverage and the random prefix introduces variability.

Back to Motivating Example. In our platooning scenario, we aim at covering behavior relevant to collisions, thus we generally set *label* = *crash* and refer to the corresponding test strategy also as Crash-Directed Testing.

3.2 Learning a Recurrent Neural Network Behavior Model

In our scenario, we are given length T sequences of vectors $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_T)$ with $\mathbf{x}_i \in \mathbb{R}^{d_x}$ representing the inputs to the hybrid system, and the task is to predict corresponding length T sequences of target vectors $\mathbf{T} = (\mathbf{t}_1, \ldots, \mathbf{t}_T)$ with $\mathbf{t}_i \in \mathbb{R}^{d_y}$ representing the outputs of the hybrid system. Recurrent Neural Networks (RNNs) are a popular choice for modelling these kinds of problems.

Algorithm 1 Output-Directed test-case generator

Input: $\mathcal{M} = \langle I, O, Q, q_0, \delta, \lambda \rangle$, $label \in O$, N_{train}
Output: TestCases : a set of test cases directed to '$label \in O$'

```
TestCases ← ∅
while |TestCases| < N_train do
  rand-len ← RandomInteger
  prefix ← RandomSequence(I, rand-len)
  q_r ← δ*(q_0, prefix)
  q'_r ← RandomState(Q)
  interfix ← PathToState(q_r, q'_r)          ▷ input sequence to q'_r
  if interfix ≠ ⊥ then                       ▷ check if path to state exists
    suffix ← PathToLabel(q_r, label)         ▷ input sequence to label
    if suffix ≠ ⊥ then                       ▷ check if path to label exists
      TestCases ← TestCases ∪ {prefix · interfix · suffix}
return TestCases
```

Given a set of N training input/output sequence pairs $\mathcal{D} = \{(\mathbf{X}_n, \mathbf{T}_n)\}_{n=1}^{N}$, the task of machine learning is to find suitable model parameters such that the output sequences $\{\mathbf{Y}_n\}_{n=1}^{N}$ computed by the RNN for input sequences $\{\mathbf{X}_n\}_{n=1}^{N}$ closely match their corresponding target sequences $\{\mathbf{T}_n\}_{n=1}^{N}$, and, more importantly, generalize well to sequences that are not part of the training set $\mathcal{D}$, i.e., the RNN produces accurate results on unseen data. To obtain suitable RNN parameters, we typically minimize a loss function describing the misfit between predictions $\mathbf{Y}$ and ground truth targets $\mathbf{T}$. Here, we achieve this through a minimization procedure known as stochastic gradient descent which works efficiently. For details on RNN learning, we refer to the extended version of this paper [2].

Back to Motivating Example. In our platooning scenario, the inputs $\mathbf{x}_i \in \mathbb{R}^2$ at time step i to the hybrid system comprise the input variables U from Section 3.1.1, i. e., the acceleration value *acc* and the orientation Δ of the leader car in radians. We preprocess the orientation Δ and transform it to $\Delta' = \Delta_i - \Delta_{i-1}$, the angular difference of orientation in radians of consecutive time steps to get rid of discontinuities when these values are constrained to a fixed interval of length 2π. The outputs $\mathbf{y}_i \in \mathbb{R}^3$ at time step i of the hybrid system comprise the values of observable output variables Y from Section 3.1.1, i. e., the velocity of the leader v_l and the first follower v_f, respectively, as well as the distance d between the leader and the first follower.

Note that RNNs are not constrained to sequences of a fixed length T. However, training with fixed-length sequences is more efficient as it allows full parallelization through GPU computations. Hence, during test-case execution, we pad sequences at the end with concrete inputs $(acc \mapsto 0, 1)$, i. e., the leader drives at constant speed at the end of every test. In rare cases the collected test data showed awkward behavior that needed to be truncated at some time step, e. g., when the leader's velocity v_l became negative. We padded the affected sequences at the beginning by copying the initial state where all cars have zero velocity. We used this padding procedure to obtain fixed-length sequences with $T = 256$.

In our experiments we use RNNs with one hidden layer of 100 neurons. Since plain RNNs are well-known to lack the ability to model long-term dependencies,

we use long short-term memory (LSTM) cells for the hidden layers [15]. To evaluate the generated training sequences, we train models for several values of training set sizes N_{train}. We used ADAM [20] implemented in Keras [7] with a learning rate $\eta = 10^{-3}$ to perform stochastic gradient descent for 500 epochs. The number of training sequences per mini-batch is set to $\min(N_{\text{train}}/100, 500)$. Each experiment is performed ten times using different random initial parameters and we report the average performance measures over these ten runs.

4 Experimental Evaluations

4.1 Predicting crashes with RNNs

We aim to predict whether a sequence of input values results in a crash, i.e., we are dealing with a binary classification problem. A sequence is predicted as positive, i.e., the sequence contains a crash, if at any time step the leader-follower distance d gets below $0.43m$ which is the length of a remote-controlled truck.

For the evaluation, we generated validation sequences with the Output-Directed Testing strategy. This strategy produces crashes more frequently than the other testing strategies which is useful to keep the class imbalance between crash and non-crash sequences in the validation set minimal. We emphasize that these validation sequences do not overlap with the training sequences that were used to train the LSTM-RNN with Output-Directed Testing sequences. The validation set[1] contains $N_{\text{val}} = 86800$ sequences out of which 17092 (19.7%) result in a crash.

For the reported scores of our binary classification task we first define the following convenient values:

True Positive (TP): #{ positive sequences predicted as positive }
False Positive (FP): #{ negative sequences predicted as positive }
True Negative (TN): #{ negative sequences predicted as negative }
False Negative (FN): #{ positive sequences predicted as negative }

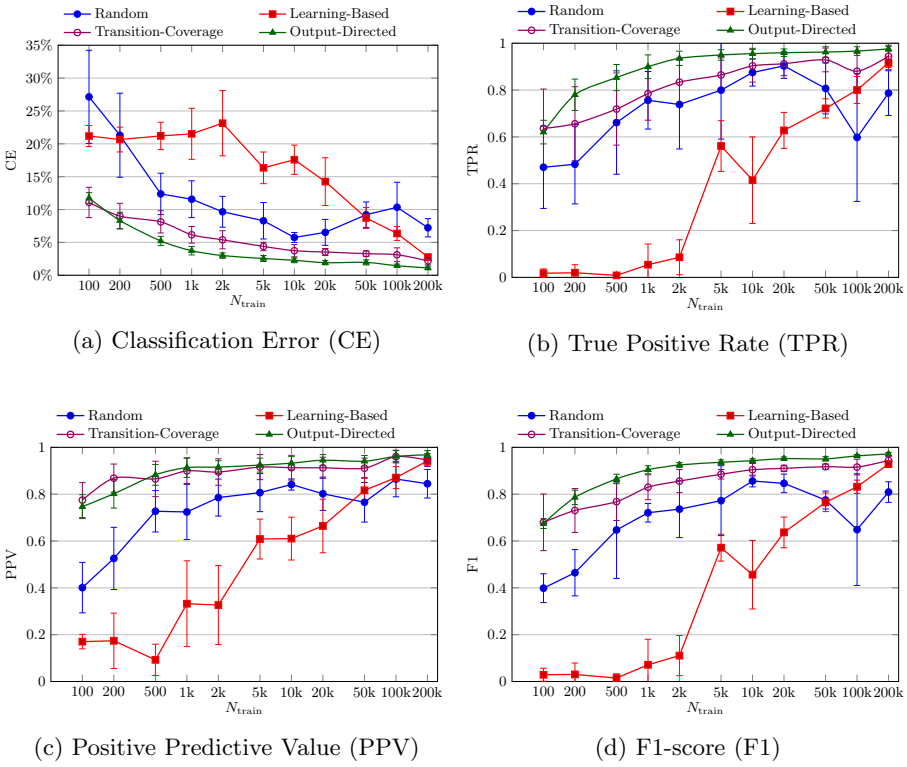
We report the following four measures: (1) the classification error (CE) in %, (2) the true positive rate (TPR), (3) the positive predictive value (PPV), and (4) the F1-score (F1). These scores are defined as

$$\text{CE} = \frac{\text{FP} + \text{FN}}{N_{\text{val}}} \times 100 \qquad \text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{PPV} = \frac{\text{TP}}{\text{TP} + \text{FP}} \qquad F1 = \frac{2\text{TP}}{2\text{TP} + \text{FP} + \text{FN}}$$

The TPR and the PPV suffer from the unfavorable property that they result in unreasonably high values if the LSTM-RNN simply classifies all sequences either as positive or negative. The F1-score is essentially the harmonic mean of

[1] This set is usually called test set in the context of machine learning, but here we adopt the term validation set to avoid confusion with model-based testing.



(a) Classification Error (CE)

(b) True Positive Rate (TPR)

(c) Positive Predictive Value (PPV)

(d) F1-score (F1)

Fig. 5: Performance measures for all testing strategies over changing N_{train}.

the TPR and the PPV so that these odd cases are ruled out. Note that while for the CE a smaller value indicates a better performance, for the other scores TPR, PPV, and F1 a higher score, i. e., closer to 1, indicates a better performance.

The average results and the standard deviations over ten runs for these scores are shown in Fig. 5. The LSTM-RNNs trained with sequences from Random Testing and LBT perform poorly on all scores especially if the number of training sequences N_{train} is small. Notably, we found that sequences generated by LBT during early rounds of automata learning are short and do not contain a lot of variability, explaining the poor performance of LBT for low N_{train}.

We can observe in Fig. 5b that Random Testing and LBT perform poorly at detecting crashes when they actually occur. Especially the performance drop of LBT at $N_{\text{train}} = 10000$ and of Random Testing at $N_{\text{train}} = 100000$ indicate that additional training sequences do not necessarily improve the capability to detect crashes as crashes in these sequences still appear to be outliers.

Training LSTM-RNNs with TCBT and Output-Directed Testing outperforms Random Testing and LBT for all training set sizes N_{train}, where the results slightly favor Output-Directed Testing. The advantage of TCBT and Output-Directed Testing becomes evident when comparing the training set size N_{train} required to achieve the performance that Random Testing achieves us-

ing the maximum of $N_{\text{train}} = 200000$ sequences. The CE of Random Testing at $N_{\text{train}} = 200000$ is 7.23% which LBT outperforms at $N_{\text{train}} = 100000$ with 6.36%, TCBT outperforms at $N_{\text{train}} = 1000$ with 6.16%, and Output-Directed Testing outperforms at $N_{\text{train}} = 500$ with 5.22%. Comparing LBT and Output-Directed Testing, Output-Directed Testing outperforms the 2.77% CE of LBT at $N_{\text{train}} = 200000$ with only $N_{\text{train}} = 5000$ sequences to achieve a 2.55% CE.

The F1-score is improved similarly: Random Testing with $N_{\text{train}} = 200000$ achieves 0.809, while TCBT achieves 0.830 using only $N_{\text{train}} = 1000$ sequences, and Output-Directed Testing achieves 0.865 using only $N_{\text{train}} = 500$ sequences. Comparing LBT and Output-Directed Testing, LBT achieves 0.929 at $N_{\text{train}} = 200000$ whereas Output-Directed Testing requires only $N_{\text{train}} = 5000$ to achieve a F1-score of 0.936. In total, the sample size efficiency of TCBT and Output-Directed Testing is two to three orders of magnitudes larger than for Random Testing and LBT.

4.2 Evaluation of the Detected Crash Times

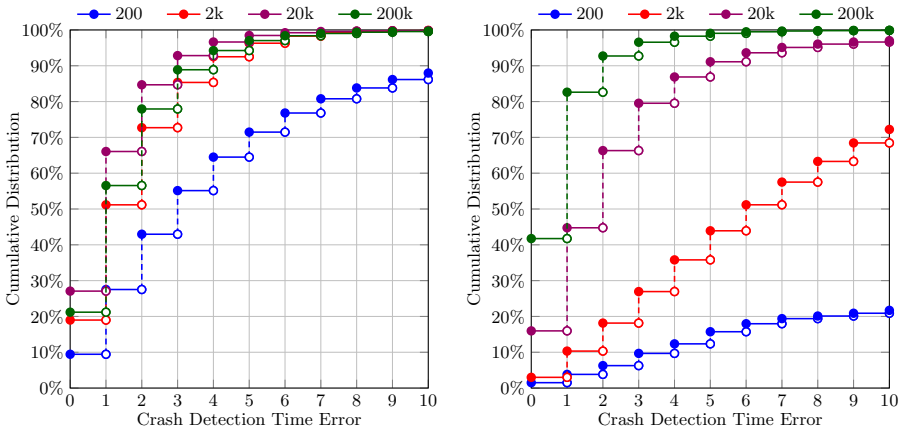
In the next experiment, we evaluate the accuracy of the crash prediction time. The predicted crash time is the earliest time step at which d drops below the threshold of $0.43m$, and the crash detection time error is the absolute difference between the ground truth crash time and the predicted crash time. Please note that the crash detection time error is only meaningful for true positive sequences.

Fig. 6 shows CDF plots describing how the crash detection time error distributes over the true positive sequences. It is desired that the CDF exhibits a steep increase at the beginning which implies that most of the crashes are detected close to the ground truth crash time. The CDF value at crash detection time error 0 indicates the percentage of sequences whose crash is detected without error at the correct time step.

As expected the results get better for larger training sizes N_{train}. Random Testing and LBT exhibit large errors and only relatively few sequences are classified without error. For Random Testing, less than 30% of the crashes in the true positive sequences are classified correctly using the maximum of $N_{\text{train}} = 200000$ sequences. On the other side, TCBT requires only $N_{\text{train}} = 20000$ sequences to classify 34.9% correctly, and Output-Directed Testing requires only $N_{\text{train}} = 2000$ to classify 41.8% correctly. Combining the results from Fig. 6 with the TPR shown in Fig. 5b strengthens the crash prediction quality even more: While TCBT and Output-Directed Testing do not only achieve a higher TPR, they also predict the crashes more accurately. Furthermore, TCBT and Output-Directed Testing classify 90.9% and 97.3% of the sequences with at most one time step error using the maximum of $N_{\text{train}} = 200000$ sequences, respectively.

5 Related Work

Verifying Platooning Strategies. Meinke [24] used LBT to analyze vehicle platooning systems with respect to qualitative safety properties, like collisions. While the automata learning setup is similar to our approach, he aimed (1)



(a) Random

(b) Learning-Based Testing

(c) Transition Coverage

(d) Output-Directed

Fig. 6: CDF plots for the difference between true crash time and predicted crash time for sequences that are correctly classified as resulting in a crash. Results are shown for all testing strategies and several training dataset sizes N_{train}.

to show how well a multi-core implementation of LBT method scales, and (2) how problem size and other factors affect scalability. Fermi et al. [12] applied rule inference methods to validate collision avoidance in platooning. More specifically, they used decision trees as classifiers for safe and unsafe platooning conditions, and they suggested three approaches to minimize the number of false negatives. Rashid et al. [30] modelled a generalized platoon controller formally in higher-order logic. They proved satisfication of stability constraints in HOL Light and showed how stability theorems can be used to develop runtime monitors.

System Identification and Safety Assurance. Determining models by using input-output data is known as *system identification* in the control systems community [21]. Such models can be useful for simulation, controller design or

diagnosis purposes. Recently, progress towards system identification techniques for hybrid systems based on the classical methods presented e.g. in the book of Ljung [21] has been made, see [39] and the references therein. In [39], single-input single-output models are considered. Furthermore, the contribution focuses on so-called piece-wise affine ARX models. We believe that the presented hybrid automata learning techniques could essentially contribute to this research field by relaxing some of the modeling assumptions.

If the model parameters and the switching mechanism is known, the problem reduces to a hybrid state estimation problem [22, 36]. Such estimators or observers are used in various problems, i.e. closed loop control, parameter estimation or diagnosis. However, the traditional methods often assume accurate and exact models. These models are mostly derived based on first principles [22], which is often not feasible in complex scenarios. This shows the advantage of our learning-based approach especially in cases without detailed model knowledge.

6 Future Work & Conclusion

We successfully combined abstract automata learning, MBT, and machine learning to learn a behavior model from observations of a hybrid system. Given a black-box hybrid system, we learn an abstract automaton capturing its discretized state-space; then, we use MBT to target a behavior of interest. This results in test suites with high coverage of the targeted behavior from which we generate a behavioral dataset. LSTM-RNNs are used to learn behavior models from the behavioral dataset. Advantages of our approach are demonstrated on a real-world case study; i. e., a platooning scenario. Experimental evaluations show that LSTM-RNNs learned with model-based data generation achieved significantly better results compared to models learned from randomly generated data, e. g., reducing the classification error by a factor of five, or achieving a relatively similar F1-score with up to three orders of magnitude fewer training samples than random testing. This is accomplished through systematic testing (i. e., automata learning, and MBT) of a black-box hybrid system without requiring a priori knowledge on its dynamics.

Motivated by the promising results shown in Sect. 4, we plan to carry out further case studies. For future research, we target runtime verification and runtime enforcement of safety properties for hybrid systems. To this end, we conjecture that a predictive behavior model enables effective runtime monitoring, which allows us to issue warnings or to intervene in case of likely safety violations.Adaptations of the presented approach are also potential targets for future research, e.g. automata learning and test-based trace generation could be interleaved in an iterative process.

Acknowledgment. This work is supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". It is also partially supported by ECSEL Joint Undertaking under Grant No.: 692455.

References

1. Aarts, F., Heidarian, F., Kuppens, H., Olsen, P., Vaandrager, F.W.: Automata learning through counterexample guided abstraction refinement. In: FM (2012)
2. Aichernig, B.K., Bloem, R., Ebrahimi, M., Horn, M., Pernkopf, F., Roth, W., Rupp, A., Tappler, M., Tranninger, M.: Learning a behavior model of hybrid systems through combining model-based testing and machine learning (full version). CoRR **abs/1907.04708** (2019), `http://arxiv.org/abs/1907.04708`
3. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Bennaceur et al. [6], pp. 74–100. , `https://doi.org/10.1007/978-3-319-96562-8_3`
4. Aichernig, B.K., Tappler, M.: Efficient active automata learning via mutation testing. Journal of Automated Reasoning (Oct 2018).
5. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. (1987)
6. Bennaceur, A., Hähnle, R., Meinke, K. (eds.): Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers, Lecture Notes in Computer Science, vol. 11026. Springer (2018). , `https://doi.org/10.1007/978-3-319-96562-8`
7. Chollet, F., et al.: Keras. `https://keras.io` (2015)
8. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering **4**(3), 178–187 (May 1978).
9. Derler, P., Lee, E.A., Sangiovanni-Vincentelli, A.L.: Modeling cyber-physical systems. Proceedings of the IEEE **100**(1), 13–28 (2012). , `https://doi.org/10.1109/JPROC.2011.2160929`
10. Dolk, V.S., Ploeg, J., Heemels, W.P.M.H.: Event-triggered control for string-stable vehicle platooning. IEEE Transactions on Intelligent Transportation Systems **18**(12), 3486–3500 (Dec 2017).
11. Duan, Y., Chen, X., Houthooft, R., Schulman, J., Abbeel, P.: Benchmarking deep reinforcement learning for continuous control. In: Balcan, M., Weinberger, K.Q. (eds.) ICML 2016. JMLR Workshop and Conference Proceedings, vol. 48, pp. 1329–1338. JMLR.org (2016), `http://jmlr.org/proceedings/papers/v48/duan16.html`
12. Fermi, A., Mongelli, M., Muselli, M., Ferrari, E.: Identification of safety regions in vehicle platooning via machine learning. In: WFCS (2018)
13. Fujiwara, S., von Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test selection based on finite state models. IEEE Transactions on Software Engineering **17**(6), 591–603 (1991).
14. Henzinger, T.A.: The theory of hybrid automata. In: LICS (1996)
15. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation **9**(8), 1735–1780 (1997)
16. Howar, F., Steffen, B.: Active automata learning in practice - an annotated bibliography of the years 2011 to 2016. In: Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers. pp. 123–148 (2018)
17. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS - lessons learned in the ZULU challenge. In: ISoLA. pp. 687–704 (2010)
18. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - A framework for active automata learning. In: CAV. pp. 487–495 (2015)

19. Kearns, M.J., Vazirani, U.V.: An Introduction to Computational Learning Theory. MIT Press, Cambridge, MA, USA (1994)
20. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. In: International Conference on Learning Representations (ICLR) (2015), arXiv: 1412.6980
21. Ljung, L.: System Identification: Theory for the User, PTR Prentice Hall Information and System Sciences Series. Prentice Hall, New Jersey (1999)
22. Lv, C., Liu, Y., Hu, X., Guo, H., Cao, D., Wang, F.: Simultaneous observation of hybrid states for cyber-physical systems: A case study of electric vehicle powertrain. IEEE Transactions on Cybernetics **48**(8), 2357–2367 (Aug 2018).
23. Manna, Z., Pnueli, A.: Verifying hybrid systems. In: Hybrid Systems (1992)
24. Meinke, K.: Learning-based testing of cyber-physical systems-of-systems: A platooning study. In: EPEW (2017)
25. Meinke, K.: Learning-based testing: Recent progress and future prospects. In: Bennaceur et al. [6], pp. 53–73. , https://doi.org/10.1007/978-3-319-96562-8_2
26. O'Shea, T.J., Hoydis, J.: An introduction to deep learning for the physical layer. IEEE Trans. Cogn. Comm. & Networking **3**(4), 563–575 (2017). , https://doi.org/10.1109/TCCN.2017.2758370
27. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. Journal of Automata, Languages and Combinatorics **7**(2), 225–246 (2002). , https://doi.org/10.25596/jalc-2002-225
28. Ploeg, J., Shukla, D.P., van de Wouw, N., Nijmeijer, H.: Controller synthesis for string stability of vehicle platoons. IEEE Transactions on Intelligent Transportation Systems **15**(2), 854–865 (April 2014).
29. Punjani, A., Abbeel, P.: Deep learning helicopter dynamics models. In: IEEE International Conference on Robotics and Automation, ICRA 2015, Seattle, WA, USA, 26-30 May, 2015. pp. 3223–3230. IEEE (2015). , https://doi.org/10.1109/ICRA.2015.7139643
30. Rashid, A., Siddique, U., Hasan, O.: Formal verification of platoon control strategies. In: SEFM (2018)
31. Rupp, A., Steinberger, M., Horn, M.: Sliding mode based platooning with non-zero initial spacing errors. IEEE Control Systems Letters **1**(2), 274–279 (Oct 2017).
32. Shahbaz, M., Groz, R.: Inferring Mealy machines. In: FM (2009)
33. Simpson, T., Booker, A., Ghosh, D., Giunta, A., Koch, P., Yang, R.J.: Approximation methods in multidisciplinary analysis and optimization: a panel discussion. Structural and Multidisciplinary Optimization **27**(5), 302–313 (2004). , https://doi.org/10.1007/s00158-004-0389-9
34. Smeenk, W., Moerman, J., Vaandrager, F.W., Jansen, D.N.: Applying automata learning to embedded control software. In: ICFEM (2015)
35. Spielberg, S., Gopaluni, R.B., Loewen, P.D.: Deep reinforcement learning approaches for process control. 2017 6th International Symposium on Advanced Control of Industrial Processes (AdCONIP) pp. 201–206 (2017)
36. Tanwani, A., Shim, H., Liberzon, D.: Observability for switched linear systems: Characterization and observer design. IEEE Transactions on Automatic Control **58**(4), 891–904 (apr 2013).
37. Vaandrager, F.W.: Model learning. Commun. ACM (2017)
38. Vasilevskii, M.P.: Failure diagnosis of automata. Cybernetics **9**(4), 653–665 (1973).
39. Vidal, R., Ma, Y., Sastry, S.S.: Hybrid system identification. In: Interdisciplinary Applied Mathematics, pp. 431–451. Springer New York (2016).