Mutation-Based Test-Case Generation with Ecdar

Larsen, Kim G.; Lorber, Florian; Nielsen, Brian; Nyman, Ulrik M.

Published in:
Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2017

DOI (link to publication from Publisher):
10.1109/ICSTW.2017.60

Publication date:
2017

Document Version
Early version, also known as pre-print

10th IEEE International Conference on Software Testing, Verification and Validation Workshops

Mutation-Based Test-Case Generation with Ecdar

Kim G. Larsen, Florian Lorber, Brian Nielsen and Ulrik M. Nyman
Department of Computer Science
Aalborg University
Aalborg, Denmark
{kgl,florber,bnielsen,ulrik}@cs.aau.dk

***Abstract*—Model-based testing is a well-known technique for automating the otherwise tedious process of testing. Test cases are automatically created from a formal model, according to some test criterion which determines when the test suite is complete. In model-based mutation testing, the test criterion is defined via faulty models, called mutants, which are used to create test cases that specifically target the modeled faults. To be able to reveal timing related faults, timed automata can be used as the test model. While model-based mutation testing has already been applied to timed automata, we show how to implement the technique more efficiently with the tool Ecdar, which belongs to the well-known UPPAAL tool family. The tool is used to perform an unbounded conformance check between the correct specification and the mutants, based on a notion of timed refinement. If a mutant does not refine the specification, Ecdar creates a strategy for reaching the non-conformance, which can be used as an adaptive test case. We applied the procedure to the timed automata model of a car alarm system, which was used in the previous approach of model-based mutation testing for timed automata, and compare the two approaches based on the results.**

I. Introduction

In the last decades testing has proven to be a popular technique for the verification of industrial systems. Via testing one either detects bugs, or raises the confidence that the system under test (SUT) works as expected. However, manual test-case generation is an error prone procedure which does not comply to safety standards. Thus, the trend went towards automated test-case generation methods, like model-based testing [24]. In model-based testing, a formal model is produced from the requirements and used to derive test cases that cover the interesting aspects. These test cases are usually produced according to specified coverage criteria, like for instance state coverage, where the produced test suite is intended to reach all states in the model.

In model-based mutation testing [3], [7], the coverage criteria is given via a set of fault models, called *mutation operators*. Each fault model specifies a certain type of fault that might occur in a model, like for instance an off-by-one fault in the enabling condition of a transition. By applying these mutation operators to the formal model a set of faulty models, called *mutants*, is generated. One can distinguish between *equivalent mutants*, where the mutation did not introduce any faulty behavior, and *non-equivalent mutants*, which do not conform to the correct specification anymore. For each non-equivalent mutant, a test case leading to the conformance violation is generated. If a (deterministic) SUT contains a bug that correlates to the fault in a mutant, the test generated for that mutant is guaranteed to detect it.

The technique can be applied to various different formalisms for the test model, and has, among others, already been applied to UML state machines [3], probabilistic finite state machines [16], and timed automata [5]. The work presented in this paper builds on the previous work on timed automata [5]. Timed automata extend traditional state machines by a means for specifying timing behavior, i.e., they are extended by clock variables which can measure the progress of time. By applying model-based mutation testing to timed automata, the generated test suite is able to specifically target timing faults in the tested systems, such as delayed outputs.

In the existing approach [5] SMT-solving and bounded model-checking were used to perform a bounded conformance check between the specification and the mutant. In this paper we propose to perform the conformance check with Ecdar [11], which allows an unbounded conformance check, provides a significant speedup compared to the previous approach and enables the generation of adaptive test strategies, which produce fewer *inconclusive* test verdicts than straightforward methods. We will present how the tool can be called for a refinement check between a specification and its mutant, and how the conformance check differs from the previous one. Then, we will show how the strategies produced by Ecdar can be used as adaptive test cases, which will (if possible) lead to the mutation despite the choices made by the SUT. Finally, we will compare the two approaches based on the timed automata model of a car alarm system, that was used to evaluate the previous approach, and discuss the pros and contras of the approaches.

The structure of the paper is as follows: first, in Section II we will present some preliminaries, including the definition of timed automata, the tool Ecdar and the workflow of model-based mutation testing. Then, in Section III, we will discuss some related work. Next, in Section IV we will present the proposed approach using Ecdar, where we discuss how the conformance check can be applied in Ecdar, and how the produced strategies can be used as test cases. Then, in Section V we will show our evaluation and discuss the advantages and disadvantages of the new approach compared to the existing one. Finally, we will conclude the paper and give an outlook of future work in Section VI.

978-1-5090-6676-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSTW.2017.60

II. Preliminaries

A. Timed Automata

Timed automata [6] are extended finite state machines, which contain clock variables for measuring the progress of time. Let $\mathcal{X}$ be a finite set of such *clock* variables. A clock *valuation* $v(x)$ is a function $v : \mathcal{X} \to \mathbb{R}_{\geq 0}$ assigning a real value to every clock $x \in \mathcal{X}$. We denote by $\mathcal{V}$ the set of all clock valuations and by $\mathbf{0}$ the valuation assigning 0 to every clock. For a valuation v and $d \in \mathbb{R}_{\geq 0}$ we define $v+d$ to be the valuation $(v+d)(x) = v(x) + d$ for all $x \in \mathcal{X}$. For a subset δ of $\mathcal{X}$, we denote by $v[\delta]$ the valuation such that for every $x \in \delta$, $v[\delta](x) = 0$ and for every $x \in \mathcal{X} \setminus \delta$, $v[\delta](x) = v(x)$. A *clock constraint* φ is a conjunction of predicates of the form $x \sim n$, where $x \in \mathcal{X}$, $n \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. Given a clock valuation v, we write $v \models \varphi$ when v satisfies φ.

*Definition 1 (*TA*):*

A timed automaton A is a tuple $(\mathcal{Q}, \hat{q}, \Sigma, \mathcal{X}, \mathcal{I}, \mathcal{G}, \mathcal{T})$, where

1) $\mathcal{Q}$ is a finite set of *locations* and $\hat{q} \in \mathcal{Q}$ is the *initial* location;
2) Σ is a finite set of observable actions;
3) $\mathcal{X}$ is a finite set of *clock* variables;
4) $\mathcal{I} : \mathcal{Q} \to LI$ is a mapping from locations to *location invariants*, where each location invariant $li \in LI$ is a conjunction of constraints of the form $true$, $x < n$ or $x \leq n$, with $x \in \mathcal{X}$ and $n \in \mathbb{N}$;
5) $\mathcal{G}$ is a set of *transition guards*, where each guard is a conjunction of constraints of the form $x \sim n$, where $x \in \mathcal{X}$, $\sim \in \{<, \leq, =, \geq, >\}$ and $n \in \mathbb{N}$;
6) $\mathcal{T} \subseteq \mathcal{Q} \times \Sigma \times \mathcal{G} \times \mathcal{P}(\mathcal{X}) \times \mathcal{Q}$ is a finite set of *transitions* of the form $(q, \alpha, g, \delta, q')$, where
   a) $q, q' \in \mathcal{Q}$ are the *source* and the *target* locations,
   b) $\alpha \in \Sigma$ is the transition *action*,
   c) $g \in \mathcal{G}$ is the transition guard,
   d) $\delta \subseteq \mathcal{X}$ is the subset of clocks to be *reset*;

The *semantics* of a TA A is given by the *timed transition system* $[[A]] = (S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma, T)$, where

1) $S = \{(q, v) \in \mathcal{Q} \times \mathcal{V} \mid v \models \mathcal{I}(q)\}$;
2) $\hat{s} = (\hat{q}, \mathbf{0})$;
3) $T \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation consisting of *timed* and *discrete* transitions such that:
   a) *Timed transitions (delay):* $(q, v) \xrightarrow{d} (q, v+d) \in T$, where $d \in \mathbb{R}_{\geq 0}$, if $v + d \models \mathcal{I}(q)$,
   b) *Discrete transitions (jump):* $(q, v) \xrightarrow{\alpha} (q', v') \in T$, where $\alpha \in \Sigma$, if there exists a transition $(q, \alpha, g, \delta, q')$ in $\mathcal{T}$, such that: (1) $v \models g$; (2) $v' = v[\delta]$ and (3) $v' \models \mathcal{I}(q')$;

We denote by $(q, v') = (q, v)$ after d the state reached from (q, v) by a delay of d, and by $(q', v') = (q, v)$ after α the state reached by performing an α transition from (q, v). We denote by $(q, v) \xrightarrow{d} \not\rightarrow$ that the delay of d is not possible in (q, v).

A *run* ρ of an TA A is a finite sequence of alternating timed and discrete transitions of the form $(q_0, v_0) \xrightarrow{d_1} (q_0, v_0 + d_1) \xrightarrow{\tau_1} (q_1, v_1) \xrightarrow{d_2} \cdots \xrightarrow{d_n} (q_{n-1}, v_{n-1} + d_n) \xrightarrow{\tau_n} (q_n, v_n)$, where $q_0 = \hat{q}$, $v_0 = \mathbf{0}$, $\tau_i = (q_{i-1}, \alpha_i, g_i, \mathcal{X}_{rst(i)}, q_i) \in \mathcal{T}$ and $\alpha_i \in \Sigma$.

A TA is called *deterministic*, iff $(q, v) \xrightarrow{\alpha} (q', v') \wedge (q, v) \xrightarrow{\alpha} (q'', v'') \implies (q', v') = (q'', v''), \alpha \in \Sigma \cup \{\epsilon\}$.

a) Timed I/O Automata: In Timed I/O Automata (TIOA), the set of actions Σ is split into two disjoint sets of input actions Σ_I and output actions Σ_O. Thus, a TIOA A_{io} is a tuple $(\mathcal{Q}, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{X}, \mathcal{I}, \mathcal{G}, \mathcal{T})$. Figure 1 illustrates a timed I/O automaton for which we have $\mathcal{Q} = \{q_0, q_1, q_2, q_3\}$, $\hat{q} = q_0$, $\Sigma_I = \{c?\}$, $\Sigma_O = \{a!, b!\}$, $\mathcal{X} = \{x\}$, $\mathcal{I} = \{q_2 \to x \leq 5\}$, $\mathcal{G} = \{x \geq 3\}$ and $\mathcal{T} = \{(q_0, a!, true, \{x\}, q_1), (q_0, b!, true, \{x\}, q_2), (q_1, c?, x \geq 3, \{\}, q_2), (q_2, a!, x \geq 4, \{\}, q_4)\}$. The semantics of timed I/O automata is given by timed input/output transition systems $[[A_{io}]] = (S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma_I, \Sigma_O, T)$, where discrete transitions are split into input actions, hereafter denoted by $(q, v) \xrightarrow{i?} (q', v')$, and output actions, denoted by $(q, v) \xrightarrow{o!} (q', v')$.

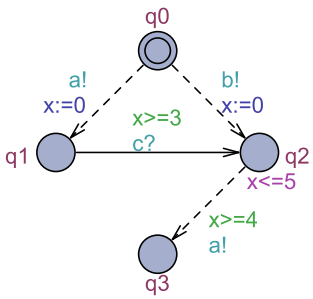

Fig. 1. A timed I/O automaton.

We denote by $out(q, v)$ the set of output actions that appear in any of the transitions leaving (q, v).

A TIOA A is input-enabled, iff in its underlying TIOTS for all locations (q, v) and all inputs $i?$, there exists a transition $(q, v) \xrightarrow{i?} (q', v')$.

A TIOA A satisfies the *independent-progress* condition, iff in its underlying TIOTS for all locations (q, v) either $\forall d \geq 0 : (q, v) \xrightarrow{d} (q, v')$ or $\exists d \geq 0 : (q, v) \xrightarrow{d} (q, v') \wedge (q, v') \xrightarrow{o!} (q', v'')$, for some $q'' \neq q'$. Thus, either the system can stay in a location forever, or there exists an output transition leaving the location. Consequently, the system will never block the progress of time.

In the presented work we only consider deterministic and input-enabled TIOA satisfying the independent-progress condition. The automaton illustrated in Figure 1 is deterministic and satisfies the independent-progress condition. However, the transitions for making it input enabled were omitted for keeping it simple.

b) Ecdar: The tool Ecdar was developed on top of UPPAAL-TIGA, implementing the timed interface theory presented by David et al. [11]. It works on timed I/O automata, where inputs are defined as controllable and outputs are defined as uncontrollable. The tool implements refinement checks, consistency checks, composition of models and a

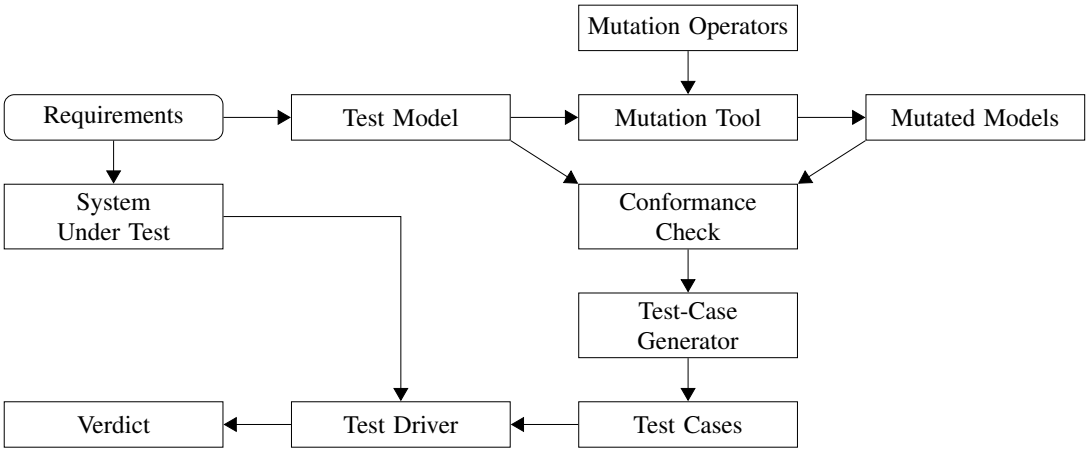

Fig. 2. Model-based mutation testing workflow.

quotient operator. In the context of this paper, we will use the refinement check for checking whether or not a mutant refines the correct model.

The notion of refinement used by Ecdar and introduced by David et al. [11] closely corresponds to the timed input-output conformance [19], and they coincide for input-enabled models. It is defined via the refinement between the underlying timed input/output transition systems:

Definition 2 (Refinement):

A TIOTS $I(S_I, \hat{s}_I, \mathbb{R}_{\geq 0}, \Sigma_I, \Sigma_O, T_I)$ refines a TIOTS $S(S_S, \hat{s}_S, \mathbb{R}_{\geq 0}, \Sigma_I, \Sigma_O, T_S)$, written $I \leq S$, iff there exists a binary relation $R \subseteq \mathcal{Q}_I \times \mathcal{Q}_S$ containing $(\hat{s}_I, \hat{s}_S)$ such that for each pair of states $(s_I, s_S) \in R$ we have:

1) whenever $s_S \xrightarrow{i?} s'_S \in T_S$ for some $s'_S \in S_S$ then $s_I \xrightarrow{i?} s'_I \in T_I$ and $(s'_I, s'_S) \in R$ for some $s'_I \in S_I$
2) whenever $s_I \xrightarrow{o!} s'_I \in T_I$ for some $s'_I \in S_I$ then $s_S \xrightarrow{o!} s'_S \in T_S$ and $(s'_I, s'_S) \in R$ for some $s'_S \in S_S$
3) whenever $s_I \xrightarrow{d} s'_I \in T_I$ for some $d \in \mathbb{R}_{\geq 0}$ then $s_S \xrightarrow{d} s'_S \in T_S$ and $(s'_I, s'_S) \in R$ for some $s'_S \in S_S$

Consequently, a TIOA A_I refines a TIOA A_S if the corresponding TIOTS $[[A_I]]$ refines the corresponding TIOTS $[[A_S]]$.

Thus, if an implementation refines a specification, then 1) for every input that can be applied to the specification, that input needs to be enabled in the implementation as well, 2) every output that can be produced by the implementation needs to be produced by the specification as well and 3) the implementation can only delay time if the specification can. Within the presented approach, the refinement check will be applied between the original specification and the slightly altered mutant, to check whether the mutant still refines the original specification. The mutated automaton from Figure 3 does not refine its original specification from Figure 1, as it allows an output at time $x = 3$ in location q_2. However, the specification refines the mutant, as the mutant covers all outputs allowed by the specification.

In the case of non-refinement, Ecdar is able to use UPPAAL-TIGA to produce strategies leading to the non-refinement. The strategy is produced according to a two player timed game on the product of the correct specification and the mutant, where the input transitions are controlled by the player, and the outputs are controlled by the opponent. The goal of a strategy is to reach some set of goal-states, denoted by $K \subseteq \mathcal{Q} \times \mathbb{R}_{\geq 0}$, in the TIOTS of a TIOA which in our case is given by the states that reveal non-conformance. A run of length n through a TIOA is a *winning run* if $\exists k.0 \leq k \leq n \wedge (q_k, v_k) \in K)$. We call the set of all winning runs and all prefixes of winning runs WR. A strategy f is a function guiding the player of a timed game towards the winning states K. A strategy can advise the player to either trigger a specific input, or to wait, denoted by λ. A state-based strategy f for a TIOTS $(S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma_I, \Sigma_O, T)$ is a partial function from S to $\Sigma_I \cup \{\lambda\}$. A run is said to be *supervised* by the strategy f, if for every step $(q_k, v_k) \xrightarrow{d_{k+1}} (q_k, v_k + d_{k+1})$ we have $\forall d' \in [0, d_{k+1}).f(q_k, v_{k+d'}) = \lambda$ and for every step $(q_k, v_k) \xrightarrow{\tau_k} (q_{k+1}, v_{k+1})$ we have $\alpha_k \in \Sigma_O \vee (\alpha_k \in \Sigma_I \wedge f(q_k, v_k) = \alpha_k)$. A strategy is a *winning strategy* on a TIOA A for the goal states K, if all supervised runs in A are in WR. A strategy is a *cooperative strategy* if it contains at least one winning run and every supervised run $r \notin WR$ contains two prefixis $p1, p2$ of length $k \geq 0$ and $k + 1$, so that $p \in WR$, $p1 \notin WR$ and $\alpha_k \in \Sigma_O$. Thus, a cooperative strategy can guide the player to the goal states, if the opponent cooperates.

B. Model-based Mutation Testing

Model-based mutation testing is a combination of model-based testing [24] and mutation testing [17]. However, unlike most classical mutation testing approaches, the technique is not used for assessing the quality of an existing testsuite, but to create a new test suite based on the tests needed to detect a given set of fault models. The general workflow of model-based mutation testing is depicted in Figure 2. The

workflow starts with the requirements, which are used (usually by individual teams) to create both the SUT and a formal test model. The test model is then processed by a mutation tool, which will insert different kinds of faults into the model. The types of faults are defined via a set of mutation operators, which may include changing the target/goal locations of transitions or the enabling conditions of guards. This produces a set of faulty test models, called mutants. These mutants usually contain one fault each, and are thus called *first-order mutants*. It is however also possible to insert multiple faults and thus produce *higher-order mutants*. Figure 3 illustrates a mutant of the TIOA illustrated in Figure 1, where an off-by-one fault was introduced into one of the guards, enabling the corresponding transition early.

After the mutation is done, the core of the test-case generation is performed, which is mainly the conformance check between the correct specification and each of the mutants. In case the introduced fault did not insert any detectable faulty behavior, the mutant is classified as equivalent with respect to the used conformance relation and discarded. Otherwise, the conformance check produces a counterexample to the conformance, i.e., a trace leading from the initial location to the transition where the mutation is revealed by the faulty behavior of the mutant. This counterexample is then turned into a test case, which means that it is extended by verdicts, and possibly made more adaptive. E.g., in the previous approach [5] the test cases were made time-adaptive, by using the concrete times from the counter example to calculate timing constraints for the test-case execution which guarantee taking the same path through the timed automaton even if the test-execution slightly deviates from the concrete times. Finally, each created test is executed on the SUT and either reveals a bug, assigning the verdict *fail* to the test execution, or runs through correctly, assigning the verdict *pass*. In special cases, when the SUT is allowed to choose between several outputs, or has a large time frame during which an output may occur, the system may choose an output (or time for an output), which is correct but not the one expected by the test case. In such cases, a non-adaptive test case may not reach its test purpose, i.e., the mutation, anymore, but also did not reveal a bug. In this case, most testing frameworks assign the verdict *inconclusive*. Depending on the test-case generation method, the produced test cases may be more or less adaptive. Non-adaptive test cases only cover the trace of the concrete counterexample, while adaptive test-cases contain additional information, in case the specification produces a different output. Thus, adaptive test-cases can steer the test-case execution back towards the goal of the test case, if a path back exists.

If the SUT contains a bug that corresponds to any of the mutations, the produced test suite is guaranteed to detect it, or, in certain cases, assign the inconclusive verdict. Additionally, the corresponding mutant can serve as an aid for detecting the bug, as it shows the location and type of the bug.

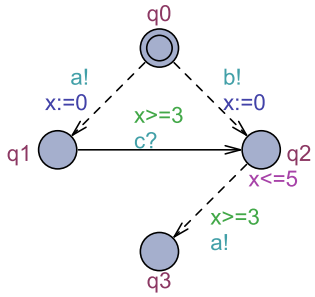

Fig. 3. A mutant of the timed I/O automaton illustrated in Figure 1.

III. Related Work

For general information on model-based testing and mutation testing, we refer to the corresponding surveys [24], [17]. Mutations have already been used for the generation of test cases both on the code level, see e.g. the work by Offutt and Untch [22], and on the model level, which will be discussed shortly. Both approaches face a lot of common problems, like the high number of mutants, or their expansive execution, and consequently, there is also an overlap in possible solutions. For instance, Xavier et al. [14] recently proposed a methodology for generating *featured mutant models*, which are models containing all possible mutants of a transition system within one model. This both enables higher-order mutants and a very efficient test execution, where the state-space only needs to be explored once. While they use the approach for validating existing test suits, the approach is model-based and could be applied for the test-case generation as well.

Model-based mutation testing was first introduced by Budd and Gobal [8] for predicate-calculus specifications. Later on, it was adapted to various formalisms, like probabilistic finite state machines [16], action systems [2] specifications and UML state machines [3]. The approaches presented so far mainly focused on testing the functional behavior of the SUT, while the presented approach uses timed automata to be able to target timing faults as well.

Timed automata have been used for test-case generation several times. The tools UPPAAL Cover [15] and UPPAAL Tron [20] were specifically developed to provide online and offline testing for timed automata. They can be used with various different testing criteria, however not with a fault-based approach. Krichen and Tripakis [18] proposed a testing framework for edge coverage, location coverage and state coverage for timed automata. Their tool, TTG, can process non-deterministic timed automata, and produce deterministic testers. Wang et al. [26] proposed an algorithm and tool support for a conformance check between timed automata, allowing non-deterministic but determinizable timed automata. However, they never applied the approach to test-case generation. There also have been several approaches using game strategies for testing of timed systems: David et al. [12] uses UPPAAL-TIGA for creating winning strategies according to manually designed test purposes. These strategies were

then translated into traditional test cases, loosing parts of their flexibility. In other work David et al. [9] also used the strategies directly for test-case execution. However, since they were still created from test purposes instead of mutants, the test execution varies from the algorithm that will be presented in Section IV. Additional work by David et al. [10], [13] extended their approach to partially observable systems and systems with concurrent behavior. Again, those approaches were based on manual test purposes and created test strategies that contain the expected behavior, while our test-strategies in their last step reflect the behavior of the mutant, and thus need a different test-execution.

Mutation of timed automata has so far been performed by AbouTrab et al. [1], Nilsson et al. [21] and Aichernig et al. [5]. AboutTrap et al. used the mutants for classical mutation testing, i.e., to assess the quality of existing test suites, not for the generation of new tests. Nilsson et al. used mutations of timed automata with tasks for test-case generation, however the mutation operators focused on the tasks, rather then on the clock resets and guards. The work by Aichernig at al. is the most related work to the presented approach, and will be the work that we compare our results with during the evaluation. They proposed model-based mutation testing for timed automata, and implemented the approach in the tool MoMuT::TA. The tool provides a mutator for timed automata, the conformance check where *timed input-output conformance (tioco)* [19] is used a the conformance relation, and in its initial version also a translation from a counterexample to time adaptive test cases. They performed the conformance check via bounded model-checking, using SMT solving. Thus, contrary to the presented approach that uses UPPAAL's zone based algorithms, they only perform a bounded conformance check. Additionally, the test cases they produced are less adaptive than the ones created by the presented approach. A closer comparison of the approaches can be found in Section V.

IV. Model-based Mutation Testing with Ecdar

A. Refinement Check

While Ecdar does not perform mutations on models[1], it provides full capabilities for the conformance check between the specification and the mutants and the generation of the strategies used for testing. Conformance is expressed as refinement, using the refinement relation introduced in Definition 2. For input-enabled, deterministic models, this refinement relation corresponds to tioco-conformance. In our approach, input enabledness is ensured via demonic completion [25] for the specification and via angelic completion [23] for the mutants. Demonic completion means that for each undefined input we create a transition with that input, which leads to a universal state. Thus, if the specified area is left, every future behavior becomes possible. Traces leading to the universal state will never yield a counterexample to the refinement, as everything refines the universal state. This fits our needs, since most *ioco*-based testing approaches [25] are only interested in testing the specified parts of the system, allowing the usage of under-specified models that do not cover all functionality. These under-specified models allow the implementation to react to additional inputs, while they must produce the correct outputs for the inputs specified by the model. Angelic completion creates self-loops for undefined inputs, indicating that those inputs are simply ignored. While a mutant is allowed to only define partial behavior of the complete specification, it is not allowed to block any input.

To apply the refinement check between a TIOA specification S and a TIOA mutant M, the call to the Ecdar verifier is simply $refinement : M <= S$. The problem is solved as a timed game, where the mutant is seen as the opponent triggering the uncontrollable actions. The goal for the specification is to find a strategy that reveals the non-refinement, regardless of the outputs chosen by the mutant. If such a strategy exists, it can be stored in a file or printed to the console.

Listing 1. The strategy for detecting the mutant of Figure 3.

```
- State: ( S.q0, M.q0)
    When you are in true,
      take M.q0 -> M.q2 {, b!, x:=0}
- State: (S.q1, M.q1)
    While you are in (S.x<3),
      wait
    When you are in (S.x==3),
      take S.q1 -> S.q2 {x>=3, c?,}
- State: (S.q2, M.q2)
    While you are in (S.x<3),
      wait
    When you are in (S.x==3),
      take M.q2 -> M.q3 {x>=3, a!,}
```

A strategy stored by Ecdar consists of a list of pairs of states, and the corresponding rules for each pair. A rule may either be a delay rule R_λ or an action rule R_α and each pair may be linked to several of both. Delay rules consist of a timing condition ϕ_δ and the command to wait, *while* the timing condition holds true. Action rules consist of a timing condition ϕ_α and the transition that shall be taken, *as soon as* the timing condition holds. Listing 1 gives an example strategy produced for the specification S and the mutant M, as depicted in Figures 1 and 3. While both automata are in the initial location there are no timing constraints, thus there is no need to wait and the tuple contains only an action rule. Note, that the b transition is uncontrollable and thus triggered by the mutant, while the action rule associated to $(S.q_1, M.q_1)$ is controllable, and thus associated to the specification. The transitions associated to the mutant are just kept in the strategy to keep it complete. During the actual test execution, the SUT will trigger the uncontrollable actions and thus they are not restricted by the strategy. Note that the clocks of both specifications may be used in the timing conditions, and that they also may be compared among each other.

Contrary to non-adaptive test cases that usually generate inconclusive verdicts if the test purpose can not be directly

[1]In our experiments we used mutants generated with MoMuT::TA.

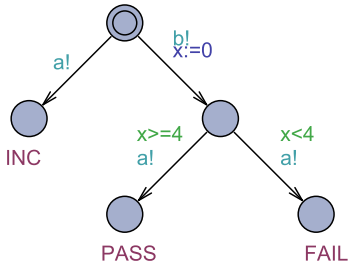

Fig. 4. A non-adaptive testcase for revealing the mutant from Figure 3.

reached, these strategies allow a far more flexible test-case execution. Consider the automaton presented in Figure 3 and the non-adaptive testcase from Figure 4, that would be generated by MoMuT::TA from the shortest counterexample. The test case reveals the mutation, if the SUT produces a $b!$ output in the initial location, while the test-case execution is aborted with an inconclusive verdict if an $a!$ is produced. The strategy provided by Ecdar is given in Listing 1. The action rule leaving $(S.q_0, M.q_0)$ is uncontrollable, and suggests that the mutant takes the $b!$ transition. However, during the test execution, the mutant will be replaced by the SUT, which has the free choice between producing an $a!$ or a $b!$. If it decides to produce an $a!$, this is no problem, as the test strategy contains instructions for q_1, thus guaranteeing the test execution to reach the mutation. However, it needs to be mentioned that there exist systems for which a winning strategy can not be built. Consider the $c?$ transition from q_1 to q_2 did not exist: in that case, there exists no winning strategy and if the SUT decides to produce an $a!$ our test-case execution can only assign an inconclusive verdict as well. Thus, we distinguish three types of outcomes of the refinement check. The output may be a winning strategy, i.e., the mutation will be reached regardless of the decisions of the SUT, a cooperative strategy [12], i.e., the mutation will be reached if the system produces the right outputs, or no strategy, if the mutant refines the original model. In case we produce a cooperative strategy in a cyclic automata, if the SUT produces the wrong output during the test execution, the strategy leads back to the initial location, and tries to reach the mutation again.

An additional advantage of Ecdar is the support for higher-order mutants, i.e. mutants with multiple faults inserted. The produced strategies will lead to the refinement violation that is the easiest to reach, just like MoMuT::TA would. However, if the test execution can not reach this goal, e.g., due to uncontrollable choices, like the choice between $a!$ and $b!$ in Figure 1, the strategies may alternatively lead to the refinement violation caused by another fault. At the moment we did not further investigate this topic, yet we plan to perform experiments with higher-order mutations as future work.

B. Strategy-driven Test Execution

The strategies produced by Ecdar only contain parts of the information needed for their execution by a test-driver. To effectively use them as test cases, the original specification

Algorithm 1 Test-Case Execution

Input: SUT, Strategy, Spec, Mutant, Bound
Output: Pass / Fail / Inconclusive

verdict = {}
step = 0
$((q_s, v_s), (q_m, v_m)) = ((\hat{q}_s, \mathbf{0}), (\hat{q}_m, \mathbf{0}))$
while $verdict = \{\}$ **do**
  if $strategy(q_s, q_m) = \emptyset \vee step = bound$ **then**
    verdict = inconclusive
    break
  end if
  for $R_\lambda \in strategy(q_s, q_m)$ **do** \\ delay
    while $sat\{\phi_\delta\}$ **do**
      wait and continuously check SUT for outputs
      if SUT produces output **then**
        break
      end if
    end while
    $d =$ waited time
    if $(q_s, v_s) \overset{d}{\not\rightarrow}$ **then**
      return fail
    end if
    $(q_s, v_s) = (q_s, v_s)$ after d
    $(q_m, v_m) = (q_m, v_m)$ after d
    if SUT produces output o **then** \\ uncontrollable
      if $o \in out(q_s, v_s)$ **then**
        if $o \in out(q_m, v_m)$ **then**
          $(q_s, v_s) = (q_s, v_s)$ after α
          $(q_m, v_m) = (q_m, v_m)$ after α
        else
          return pass
        end if
      else
        return fail
      end if
    end if
  end for
  for $R_\alpha \in strategy(q_s, q_m)$ **do** \\ controllable
    if $sat\{\phi_\alpha\}$ **then**
      trigger input α on SUT
      $(q_s, v_s) = (q_s, v_s)$ after α
      $(q_m, v_m) = (q_m, v_m)$ after α
    end if
  end for
  step = step + 1
end while

and the mutant need to be provided as well. While the strategy provides the delays and transitions that need to be taken in order to reach the mutation, the original specification is needed to validate all occurring outputs, and the mutant is needed to detect whether an incorrect output of the SUT corresponds to the mutant, or whether we just revealed another bug. Thus, both the specification and the mutant need to be simulated during the test-case execution, so their current locations and clock values can be assessed by the test adapter. In case the strategy is a cooperative strategy, i.e., not a winning strategy, it may produce a loop, trying to reach the mutation even though it might not be reachable. For such cases, we need to provide an upper bound for the test-case execution. Algorithm 1 gives an abstract overview on how the test execution of a strategy, extended by the specification and the corresponding mutant, works. We use the variables (q_s, v_s) and (q_m, v_m) to keep track of the current states of the TIOTS of both the specification and the mutant and initialize them with their initial locations. Then, as long as we did not yet assign any verdict, we first check whether the upper bound is reached, or whether we reached a state tuple which is not specified by the strategy. If that happened, it means that we reached a state from which the test goal is unreachable, and we assign the verdict inconclusive. Otherwise, we look up the delay and action rules for the current locations of the specification and the mutant.

While the condition of one of the delay rules is satisfied, we wait. During waiting, we continuously check the SUT for outputs. If it produces an output, we break the waiting. The waited time is stored in the delay variable d. If the delay rule required to wait longer than the current location invariant of the specification allows (such strategies are for instance produced by mutants were the invariant is violated) and the SUT did not interrupt the waiting with an output, we return the verdict *fail*, otherwise we update the current state variables according to the delay. If during/after waiting an output was observed, there are several possibilities: if the output is correct according to the specification, there are two choices: either the output is also correct according to the mutant, in which case we can update the current states and go on with the next step or the output is correct according to the specification, but the mutant required a different output. In that case we reached the mutation and the SUT behaved according to the specification. Consequently, we can terminate the test execution and return the verdict pass. If the output was not correct according to the specification, a bug was detected and we issue the verdict *fail*. Note that at this point, one could be more fine grained, and check whether the mutant would have allowed that output. If so, we know that we revealed that particular mutation in the SUT, instead of detecting just any bug.

After the outputs of the SUT are processed, we search for enabled action rules. Due to the deterministic nature of the strategies, only one rule is enabled at a time. The action rule corresponds to a controllable action which is simply executed on the SUT. Additionally the current states of the specification and the mutant are updated.

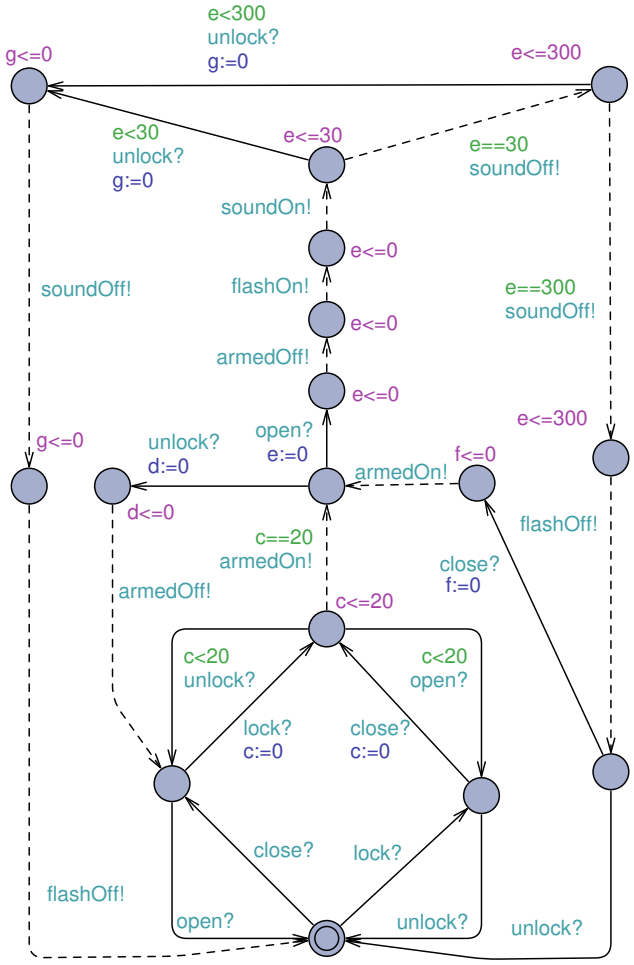

Fig. 5. The TIOA of the car alarm system.

The presented algorithm was kept as simple as possible. Thus, it does not differentiate between catching any bug, and catching the bug specified by the mutant, which would require adding a fourth verdict. We did however already pointed out what would need to be changed, in order to do this differentiation. The big advantage in that would be the aid in fault localization. As already discussed in previous work [4], selecting a set of model mutants that directly correspond to the faulty behavior of an incorrect SUT may aid the programmer both with regards to the location of the bug, as well as with regards to the type of the bug.

V. Evaluation

A. Car Alarm System

We evaluate our approach by comparison to the results of the existing model-based mutation testing approach for timed automata [5]. They applied the technique to a car alarm system, modeled as a timed automaton with inputs and outputs. Basically, the car alarm system allows as inputs the unlocking, locking, closing and opening of the cars door, and prompts

as outputs the signals for arming, unarming, and turning the sound and flash alarms on and off. We use the same model and the same set of mutants, even though we had to make several small adjustments:

- Instead of keeping the mutant and the specification in separate files, the specification was appended to each of the mutants as a separate template.
- All synchronization channels are transformed to be broadcast channels.
- All input transitions are marked as controllable, all output transitions as uncontrollable.
- In the system declaration, the specification and the mutant are defined as IO automata, including the specification of their inputs and outputs.

The TIOA of the car alarm system is illustrated in Figure 5. Uncontrollable transitions are marked via dotted lines. The transitions for making the model input enabled are omitted for presentational purposes.

We took the mutants that were used for the original study, adjusted them as already mentioned, and applied the refinement checks. For the 1075[2] mutants of the car alarm system specification, Ecdar found 677 counterexamples to the refinement and produced the corresponding strategies. 397 mutants refined the original specification, which includes 226 inconsistent and 39 non-deterministic mutants that could not be processed. Inconsistent mutants violate properties like for example the 'independent progress' property, as the mutation may remove an output from a time restricted location, allowing the mutants to block the progress of time. These mutants do not correspond to real faulty implementations, and are thus not relevant for the test-case generation anyway. The refinement check and the creation of the strategies took 164.5 seconds.

We also developed a test driver according to Algorithm 1 and applied the test suite to the faulty Java implementations used in [5] to see whether the test cases produced by the model mutants are able to detect corresponding faults in real implementations. We were able to detect every fault, and thus achieved a 100% mutation score. Table I summarizes the previous and the new results. While the number of test cases is in the same order of magnitude for both approaches, and both approaches were able to detect all faulty Java implementation, the runtime of the test-case generation is significantly faster for the new approach with Ecdar, giving a speedup factor of over 30 for the car alarm system. Due to the fact that some mutants could not be processed, Ecdar produces a smaller test suite. Since the test suite was still able to capture all faulty implementations, the smaller size actually provides a small advantage, as it reduces the test-case execution time.

The car alarm system is modeled in a very restrictive way. The guards of input and output transitions are non-overlapping, to ensure that for each location at any point of time only either inputs or outputs are enabled. Additionally the model does not contain any uncontrollable choices. Thus, also via non-adaptive test-case generation no test cases with inconclusive verdicts are generated. However, if the restrictions on the inputs in the model are weakened, 117 of the non-adaptive test-cases contain traces leading to inconclusive verdicts, while none of the adaptive test-cases would.

[2]24 of the original 1099 mutants were discarded, as two mutation operators produced duplicated mutants, i.e., mutants that were equivalent among each other, but not necessarily to the original specification, where we only kept one set.

B. Pros and Cons

The results presented in the previous subsection show that the implementation of model-based mutation testing via Ecdar is very efficient with respect to runtime, compared to the previous approach based on bounded model-checking and SMT solving. However, both approaches have their individual benefits, which are discussed below:

- *Runtime.* In terms of runtime, Ecdar definitely beats MoMuT::TA. While this may partially be based on the maturity of the tools, the different programming language used, and probably other incomparable factors, it still indicates that bounded model-checking can not compete with the dedicated zone-based symbolic on-the-fly algorithm implemented in Ecdar.
- *Efficiency.* Both test suites were able to detect all of the faulty Java implementation, and their size was in the same order of magnitude. Thus, in terms of efficiency, the two approaches seem to be almost equivalent.
- *Expressiveness.* The two approaches allow a rather different set of model-elements. While the tool ::TA is restricted to classical timed automata, it allows any type of expression that can be processed by the used SMT solver in the guard, including negation and disjunction. Ecdar on the other hand allows C-like functions, the use of data variables and urgent/committed locations. However, excessive use of data variables in the model might drastically increase the runtime as the states would be enumerated explicitly.
- *Complexity of test-case execution.* The proposed test-case execution algorithm is more complex than the previous one, as it needs to simulate both the specification and the mutant in the background, in order to determine the next states (which are needed to consult the strategy), and to classify the outputs of the SUT. The previous approach simply provided a trace leading to the goal state, attached with the verdicts for the individual outputs.
- *Adaptiveness of test-case execution.* While the test-cases described in [5] are time adaptive, and can vary the time of the inputs, according to the timing of previous outputs, they will produce an 'inconclusive' verdict if the SUT produces different (but correct) outputs than expected by the test case. The strategies produced by Ecdar are able to steer the test-case execution back to the intended path.
- *Non-deterministic mutants.* In the presented study, all non-deterministic mutants were simply neglected, as they do not conform to the restrictions by Ecdar, as some of the theories implemented by Ecdar only work for deterministic automata. In the previous work, only the

	# Mutants	# Tests	TCG Time [sec]	Mutation Score
MoMuT::TA	1099	628	3 798	100%
Ecdar	1074	677	164.5	100%

TABLE I
A COMPARISON OF TEST-CASE GENERATION RESULTS.

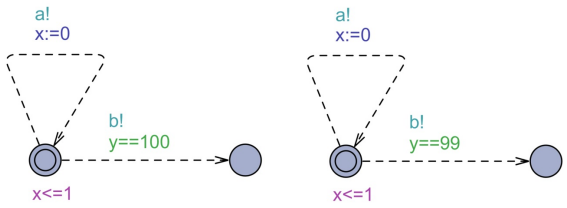

Fig. 6. A specification and its mutant, demonstrating the capabilities of an unbounded conformance check.

specification was expected to be deterministic, while non-determinism in the mutants did not pose a problem. In cases where the faulty SUT shows non-deterministic behavior, these mutants may reflect the behavior of the SUT more accurate than the deterministic ones. However, the test cases produced for the non-deterministic mutants would only cover one deterministic trace, and would need to be executed multiple times, to ensure capturing the fault.

- *Boundedness.* The previous approach computed a bounded conformance check between the specification and the mutant, thus only checking whether the mutant conforms for the first k steps of the execution. While the bound can usually be set high enough for the detection of all mutants, an upper bound has to be approximated, and if some mutants exceed it, their corresponding test cases will not be generated. Consider the example presented in Figure 6. The mutant could only be detected with a search depth of at least 100. For more realistic examples, the needed search depth may be a lot harder to determine and increasing the depth drastically increases the runtime, which is avoided by Ecdar.

VI. CONCLUSIONS

In this paper we have demonstrated how to implement model-based mutation testing via the tool Ecdar. We showed how to use its refinement check for model-based mutation testing, how the produced strategies can be interpreted as adaptive test cases, and defined an algorithm for the test-case execution. We compared the approach to an existing implementation of model-based mutation testing for timed automata which showed a high speedup, while maintaining the quality of the test suite with respect to the mutation score and improving its quality with respect to adaptiveness.

In future work we intend to provide further features to Ecdar, including an automated way for turning models input-enabled, providing more information in the produced strategies and a model-mutator. We also plan further experimental evaluations, including higher-order mutants.

ACKNOWLEDGMENT

The research leading to these results has received funding from the Danish Innovation Center DiCyPS (www.dicyps.dk), the ERC Advanced Grant LASSO: "Learning, Analysis, Synthesis and Optimization of Cyber Physical Systems" as well as the H2020-ECSEL-JU ENABLE-S3 European Initiative to Enable Validation for Highly Automated Safe and Secure Systems under grant agreement N°692455.

REFERENCES

[1] M. S. AbouTrab, S. Counsell, and R. M. Hierons. Specification mutation analysis for validating timed testing approaches based on timed automata. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 660–669, July 2012.

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Uml in action: A two-layered interpretation for testing. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, January 2011.

[3] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Softw. Test. Verif. Reliab.*, 25(8):716–748, December 2015.

[4] Bernhard K. Aichernig, Klaus Hörmaier, and Florian Lorber. *Debugging with Timed Automata Mutations*, pages 49–64. Springer International Publishing, Cham, 2014.

[5] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. *Time for Mutants — Model-Based Mutation Testing with Timed Automata*, pages 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[7] Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for simulink models. In *Proceedings of the 8th International Conference on Formal Methods for Components and Objects*, FMCO'09, pages 208–227, Berlin, Heidelberg, 2010. Springer-Verlag.

[8] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63 – 73, 1985.

[9] A. David, K. G. Larsen, S. Li, and B. Nielsen. A game-theoretic approach to real-time system testing. In *2008 Design, Automation and Test in Europe*, pages 486–491, March 2008.

[10] A. David, K. G. Larsen, S. Li, and B. Nielsen. Timed testing under partial observability. In *2009 International Conference on Software Testing Verification and Validation*, pages 61–70, April 2009.

[11] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. Timed i/o automata: A complete specification theory for real-time systems. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 91–100, New York, NY, USA, 2010. ACM.

[12] Alexandre David, Kim G. Larsen, Shuhao Li, and Brian Nielsen. Cooperative testing of timed systems. *Electronic Notes in Theoretical Computer Science*, 220(1):79 – 92, 2008.

[13] Alexandre David, Kim Guldstrand Larsen, Shuhao Li, Marius Mikucionis, and Brian Nielsen. Testing real-time systems under uncertainty. In *Proceedings of the 9th International Conference on Formal Methods for Components and Objects*, FMCO'10, pages 352–371, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 655–666, New York, NY, USA, 2016. ACM.

[15] Anders Hessel and Paul Pettersson. Cover-a test-case generation tool for timed systems. *Testing of Software and Communicating Systems*, pages 31–34, 2007.

[16] R. M. Hierons and M. G. Merayo. Mutation testing from probabilistic finite state machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 141–150, Sept 2007.

[17] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, September 2011.

[18] Moez Krichen and Stavros Tripakis. *Real-Time Testing with Timed Automata Testers and Coverage Criteria*, pages 134–151. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[19] Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Form. Methods Syst. Des.*, 34(3):238–304, June 2009.

[20] Marius Mikucionis, Brian Nielsen, and Kim G. Larsen. Real-time system testing on-the-fly. In Kaisa Sere and Marina Waldén, editors, *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Abo Akademi, Department of Computer Science, Finland. Abstracts.

[21] Robert Nilsson, Jeff Offutt, and Jonas Mellin. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science*, 164(4):97 – 114, 2006.

[22] A. Jefferson Offutt and Roland H. Untch. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.

[23] Jan Tretmans. Formal methods and testing. chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[24] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5):297–312, August 2012.

[25] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100, Berlin, Germany, 2004. Springer Verlag.

[26] Ting Wang, Jun Sun, Yang Liu, Xinyu Wang, and Shanping Li. *Are Timed Automata Bad for a Specification Language? Language Inclusion Checking for Timed Automata*, pages 310–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.