Time to Learn – Learning Timed Automata from Tests

Martin Tappler[1(✉)], Bernhard K. Aichernig[1], Kim Guldstrand Larsen[2], and Florian Lorber[2]

[1] Institute of Software Technology, Graz University of Technology, Graz, Austria
{martin.tappler,aichernig}@ist.tugraz.at
[2] Department of Computer Science, Aalborg University, Aalborg, Denmark
{kgl,florber}@cs.aau.dk

Abstract. Model learning has gained increasing interest in recent years. It derives behavioural models from test data of black-box systems. The main advantage offered by such techniques is that they enable model-based analysis without access to the internals of a system. Applications range from fully automated testing over model checking to system understanding. Current work focuses on learning variations of finite state machines. However, most techniques consider discrete time. In this paper, we present a novel method for learning timed automata, finite state machines extended with real-valued clocks. The learning method generates a model consistent with a set of timed traces collected via testing. This generation is based on genetic programming, a search-based technique for automatic program creation. We evaluate our approach on **44** timed systems, comprised of four systems from the literature (two industrial and two academic) and **40** randomly generated examples.

1 Introduction

Test-based model-learning techniques have gained increasing interest in recent years. Basically, these techniques derive formal system models from (test) observations. They therefore enable model-based reasoning about software systems while requiring only limited knowledge about the system at hand. Put differently, such techniques allow for model-based verification of black-box systems if they are amenable to testing.

Peled et al. [39] performed pioneering work in this area by introducing *Black Box Checking*, automata-based model checking for black-box systems. It involves interleaved model learning, model checking and conformance testing and built the basis for various follow-up works [11,17], including model checking of network protocols [12] and differential testing on the model level [7,43]. The framework we target is shown in Fig. 1. In the simplest case,

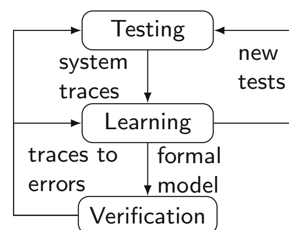


Fig. 1. General framework: test-based learning for verification.

© Springer Nature Switzerland AG 2019
É. André and M. Stoelinga (Eds.): FORMATS 2019, LNCS 11750, pp. 216–235, 2019.
https://doi.org/10.1007/978-3-030-29662-9_13

we interact with a system by testing, learn a model from system traces (logs) and then perform verification. Feedback loops are also possible: we can derive additional tests from the preliminary learned model, and we could use counterexample traces from model checking as tests.

Learning-based verification has great potential, but applications often use modelling formalisms with low expressiveness such as Mealy machines. This can be attributed to the availability of efficient implementations of learning algorithms for variations of finite automata, e.g. in LearnLib [22], and comparably low support for richer automata types; especially timed systems have received little attention. In addition, many of the proposed methods are not supported by implementations. Notable works include learning of deterministic real-time automata and a probabilistic variant thereof [46,47] by Verwer et al. and techniques for learning deterministic event-recording automata described by Grinchtein et al. [15,16]. The existing solutions on timed automata contain several limitations. Real-time automata are restricted to one clock, which is reset in every transition. Thus, they can only reason about delays in the current location, but not keep track of delays since earlier events. Event-recording automata are also less expressive than timed automata, and their learning has a high runtime complexity. To the best of our knowledge, the proposed solution is the first to implement a learning technique for general input-output timed automata, as used in the model-checker Uppaal [8]. Such automata, usually do not have canonical forms [15] which complicates the development of learning techniques, therefore we follow a metaheuristic approach. The only restrictions required by our approach are that it considers the systems under learning to be output-urgent, deterministic and with isolated outputs (a special form of output determinism). While specifications are generally vague, especially on timing, and leave freedom to the actual implementation, implementations themselves are generally considered to reflect one specific choice satisfying the specification, and implement it deterministically [10]. Thus, since we learn from concrete implementations, we do not see these restrictions as too limiting. Notably, we consider input-enabled systems which makes our approach well-suited for a testing-based setting.

Scope and Contribution. Here, we focus on the *learning* part in Fig. 1. Generally, model learning may be performed either passively or actively [20]. Passive learning uses preexisting data, such as system logs or existing test data, as basis, while active learning actively queries the system, e.g. by testing, to gain relevant information. We use a form of genetic programming [25] to passively learn a deterministic timed automaton (TA) consistent with a given set of test cases. We evaluate this approach, a meta-heuristic search, on four manually created TA and several randomly generated TA. The evaluation demonstrates that the search reliably converges to a TA consistent with test cases given as training data. Furthermore, we simulate learned TA on independently produced test data to show that our identified solutions generalise well, thus do not overfit to training data. Our technique is passive, but active extensions are possible by testing based on intermediate versions of the learned model. Such an active approach is currently under development and first evaluations show promising results.

Our contribution is threefold: (1) We show that TA can be genetically programmed and present the corresponding parameters and techniques. (2) We implemented these techniques in a tool available for download [45]. (3) The evaluation results may serve as a benchmark for alternative TA learning methods.

Structure. Section 2.1 contains background information on TA and genetic programming. Section 3 describes our approach to learning TA. Applications of this approach are presented in Sect. 4. In Sect. 5, we provide a summary and discuss related work, as well as potential extensions.

2 Preliminaries

2.1 Timed Automata

TA are finite automata enriched with real-valued variables called clocks [6]. Clocks measure the progress of time which elapses while an automaton resides in some location. Transitions can be constrained based on clock values and clocks may be reset on transitions. We denote the set of clocks by $\mathcal{C}$ and the set of guards over $\mathcal{C}$ by $\mathcal{G}(\mathcal{C})$. Guards are conjunctions of constraints of the form $c \oplus k$, with $c \in \mathcal{C}, \oplus \in \{>, \geq, \leq, <\}, k \in \mathbb{N}$. Transitions are labelled by input and output actions, denoted by Σ_I and Σ_O respectively, with $\Sigma = \Sigma_I \cup \Sigma_O$. Input labels are suffixed by ? and output labels end with !. A TA over $(\mathcal{C}, \Sigma)$ is a triple $\langle L, l_0, E\rangle$, where L is a finite non-empty set of locations, $l_0 \in L$ is the initial location and E is the set of edges, with $E \subseteq L \times \Sigma \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$. We write $l \xrightarrow{g,a,r} l'$ for an edge $(l, g, a, r, l') \in E$ with guard g, label a, and clock resets r.

Example 1 (Train TA Model). Figure 2 shows a TA model of a train, for which we have $\Sigma_I = \{start?, stop?, go?\}$, $\Sigma_O = \{appr!, enter!, leave!\}$, $\mathcal{C} = \{c\}$, $L = \{l_0, \ldots, l_5\}$, and $E = \{l_0 \xrightarrow{\top, start?, \{c\}} l_1, \ldots\}$. From initial location l_0, the train accepts the input *start*?, resetting clock c. After that, it can produce the output *appr*! if $c \geq 5$, i.e the train may approach 5 time units after it is started.

The semantics of a TA is given by a timed transition system (TTS) $\langle Q, q_0, \Sigma, T\rangle$, with states $Q = L \times \mathbb{R}_{\geq 0}{}^{\mathcal{C}}$, initial state q_0, and transitions $T \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$, for which we write $q \xrightarrow{e} q'$ for $(q, e, q') \in T$. A state $q = (l, \nu)$ is a pair consisting of a location l and a clock valuation ν. For $r \subseteq \mathcal{C}$, we denote resets of clocks in r by $\nu[r]$, i.e. $\forall c \in r : \nu[r](c) = 0$ and $\forall c \in \mathcal{C} \backslash r : \nu[r](c) = \nu(c)$. Let $(\nu+d)(c) = \nu(c)+d$ for $d \in \mathbb{R}_{\geq 0}, c \in \mathcal{C}$ denote the progress of time and $\nu \models \phi$ denote that valuation ν satisfies formula ϕ. Finally, $\mathbf{0}$ is the valuation assigning zero to all clocks and the initial state q_0 is $(l_0, \mathbf{0})$. Transitions of TTSs are either delay transitions $(l, \nu) \xrightarrow{d} (l, \nu + d)$ for a delay $d \in \mathbb{R}_{\geq 0}$, or discrete transitions $(l, \nu) \xrightarrow{a} (l', \nu[r])$ for an edge $l \xrightarrow{g,a,r} l'$

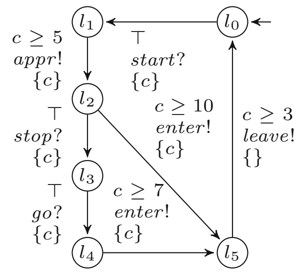


Fig. 2. Train TA.

such that $\nu \models g$. Delays are usually further constrained, e.g. by invariants [19] limiting the sojourn time in locations.

Timed Traces. We use the terms timed traces and test sequences similarly to [41]. The latter are sequences of inputs and corresponding execution times, while the former are sequences of inputs and outputs, together with their times of occurrence (produced in response to a test sequence). A test sequence ts is an alternating sequence of non-decreasing time stamps t_j and inputs i_j, i.e. $ts = t_1 \cdot i_1 \cdots t_n \cdot i_n \in (\mathbb{R}_{\geq 0} \times \Sigma_I)^*$ with $\forall j \in \{1, \ldots, n-1\} : t_j \leq t_{j+1}$. Informally, a test sequence prescribes that i_j should be executed at time t_j. A timed trace $tt \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ consists of inputs interleaved with outputs produced by a timed system. Analogously to test sequences, its timestamps are non-decreasing.

Assumptions on Timed Systems. Testing based on TA often places further assumptions on TA [19,41]. Since we learn models from tests we make similar assumptions (closely following [19]). We describe these assumptions on the level of semantics and use $q \xrightarrow{a}$ to denote $\exists q' : q \xrightarrow{a} q'$ and $q \not\xrightarrow{a}$ for $\nexists q' : q \xrightarrow{a} q'$:

1. *Determinism.* A TA is deterministic iff for every state $s = (l, \nu)$ and every action $a \in \Sigma$, whenever $s \xrightarrow{a} s'$, and $s \xrightarrow{a} s''$ then $s' = s''$.
2. *Input Enabledness.* A TA is input enabled iff for every state $s = (l, \nu)$ and every input $i \in \Sigma_I$, we have $s \xrightarrow{i}$.
3. *Output Urgency.* A TA shows output-urgent behaviour if outputs occur immediately as soon as they are enabled, i.e. for $o \in \Sigma_O$, if $s \xrightarrow{o}$ then $s \not\xrightarrow{d}$ for all $d \in \mathbb{R}_{\geq 0}$. Thus, outputs must not be delayed.
4. *Isolated Outputs.* A TA has isolated outputs iff whenever an output may be executed, then no other output is enabled, i.e. if $\forall o \in \Sigma_O, \forall o' \in \Sigma_O : q \xrightarrow{o}$ and $q \xrightarrow{o'}$ implies $o = o'$.

It is necessary to place restrictions on the sojourn time in locations to establish output urgency. Deadlines provide a simple way to model the assumption that systems are output urgent [9]. With deadlines it is possible to model eager actions. We use this concept and implicitly assume all learned output edges to be eager. This means that outputs must be produced as soon as their guards are satisfied. For that, we extend the semantics given above by adding the following restriction: delays $(l, \nu) \xrightarrow{d} (l', \nu + d)$ are only possible if $\forall\, d' \in \mathbb{R}_{\geq 0}, d' < d :$ $\nu + d' \models \neg \bigvee_{g \in G_O} g$, where $G_O = \{g | \exists l', a, r : l \xrightarrow{g,a,r} l', a \in \Sigma_O\}$ are the guards of outputs in location l. To avoid issues related to the exact time at which outputs should be produced, we further restrict the syntax of TA by disallowing strict lower bounds for output edges. Uppaal [27] uses invariants rather than deadlines to limit sojourn time. In order to analyse TA using Uppaal, we use the translation given in [14]. We implicitly add self-loops to all states $s = (l, \nu)$ for inputs i undefined in s, i.e. we add $(l, \nu) \xrightarrow{i} (l, \nu)$ if $\nu \not\models \bigvee_{\exists l', r : l \xrightarrow{g,i,r} l'} g$. This ensures input enabledness while avoiding TA cluttered with input self-loops. It also allows to ignore input enabledness during genetic programming, e.g. mutations may remove input edges.

The assumptions placed on systems under test (SUTs) ensure testability [19]. Assuming that SUTs can be modelled in some modelling formalism is usually referred to as *testing hypothesis*. Placing the same assumptions on learned models simplifies checking conformance between model and SUT. The execution of a test sequence on such a model uniquely determines a response [41], and due to input enabledness we may execute any test sequence. This allows us to use equivalence as conformance relation between learned models and SUT. What is more, we can approximate checking equivalence between the learned models and the SUT by executing test sequences on the models and check for equivalence between the SUT's responses and the response predicted by the models.

2.2 Genetic Programming

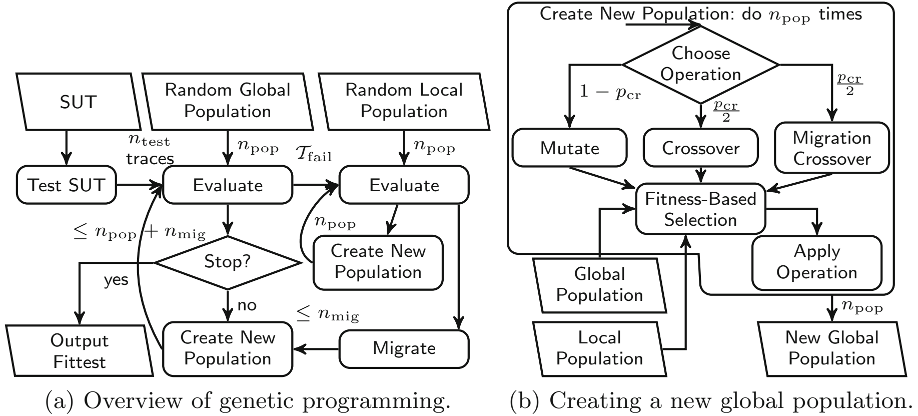
Genetic programming [25] is a search-based technique to automatically generate programs exhibiting some desired behaviour. Like *Genetic Algorithms* [35], it is inspired by nature. Programs, also called individuals, are iteratively refined by: (1) fitness-based selection followed by (2) operations altering program structure, like mutation and crossover. Fitness measures are problem-specific and may for instance be based on tests. In this case, one could assign a fitness value proportional to the number of tests passed by an individual. The following basic functioning principle underlies genetic programming.

1. Randomly create an initial population.
2. Evaluate the fitness of each individual in the population.
3. If an acceptable solution has been found or the maximum number of iterations has been performed: **stop** and output the best individual
4. Otherwise repeatedly select an individual based on fitness and apply one of:
   Mutation: change a part of the individual creating a new individual.
   Crossover: select another individual according to its fitness and combine both individuals to create offspring.
   Reproduction: copy the individual to create a new equivalent individual.
5. Form a new population from the new individuals and go to Step 2.

Due to their nature, genetic algorithms and genetic programming lend themselves to parallelisation. Several populations may, e.g., be evolved in parallel, which is particularly useful if speciation is applied [37]. In speciation, different subpopulations explore different parts of the search space. Information is commonly exchanged between subpopulations by migrating individuals.

3 Genetic Programming for Timed Automata

Figure 3a provides an overview of the steps we perform, while Fig. 3b shows the creation of a new population in more detail. We first test the SUT, by generating and executing n_{test} test sequences to collect n_{test} timed traces. Our goal is then to genetically program a TA consistent with the collected timed traces. Put differently, we want to generate a TA that produces the same outputs as the SUT in



(a) Overview of genetic programming. (b) Creating a new global population.

Fig. 3. Overview of the learning process.

response to the inputs of the test sequences. For the following discussion, we say that a TA passes a timed trace t if it produces the same outputs as the SUT when simulating the test sequence corresponding to t. Otherwise it fails t. In addition to passing all timed traces, the final TA shall be deterministic. This is achieved by assigning larger fitness values to deterministic solutions. Both mutation and crossover can create non-deterministic intermediate solutions, which might help the search in the short-term and will be resolved in future generations.

Generally, we evolve two populations of TA simultaneously, a global population, evaluated on all the traces, and a local population, evaluated only on the traces that fail on the fittest automaton of the global population. Both are initially created equally and contain n_{pop} TA. After initial creation, the global population is evaluated on all n_{test} traces. During that, we basically test the TA and check how many traces each TA passes and assign fitness values accordingly, i.e., the more passing traces the fitter. Additionally, we add a fitness penalty for model size. The local population is evaluated only on a subset $\mathcal{T}_{\text{fail}}$ of the traces. This subset $\mathcal{T}_{\text{fail}}$ contains all traces which the fittest TA fails, and which likely most of the other TA fail as well. With the local population, we are able to explore new parts of the search space more easily since we may ignore functionality already modelled by the global population. We integrate functionality found via this local search into the global population through migration and migration combined with crossover. To avoid overfitting to a low number of traces, we ensure that $\mathcal{T}_{\text{fail}}$ contains at least $\frac{n_{\text{test}}}{100}$ traces. If there are fewer actually failing traces, we add randomly chosen traces from all n_{test} traces to $\mathcal{T}_{\text{fail}}$.

After evaluation, we stop if we either reached the maximum number of generations g_{max}, or the fittest TA passes all traces and has not changed in g_{change} generations. Note that two TA passing all traces may have different fitness values depending on model size, i.e. g_{change} controls how long we try to decrease the size of the fittest TA. The rationale behind this is that smaller TA are less complex and simpler to comprehend.

If not stopped, we create new populations of TA, which works slightly differently for the local and the global population. Figure 3b illustrates the creation of a new global population. Before creating new TA, existing TA may migrate from the local to the global population. For that, we check each of the fittest n_{mig} local TA and add it to the global population if it passes at least one trace from $\mathcal{T}_{\mathrm{fail}}$. We generally set n_{mig} to $\frac{5n_{\mathrm{pop}}}{100}$, i.e. the top five percent of the local population are allowed to migrate. After migration, we create n_{pop} new TA through the application of one of three operations:

- *with probability* $1 - p_{\mathrm{cr}}$: mutate a TA from the global population
- *with probability* $\frac{p_{\mathrm{cr}}}{2}$: crossover of two TA from the global population
- *with probability* $\frac{p_{\mathrm{cr}}}{2}$: crossover of two TA, one from each population

The rationale behind migration combined with crossover is that migrated TA may have low fitness from a global point of view and will therefore not survive selection. They may, however, have desirable features which can be transferred via crossover. For the local population, we perform the same steps, but without any migration, in order to keep the local search independent. Once we have new populations, we start a new generation by evaluating the new TA.

A detail not illustrated in Fig. 3a is our implementation of *elitism* [35]. We always keep track of the fittest TA found so far for both populations. In each generation, we add these fit TA to their respective populations after mutation.

Parameters. Our implementation could be controlled by a large number of parameters. To ease applicability and to avoid the need for meta-optimisation of parameter settings for a particular SUT, we fixed as many as possible to constant values. The actual values, like $\frac{5n_{\mathrm{pop}}}{100}$ for n_{mig}, are motivated by experiments. The remaining parameters can usually be set to default values or chosen based on guidelines. For instance, n_{pop}, g_{max}, and n_{test} may be chosen as large as possible, given available memory and maximum computation time.

3.1 Creation of Initial Random Population

We initially create n_{pop} random TA, parameterised by: (1) the labels Σ_I and Σ_O, (2) the number of clocks n_{clock}, and (3) the appr. largest constant in clock constraints c_{max}. Note, c_{max} is an approximation, because mutations may increase constants. Each TA has initially only two locations, as we intend to increase size and thereby complexity only through mutation and crossover. Moreover, it is assigned the given action labels and has a set of n_{clock} clocks. During creation, we add random edges, such that at least one edge connects the initial location to the other location. We create edges entirely randomly, whereby the number of constraints in guards as well as the number of clock resets are geometrically distributed with fixed parameters. The edge label, the relational operators and constants in constraints are chosen uniformly at random from the respective sets Σ, $\{<, \leq, \geq, >\}$, and $[0\,..\,c_{\mathrm{max}}]$ (operators for outputs exclude $>$). The source and target locations are also chosen uniformly at random from the set of locations, i.e. initially we choose from two locations. If the required number of clocks

is not known a priori, we suggest setting $n_{\text{clock}} = 1$ and increasing it only if it is not possible to find a valid TA. A similar approach could be used for $c_{\max}$.

3.2 Fitness Evaluation

Simulation. We simulate the TA to evaluate their fitness. Above, we discussed failing and passing traces, but evaluation is more fine grained. We execute the inputs of each timed trace and observe produced outputs until (1) the simulation is complete, (2) an expected output is not observed, or (3) output isolation is violated (output non-determinism).

In general, if $\mathcal{T}$ is a deterministic, input-enabled TA with isolated and urgent outputs and *ts* is a test sequence, then executing *ts* on $\mathcal{T}$ uniquely determines a timed trace *tt* [41]. By the testing hypothesis, the SUT fulfils these assumptions. However, TA generated through mutation and crossover are input-enabled, but may show non-deterministic behaviour. Hence, simulating a test sequence or a timed trace on a generated TA may follow multiple paths of states. Some of these paths may produce the expected outputs and some may not. Our goal is to find a TA that is both correct, i.e. produces the same outputs as the SUT, and is deterministic. Consequently, we reward these properties with positive fitness.

The simulation function $\textsc{sim}(\mathcal{G}, tt)$ simulates a timed trace *tt* on a generated TA $\mathcal{G}$ and returns a set of timed traces. It uses the TTS semantics but does not treat outputs as urgent outputs. From the initial state $(l_0, \mathbf{0})$, where l_0 is the initial location of $\mathcal{G}$, it performs the following steps for each $t_i e_i \in tt$ with $t_0 = 0$:

1. From state $q = (l, \nu)$
2. Delay for $d = t_i - t_{i-1}$ to reach $q^d = (l, \nu + d)$
3. If $e_i \in \Sigma_I$, i.e. it is an input:
   - 3.1. If $\exists o \in \Sigma_O, d^o \leq d : (l, \nu + d^o) \xrightarrow{o}$, i.e. an output would have been possible while delaying or at time t_i
     $\rightarrow$ then mark e_i
   - 3.2. If $\exists q^1, q^2, q^1 \neq q^2 : q^d \xrightarrow{e_i} q^1 \wedge q^d \xrightarrow{e_i} q^2$
     $\rightarrow$ then mark e_i
   - 3.3. For all q' such that $q^d \xrightarrow{e_i} q'$
     $\rightarrow$ carry on exploration with q'
4. If $e_i \in \Sigma_O$, i.e. it is an output:
   - 4.1. If $\exists o \in \Sigma_O, d^o < d : (l, \nu + d^o) \xrightarrow{o}$, i.e. an output would have been possible while delaying
     $\rightarrow$ stop exploration
   - 4.2. If $\exists q^1, q^2, q^1 \neq q^2 : q^d \xrightarrow{e_i} q^1 \wedge q^d \xrightarrow{e_i} q^2$ or $\exists o, o \neq e_i : q^d \xrightarrow{o}$
     $\rightarrow$ stop exploration
   - 4.3. If there is a q' such that $q^d \xrightarrow{e_i} q'$
     $\rightarrow$ carry on exploration with q'

The procedure shown above allows for two types of non-determinism. During delays before executing an input, we may ignore outputs (3.1) and we may explore multiple paths with inputs (3.3). We mark these inputs to be non-deterministic, through (3.1 and 3.2). Since we explore multiple paths, a single

input e_i may be marked along one path but not marked along another path. In contrast, we do not explore non-deterministic outputs, leading to lower fitness for respective traces. This avoids issues with trivial TA which produce each output all the time. Such TA would completely simulate all traces non-deterministically, but would not be useful.

During exploration, $\textsc{sim}(\mathcal{G}, tt)$ collects and returns timed traces tts, which are prefixes of tt but with marked and unmarked inputs. For fitness computation, we defined four auxiliary functions. The first one assigns a simulation verdict:

$$\textsc{verdict}(tts) = \begin{cases} \texttt{PASS} & \text{if } |tts| = 1 \land tt \in tts \\ \texttt{NONDET} & \text{if } |tts| > 1 \land \exists tt' \in tts : |tt'| = |tt| \\ \texttt{FAIL} & \text{otherwise} \end{cases}$$

A TA, which produces a `PASS` verdict for all timed traces, behaves equivalently to the SUT for these traces. `NONDET` is returned in case of non-determinism with at least one correct execution path. Function $\textsc{steps}(tts)$ returns the maximum number of deterministic steps, and $\textsc{out}(tts)$ returns the number of outputs along the longest traces in tts. Finally, $\textsc{size}(\mathcal{G})$ returns the number of edges.

Fitness Computation. In order to compute the fitness, we assign the weights $w_{\texttt{PASS}}$, $w_{\texttt{NONDET}}$, $w_{\texttt{FAIL}}$, w_{STEPS}, w_{OUT}, and w_{SIZE} to the gathered information of $\mathcal{G}$. Basically, we give some positive fitness for deterministic steps, correctly produced outputs, and verdicts, but penalise size. Let $\mathcal{TT}$ be the timed traces on which $\mathcal{G}$ is evaluated. The fitness $\textsc{fit}(\mathcal{G})$ is then (note that $w_{\textsc{verdict}(tts)}$ evaluates to one of $w_{\texttt{PASS}}$, $w_{\texttt{NONDET}}$, or $w_{\texttt{FAIL}}$):

$$\textsc{fit}(\mathcal{G}) = \sum_{tt \in \mathcal{TT}} \textsc{fit}(\mathcal{G}, tt) - w_{\text{SIZE}}\ \text{SIZE}(\mathcal{G}) \quad \text{where}$$

$$\textsc{fit}(\mathcal{G}, tt) = w_{\textsc{verdict}(tts)} + w_{\text{STEPS}}\ \text{STEPS}(tts) + w_{\text{OUT}}\ \text{OUT}(tts) \text{ and } tts = \textsc{sim}(\mathcal{G}, tt)$$

Fitness evaluation adds further parameters. We identified guidelines for choosing them adequately. We generally set $w_{\texttt{FAIL}} = 0$ and use w_{OUT} as basis for other weights. Usually, we set $w_{\text{STEP}} = w_{\text{OUT}}/2$ and $w_{\texttt{PASS}} = k \cdot l \cdot w_{\text{OUT}}$, where l is the average length of test sequences and k is a small natural number, e.g. 4. More important than the exact value of k is setting $w_{\texttt{NONDET}} = w_{\texttt{PASS}}/2$ which gives positive fitness to correctly produced timed traces but with a bias towards deterministic solutions. The weight w_{SIZE} should be chosen low, such that it does not prevent adding of necessary edges. We usually set it to w_{STEP}. It needs to be non-zero, though. Otherwise an acceptable solution could be a tree-shaped automaton exactly representing $\mathcal{TT}$ without generalisation. As noted at the beginning of this section, we assign larger fitness to solutions that accept a larger portion of the traces deterministically, as our goal is to learn deterministic TA.

As noted above, a TA $\mathcal{T}$ producing only `PASS` verdicts behaves equivalently to the SUT with respect to $\mathcal{TT}$, i.e. $\mathcal{T}$ is "approximately trace equivalent" to the SUT. Due to the restriction to deterministic output-urgent systems, trace inclusion and trace equivalence coincide. As a result, a TA producing a `FAIL` verdict is neither an under- nor an over-approximation.

3.3 Creation of New Population

Table 1 lists all implemented mutation operators for TA. Whenever an operator selects an edge or a location, the selection is random, but favours locations and edges which are associated with faults and non-deterministic behaviour. We augment TA with such information during fitness evaluation. To create an edge, we create random guards and reset sets, and choose a random label, like for the initial creation of TA.

Table 1. Mutation operators

Name	Short description
Add constraint	Add a guard constraint to an edge
Change guard	Select edge and create a random guard if the edge does not have a guard, otherwise mutate a constraint of its guard
Change target	Change the target location of an edge
Remove guard	Remove either all or a single guard constraint from an edge
Change resets	Remove clocks from or add clocks to the clock resets of an edge
Remove edge	Remove a selected edge
Add edge	Add an edge connecting randomly chosen locations
Sink location	Add a new location
Merge location	Merge two locations
Split location	Split a location l by creating a new location l' and redirecting an edge reaching l to l'
Add location	Add a new location and two edges connecting the new location to existing locations
Split edge	Replace an edge e with either the sequence $e' \cdot e$ or $e \cdot e'$ where e' is a new random edge (adds a location to connect e and e')

The mutation operators form three groups separated by double horizontal lines. The first and largest group contains basic operators, which are sufficient to create all possible automata. The second group is motivated by the basic principle behind automata learning algorithms. Passive algorithms often start with a tree-shaped representation of traces and transform this representation into an automaton via iterated state-merging [20]. Active learning algorithms on the other hand usually start with a low number of locations and add new locations if necessary. This can be interpreted as splitting of existing locations, an intuition which also served as a basis for test-case generation in active automata learning [5]. The last two operators are motivated by observations during experiments: *add location* increases the automaton size but avoids creating deadlock states, unlike the operator *sink location*. *Split edge* addresses issues related to input enabledness, where an input i is implicitly accepted without changing state, although an edge labelled i should change the state. The operator aims

to introduce such edges. For mutation, we generally select one of the operators uniformly at random.

In addition to mutation, we apply a simplification procedure. It changes the syntactic representation of TA without affecting semantics, by, e.g., removing unreachable locations. For further details regarding simplification, migration, selection and crossover we refer to our technical report [44].

3.4 Implementation

The presented algorithms have been implemented in a tool shown in Fig. 4 and can be found online [45]. The tool supports customisation of almost all relevant parameters. When selecting one of the presented experiments, the tool will propose the same values that were used in the evaluation presented in Sect. 4.

While the tool is general enough to learn from any set of timed traces given in the correct format, the prototype is currently only meant for evaluating the examples presented in this paper. A full release of the tool is planned soon.

The tool implements the presented automatic genetic-programming process, with the possibility to inspect the current status of the search, like the accepted traces by the current population. In case the search gets stuck, the tool also allows for manual changes to be performed, enabling semi-automatic modelling.

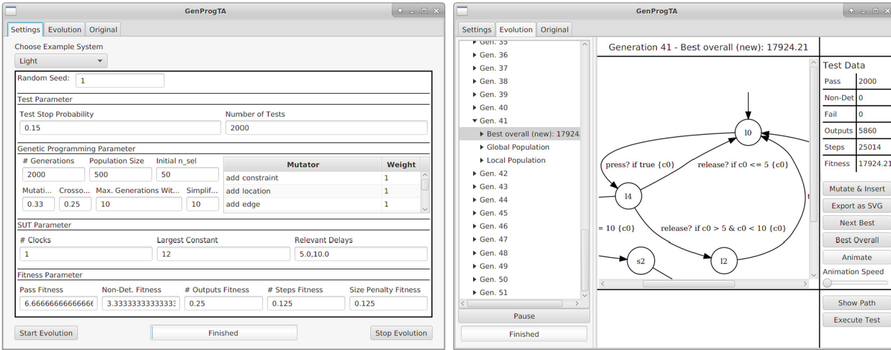


Fig. 4. Two screenshots of the genetic programming tool for time automata, illustrating the possible configurations (left) and the screen to view intermediate results (right).

4 Case Studies

Our evaluation is based on four manually created and 40 randomly generated TA, which serve as our SUTs. Using TA provides us with an easy way of checking whether we found the correct model. However, our approach and our tool are general enough to work on real black-box implementations. Our algorithms are implemented in Java. A demonstrator with a GUI is available in the supplementary material, which also includes Graphviz dot-files of the TA [45]. The demonstrator allows repeating all experiments presented in the following with

freely configurable parameters. Moreover, the search progress can be inspected anytime. The user interface lists the fittest TA for each generation and visualises each of them along with the timed traces used for learning.

For the evaluation, we generated timed traces by simulating n_{test} random test sequences on the SUTs. The inputs in the test sequences were selected uniformly at random from the available inputs. The lengths of the test sequences are geometrically distributed with a parameter p_{test}, which is set to 0.15 unless otherwise noted. To avoid trivial timed traces, we ensure that all test sequences cause at least one output to be produced. The delays in test sequences were chosen probabilistically in accordance with the user-specified largest constant $c_{\max}$. Additionally, one could specify important constants used in the SUTs, gathered from a requirements document if available. Specifying appropriate delays helps to ensure that the SUTs are covered sufficiently well by the test sequences.

Measurement Setup and Criteria. The measurements were done on a notebook with 16 GB RAM and an Intel Core i7-5600U CPU operating at 2.6 GHz. Our main goal is to show that we can learn models in a reasonable amount of time, but further improvements are possible, e.g., via parallelisation. We use a training set and a test set for evaluation, each containing n_{test} timed traces. First, we learn from the training set until we find a TA which produces a `PASS` verdict for all traces. Then, we simulate the traces from the test set and report all traces leading to a verdict other than `PASS` as erroneous. Note that since we generate the test set traces through testing, there are no negative traces. In other words, all traces are observable and can be considered positive. Consequently, notions like precision and recall do not apply to our setting.

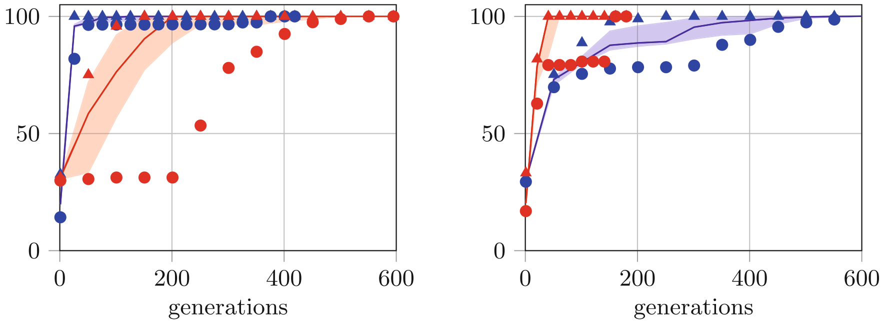
Our four manually created TA, with number of locations and $c_{\max}$ in parentheses, are called car alarm system (CAS) $(14, 30)$, Train $(6, 10)$, Light $(5, 10)$, and particle counter (PC) $(26, 10)$. All of them use one clock. The CAS is an industrial case study, which served as a benchmark for test-case generation for timed systems [3]. Different versions of the Train and Light have been used as examples in real-time verification [8] and variants of them are distributed as demo examples with the real-time model-checker Uppaal [27] and the real-time testing tool Uppaal Tron [18]. The particle counter (PC) is the second industrial case study. Untimed versions of it were examined in model-based testing [2].

In addition to the manually created timed systems, we have four categories of random TA, each containing ten TA: C15/1, C20/1, C6/2, C10/2, where the first number gives the number of locations and the second the number of clocks. TA from the first two categories have alphabets containing 5 distinct inputs and 5 distinct outputs, while the TA from the other two categories have 4 inputs and 4 outputs. For all random TA, we have $c_{\max} = 15$.

We used similar configurations for all experiments. Following the suggestions in Sect. 3, we set the fitness weights to $w_{\text{OUT}} = 0.25$, $w_{\text{STEPS}} = \frac{w_{\text{OUT}}}{2} = w_{\text{SIZE}}$, $w_{\texttt{PASS}} = \frac{4 w_{\text{OUT}}}{p_{\text{test}}}$, $w_{\texttt{NONDET}} = \frac{\texttt{PASS}}{2}$, and $w_{\texttt{FAIL}} = 0$, with the exception of CAS. Since the search frequently got trapped in minima with non-deterministic behaviour, we set $w_{\text{OUT}} = \frac{w_{\text{STEPS}}}{2}$, i.e. we valued deterministic steps more than outputs, and $w_{\texttt{NONDET}} = -0.5$, i.e. we added a small penalty for non-determinism. Other than

Table 2. Measurement results

TA	Test set errors	Generations	Time
CAS	0	147/246.0/305.8/595	27.3 min/57.2 min/1.2 h/2.7 h
Train	0	50/71.0/83.4/180	2.9 min/4.7 min/4.8 min/9.1 min
Light	0	42/77.5/84.5/240	3.2 min/7.4 min/8.7 min/31.1 min
PC	0	278/685.5/554.9/859	3.0 h/8.7 h/7.3 h/10.6 h
C15/1	0/2.0/1.8/6	201/404.5/401.3/746	1.4 h/3.1 h/3.2 h/6.6 h
C20/1	0/0.0/1.0/6	45/451.0/665.8/1798	23.4 min/6.7 h/7.4 h/18.3 h
C6/2	0/0.0/0.5/3	18/68.5/176.9/709	9.4 min/43.9 min/1.8 h/7.6 h
C10/2	0/2.5/2.6/8	73/239.0/344.9/984	35.8 min/3.1 h/3.4 h/9.3 h



(a) Percentages for the Light (blue) and the CAS (red).

(b) Percentages for the PC (blue) and the Train (red).

Fig. 5. Percentages of accepted test steps of fittest individual. (Color figure online)

that, we set $g_{\mathrm{max}} = 3000$, $n_{\mathrm{pop}} = 2000$, the initial $n_{\mathrm{sel}} = \frac{n_{\mathrm{pop}}}{10}$, $n_{\mathrm{test}} = 2000$, $p_{\mathrm{cr}} = 0.25$, $g_{\mathrm{change}} = 10$, $p_{\mathrm{mutinit}} = 0.33$, and $g_{\mathrm{simp}} = 10$, with the following exceptions. Train and Light require less effort, thus we set $n_{\mathrm{pop}} = 500$. The categories C10/2, C15/1, and C20/1 require more thorough testing, so we configured $n_{\mathrm{test}} = 4000$ for C10/2 and C15/1, and $n_{\mathrm{test}} = 6000$ with $p_{\mathrm{test}} = 0.1$ for C20/1. We determined the settings for n_{test} experimentally, by manually inspecting if the intermediate learned TA were approximately equivalent to the true models, so as to ensure that the training sets adequately cover the relevant behaviour.

All learning runs were successful by finding a TA without errors on the training set, except for two cases, one in C10/2 and one in C20/1. For the first, we repeated the experiment with a larger population $n_{\mathrm{pop}} = 6000$, resulting in successful learning. For the random TA in C20/1, we observed a similar issue as for CAS, i.e. non-determinism was an issue, but used another solution. In some cases, crossover may introduce non-determinism, thus we decreased the probability for crossover p_{cr} to 0.05 and learned the correct model. Hence, we are able to learn TA that are consistent with given trace data via genetic programming.

Table 2 shows the learning results. The column *test set error* contains 0, if there were no errors on the test set. Otherwise, each cell in the table contains,

from left to right, the minimum, the median and the mean, and the maximum computed over 10 runs for manually created TA and over 10 runs for each random category, i.e. one run per random TA.

Figure 5a and b illustrate the percentage of correct steps when simulating the test cases on the intermediate learned models. The solid line represents the median out of the 10 runs, the dots represent the minimum, the triangles the maximum and the coloured area is the area between first and third quartile. One can see a steep rise in the early generations, while later generations are mainly needed to minimise the learned models, which already correctly incorporate all test steps. The CAS is the model with the slowest initial learning, where, in the worst case, the first 200 generations did not improve the model.

The test set errors are generally low, so our approach generalises well and does not simply overfit to the training data. We also see that manually created systems produced no test set errors. While the more complex, random TA led to errors. However, for them the relative number of errors was at most two thousandths (8 errors out of 4000 tests). Such errors may, e.g., be caused by slightly too loose or too strict guards on inputs.

The computation time of at most 18.3 h seems acceptable, especially considering that fitness evaluation, as the most time-consuming part, is parallelisable. Finally, we want to emphasise that we identified parameters which almost consistently produced good results. In the exceptions where this was not the case, it was simple to adapt the configuration.

The size of our TA in terms of number of locations ranges between 5 and 26. To model real-world systems, it is therefore necessary to apply abstraction during the testing phase, which collects timed traces. Since model learning requires thorough testing, abstraction is commonly used in this area. Consequently, this requirement is not a strong limitation. Several applications of automata learning show that implementation flaws can be detected by analysing learned abstract models, e.g., in protocol implementations [12,40,43].

In conclusion, we have shown that we can learn models that are consistent with given training data and that these models generalise to test data that is produced equally, but which does not overlap with the training data. Since we learn from randomly generated data in our experiments, the learned models may not be equivalent to true underlying models. However, a manual inspection revealed that we generally learned correct models, with the exception of slight discrepancies in behaviour in some cases. We are currently working on extending our work to actively search for counterexamples to equivalence which could potentially provide stronger guarantees (see also our discussion on future work).

5 Conclusion

Summary. We presented an approach to learn deterministic TA with urgent outputs, an important subclass for testing timed systems [19]. The learned models may reveal flaws during manual inspection and enable verification of black-box systems via model-checking. Genetic programming serves as a basis. In our

implementation of it, we parallelise search by evolving two populations simultaneously and developed techniques for mutation, crossover, and for a fine-grained fitness-evaluation. While, due to the heuristic nature of the proposed method, we cannot provide a convergence proof, we provide empirical evidence that the method performs well, can cope with big state spaces and generally converges to a solution consistent with given trace data. We evaluated the technique on non-trivial TA with up to 26 locations. We could learn all 44 TA models, only two random TA needed a small parameter adjustment.

Related Work. Verwer et al. [46,47] passively learned real-time automata via state-merging. These TA measure the time between two consecutive events and use guards in the form of intervals, i.e. they have a single clock which is reset on every transition. They do not distinguish between inputs and outputs. Improvements of [47] were presented in [34]. Similarly, Mao et al. applied state-merging to learn continuous time Markov chains [32]. A state-merging-based learning algorithm for more general stochastic timed systems has been proposed by de Matos Pedro et al. [33]. They target learning generalised semi-Markov processes, which are generated by stochastic timed automata. All these techniques have in common that they consider systems where the relation between events is fully described by a system's structure. Pastore et al. [38] learn specifications capturing the duration of (nested) operations in software systems. A timed trace therefore includes for each operation its start and end, i.e. the trace records two related events. Their algorithm is based on the passive learning technique *k-Tail*.

Grinchtein et al. [15,16] described active learning approaches for deterministic event-recording automata, a subclass of TA with one clock per action. The clock corresponding to an action is reset upon its execution essentially recording the time since the action has occurred. While the expressiveness of these automata suffices for many applications, the runtime complexity of the described techniques is high and may be prohibitive in practice. Currently, there is no implementation to actually measure runtime. Furthermore, this kind of TA cannot model certain timing patterns, e.g., in the case of input enabledness where always resetting a clock may not be appropriate. Lin et al. [29] also presented an active learning algorithm for event-recording automata and applied it to learn assumptions in compositional verification via assume-guarantee reasoning [30].

Meta-heuristic search as an alternative to classical automata learning has been proposed by Lai et al. for finite state machines [26]. They apply genetic algorithms and assume the number of states to be known. Lucas and Reynolds compared state-merging and evolutionary algorithms, but also fixed the number of states for runs of the latter [31].

Lefticaru et al. similarly assume the number of states to be known and generate state machine models via genetic algorithms [28]. Their goal, however, is to synthesise a model satisfying a specification given in temporal logics. Early work suggesting such an approach was performed by Johnson [23], which like our approach does not require the solution size to be known. In contrast, Johnson does not apply crossover. Further synthesis work from Katz and Peled [24] tries to infer a correct program or model on the source code level, while we aim at syn-

thesising a model representing a black-box system. Nenzi et al. [36] presented an evolutionary algorithm for mining specifications in signal temporal logic (STL) distinguishing between regular and anomalous system behaviour. An important difference to our work is that they perform a classification task, while we learn models producing the same traces as the systems under consideration.

Evolutionary methods have been combined with testing in several areas: Abdessalem et al. [1] use evolutionary algorithms for the generation of test scenarios and learn decision trees to identify critical scenarios. Using the learned trees, they can steer the test generation towards critical scenarios. The tool Evosuite by Fraser and Arcuri [13] uses genetic operators for optimising whole test suites at once, increasing the overall coverage, while reducing the size of the test suite. Walkinshaw and Fraser presented Test by Committee, test-case generation using uncertainty sampling [49]. The approach is independent of the type of model that is inferred and an adaption of Query By Committee, a technique commonly used in active learning. In their implementation, they infer several hypotheses at each stage via genetic programming, generate random tests and select those tests which lead to the most disagreement between the hypotheses. In contrast to most other works considered, their implementation infers non-sequential programs. It infers functions mapping from numerical inputs to single outputs.

The work by Steffen et al. [21,42] is another good showcase for the strong possible relation between testing and model learning. They combine both areas, by performing black box tests and using the results to generate a model. Contrary to our work, they perform active learning, i.e., they use the intermediate versions of the learned models to guide the test generation. For a more comprehensive overview of combinations of learning and testing, we refer to [4].

Future Work. As indicated above, our technique is entirely passive, i.e. we learn from a set of timed traces (test observations), collected beforehand by random testing. There is no feedback from genetic programming to testing. In contrast to this, model-based testing could be applied to find discrepancies between the SUT and learned models [48]. These may then be used to iteratively improve the models. Active testing based on intermediate learned models may improve coverage of the SUT while requiring fewer tests, since we would benefit from additional knowledge about the system behaviour. This may therefore lead to improved accuracy of the model and increased performance through a reduction of tests and testing time. We are currently investigating this approach.

Assuming output urgency helps to approximate equivalence checks by "testing" candidate automata during learning. However, such models do not allow for uncertainty with respect to output timing. Relaxing this limitation represents an important next step. We are also currently working on this topic.

We demonstrated that TA can be genetically programmed, i.e. their structure is amenable to iterative refinement via mutation and crossover. Therefore, we could apply the same approach, but base the fitness evaluation on model checking by adapting the technique presented by Katz and Peled [24], to synthesise TA satisfying some properties. This would enable learning a black-box system, which

may contain errors, and synthesising a controller ensuring that those errors do not lead to observable system failures.

Acknowledgment. The work of B. Aichernig and M. Tappler has been carried out as part of the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". The work of K. Larsen and F. Lorber has been conducted within the ENABLE-S3 project that has received funding from the ECSEL Joint Undertaking under grant agreement no. 692455. This joint undertaking receives support from the European Union's Horizon 2020 research and innovation programme and Austria, Denmark, Germany, Finland, Czech Republic, Italy, Spain, Portugal, Poland, Ireland, Belgium, France, Netherlands, United Kingdom, Slovakia, Norway. We would like to thank student Andrea Pferscher for her help in implementing the demonstrator. We also want to thank the anonymous reviewers for their insightful comments and suggestions.

References

1. Abdessalem, R.B., Nejati, S., Briand, L.C., Stifter, T.: Testing vision-based control systems using learnable evolutionary algorithms. In: ICSE 2018, pp. 1016–1026. ACM (2018). https://doi.org/10.1145/3180155.3180160
2. Aichernig, B.K., et al.: Model-based mutation testing of an industrial measurement device. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 1–19. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09099-3_1
3. Aichernig, B.K., Lorber, F., Ničković, D.: Time for mutants — model-based mutation testing with timed automata. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 20–38. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38916-0_2
4. Aichernig, B.K., Mostowski, W., Mousavi, M.R., Tappler, M., Taromirad, M.: Model learning and model-based testing. In: Bennaceur, A., Hähnle, R., Meinke, K. (eds.) Machine Learning for Dynamic Software Analysis: Potentials and Limits. LNCS, vol. 11026, pp. 74–100. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96562-8_3
5. Aichernig, B.K., Tappler, M.: Efficient active automata learning via mutation testing. J. Autom. Reason. (2018). https://doi.org/10.1007/s10817-018-9486-0
6. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
7. Argyros, G., Stais, I., Jana, S., Keromytis, A.D., Kiayias, A.: SFADiff: automated evasion attacks and fingerprinting using black-box differential automata learning. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 1690–1701. ACM (2016). https://doi.org/10.1145/2976749.2978383
8. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
9. Bornot, S., Sifakis, J., Tripakis, S.: Modeling urgency in timed systems. In: de Roever, W.-P., Langmaack, H., Pnueli, A. (eds.) COMPOS 1997. LNCS, vol. 1536, pp. 103–129. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49213-5_5

10. David, A., Larsen, K.G., Legay, A., Nyman, U., Wasowski, A.: Timed I/O automata: a complete specification theory for real-time systems. In: Johansson, K.H., Yi, W. (eds.) HSCC 2010, pp. 91–100. ACM (2010). https://doi.org/10.1145/1755952.1755967
11. Elkind, E., Genest, B., Peled, D., Qu, H.: Grey-box checking. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 420–435. Springer, Heidelberg (2006). https://doi.org/10.1007/11888116_30
12. Fiterău-Broştean, P., Janssen, R., Vaandrager, F.: Combining model learning and model checking to analyze TCP implementations. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 454–471. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_25
13. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: SIGSOFT/FSE 2011, pp. 416–419. ACM (2011). https://doi.org/10.1145/2025113.2025179
14. Gómez, R.: A compositional translation of timed automata with deadlines to Uppaal timed automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 179–194. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04368-0_15
15. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. Theor. Comput. Sci. **411**(47), 4029–4054 (2010). https://doi.org/10.1016/j.tcs.2010.07.008
16. Grinchtein, O., Jonsson, B., Pettersson, P.: Inference of event-recording automata using timed decision trees. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 435–449. Springer, Heidelberg (2006). https://doi.org/10.1007/11817949_29
17. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46002-0_25
18. Hessel, A., Larsen, K.G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: Testing real-time systems using UPPAAL. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing. LNCS, vol. 4949, pp. 77–117. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78917-8_3
19. Hessel, A., Larsen, K.G., Nielsen, B., Pettersson, P., Skou, A.: Time-optimal real-time test case generation using Uppaal. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 114–130. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24617-6_9
20. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, New York (2010)
21. Hungar, H., Margaria, T., Steffen, B.: Test-based model generation for legacy systems. In: ITC 2003, pp. 971–980. IEEE (2003). https://doi.org/10.1109/TEST.2003.1271084
22. Isberner, M., Howar, F., Steffen, B.: The open-source LearnLib - a framework for active automata learning. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 487–495. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_32
23. Johnson, C.G.: Genetic programming with fitness based on model checking. In: Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., Esparcia-Alcázar, A.I. (eds.) EuroGP 2007. LNCS, vol. 4445, pp. 114–124. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-71605-1_11

24. Katz, G., Peled, D.: Synthesizing, correcting and improving code, using model checking-based genetic programming. STTT **19**(4), 449–464 (2017). https://doi.org/10.1007/s10009-016-0418-1
25. Koza, J.R.: Genetic Programming - On the Programming of Computers by Means of Natural Selection. Complex adaptive systems. MIT Press, Cambridge (1993)
26. Lai, Z., Cheung, S.C., Jiang, Y.: Dynamic model learning using genetic algorithm under adaptive model checking framework. In: QSIC 2006, pp. 410–417. IEEE (2006). https://doi.org/10.1109/QSIC.2006.25
27. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT **1**(1–2), 134–152 (1997). https://doi.org/10.1007/s100090050010
28. Lefticaru, R., Ipate, F., Tudose, C.: Automated model design using genetic algorithms and model checking. In: BCI 2009, pp. 79–84. IEEE (2009). https://doi.org/10.1109/BCI.2009.15
29. Lin, S.-W., André, É., Dong, J.S., Sun, J., Liu, Y.: An efficient algorithm for learning event-recording automata. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 463–472. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_35
30. Lin, S., André, É., Liu, Y., Sun, J., Dong, J.S.: Learning assumptions for compositional verification of timed systems. IEEE Trans. Softw. Eng. **40**(2), 137–153 (2014). https://doi.org/10.1109/TSE.2013.57
31. Lucas, S.M., Reynolds, T.J.: Learning DFA: evolution versus evidence driven state merging. In: CEC 2003, pp. 351–358. IEEE (2003). https://doi.org/10.1109/CEC.2003.1299597
32. Mao, H., Chen, Y., Jaeger, M., Nielsen, T.D., Larsen, K.G., Nielsen, B.: Learning deterministic probabilistic automata from a model checking perspective. Mach. Learn. **105**(2), 255–299 (2016). https://doi.org/10.1007/s10994-016-5565-9
33. de Matos Pedro, A., Crocker, P.A., de Sousa, S.M.: Learning stochastic timed automata from sample executions. In: Margaria, T., Steffen, B. (eds.) ISoLA 2012. LNCS, vol. 7609, pp. 508–523. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34026-0_38
34. Mediouni, B.L., Nouri, A., Bozga, M., Bensalem, S.: Improved learning for stochastic timed models by state-merging algorithms. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 178–193. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57288-8_13
35. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
36. Nenzi, L., Silvetti, S., Bartocci, E., Bortolussi, L.: A robust genetic algorithm for learning temporal specifications from data. In: McIver, A., Horvath, A. (eds.) QEST 2018. LNCS, vol. 11024, pp. 323–338. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99154-2_20
37. Nowostawski, M., Poli, R.: Parallel genetic algorithm taxonomy. In: KES 1999, pp. 88–92. IEEE (1999). https://doi.org/10.1109/KES.1999.820127
38. Pastore, F., Micucci, D., Mariani, L.: Timed k-tail: automatic inference of timed automata. In: ICST 2017, pp. 401–411 (2017). https://doi.org/10.1109/ICST.2017.43
39. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. JALC **7**(2), 225–246 (2002)
40. de Ruiter, J., Poll, E.: Protocol state fuzzing of TLS implementations. In: USENIX Security 2015, pp. 193–206. USENIX Association (2015). https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

41. Springintveld, J., Vaandrager, F.W., D'Argenio, P.R.: Testing timed automata. Theor. Comput. Sci. **254**(1–2), 225–257 (2001). https://doi.org/10.1016/S0304-3975(99)00134-6
42. Steffen, B., Howar, F., Merten, M.: Introduction to active automata learning from a practical perspective. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 256–296. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21455-4_8
43. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: ICST 2017, pp. 276–287 (2017). https://doi.org/10.1109/ICST.2017.32
44. Tappler, M., Aichernig, B.K., Larsen, K.G., Lorber, F.: Learning timed automata via genetic programming. CoRR abs/1808.07744 (2018). http://arxiv.org/abs/1808.07744
45. Tappler, M., Pferscher, A.: Supplementary Material for "Learning Timed Automata via Genetic Programming" (2019). https://doi.org/10.6084/m9.figshare.5513575.v1. https://figshare.com/articles/Supplementary_Material_for_Learning_Timed_Automata_via_Genetic_Programming_/5513575
46. Verwer, S., De Weerdt, M., Witteveen, C.: An algorithm for learning real-time automata. In: Benelearn 2007 (2007)
47. Verwer, S., de Weerdt, M., Witteveen, C.: A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In: Sempere, J.M., García, P. (eds.) ICGI 2010. LNCS (LNAI), vol. 6339, pp. 203–216. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15488-1_17
48. Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 305–320. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05089-3_20
49. Walkinshaw, N., Fraser, G.: Uncertainty-driven black-box test data generation. In: ICST 2017, pp. 253–263 (2017). https://doi.org/10.1109/ICST.2017.30