









Learning Deterministic One-Clock Timed Automata via Mutation Testing

Xiaochen Tang¹ , Wei Shen¹ , Miaomiao Zhang¹  (✉), Jie An² ,
Bohua Zhan^{3,4} , and Naijun Zhan^{3,4} 

¹ School of Software Engineering, Tongji University, Shanghai, China
{xiaochen9697, weishen, miaomiao}@tongji.edu.cn

² Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

³ State Key Laboratory of Computer Science, Institute of Software,
CAS, Beijing, China

⁴ University of Chinese Academy of Sciences, Beijing, China

Abstract. In active learning, an equivalence oracle is supposed to answer whether a hypothesis model is equivalent to the system under learning. Its implementation in real applications is considered a major bottleneck for active automata learning. The problem is especially difficult in the context of learning timed automata due to the infinitely large state space involved. In this paper, following the framework of combining mutation analysis and random testing, we propose an implementation of equivalence oracle in the context of learning deterministic one-clock timed automata (DOTAs). This includes two learning-friendly mutation operators, a heuristic test-case generation method, and a score-based test-case selection method. We implemented a prototype applying our approach by extending an existing tool on active learning of DOTAs and conducted extensive experiments. The results indicate that our method improves upon existing methods on the rate of learning correct models, the number of test cases required, and accumulated delay time in test cases.

Keywords: Active learning · Timed automata · Model-based mutation testing

1 Introduction

Active (model) learning [28] has emerged as a highly effective technique for learning the model of a system under learning (SUL). Most of active learning methods follow the L^* framework proposed by Angluin [12]. The learning process to achieve a hypothesis of the SUL can be viewed as an interaction between a *learner* and a *teacher*, where the learner asks *membership queries* (MQs) and *equivalence queries* (EQs) to a teacher who holds oracles to answer these queries. The former corresponds to a single test of the SUL to check whether a sequence of actions

This work has been partially funded by NSFC under grant No. 61972284, 62032019, 62032024, 62192732, 62192730, and 61625206, by DFG project 389792660-TRR 248.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
A. Bouajjani et al. (Eds.): ATVA 2022, LNCS 13505, pp. 233–248, 2022.
https://doi.org/10.1007/978-3-031-19992-9_15

can be executed. EQs check whether a learned hypothesis represents the SUL. The teacher either answers affirmatively or generates a counterexample showing the difference between the SUL and the hypothesis. Compared to active learning of deterministic finite state automata (DFAs) [12], learning timed automata [7] is much more complex since it involves an infinite set of timed actions and clock reset information while the alphabets of DFAs is finite. Among the existing works [8–10, 16, 17, 29] on active learning of the different timed models, An et al. proposed an active learning method for deterministic one-clock timed automata (DOTAs) in [8]. Inherited from L^* , this method also assumes an ideal setting where the EQs can be answered exactly by an oracle. However, exact equivalence oracles are usually unrealistic in most practical situations, which is a well-known problem that learning methods based on L^* in practice face and can be considered as “the true bottleneck of automata learning” [13].

To address the issues mentioned above, various attempts have been carried out. For real applications, one of the most widely studied approach for EQs is *conformance testing* [1, 13–15, 22, 25]. However, the size of the constructed test suite is usually exponential in the number of states of the SUL, which makes it inefficient in many industrial scenarios. Another limitation is that most of the existing methods for timed systems do not consider the accumulated delay time of test suites. Therefore, a new target for conformance testing is to find counterexamples fast, rather than trying to prove equivalence [19]. Model-based testing [27], a popular technique for automated test-case generation, can be used as an approach for conformance testing. Commonly relying on some coverage criterion, it produces new test cases until that criterion is satisfied. Model-based mutation testing [2, 3, 23] uses *faults* as such a criterion: the original model is modified by different fault injections, called *mutation operators*, which results in a set of faulty models called *mutants*. In [6], Aichernig et al. combined random testing and mutation analysis [11] to learn Mealy machines and show their effectiveness. Here random testing is used to achieve high variability of tests, while mutation analysis is used to ensure appropriate coverage.

In this paper, we propose a conformance testing approach combining random testing and mutation-based testing to replace exact EQs in the active learning of DOTAs. Even though many existing studies proposed mutation operators for timed automata, which generate mutants covering specific faults [4, 26, 31], there are many redundancies among these mutation operators, and the mutants generated by these operators are possibly non-deterministic. These make the existing mutation operators not well-suited to the context of active automata learning. Thus, we design two mutation operators to address DOTAs learning. The approach we presented aims at finding counterexamples quickly, and reducing the total amount of time of executing test cases, in addition to the reduction of tests as in [6]. Moreover, to design the test suits, we take previous counterexamples into consideration and the modifications between two successive hypotheses in the learning procedure for DOTAs. Our contributions are summarized as follows.

- A heuristic algorithm for random test-case generation. We take counterexamples into consideration and apply three heuristics to random testing, aiming at generating more useful test cases.

- Two learning-friendly mutation operators. In contrast to generating only first-order mutants [4, 26, 31], which usually contain one fault, we take into account the deterministic behaviours of the model and the modifications between two successive hypotheses obtained in the learning process, so that more faults are considered in the construction of mutants.
- A mutation and score-based selection of test cases. In addition to mutation coverage, we also take the length and accumulated delay time of the test cases into consideration to achieve faster testing.
- An implementation of our method. To investigate the effectiveness and efficiency of our method, we extend the prototype tool for DOTAs learning [8] and compare our method with various existing methods.

The rest of the paper is organized as follows. In Sect. 2, we review the learning algorithm for DOTAs in [8] and the model-based mutation testing framework. In Sect. 3, we describe the mutation-based testing in the context of active learning of DOTAs in detail. In Sect. 4, we introduce two mutation operators used in the mutation-based testing framework. The experimental results are reported in Sect. 5. Finally, Sect. 6 concludes this paper.

2 Preliminaries

2.1 Deterministic One-Clock Timed Automata

In this paper, we consider a subclass of timed automata [7] that are deterministic and contain only a single clock, called *Deterministic One-Clock Timed Automata* (DOTAs). Let \mathbb{N} be the natural numbers and $\mathbb{R}_{\geq 0}$ be the non-negative real numbers. We use \top to stand for true and \perp for false. Let $\mathbb{B} = \{\top, \perp\}$. Let c be the clock variable, denote by Φ_c the set of clock constraints of the form $\phi ::= \top \mid c \bowtie m \mid \phi \wedge \phi$, where $m \in \mathbb{N}$ and $\bowtie \in \{=, <, >, \leq, \geq\}$.

Definition 1 (One-clock timed automata). *A one-clock timed automaton (OTA) is a 6-tuple $\mathcal{A} = (\Sigma, Q, q_0, F, c, \Delta)$, where Σ is a finite set of actions, Q is a finite set of locations, q_0 is the initial location, $F \subseteq Q$ is a set of final locations, c is the unique clock and $\Delta \subseteq Q \times \Sigma \times \Phi_c \times \mathbb{B} \times Q$ is a finite set of transitions.*

A transition $\delta \in \Delta$ is a 5-tuple (q, σ, ϕ, b, q') , where $q, q' \in Q$ are the source and target locations respectively, $\sigma \in \Sigma$ is an action, $\phi \in \Phi_c$ is a clock constraint, and b is the reset indicator. Such δ allows a jump from q to q' by performing an action σ if the current clock valuation ν satisfies the constraint ϕ . We also call ϕ as a *guard*. Meanwhile, clock c is reset to zero if $b = \top$ and remains unchanged otherwise. A *clock valuation* is a function $\nu : c \mapsto \mathbb{R}_{\geq 0}$ that assigns a non-negative real number to the clock. For $t \in \mathbb{R}_{\geq 0}$, let $\nu + t$ be the clock valuation with $(\nu + t)(c) = \nu(c) + t$. A *state* is a pair (q, ν) , where $q \in Q$ and ν is a clock valuation. A *timed action* is a pair (σ, t) that indicates the action σ is applied after t time units since the occurrence of the previous action. A *timed trace* is a

sequence $\omega = (\sigma_1, t_1) (\sigma_2, t_2) \dots (\sigma_n, t_n)$ of timed actions $(\sigma_i, t_i) \in \Sigma \times \mathbb{R}_{\geq 0}$. A finite *run* ρ of \mathcal{A} over a timed trace $\omega = (\sigma_1, t_1) (\sigma_2, t_2) \dots (\sigma_n, t_n)$ is a sequence of timed states and timed actions $\rho = (q_0, \nu_0) \xrightarrow{\sigma_1, t_1} (q_1, \nu_1) \xrightarrow{\sigma_2, t_2} \dots \xrightarrow{\sigma_n, t_n} (q_n, \nu_n)$ where $\nu_0 = 0$, and for all $1 \leq i \leq n$ there exists a transitions $(q_{i-1}, \sigma_i, \phi_i, b_i, q_i) \in \Delta$ such that $\nu_{i-1} + t_i$ satisfies ϕ_i , and $\nu_i(c) = 0$ if $b_i = \top$, $\nu_i(c) = \nu_{i-1}(c) + t_i$ otherwise. Since time values t_i represents *delay* times, we call such a timed trace a *delay-timed word*. The delay-timed word is observed outside from the view of the global clock. On the other hand, the behavior can also be observed inside from the view of the local clock. This results in a *logical-timed word* of the form $\gamma = (\sigma_1, \mu_1) (\sigma_2, \mu_2) \dots (\sigma_n, \mu_n)$ with $\mu_i = t_i$ if $i = 1$ or $b_{i-1} = \top$, otherwise $\mu_i = \mu_{i-1} + t_i$. The time spent in a timed trace ω , denoted $time(\omega)$ is the sum of all delays in ω , for example, $time(\epsilon) = 0$ and $time((a, 1.0)(b, 1.5)) = 2.5$.

Definition 2 (Deterministic OTA). *An OTA is a deterministic one-clock timed automaton (DOTA) if there is at most one run for a given timed word.*

A DOTA \mathcal{A} is *complete* if for any location q and action σ , the constraints form a partition of $\mathbb{R}_{\geq 0}$. Any incomplete DOTA \mathcal{A} can be transformed into a complete DOTA accepting the same timed language by adding a non-accepting *sink* location q_{sink} , and adding transitions to the sink location for each unavailable action [8]. We therefore assume that we are working with complete DOTAs.

2.2 Active Learning Algorithm for DOTAs

In this section, we provide a brief description of the active learning algorithm [8] for a black-box SUL which can be represented by a DOTA \mathcal{A} . The existing work distinguishes two learning scenarios: learning from a *normal teacher* or a *smart teacher*. As the work in this paper concerns EQs only, it applies to both normal teacher and smart teacher scenarios. For the experiments, we mainly consider the case of smart teachers. In practical applications, this corresponds to executing the test case, where information about clock-resets is known by code instrumentation or watchdogs (refer to the concept of testable systems in [15, 18]). The learner maintains an *observation table* to collect the answers of MQs. The table will be transformed to a DOTA \mathcal{H} as a hypothesis if it satisfies several preparedness conditions. The learner then performs an EQ by submitting \mathcal{H} to the teacher. In theory, we assume that the teacher holds an equivalence oracle to answer EQs, returning whether the timed languages of \mathcal{H} and \mathcal{A} are equivalent. If the answer is no, the teacher also returns a logical-timed word with reset information as a counterexample. The learner then performs more MQs guided by the counterexample. The learning loop terminates when an EQ returns a positive answer. We refer to [8] for more details.

However, since such equivalence oracles may not exist in practical situations, the equivalence oracle is often achieved through conformance testing, i.e., asking a lot of MQs to answer a single EQ. If, for every MQ, the output produced by the SUL is consistent with hypothesis \mathcal{H} , the answer to the EQ is “Yes”. Otherwise, the answer “No” is provided, together with a counterexample that

indicates a difference between \mathcal{H} and the SUL. In this paper, we address the implementation of equivalence oracle through a combination of random testing and mutation analysis.

2.3 Model-Based Mutation Testing

Model-based mutation testing [2, 3, 23] is a promising technique combining the central ideas of mutation testing [21] and model-based testing [27]. By making some adaptations, it can be regarded as an equivalence oracle in the context of active learning. The process starts with the current hypothesis \mathcal{H} . A set of mutants from \mathcal{H} are generated by mutation operators. Once all mutants are created, the actual test suite generation starts. The original \mathcal{H} is compared to each mutant via an equivalence check (this can be done exactly since models for both \mathcal{H} and the mutant are available). If a mutant \mathcal{M} is not equivalent to \mathcal{H} , the checking procedure returns a trace that serves as a witness, and this trace can be converted into a test case.

The equivalence checks in the above process can be computationally expensive. Therefore, the work in [6] considers a new model-based mutation testing framework combined with random testing for learning Mealy machines, and the experiments have demonstrated that a combination of random exploration and mutation-based test-case generation is beneficial. Briefly, the framework includes the following steps to generate test suites for conformance testing. First, it utilizes random testing to generate a large set of test cases \mathbf{T} . Then it analyzes the mutation coverage of each test case in \mathbf{T} , i.e., it executes each test case and determines which of the mutants produces outputs different from \mathcal{H} . Finally, the test suite is created by selecting a subset of \mathbf{T} based on the computed mutation coverage. After that, the conformance testing between \mathcal{H} and the SUL can be conducted by executing test cases of the test suite on both respectively. The test case producing different outputs between \mathcal{H} and the SUL is a counterexample to the equivalence, which is utilized to further refine the current hypothesis \mathcal{H} .

3 Mutation-Based Testing for DOTAs

In this section, we introduce our mutation-based testing process for solving the EQs in learning DOTAs. We first describe the whole process and then present the heuristic method to generate test cases and the mutation-based selection of the test suite. The details on the mutation operators and mutation generation are described in Sect. 4.

3.1 The Process Overview

Following the idea in [6], we use a combination of random testing, to achieve high variability of tests, and mutation analysis, to address coverage appropriately. However, given the particular characteristics of DOTAs learning, that is, counterexample processing will generate two kinds of modifications between two

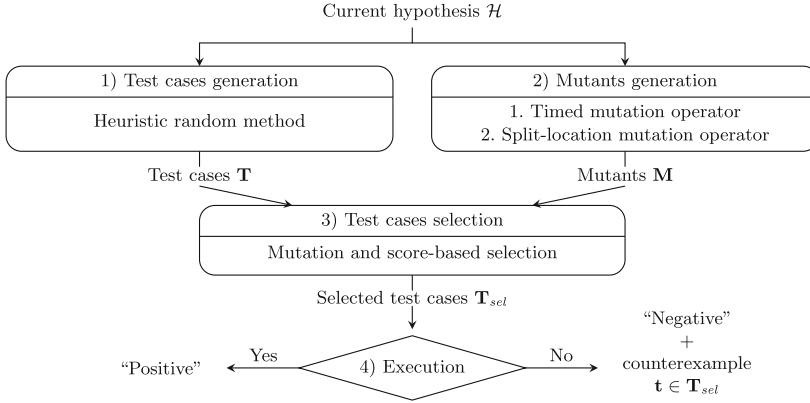


Fig. 1. The overview of mutation-testing-based equivalence checking for learning DOTAs.

successive hypotheses, we have to design new mutation operators and adapt the framework accordingly. The whole process is depicted in Fig. 1. The input for this process is a hypothesis \mathcal{H} , a learned intermediate DOTA, while the output is the answer of an EQ.

1. From the model \mathcal{H} , we first develop a heuristic test-case generation algorithm to obtain a large set of test cases \mathbf{T} (see Sect. 3.2).
2. Independently, we generate a set of mutants \mathbf{M} from \mathcal{H} based on the timed mutation operator (see Sect. 4.1) and the split-location mutation operator (see Sect. 4.2).
3. Using a score-based test-case selection method, together with mutation analysis, a subset \mathbf{T}_{sel} of \mathbf{T} is selected (see Sect. 3.3). The purpose is to select a subset of test cases from \mathbf{T} that are likely to distinguish between the original hypothesis \mathcal{H} and the mutants as the test suite to execute, i.e. to select the test cases that cover the mutants.
4. Finally, we execute all test cases in \mathbf{T}_{sel} on the hypothesis \mathcal{H} and the SUL respectively. A test case is a counterexample if producing different outputs on the SUL and \mathcal{H} . If such a counterexample is found, it is returned to the learning algorithm. Otherwise, the EQ returns a positive answer.

Different from [6] using only one mutation operator for generating mutants, we hereby use two mutation operators to cover the different possibilities of mutations for timed automata. This produces a set of mutants helpful for generating a more complete test suite that is able to find potential differences between \mathcal{H} and the SUL. Our experiments have shown that both mutation operators are necessary for improving the rate of learning the correct model of the SUL. In addition, to avoid interleaving complexity, instead of simultaneously using two mutation operators to generate test cases, we divide the process into two phases using two operators respectively. In our case, we choose to first use the timed

Algorithm 1. Heuristic test-case generation

```

Input: hypothesis  $\mathcal{H} = (\Sigma, Q, q_0, F, c, \Delta)$ ;           12:    $\Delta' \leftarrow \{\delta \mid \delta = (q, \sigma, \phi, b, q') \in$ 
      the previous counterexample  $ctx$ ;                        $\Delta \wedge q = q_c \wedge q' = q_{sink}\}$ ;
      the maximal length  $len$  of “ $xy$ ”                       13:   if  $\Delta' \neq \emptyset$  then
      part of a test case; three probabili-                 14:      $(q, \sigma, \phi, b, q') \leftarrow getRandom(\Delta')$ ;
      ty values  $p_{start}$ ,  $p_{valid}$  and  $p_{stop}$ .                 15:      $t \leftarrow getRandomDelay(\nu, \phi)$ ;
Output: a test case  $\mathbf{t}$ .                                     16:     if  $t \neq None$  then
1:  $\mathbf{t} \leftarrow \epsilon$ ;                                       17:        $\mathbf{t} \leftarrow \mathbf{t} \cdot (\sigma, t)$ ;
2:  $q_c \leftarrow q_0$ ;  $\nu \leftarrow 0$ ;                         18:        $q_c, \nu \leftarrow execute((q_c, \nu), (\sigma, t))$ ;
3:  $Q_{visited} \leftarrow \emptyset$ ;                               19:        $Q_{visited} \leftarrow Q_{visited} \cup \{q_c\}$ ;
4: if  $\mathbf{prob}(p_{start})$  then  $\mathbf{t} \leftarrow ctx$ ;                 20:     if  $\mathbf{prob}(p_{stop})$  then
5:   for  $i \leftarrow 0$  to  $|ctx| - 1$  do                          21:       break;
6:      $q_c, \nu \leftarrow execute((q_c, \nu), ctx[i])$ ;          22:   if  $Q \setminus Q_{visited} \neq \emptyset$  then
7:      $Q_{visited} \leftarrow Q_{visited} \cup \{q_c\}$ ;            23:      $q_t \leftarrow getRandomLocation(Q \setminus Q_{visited})$ ;
8: while  $|\mathbf{t}| < len$  do                                       24:      $\omega \leftarrow findTimedTrace((q_c, \nu), q_t)$ ;
9:   if  $\mathbf{prob}(p_{valid})$  then                                       25:     if  $\omega \neq \epsilon$  then
10:     $\Delta' \leftarrow \{\delta \mid \delta = (q, \sigma, \phi, b, q') \in$  26:        $\mathbf{t} \leftarrow \mathbf{t} \cdot \omega$ ;
       $\Delta \wedge q = q_c \wedge q' \neq q_{sink}\}$ ;                 27:   return  $\mathbf{t}$ ;
11:  else

```

mutation operator only, and if no counterexample is found, then use the split-location mutation operator and repeat Step 2 to Step 4.

3.2 Heuristic Test-Case Generation

Random testing is a widely-used method and has been integrated in the learning library LearnLib [20] as an EQ method for untimed models. In this section, we apply three heuristics to generate test cases randomly, aiming at generating more useful test cases. Algorithm 1 presents the generation process, containing three main steps corresponding to the heuristics. The inputs include the current hypothesis \mathcal{H} and several relevant parameters, and the output is a test case $\mathbf{t} \in (\Sigma \times \mathbb{R}_{\geq 0})^*$ of the form xyz , where x is the prefix, y a random sequence of timed actions, and z the suffix. Function $\mathbf{prob}(p)$ returns true with probability p and false with probability $1 - p$. The generation process is performed many times to generate a large-size (can be parameterized by the user) test set \mathbf{T} , whose size is related to the number of actions and transitions, and the timing parameters of \mathcal{H} (see Sect. 5).

1. Firstly, according to our observation that a counterexample is often prefixed with its previous counterexample, we reuse the previous counterexample ctx as the prefix x with probability p_{start} (Line 4 to Line 7).
2. Then, consider for many reactive systems, from the current timed state, randomly selecting an action is likely to transit to a sink location, since not all timed actions can be executed or make sense at the current state. Therefore, we prefer to explore non-sink locations with probability p_{valid} when using random walking method to find timed actions (σ, t) . Such timed actions form the segment y extending the test case (Line 8 to Line 21). The parameter len

Algorithm 2. Mutation and score-based test-case selection

Input: \mathbf{M} ; $\mathbf{T}_{\mathcal{M}}$ for all $\mathcal{M} \in \mathbf{M}$; $V_{\mathbf{t}}$ for all $\mathbf{t} \in \mathbf{T}$. Output: a subset of test cases \mathbf{T}_{sel} .	4:	if $\mathbf{T}_{\mathcal{M}_{opt}} \neq \emptyset$ and $\mathbf{T}_{\mathcal{M}_{opt}} \cap \mathbf{T}_{sel} = \emptyset$ then
1: $\mathbf{T}_{sel} \leftarrow \emptyset$;	5:	$\mathbf{t}_{opt} \leftarrow \operatorname{argmax}_{\mathbf{t} \in \mathbf{T}_{\mathcal{M}_{opt}}} V_{\mathbf{t}}$;
2: while $\mathbf{M} \neq \emptyset$ do	6:	$\mathbf{T}_{sel} \leftarrow \mathbf{T}_{sel} \cup \{\mathbf{t}_{opt}\}$;
3: $\mathcal{M}_{opt} \leftarrow \operatorname{argmin}_{\mathcal{M} \in \mathbf{M}} \mathbf{T}_{\mathcal{M}} $;	7:	$\mathbf{M} \leftarrow \mathbf{M} \setminus \{\mathcal{M}_{opt}\}$;
	8:	return \mathbf{T}_{sel} ;

limits the maximal length of the “ xy ” part. The exploring process stops with probability p_{stop} at the end of each round.

3. Finally, we add the path from the current location to a non-visited location as suffix z to increase the coverage of each test case (Line 22 to Line 26).

3.3 Mutation and Score-Based Test-Case Selection

In order to improve the mutation coverage of test cases, we define a special output function in response to a given delay-timed word. Let $\mathcal{D} = \{+, -\}$ be the output domain, indicating whether the trace is accepted (+) or not (-).

Definition 3 (Output function). *Given a test case (delay-timed word) $\mathbf{t} = (\sigma_1, t_1) (\sigma_2, t_2) \cdots (\sigma_n, t_n)$ and a (complete) DOTA $\mathcal{A} = (\Sigma, Q, q_0, F, c, \Delta)$, corresponding a run $\rho = (q_0, \nu_0) \xrightarrow{\sigma_1, t_1} (q_1, \nu_1) \xrightarrow{\sigma_2, t_2} \cdots \xrightarrow{\sigma_n, t_n} (q_n, \nu_n)$ in \mathcal{A} , the output function for the test case is defined as $out_{\mathcal{A}}(\mathbf{t}) = o_1 o_2 \cdots o_n$, where $o_i = +$ if $q_i \subseteq F$ and $o_i = -$ otherwise.*

Given two models \mathcal{A}_1 and \mathcal{A}_2 , we say \mathbf{t} *passes* if $out_{\mathcal{A}_1}(\mathbf{t}) = out_{\mathcal{A}_2}(\mathbf{t})$, otherwise, it *fails* and serves as a counterexample to the equivalence.

After the heuristic test-case generation described previously, we have obtained a large-size test set \mathbf{T} . However, it may contain just a small number of test cases that can be counterexamples due to the randomness. Therefore, we further need to select a subset \mathbf{T}_{sel} from \mathbf{T} consisting of test cases that are more likely to be counterexamples. Normally, the selection is based purely on a kind of coverage, e.g. location or transition coverage. However, unlike testing for Mealy machines [6, 14] or other finite labeled transition systems, for timed systems, we should also consider the time elapsed in two consecutive input actions. Therefore, the main objective in addition to the number of test cases is to reduce the accumulated delay time of all test cases used in conformance testing, provided that the maximum mutant coverage is achieved. Our selection process is based on a set of mutants of the hypothesis \mathcal{H} . We leave the details of mutation generation in Sect. 4 and suppose that a set of mutants \mathbf{M} has been generated.

Algorithm 2 presents the mutation and score-based selection method, which considers several factors of the test cases. At beginning, we need to prepare the inputs. First, we associate each test case $\mathbf{t} \in \mathbf{T}$ with a set of mutants $\mathbf{M}_{\mathbf{t}}$ covered by \mathbf{t} , i.e. $\mathbf{M}_{\mathbf{t}} = \{\mathcal{M} \in \mathbf{M} \mid out_{\mathcal{H}}(\mathbf{t}) \neq out_{\mathcal{M}}(\mathbf{t})\}$, and associate each

mutant $\mathcal{M} \in \mathbf{M}$ with a set of test cases $\mathbf{T}_{\mathcal{M}} \in \mathbf{T}$ that can cover \mathcal{M} , i.e. $\mathbf{T}_{\mathcal{M}} = \{\mathbf{t} \in \mathbf{T} \mid out_{\mathcal{H}}(\mathbf{t}) \neq out_{\mathcal{M}}(\mathbf{t})\}$. Then, we use the following four attributes to decide whether a test case \mathbf{t} is selected: (1) $time(\mathbf{t})$ is the total delay time of \mathbf{t} , (2) $|\mathbf{t}|$ is the length of \mathbf{t} , (3) $|\mathbf{M}_{\mathbf{t}}|$ is the mutation coverage of \mathbf{t} , and (4) $|\mathbf{C}_{\mathbf{t}}|$ is the transition coverage of \mathbf{t} . After normalization for the attributes, we acquire the score $V_{\mathbf{t}} = a \cdot (1 - time(\mathbf{t}))' + b \cdot (1 - |\mathbf{t}|)' + c \cdot |\mathbf{M}_{\mathbf{t}}|' + d \cdot |\mathbf{C}_{\mathbf{t}}|'$, where a, b, c, d are the weights. Upon obtaining $\mathbf{T}_{\mathcal{M}}$ for each mutant \mathcal{M} and $V_{\mathbf{t}}$ for each test case \mathbf{t} , the algorithm follows the basic idea that the higher the score value, the more likely the test case will be selected. So, at each round, first select the mutant \mathcal{M} covered by the least number of test cases (Line 3). If the currently selected test cases \mathbf{T}_{sel} cannot cover it, the test case \mathbf{t} with the largest score $V_{\mathbf{t}}$ is chosen from $\mathbf{T}_{\mathcal{M}_{opt}}$ and added to \mathbf{T}_{sel} (Line 4 to Line 6). Then remove \mathcal{M} from \mathbf{M} . The steps repeat until all mutants covered by at least one test case are considered, i.e. the selected test cases have been able to achieve the maximum possible coverage of the mutant set. Mutants that cannot be covered by any test case are removed by constraint $\mathbf{T}_{\mathcal{M}_{opt}} \neq \emptyset$ in Line 4.

4 Learning-Friendly Mutation Operators for DOTAs

In order to provide the mutants of hypothesis \mathcal{H} for the processes in Sect. 3, we need to design suitable mutation operators for DOTAs. Considering the learning method in [8], we find that counterexample handling will lead to generating two kinds of modifications between the successive hypotheses \mathcal{H} and \mathcal{H}' . Similar to the terms used in [24], the first is called *expansive modification*, which means that \mathcal{H}' has more locations and/or transitions than \mathcal{H} . While the second is called *non-expansive modification*, which implies that only the timed constraints and/or the reset indicators of some transitions differ between \mathcal{H} and \mathcal{H}' . Inspired by the observations, we design two *mutation operators*. The first one is *timed mutation operator* given in Sect. 4.1, which includes a series of mutation operations specific to DOTAs learning, corresponding to the transition changes of the expansive and non-expansive modification. The second, *split-location mutation operator* given in Sect. 4.2, is closely related to [6], corresponding to the location change of the *expansive modification*. In terms of the two designed operators, all the mutants generated from \mathcal{H} are still deterministic automata.

4.1 Timed Mutation Operator

Given the current hypothesis \mathcal{H} , the timed mutation operator is conducted on every transition in turn to generate mutants. Consider a transition $\delta = (q, \sigma, \phi, b, q')$, the basic idea is as follows. For the timed interval ϕ , we will first slice it into sub-intervals as a partition (see Definition 4), resulting in several new transitions. Then we conduct two operations (see Definition 5) on each new transition to generate mutants. One operation is to change the target location of one transition, which helps us to modify the timed interval ϕ . The other operation is to change the reset indicator of one transition. Actually, we can also apply the two operations to some transitions at the same time.

Algorithm 3. Mutants generation via the timed mutation operator

Input: a DOTA $\mathcal{H} = (\Sigma, Q, q_0, F, c, \Delta)$; the greatest integer constant B ; a slicing step w . OUTPUT: a set of mutants \mathbf{M} .	4:	for each $\delta_s \in \Delta_s$ do
	5:	$\Delta_m \leftarrow rt(\delta_s) \cup fl(\delta_s) \cup fl \circ rt(\delta_s)$;
	6:	for each $\delta_m \in \Delta_m$ do
1: $\mathbf{M} \leftarrow \emptyset$	7:	$\mathcal{M} \leftarrow (\Sigma, Q, q_0, F, c, \Delta \setminus \{\delta\}) \cup \Delta_s \setminus \{\delta_s\} \cup \{\delta_m\}$;
2: for each $\delta = (q, \sigma, \phi, b, q') \in \Delta$ do	8:	$\mathbf{M} \leftarrow \mathbf{M} \cup \{\mathcal{M}\}$;
3: $\Delta_s \leftarrow \{(q, \sigma, \phi_s, b, q') \mid \phi_s \in S_g(\phi, B, w)\}$;	9:	return \mathbf{M} ;

Definition 4 (Slicing timed interval). Let B be the greatest integer constant appearing in the DOTA to be learned (can also be set by the user), and $w \in \mathbb{N}_{>0}$ be the slicing step. Given a timed interval $\langle \alpha, \beta \rangle$, where $\langle \cdot, \cdot \rangle \in \{(\cdot, \cdot), [\cdot, \cdot], (\cdot, \cdot], [\cdot, \cdot)\}$, the slicing can generate a partition of $\langle \alpha, \beta \rangle$ as follows:

- If $\beta > B$ (including $\beta = \infty$), $S_g(\langle \alpha, \beta \rangle, B, w)$
 $= \{ \langle \alpha, \alpha + w \rangle \mid \alpha + w \leq B \} \cup \{ \langle \alpha + w * i, \alpha + w * i \rangle \mid \alpha + w * i \leq B, i \in \mathbb{N}_{>0} \}$
 $\cup \{ \langle \alpha + w * i, \alpha + w * (i + 1) \rangle \mid \alpha + w * (i + 1) \leq B, i \in \mathbb{N}_{>0} \}$
 $\cup \{ \langle \alpha + w * i, \beta \rangle \mid \alpha + w * i \leq B \wedge \alpha + w * (i + 1) > B, i \in \mathbb{N}_{>0} \}$
- If $\beta \leq B$, $S_g(\langle \alpha, \beta \rangle, B, w)$
 $= \{ \langle \alpha, \alpha + w \rangle \mid \alpha + w < \beta \} \cup \{ \langle \alpha + w * i, \alpha + w * i \rangle \mid \alpha + w * i < \beta, i \in \mathbb{N}_{>0} \}$
 $\cup \{ \langle \alpha + w * i, \alpha + w * (i + 1) \rangle \mid \alpha + w * (i + 1) < \beta, i \in \mathbb{N}_{>0} \}$
 $\cup \{ \langle \alpha + w * i, \beta \rangle \mid \alpha + w * i < \beta \wedge \alpha + w * (i + 1) \geq \beta, i \in \mathbb{N}_{>0} \}$

Therefore, for a transition $\delta = (q, \sigma, \phi, b, q')$ in \mathcal{H} , ϕ is sliced into several timed intervals $S_g(\phi, B, w)$. This implies that instead of δ , a new transition set $\Delta_s = \{(q, \sigma, \phi_s, b, q') \mid \phi_s \in S_g(\phi, B, w)\}$ with $|\Delta_s| = |S_g(\phi, B, w)|$ is generated. Obviously, if the slicing step $w = 1$, the intervals in S_g are *regions* [7].

Definition 5 (Timed mutation operations). Given a sliced transition $\delta_s = (q, \sigma, \phi_s, b, q') \in \Delta_s$, the timed mutation operator includes the following two operations: (1) *Re-target*: $rt(\delta_s) = \{(q, \sigma, \phi_s, b, q'') \mid q'' \in Q \setminus q'\}$; (2) *Flop-reset*: $fl(\delta_s) = \{(q, \sigma, \phi_s, b', q') \mid b' \in \mathbb{B} \setminus b\}$.

Algorithm 3 presents the procedure generating mutants from \mathcal{H} using the timed mutation operator. First, for each transition δ , we build the sliced transition set Δ_s (Line 3). Second, for each sliced transition $\delta_s \in \Delta_s$, we conduct the mutation operations rt and fl separately and both on it, and thus get a mutated transition set Δ_m (Line 5). Then, for each mutated transition $\delta_m \in \Delta_m$, we can build a mutant by removing δ, δ_s , and adding the new mutated transition δ_m to the transition set (Line 7). Therefore, every mutant is obtained from \mathcal{H} via the options of changing the timed constraints, or flopping the reset indicator, or adding new transitions, or a combination of the above three, so that the mutant gets closer to the successor hypothesis \mathcal{H}' . To simplify a mutant, we merge two transitions if they have the same source location, target location, action, and reset indicator respectively. For example, given two transitions $(q, \sigma, [2, 4], b, q')$ and $(q, \sigma, (4, 5], b, q')$, the merged transition is $(q, \sigma, [2, 5], b, q')$.

4.2 Split-Location Mutation Operator

The *split-location mutation operator* mainly involves modification on locations while not the timed information on transitions, which was first introduced in [6] for the learning of Mealy machines. To deal with DOTAs, we make some modifications for the operator that includes the execution of the following steps: (1) making abstraction from a DOTA to a DFA by labeling every transition with a different abstract action u , (2) mutating the DFA using split-location operator referring to [6], and (3) transforming the mutated DFA back to a DOTA as a mutant of the original DOTA. In order to instantiate the split-location mutation operator in our implementation and experiments, we also need two parameters: n_{acc} is an upper bound on the number of access sequences leading to a split location and k is the length of a distinguishing sequence.

5 Implementation and Experiments

To further investigate the efficiency of the proposed method, we extend the existing DOTAs learning prototype tool in [8] with the proposed EQ implementation and evaluate it on a set of DOTAs. The experiments are meant to check whether the proposed technique is an effective implementation of equivalence oracle to find counterexamples for incorrect hypotheses under the DOTAs learning setting. The prototype tool and experiments are available on the tool page https://github.com/Anna9697/mut_learn_DOTAs.

5.1 Case Studies

First, we evaluated the DOTAs learning with mutation-based testing on 18 randomly generated DOTAs. We divided them into 6 groups depending on the number of locations ($|Q|$), the number of untimed actions ($|\Sigma|$), and the maximum constant appearing in the models (B). In addition, there are also three manually created examples from the real world: a lamp touch control model (Lamp) from [5], a coffee vending machine model (Coffee) from [30], and the model of TCP protocol (TCP) from [8].

For each case, we executed 15 times to acquire the average number of tests ($\#tests$) and actions ($\#actions$), the average accumulated delay time in tests (t_{delay}), and the number of correct models learned (n_{exact}). We used the same exact equivalence oracle in [8] to judge whether the learned automata were completely correct or not. The related parameters to run the experiments are as follows. To generate test cases via Algorithm 1, we set parameters $p_{start} = 0.4$, $p_{valid} = 0.8$, $p_{stop} = 0.05$ and $len = 2 \cdot |Q|$, where $|Q|$ is the number of locations of models in each group, and sampled delay-time value at a granularity of 0.5 with the upper bound $1.5 \cdot B$. We obtain a test suite \mathbf{T} with size $|\mathbf{T}| = 30 \cdot |Q_{\mathcal{H}}| \cdot |\Sigma_{\mathcal{H}}| \cdot B$ by repeatedly calling Algorithm 1, where $|Q_{\mathcal{H}}|$ and $|\Sigma_{\mathcal{H}}|$ are the number of locations and actions of the current hypothesis \mathcal{H} respectively. In the mutation generation, for the timed mutation operator, we set the slicing step w to the minimal *duration* of the constraints of the models¹. While for split-location mutation

¹ Set $w = 1$ if no additional information is known.

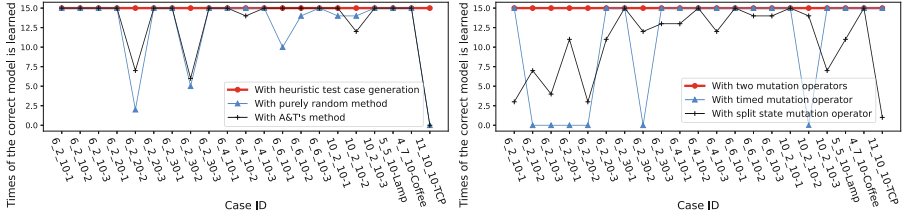
Table 1. Experimental results of case studies.

Case ID	Mutation_new				Mutants_checking				Heuristic_random_testing			
	#tests	#actions	t_{delay}	n_{exact}	#tests	#actions	t_{delay}	n_{exact}	#tests	#actions	t_{delay}	n_{exact}
6.2.10-1	443.1	2592.1	9012.3	15	2640.2	9295.9	28774.7	15	3372.0	14405.0	46005.1	15
6.2.10-2	559.9	3254.1	15934.2	15	3004.6	9141.6	36273.7	15	4204.3	20711.0	97402.2	15
6.2.10-3	967.1	6218.8	21540.9	15	9389.0	32769.5	94801.4	15	5640.5	34557.3	132976.6	12
6.2.20-1	1085.8	7475.5	41588.0	15	11616.6	42426.0	171239.0	15	6957.6	36032.3	206737.9	15
6.2.20-2	614.4	3669.9	21300.3	15	3572.5	13607.7	58794.7	15	6885.5	33562.3	181704.7	15
6.2.20-3	1428.3	8817.3	59688.2	15	12931.5	40198.2	233110.7	15	9529.8	53712.3	387141.1	15
6.2.30-1	859.6	6626.6	54486.0	15	4244.8	13544.9	100506.4	15	9238.7	61385.1	729274.2	15
6.2.30-2	2381.5	18481.1	181969.8	15	43635.6	155341.7	1236780.6	15	17205.5	112464.1	1346503.8	15
6.2.30-3	1321.1	7378.7	68047.2	15	9877.7	27839.6	259687.3	15	9600.3	41289.2	419800.6	15
6.4.10-1	1003.0	7129.3	24928.0	15	7582.0	19563.7	57116.9	15	6430.1	27362.5	90757.6	15
6.4.10-2	797.5	5618.1	15934.7	15	8857.2	24367.8	65591.0	15	6042.5	30837.7	103849.5	15
6.4.10-3	805.9	5299.8	18883.0	15	9668.0	27604.1	60054.4	15	6605.8	33476.7	124421.4	15
6.6.10-1	1052.0	5822.4	19984.1	15	13569.4	31987.9	70455.2	15	9058.4	40925.5	125318.0	15
6.6.10-2	956.8	5955.3	29055.3	15	16273.2	39944.4	151809.1	15	11260.3	50044.5	193209.0	15
6.6.10-3	1243.9	7204.4	25286.5	15	11422.4	25626.3	68280.0	15	9161.1	43958.1	166033.1	15
10.2.10-1	883.7	8550.7	24911.4	15	13000.7	61353.5	105297.1	15	5168.2	39441.7	128707.0	15
10.2.10-2	1512.5	12905.2	38655.8	15	5826.2	26173.5	50920.1	15	7016.8	50663.2	177461.2	15
10.2.10-3	1398.5	12173.1	49605.8	15	22901.7	96580.9	265156.0	15	8166.5	63442.1	253596.7	15
5.5.10-Lamp	568.3	3396.8	18240.8	15	3076.3	8113.6	31719.7	15	11776.9	51956.6	287967.0	15
4.7.10-Coffee	766.3	3585.8	12264.1	15	6329.7	13279.9	33141.1	15	7374.7	35100.8	126433.5	15
11.10.10-TCP	4525.6	26779.9	87720.3	15	160482.1	480235.5	841312.0	15	36427.1	206505.9	483336.1	15

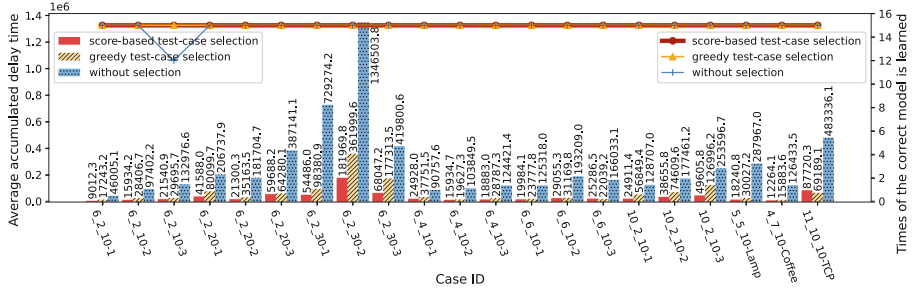
operator, we set $n_{acc} = 8$ and $k = 1$. In the mutation-based selection of test cases, we set the weights with $a = 0.4$, $b = 0.4$, $c = 0.6$, and $d = 0.2$ to calculate the scores.

We refer to our mutation-based testing for DOTAs learning as **Mutation_new**. We also set two baseline methods, **Mutants_checking** and **Heuristic_random_testing**. In the former, we first generate mutants using the method in Sect. 4 and then generate test cases by equivalence checking between the current hypothesis and each mutant as introduced in [23]. Hence, a test case is a timed word showing the violation of equivalence. In the latter, we directly use the test cases \mathbf{T} generated by repeating Algorithm 1 without any mutation or selection. We compare our technique with the two baseline methods. To ensure the comparability of the different techniques, the parameter settings are the same as those mentioned previously.

The experimental results of the three methods are given in Table 1. It shows that the three methods can learn the correct model in all cases except for one failure for **Heuristic_random_testing** on model 6.2.10-3. Our method takes the least number of test cases, actions, and accumulated delay time on all cases, beating the two baseline methods by about an order of magnitude. Among the baseline methods, **Mutants_checking** costs less on average than **Heuristic_random_testing**, but the comparison is highly variable across cases. In order to evaluate the quality of the incorrect learned model for 6.2.10-3, we randomly generated extra 50000 test cases to test the learned model. The passing rate is 99.27%. Additionally, we analyze why **Heuristic_random_testing** failed once but **Mutation_new** did not. As we know, for an EQ, \mathbf{T}_{sel} is a subset of \mathbf{T} .



(a) The comparison on the number of times the correct models learned using different test-case generation methods. (b) The comparison on the number of times the correct models learned using two mutation operators separately and both.



(c) The comparisons on average accumulated delay time in test cases and the number of times the correct models learned using different test-case selection methods.

Fig. 2. Experimental results of the evaluation of improvements.

The found counterexamples for an EQ may be different using two methods, thus leading to different hypotheses which will affect the learning process further.

5.2 Evaluation of Improvements

We continue to use the cases in Table 1 to evaluate the improvements of the three main contributions in the context of learning: heuristic test-case generation method, two special mutation operators, and score-based test-case selection. The following three experiments are conducted:

- E1. Comparison of the heuristic test-case generation and the baseline.
- E2. Comparison of the algorithm with and without the two mutation operators.
- E3. Comparison of the mutation and score-based test-case selection, the greedy test-case selection, and without selection.

E1. Evaluation of the Heuristic Test-Case Generation. The quality of the test cases \mathbf{T} obtained using heuristic test-case generation is critical, as the test cases we execute on the system are selected from \mathbf{T} . In order to evaluate our heuristic test-case generation method, we compare it with a purely random method (randomly select actions and delay times to form timed traces) and the A&T's method (another random testing approach discussed in [6]). In other words, in

the whole testing process, the experiments conducted only differ in the test-case generation method. For A&T’s method, we set values to the parameters in the method as $p_{retry} = 0.9$, $p_{stop} = 0.05$, $l_{infix} = |Q|/2$, and $maxSteps = 2 \cdot |Q|$. For each case, the original test suites generated by the three methods are of the same size. Still, for each case, we learn the models 15 times respectively using each method and observe the times of the correct models are learned. The results are shown in Fig. 2(a). It can be found that our heuristic method performs much better than the other two methods on learning out correct models.

E2. Evaluation of the Two Mutation Operators. As described in Sect. 3.1, during the process of mutation-based testing for DOTAs, we design and adopt two kinds of mutation operators to generate mutants: timed mutation operator and split-location mutation operator. We would like to evaluate the efficiency of the two operators. For each case, we learn the models 15 times respectively using only the timed mutation operator, or only the split-location operator, or both to generate mutants, and observe the times of the correct models are learned. The results, given in Fig. 2(b), shows that although for some cases we are able to learn correct models 15 times using a single mutation operator, using two operators together gives a significant improvement on the rate of learning correct models. Therefore, it is necessary to use both operators in mutation-based testing in the context of DOTAs learning.

E3. Evaluation of the Mutation and Score-Based Test-Case Selection. Our mutation-score-based test-case selection algorithm considers various attributes and guarantees mutation coverage at the same time. We compared the method with a greedy test-case selection method [6] which only guarantees that the test suite selected provides maximum coverage of the mutants. The experiments are conducted differently only in the selection of test cases. Running all cases without any test-case selection procedure is as the baseline. We run each experiment for 15 times on each case and the results are shown in Fig. 2(c). Considering the number of times the correct model is learned, both selection methods performed better than the baseline (this is because the found counterexamples for an EQ are different by different methods, which lead to different hypotheses and will affect the learning process further). On most cases, our selection approach has the least accumulated delay time except for case 6_6_10-3 and case 11_10_10-TCP. However, for these two cases, we can still achieve better results than the greedy test-case selection method by adjusting parameters.

6 Conclusion

We presented a conformance testing approach combining random testing and model-based mutation testing, which can be used for EQs in the active learning of DOTAs. The experimental results show the effectiveness and efficiency of our two learning-friendly mutation operators and several heuristics in the generation and selection of test cases. Since the performance depends on the instantiation of parameters and we set parameters according to our experience, one possible

future work is to determine automatic methods for setting or online adaption of parameters according to the learning scenarios.

References

1. Aarts, F., Kuppens, H., Tretmans, J., Vaandrager, F., Verwer, S.: Improving active Mealy machine learning for protocol conformance testing. *Mach. Learn.* **96**, 189–224 (2013). <https://doi.org/10.1007/s10994-013-5405-0>
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., Brandl, H., Jöbstl, E., Krenn, W., Schlick, R., Tiran, S.: Killing strategies for model-based mutation testing. *Softw. Test. Verification Reliab.* **25**(8), 716–748 (2015). <https://doi.org/10.1002/stvr.1522>
4. 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
5. Aichernig, B.K., Pferscher, A., Tappler, M.: From passive to active: learning timed automata efficiently. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NFM 2020. LNCS, vol. 12229, pp. 1–19. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-55754-6_1
6. Aichernig, B.K., Tappler, M.: Efficient active automata learning via mutation testing. *J. Autom. Reason.* **63**(4), 1103–1134 (2018). <https://doi.org/10.1007/s10817-018-9486-0>
7. Alur, R., Dill, D.L.: A theory of timed automata. *Theoret. Comput. Sci.* **126**(2), 183–235 (1994). [https://doi.org/10.1016/0304-3975\(94\)90010-8](https://doi.org/10.1016/0304-3975(94)90010-8)
8. An, J., Chen, M., Zhan, B., Zhan, N., Zhang, M.: Learning one-clock timed automata. In: TACAS 2020. LNCS, vol. 12078, pp. 444–462. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_25
9. An, J., Wang, L., Zhan, B., Zhan, N., Zhang, M.: Learning real-time automata. *Sci. China Inf. Sci.* **64**(9), 1–17 (2021). <https://doi.org/10.1007/s11432-019-2767-4>
10. An, J., Zhan, B., Zhan, N., Zhang, M.: Learning nondeterministic real-time automata. *ACM Trans. Embed. Comput. Syst.* **20**(5s), 1–26 (2021). <https://doi.org/10.1145/3477030>
11. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Software Eng.* **32**(8), 608–624 (2006). <https://doi.org/10.1109/TSE.2006.83>
12. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
13. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 175–189. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31984-9_14
14. Chow, T.: Testing software design modeled by finite-state machines. *IEEE Trans. Software Eng.* **3**, 178–187 (1978). <https://doi.org/10.1109/TSE.1978.231496>
15. En-Nouaary, A., Dssouli, R., Khendek, F.: Timed Wp-method: Testing real-time systems. *IEEE Trans. Software Eng.* **28**(11), 1023–1038 (2002). <https://doi.org/10.1109/TSE.2002.1049402>

16. Grinchtein, O., Jonsson, B., Leucker, M.: Learning of event-recording automata. *Theoret. Comput. Sci.* **411**(47), 4029–4054 (2010). <https://doi.org/10.1016/j.tcs.2010.07.008>
17. Henry, L., Jéron, T., Markey, N.: Active learning of timed automata with unobservable resets. In: Bertrand, N., Jansen, N. (eds.) *FORMATS 2020*. LNCS, vol. 12288, pp. 144–160. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-57628-8_9
18. Howar, F., Jonsson, B., Vaandrager, F.: Combining black-box and white-box techniques for learning register automata. In: Steffen, B., Woeginger, G. (eds.) *Computing and Software Science*. LNCS, vol. 10000, pp. 563–588. Springer, Cham (2019). https://doi.org/10.1007/978-3-319-91908-9_26
19. Howar, F., Steffen, B., Merten, M.: From ZULU to RERS. In: Margaria, T., Steffen, B. (eds.) *ISoLA 2010*. LNCS, vol. 6415, pp. 687–704. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16558-0_55
20. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib. 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
21. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.* **37**(5), 649–678 (2011). <https://doi.org/10.1109/TSE.2010.62>
22. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. *Formal Methods Syst. Des.* **34**(3), 238–304 (2009). <https://doi.org/10.1007/s10703-009-0065-1>
23. Larsen, K.G., Lorber, F., Nielsen, B., Nyman, U.: Mutation-based test-case generation with Ecdar. In: *ICST Workshops 2017*, pp. 319–328. IEEE (2017). <https://doi.org/10.1109/ICSTW.2017.60>
24. Maler, O., Mens, I.-E.: Learning regular languages over large alphabets. In: Ábrahám, E., Havelund, K. (eds.) *TACAS 2014*. LNCS, vol. 8413, pp. 485–499. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_41
25. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. *J. Autom. Lang. Comb.* **7**(2), 225–246 (2002). <https://doi.org/10.25596/jal-2002-225>
26. Trab, M.S.A., Counsell, S., Hierons, R.M.: Specification mutation analysis for validating timed testing approaches based on timed automata. In: *COMPSAC 2012*, pp. 660–669. IEEE Computer Society (2012). <https://doi.org/10.1109/COMPSAC.2012.93>
27. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. *Softw. Test. Verification Reliab.* **22**(5), 297–312 (2012). <https://doi.org/10.1002/stvr.456>
28. Vaandrager, F.: Model learning. *Commun. ACM* **60**(2), 86–95 (2017). <https://doi.org/10.1145/2967606>
29. Vaandrager, F., Bloem, R., Ebrahimi, M.: Learning mealy machines with one timer. In: Leporati, A., Martín-Vide, C., Shapira, D., Zandron, C. (eds.) *LATA 2021*. LNCS, vol. 12638, pp. 157–170. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-68195-1_13
30. Van Beek, D., Man, K., Reniers, M., Rooda, J., Schiffelers, R.: Syntax and semantics of timed Chi. *J. Symb. Comput.* *JSC* (2005)
31. Vega, J.J.O., Perrouin, G., Amrani, M., Schobbens, P.: Model-based mutation operators for timed systems: a taxonomy and research agenda. In: *QRS 2018*, pp. 325–332. IEEE (2018). <https://doi.org/10.1109/QRS.2018.00045>