Master Thesis

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerlöfs Vej 300
DK-9220 Aalborg Ø
http://www.cs.aau.dk

Title:
Effortless Fault Localisation: Conformance Testing of Real-Time Systems in Ecdar

Subject:
Semantics and Verification

Project Period:
Spring Semester 2018

Project Group:
deis104f18

Participants:
Tobias Rosenkrantz Gundersen
Christian Ovesen

Supervisor:
Ulrik Nyman

Pages:
14

Date of Completion:
15$^{\text{th}}$ of June 2018

Summary

It is essential to ensure the correctness of safety-critical systems. Model checking can prove the correctness of a design. However, a correct design does not ensure the correctness of its implementation. We could test this correctness by writing unit tests by hand. However, this is time-consuming and error-prone, as we would manually derive them from the requirements of the System Under Test (SUT).

Instead, we can test if the SUT conforms to the model proven to be correct. Model checkers for real-time systems focus on timing aspects, which are essential for safety-critical real-time systems. When testing based on a timed model, the tests will focus on timed aspects as well.

One model-based testing technique is Model-Based Mutation Testing (MBMT). It has already been applied to timed automata with simulated time.

The model checker Ecdar can perform unbounded conformance checks, and it can perform adaptive test-cases, which produce fewer inconclusive test verdicts than non-adaptive test-cases. In this paper we present an extension of Ecdar that integrates conformance testing into the tool in order to improve productiveness and reliability. It uses MBMT that – contrary to similar approaches – is fault-based, proving the absence of certain types of faults. The extension also generates test-cases solely based on the test model. Thus, the tester does not need to provide any other constructs unlike what is required for some other methods.

The extension is an open-source IDE using the Ecdar engine. It can model Timed I/O Automata (TIOA), verify them, and test a system based on a TIOA.

We start this paper by discussing work related to MBMT. We discuss different approaches to MBMT. Furthermore, we discuss the tools MoMuT::TA, Uppaal TRON, CO✓ER, and Uppaal Yggdrasil

We present preliminaries, where we include a definition of TIOA. Compared to previous work, our definition is expanded to support local variables, as these are supported by the Ecdar engine. We also show the definition of timed I/O transition systems and refinement. Lastly, we describe the workflow of MBMT.

After preliminaries we present the mutation operators that we use; we reuse six from a previous paper and define five ourselves. Ecdar mutates a test model through the 11 mutation operators and uses the Ecdar engine to generate strategies that we use for test-case generation. The tool then executes the test-cases on the SUT. A strategy contains rules that we use to determine what inputs to send to the SUT and how long we delay before sending them. From this, we guide the test execution until certain criteria are met. Then we can give the test-case a verdict.

We introduce the concept of primary fails for when the mutant recognises a failing action from the SUT. The tool utilises such fails to perform automatic fault localisation.

The tool can execute test-cases using either real-time or simulated time. Testing using real-time enables testing of physical systems. Testing using simulated time allows for a significant speed-up. To further speed up testing, the tool parallelises test-case generation and test execution.

We present a case study that we use to test our integration. We use a system that is based on a model from another paper. We implement the system and use the independent mutation testing tool PIT on our implementation to generate 140 faulty systems.

We conduct both a test using real-time and one using simulated time during the case study to assess that testing with Ecdar is feasible. It took 0.61 seconds to mutate, 30 seconds to generate test-cases, 3 hours and 52 minutes to execute all test-cases using real-time and 26 seconds to execute all test-cases using simulated time.

From this case study we observe that our integration could detect all faults and could achieve primary fails for all but two faulty systems. We observe that our new operators can detect faults and achieve primary fails that the existing six operators cannot detect and achieve, respectively. This shows that the new operators improve the ability to detect and locate faults.

We then showcase the UI of modelling in Ecdar and the UI of mutation testing in Ecdar. We conclude on our contributions and, lastly, we discuss future work. As future work, we propose supporting for testing integrated systems, conducting an industrial case study, improving fault detection by using higher-order

mutants, reducing test execution and test-case generation time, and including a command-line interface to enable conformance testing with Ecdar during continuous integration.

Effortless Fault Localisation: Conformance Testing of Real-Time Systems in Ecdar

Tobias R. Gundersen Christian Ovesen
Aalborg University, Denmark
tgunde13@student.aau.dk covese13@student.aau.dk

Model checking of real-time systems has evolved throughout the years. Recently, the model checker Ecdar, using timed I/O automata, was used to perform compositional verification. However, in order to fully integrate model checking of real-time systems into industrial development, we need a productive and reliable way to test if such a system conforms to its corresponding model. Hence, we present an extension of Ecdar that integrates conformance testing into a new IDE that now features modelling, verification, and testing. The new tool uses model-based mutation testing, requiring only the model and the system under test, to locate faults and to prove the absence of certain types of faults. It supports testing using either real-time or simulated time. It parallelises test-case generation and test execution to provide a significant speed-up. We also introduce new mutation operators that improve the ability to detect and locate faults. Finally, we conduct a case study with 140 faulty systems, where Ecdar detects all faults.

1 Introduction

It is essential to ensure the correctness of safety-critical systems. Model checking can prove the correctness of a design. However, a correct design does not ensure the correctness of its implementation. We could test this correctness by writing unit tests by hand. However, this is time-consuming and error-prone, as we would manually derive them from the requirements of the System Under Test (SUT).

Instead, we can test if the SUT conforms to the model proven to be correct. Model checkers for real-time systems focus on timing aspects, which are essential for safety-critical real-time systems. When testing based on a timed model, the tests will focus on timed aspects as well.

One model-based testing technique is Model-Based Mutation Testing (MBMT). It has already been applied to Timed Automata (TA) with simulated time [3, 14]. Larsen et al. [14] use the model checker Ecdar [6] to perform unbounded conformance checks, to provide a significant speed-up compared to [3], and to enable adaptive test-cases, which produce fewer inconclusive test verdicts.

The main contributions of this paper are (1) an extension of Ecdar that integrates conformance testing of real-time systems, using only the model and the SUT, into the tool; (2) automatic fault localisation through the introduction of primarily failed test-cases; and (3) support for testing using either real-time or simulated time.

Other contributions are parallelisation of test-case generation and test execution to provide a significant speed-up and the introduction of new mutation operators that improve the ability to detect and locate faults.

The rest of the paper is structured as follows: First, in Section 2 we discuss related work. In Section 3 we present preliminaries, including the definition of Timed I/O Automata (TIOA) and the workflow of MBMT. In Section 4 we discuss the integration of MBMT into Ecdar. In Section 5 we present a case study of our extension. In Section 6 we showcase the extension. Finally, in Section 7 we conclude the paper and outline ideas for future work.

To appear in EPTCS.

2 Related Work

MBMT has already been applied to probabilistic finite state machines [11] and UML state machines [1]. These approaches mainly focus on testing functional behaviour. Aichernig et al. [3] propose to use MBMT for TA and presents the tool MoMuT::TA[1] that implements test-case generation.

Larsen et al. [14] propose to use Ecdar to perform unbounded conformance checks, to provide a significant speed-up compared to [3], and to enable adaptive test-cases, which produce fewer inconclusive test verdicts.

Lorber et al. [15] propose an approach to combine MBMT and Timed Computation Tree Logic (TCTL) properties used for verification; from a set of generated mutants, they check if the mutants satisfy the properties. For each violation, they use the counterexample as a test-case. This way, the approach only model checks individual properties rather than performing a potentially more time-consuming full conformance check. Also, the approach generates fewer test-cases, which takes less time to execute, and the test-cases focus on the safety-critical properties derived from the requirements of the system.

Devroey et al. [8] propose another way to reduce generation and testing time. They propose to encode each mutant as a product in a software product line; instead of generating individual mutants, they propose to generate a single featured mutant model that – when configured – can represent any mutant. This way, the mutants can share execution, allowing for testing in a single run. While the approach is for assessing the quality of an existing test suite, it could also be applied for test-case generation.

There exist tools for model-based testing using TA. The tool UPPAAL TRON [9] can conduct conformance tests on systems in real-time and thus can be applied to physical systems. It uses an environment TA to decide which inputs and delays to trigger and detects whether the SUT produces allowed outputs. A tester can easily construct a simple, permissive environment, but at the cost of simply triggering random inputs and delays. Otherwise, they can construct environment models that steer the execution towards critical areas.

The tool CO✓ER [10] generates test-cases based on TA and TCTL. A tester can reuse the TCTL properties used for verifying the test model. The tester also needs to specify monitoring automata that describe coverage criteria.

The tool UPPAAL Yggdrasil [13] is similar to CO✓ER. It generates test-cases based on TA, TCTL properties, and random search.

Contrary to these three tools, the presented Ecdar extension uses a fault-based approach that can prove the absence of certain types of faults. Furthermore, it generates test-cases only based on a test model. Thus, an Ecdar tester need only provide the test model and the SUT; they do not need to perform the time-consuming and error-prone tasks of constructing environment models, TCTL properties, or monitoring automata.

3 Preliminaries

In this section we define TIOA, their underlying transition systems, determinism, input-enabledness, and refinement. Finally, we describe the workflow of MBMT.

[1] `https://momut.org/`

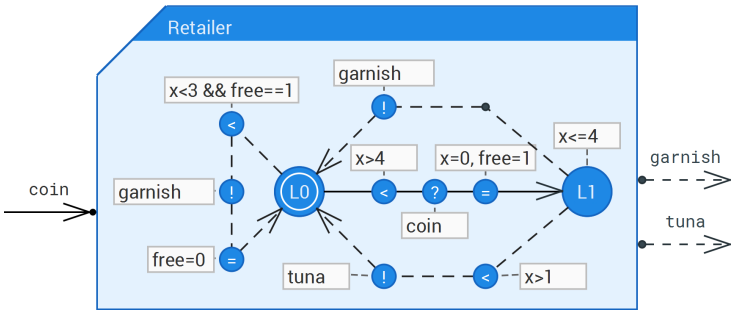


Figure 1: An Ecdar TIOA *Retailer* for a fish retailer.

3.1 Timed I/O Automata

A TIOA [7] is a syntactical, finite representation of a timed system. It is a tuple $(\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, \mathcal{E}, \mathcal{I})$, where:

- $\mathcal{Q}$ is a finite set of locations.
- $q_0 \in \mathcal{Q}$ is the initial location.
- $\mathcal{C}$ is a finite set of clocks used to represent time.
- $\mathcal{V}$ is a finite set of integer variables local to the automata. Each variable $v \in \mathcal{V}$ has lower and upper bounds $v_{min}, v_{max} \in \mathbb{Z}$ and an initial value $v_0 \in [v_{min}, v_{max}]$.
- Σ is a finite set of observable actions partitioned into inputs (Σ_i) and outputs (Σ_o).
- $\mathcal{E}$ is a finite set of edges of the form $e = (q, g, \sigma, R, u, q')$, where:
  - $q, q' \in \mathcal{Q}$ are the source and target locations, respectively.
  - g (the *guard*) is a conjunction $\bigwedge_{b \in B_e} b$, i.e. B_e is the set of basic constraints of the guard of edge e. Each basic constraint b is of the form $x \circ c$, where $x \in \mathcal{C} \cup \mathcal{V}$, $\circ \in \{<, \leq, =, \neq, \geq, >\}$, and $c \in \mathbb{Z}$. The guard must be satisfied when executing the edge.
  - $\sigma \in \Sigma$ is the observable action.
  - $R \subseteq \mathcal{C}$ is the set of clocks to reset.
  - $u : \mathcal{V} \to \mathbb{Z}$ is an update of some of the variables. For each such variable v, $u(v) \in [v_{min}, v_{max}]$.
- $\mathcal{I} : \mathcal{Q} \to \mathcal{U}(\mathcal{C})$ is a set of invariants for some of the locations. We write $\mathcal{U}(\mathcal{C})$ for the set of constraints over $\mathcal{C}$ of the form $x \circ c$, where $x \in \mathcal{C}$, $\circ \in \{<, \leq\}$, and $c \in \mathbb{N}$. An invariant must be satisfied when entering and while in the respective location.

Contrary to [7], our definition of TIOA includes local variables, since the Ecdar engine supports them. Figure 1 illustrates the TIOA for which $\mathcal{Q} = \{L0, L1\}$, $q_0 = L0$, $\mathcal{C} = \{x\}$, $\mathcal{V} = \{free\}$, $free_0 = free_{min} = 0$, $free_{max} = 1$, $\Sigma_i = \{coin?\}$, $\Sigma_o = \{garnish!, tuna!\}$, $\mathcal{E} = \{(L0, x < 3 \wedge free = 1, garnish!, \emptyset, \{(free, 0)\}, L0), (L0, x > 4, coin?, \{x\}, \{(free, 1)\}, L1), (L1, true, garnish!, \emptyset, \emptyset, L0), (L1, x > 1, tuna!, \emptyset, \emptyset, L0)\}$, and $\mathcal{I} = \{(L1, x \leq 4)\}$.

We denote by $\mathcal{A}$ the set of all TIOA. We denote by $\mathcal{E}_i$ the input edges $\{e = (q, g, \sigma, R, u, q') \mid e \in \mathcal{E} \wedge \sigma \in \Sigma_i\}$, and by $\mathcal{E}_o$ the output edges $\{e = (q, g, \sigma, R, u, q') \mid e \in \mathcal{E} \wedge \sigma \in \Sigma_o\}$.

3.2 Timed I/O Transition Systems

A Timed I/O Transition System (TIOTS) [7] S is the semantic representation induced by a TIOA A, written $S = [\![A]\!]_{sem}$. It is a tuple $(St, s_0, \Sigma, \rightarrow)$ where:

- St is a set of states.

- $s_0 \in St$ is the initial state.
- Σ is a finite set of observable actions partitioned into inputs (Σ_i) and outputs (Σ_o).
- $\rightarrow \subseteq St \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times St$ is a transition relation.

We write $s \xrightarrow{a} s'$ instead of $(s,a,s') \in \rightarrow$. We write $s \xrightarrow{a}$ for $\exists s'.\, s \xrightarrow{a} s'$. We use $i?$, $o!$, and d to range over inputs, outputs, and delays ($\mathbb{R}_{\geq 0}$), respectively.

Determinism: A TIOTS is *deterministic* iff $\forall s,s',s'' \in St.\, \forall \sigma \in \Sigma.\, s \xrightarrow{\sigma} s' \wedge s \xrightarrow{\sigma} s'' \implies s' = s''$. That is, whenever the transition system can perform an action, there is always only one transition to take with that action. As an example, the transition system of *Retailer* in Figure 1 is deterministic.

Input-Enableness: A TIOTS is *input-enabled*, iff $\forall s \in St.\, \forall i? \in \Sigma_i.\, s \xrightarrow{i?}$. That is, it can always accept any of its defined inputs. As an example, the transition system of *Retailer* in Figure 1 does not accept a *coin*? in $L1$, and thus it is not input-enabled.

We can use *angelic completion* [17] to transform an automaton to one with an input-enabled transition system: For each input, it receives new self-loops for each state that did not accept that input. It corresponds to ignoring those inputs.

Alternatively, we can use *demonic completion* [4] to transform an automaton to one with an input-enabled transition system: Instead of self-loops, it has new edges leading to a *universal location*. In a universal location, every possible behaviour defined by the automata is enabled. That is, it can continuously both delay indefinitely and perform every action Σ.

3.3 Refinement

Refinement [7] compares the behaviour of two deterministic, input-enabled transition systems. A TIOTS $T = (St^T, t_0, \Sigma, \rightarrow^T)$ *refines* a TIOTS $S = (St^S, s_0, \Sigma, \rightarrow^S)$, written $T \leq S$, iff there exists a binary relation $R \subseteq St^T \times St^S$ containing (t_0, s_0) such that for each pair of states $(t,s) \in R$ we have:

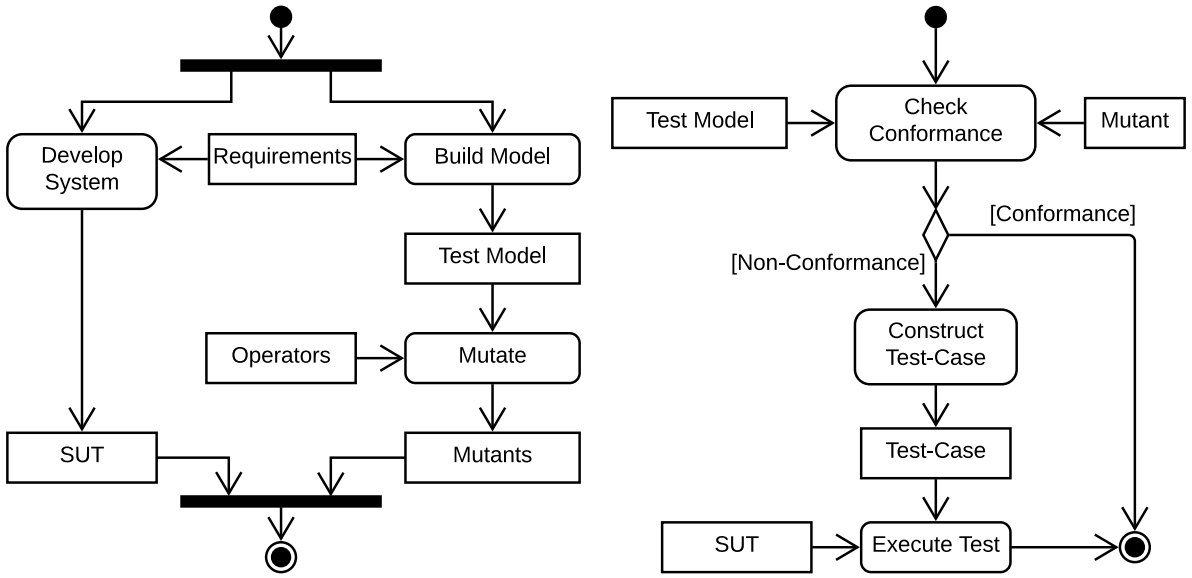
- $\forall s' \in St^S.\, \forall i? \in \Sigma_i.\, s \xrightarrow{i?}^S s' \implies \exists t' \in St^T.\, t \xrightarrow{i?}^T t' \wedge (t',s') \in R$.
- $\forall t' \in St^T.\, \forall o! \in \Sigma_o.\, t \xrightarrow{o!}^T t' \implies \exists s' \in St^S.\, s \xrightarrow{o!}^S s' \wedge (t',s') \in R$.
- $\forall t' \in St^T.\, \forall d \in \mathbb{R}_{\geq 0}.\, t \xrightarrow{d}^T t' \implies \exists s' \in St^S.\, s \xrightarrow{d}^T s' \wedge (t',s') \in R$.

$T \leq S$ represents that T has less behaviour than or equal behaviour to S. A TIOA A_1 refines another TIOA A_2, written $A_1 \leq A_2$, iff $[\![A_1]\!]_{sem} \leq [\![A_2]\!]_{sem}$. That is, there is a corresponding refinement between their underlying TIOTSs.

3.4 Model-Based Mutation Testing

In MBMT we construct a test suite based on mutants of a test model. Firstly, based on requirements of a system, we develop the SUT and a test model that it should conform to (see Figure 2a). We then mutate the test model according to some selected mutation operators. An operator represents certain ways of changing the model, e.g. changing the source location of an arbitrary edge to an arbitrary location. A mutant is the result of a single application of an operator and represents a potential fault.

For each mutant, we check if it conforms to the test model (see Figure 2b). If it does, it does not introduce any observable faults. Thus, we discard it. Otherwise, the conformance check provides a counterexample, i.e. a way to potentially reveal the fault. We use the counterexample as a test-case by applying it on the SUT. A test passes iff the SUT is shown to behave according to the test model and not according to the mutant.



(a) Activity to create a SUT, a test model, and mutants.

(b) Activity to generate and execute a test-case.

Figure 2: UML 2.5 [16] activity diagrams of MBMT.

Assuming determinism, for all passing test-cases, MBMT guarantees that the faults represented by their corresponding mutants do not exist in the SUT. A failed test-case is an aid for debugging as presented in [2]; a failed test-case can show the location and type of the fault and provides a way to reproduce it.

Adaptiveness Test-cases can be more or less adaptive. A non-adaptive test-case only covers a trace with its counterexample; if the test model has multiple choices of delaying and outputting, a non-adaptive test-case can only handle one of the choices. If the SUT performs an unexpected, allowed delay or output, we assign it the *inconclusive* verdict.

An *adaptive* test-case can handle various choices made by the SUT and steer the execution towards the fault represented by the mutant.

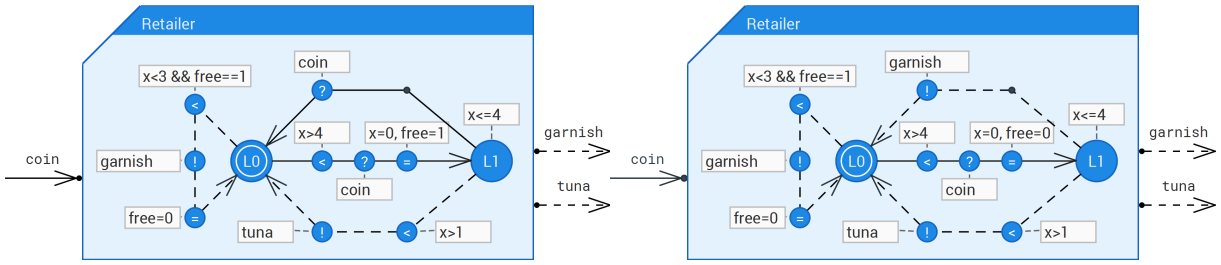
4 Integration

We divide testing with Ecdar into three steps: Mutation, test-case generation, and test execution. In this section, we discuss each of these steps and then discuss performance.

4.1 Mutation

A mutation operator is a function $\mathcal{M} : \mathcal{A} \to 2^{\mathcal{A}}$. We use the following mutation operators defined by Aichernig et al. [3]:

- $\mathcal{M}_s$ replaces the *source* location of an edge with another location.
- $\mathcal{M}_t$ replaces the *target* location of an edge with another location.
- $\mathcal{M}_o$ replaces the action of an edge with a (different) *output*.



(a) Mutant in $\mathcal{M}_i(Retailer)$. The action of an edge from $L1$ to $L0$ is changed from output *garnish*! to input *coin*?.

(b) Mutant in $\mathcal{M}_{vu}(Retailer)$. The variable update of the edge from $L0$ to $L1$ is changed from $(free,1)$ to $(free,0)$.

Figure 3: Two mutants of the *Retailer* TIOA on Figure 1.

$\mathcal{M}_{inv}$ loosens a constraint in an *invariant* by 1 time unit (e.g. $x <= 2$ would be loosened to $x <= 3$). We do not tighten invariants, as this would result in a conformance.

$\mathcal{M}_{sl}$ changes the target location of an edge to a new *sink location*. Sink locations accept but ignore all inputs.

$\mathcal{M}_c$ inverts a *clock* reset on an edge; if the clock was originally reset, the reset is removed, otherwise a reset is added.

We define the following new mutation operators. In the definition of the operators, we mutate a TIOA $S = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, \mathcal{E}, \mathcal{I})$:

$\mathcal{M}_i$ replaces the action of an edge with a (different) *input*. This creates $|\mathcal{E}_i|(|\Sigma_i|-1)+|\mathcal{E}_o||\Sigma_i|$ mutants. Aichernig et al. [3] only replaces actions with outputs (the operator $\mathcal{M}_o$). A TIOA $M \in \mathcal{M}_i(S)$ iff $M = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, (\mathcal{E} \setminus \{e_S\}) \cup \{e_M\}, \mathcal{I})$, such that $e_S = (q, g, \sigma_S, R, u, q') \in \mathcal{E}$, $e_M = (q, g, \sigma_M, R, u, q')$, $\sigma_M \in \Sigma_i$, and $\sigma_S \neq \sigma_M$. An example of such a mutant is given is Figure 3a.

$\mathcal{M}_{gc}$ adds or subtracts 1 to or from a *guard constant*. This creates $2\sum_{e\in\mathcal{E}}|B_e|$ mutants. A TIOA $M \in \mathcal{M}_{gc}(S)$ iff $M = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, (\mathcal{E} \setminus \{e_S\}) \cup \{e_M\}, \mathcal{I})$, such that $e_S = (q, g_S, \sigma, R, u, q') \in \mathcal{E}$, $e_M = (q, g_M, \sigma, R, u, q')$, $g_S = \bigwedge_{i\in I}(x_i \circ_i c_i^S)$, $g_M = \bigwedge_{i\in I}(x_i \circ_i c_i^M)$, $\exists i' \in I.\, c_{i'}^S \pm 1 = c_{i'}^M$, and $\forall i \in I \setminus \{i'\}.\, c_i^S = c_i^M$.

$\mathcal{M}_{goc}$ changes a *guard operator* with a *clock* as its left side. This creates up to $2\sum_{e\in\mathcal{E}}|B_{e,\mathcal{C}}|$ mutants, where $B_{e,\mathcal{C}} = \{b \in B_e \mid b = x \circ c \wedge x \in \mathcal{C}\}$. Time is continuous. Thus, in practice we expect that for all clock valuations v we have that $\forall c \in \mathbb{Z}.\, v < c \iff v \leq c \wedge v > c \iff v \geq c \wedge v \neq c$. For this reason, we only mutate with $\leq$ and $>$. With reduced number of guard operators, we reduce generation and test execution time. $\mathcal{M}_{goc}$ overlaps with two mutation operators in [3]: μ_{cg} that in a single mutation changes all operators in a guard to one among $\{<, \leq, =, \geq, >\}$, and μ_{ng} that negates a guard. Since a mutation is a simple syntactic change [12], we combine these two operators into $\mathcal{M}_{goc}$ that only changes a single operator. A TIOA $M \in \mathcal{M}_{goc}(S)$ iff $M = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, (\mathcal{E} \setminus \{e_S\}) \cup \{e_M\}, \mathcal{I})$, such that $e_S = (q, g_S, \sigma, R, u, q') \in \mathcal{E}$, $e_M = (q, g_M, \sigma, R, u, q')$, $g_S = \bigwedge_{i\in I}(x_i \circ_i^S c_i)$, $g_M = \bigwedge_{i\in I}(x_i \circ_i^M c_i)$, $\exists i' \in I.\, x_{i'} \in \mathcal{C} \wedge \circ_{i'}^M \in \{\leq, >\} \setminus \{\circ_{i'}^S\}$, and $\forall i \in I \setminus \{i'\}.\, \circ_i^S = \circ_i^M$.

$\mathcal{M}_{gov}$ changes a *guard operator* with a *variable* as its left side. This creates $5\sum_{e\in\mathcal{E}}|B_{e,\mathcal{V}}|$ mutants, where $B_{e,\mathcal{V}} = \{b \in B_e \mid b = x \circ c \wedge x \in \mathcal{V}\}$. A TIOA $M \in \mathcal{M}_{gov}(S)$ iff $M = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, (\mathcal{E} \setminus \{e_S\}) \cup \{e_M\}, \mathcal{I})$, such that $e_S = (q, g_S, \sigma, R, u, q') \in \mathcal{E}$, $e_M = (q, g_M, \sigma, R, u, q')$, $g_S = \bigwedge_{i\in I}(x_i \circ_i^S c_i)$, $g_M = \bigwedge_{i\in I}(x_i \circ_i^M c_i)$, $\exists i' \in I.\, x_{i'} \in \mathcal{V} \wedge \circ_{i'}^M \in \{<, \leq, =, \neq, \geq, >\} \setminus \circ_{i'}^S$, and $\forall i \in I \setminus \{i'\}.\, \circ_i^S = \circ_i^M$.

$\mathcal{M}_{vu}$ assigns a value to a local *variable* in an *update* property. If the variable is already being assigned in this property, the mutating assignment overrides the existing one. If the existing and mutating assignment values are equal, the corresponding mutant is not created. This creates up to $|\mathcal{E}|\sum_{v\in\mathcal{V}}(v_{max}-v_{min}+1)$ mutants. A TIOA $M \in \mathcal{M}_{vu}(S)$ iff $M = (\mathcal{Q}, q_0, \mathcal{C}, \mathcal{V}, \Sigma, (\mathcal{E} \setminus \{e_S\}) \cup \{e_M\}, \mathcal{I})$, such that $e_S = (q, g, \sigma, R, u_S, q') \in \mathcal{E}$, $e_M = (q, g, \sigma, R, u_M, q')$, $\exists v' \in \mathcal{V}.\, u_M(v') \neq u_S(v') \vee (v' \notin \text{dom}(u_S) \wedge v' \in \text{dom}(u_M))$, and $\forall v \in \mathcal{V} \setminus \{v'\}.\, u_M(v) = u_S(v) \vee (v \notin \text{dom}(u_S) \wedge v \notin \text{dom}(u_M))$. An example of such a mutant is given is Figure 3b.

The definition of mutation operators is simplified compared to the Ecdar implementation for better understanding. For instance, the implementation can also handle more complex constraints, e.g. constraints with addition, subtraction, and mixed use of constants, clocks, and variables on either side. Also, it does not mutate certain locations and edges. For instance, it is sometimes inappropriate to mutate the universal location and its outgoing edges.

4.2 Test-Case Generation

We use the approach presented in [14] to perform conformance checks. That is, we use the Ecdar engine to determine if the mutant refines the test model. Since refinement assumes determinism, we discard non-deterministic models.

Refinement also assumes input-enabledness. However, we do not want to force the modeller to make the test model input-enabled. Rather, the behaviour missing in order to be input-enabled is not relevant for the SUT and thus should not be tested for.

Instead, we apply demonic completion on the test model and angelic completion on the mutants like the approach presented in [14]. This way, traces leading to missing behaviour will transition the test model into the universal location. Everything refines the universal location. Thus, mutants resulting in such traces will not yield a counterexample for the refinement.

The Ecdar engine solves a refinement check as a timed game. To check if $T \leq S$, the goal of the game is to find a strategy for revealing the non-refinement $T \not\leq S$ by triggering delays and inputs, and observing outputs. In the case of a non-refinement, the Ecdar engine produces a strategy. A strategy can handle various choices made by the SUT and can steer the execution towards the goal. Thus, it provides us with an adaptive test-case.

4.3 Test Execution

We develop a test driver that executes the generated test-cases on a SUT. The test driver implementation is inspired by the algorithm presented by Larsen et al. [14].

The test driver communicates with the SUT over its standard I/O streams. We treat the SUT as a black box where the inputs and outputs we send and receive are the same as those represented in the test model. The driver can test systems using either real-time or simulated time. Using real-time, we can test physical systems. However, using real-time is often significantly slower because it may need to perform physical delays. When simulating time, rather than physically delaying the test, the test driver computes and sends to the SUT how long the SUT is allowed to simulate delay without interruptions through an input, and the SUT answers how long it actually simulated.

If the SUT terminates while testing, we treat it as a sink location. This allows us to finish a test if the program terminates.

		Test model	
		✓	✗
Mutant	✓	Continue	Primary fail
	✗	Pass	Other fail

Table 1: What action to take based on whether the test model and the mutant can (✓) or cannot (✗) simulate a delay or an output produced by the SUT.

Rules A strategy consists of delay, input, and output rules, all with disjoint conditions. The test driver checks which rule in the strategy is satisfied for the current states of the test model and the mutant.

If the satisfied rule is an *input rule*, we send an input to the SUT. For a *delay rule* the test driver performs a delay until the rule is no longer satisfied or the SUT has produced an output. The test driver cannot force the SUT to perform an output. It instead must wait for the SUT to produce one. Thus, we treat an *output rule* as a delay rule.

Aborting A location with no invariant but with outgoing output edges with fully permissive guards implies that the SUT can wait forever before outputting. This behaviour causes output rules to suggest that we wait indefinitely until the SUT (hopefully) outputs. To avoid this, we introduce a maximum wait time; if this is exceeded, we abort the current test.

If the SUT avoids the states needed to determine the existence of a fault by looping among the same set of states, the strategy will suggest that we loop indefinitely among the same set of rules. To avoid this, we enforce a bound. Whenever we change the current rule, we increment a step value. If the value exceeds the bound, we abort the current test.

Verdicts As described in Section 3.4 a test can pass, fail or be inconclusive. We simulate the actions of the SUT on the test model and the mutant to determine if the current test passes or fails according to Table 1. A test passes iff the test model can simulate a delay or an output, but the mutant cannot.

A test fails iff the test model cannot simulate a delay or an output. If a test fails, we simulate the failing action on the mutant. If the mutant can perform it, then we have found a fault that is recognised by the mutant. This mutant is especially helpful for locating the fault. We call these types of fails primary fails.

A test is inconclusive, if there are no applicable rules for the current states of the models or if we abort the test.

4.4 Performance

Generation is a computationally heavy task for which the CPU is the bottleneck. However, we call the Ecdar engine through its command-line interface, which causes delays, making the CPU underutilised. To speed up generation, we generate multiple test-cases in parallel, making Ecdar able to fully utilise the CPU.

Executing a test is not necessarily a computationally heavy task for the test device. Thus, we could speed up testing by generating and executing tests in a pipeline manner. However, we want a consistent and realistic test environment. Running computationally heavy tasks in the background while testing will slow down the test driver, which can cause false positive verdicts (e.g. if the SUT would have outputted too early). Thus, we test only after we have generated all test-cases.

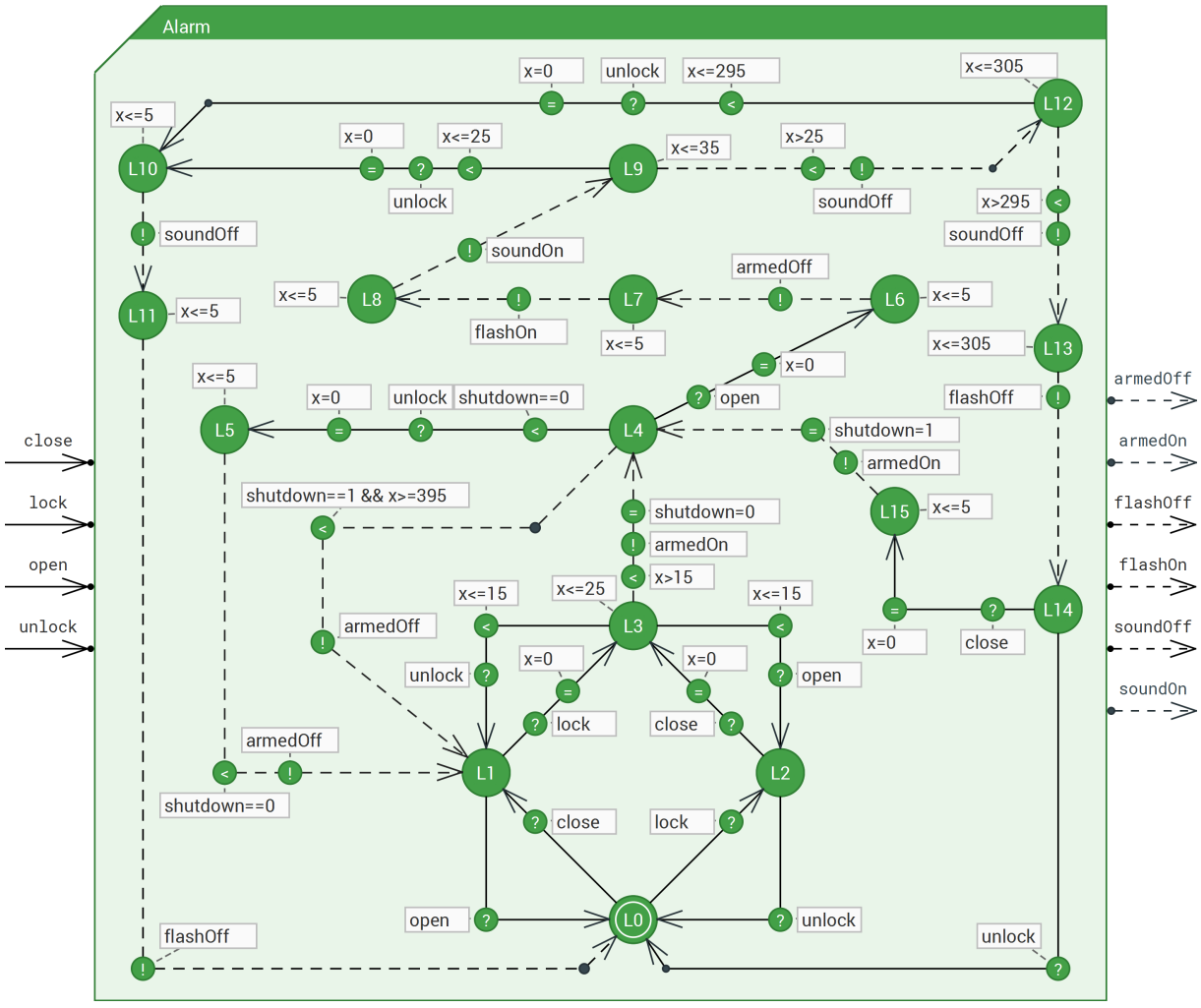


Figure 4: An Ecdar TIOA *Alarm* for a car alarm system.

Some systems allow running multiple concurrent instances of them without problems. For this reason, Ecdar allows running multiple concurrent instances of the SUT, which speeds up test execution. This is especially useful for real-time testing, as the test driver might wait a long time for the SUT to output. By default, we run only one concurrent instance. We leave it up to the user to set a low enough limit on the maximum number of concurrent instances.

5 Case Study

We implement a modified version of the car alarm system from [14] to be used as a case study for the purpose of evaluating Ecdar. The system represents a car alarm that is armed and triggered if someone opens a door without unlocking the car first. If triggered, it can be disarmed or – if left alone – it disarms itself after a set duration. As we have defined mutation operators that handle variables, we add a *shutdown* variable. Furthermore, we make the model robust with regards to time constraints in order to support testing using real-time. The model of the modified system can be seen on Figure 4.

$\mathcal{M}$	#	Set 1	Set 2	Set 3	Set 4	Set 5
$\mathcal{M}_s$	203					
$\mathcal{M}_t$	333					
$\mathcal{M}_o$	143					
$\mathcal{M}_{inv}$	7					
$\mathcal{M}_{sl}$	26					
$\mathcal{M}_c$	7					
$\mathcal{M}_i$	30					
$\mathcal{M}_{gc}$	10					
$\mathcal{M}_{goc}$	8					
$\mathcal{M}_{gov}$	4					
$\mathcal{M}_{vu}$	1					
Total	772					

Table 2: Results of testing the faulty systems. We denote by # the number of generated test-cases. For each combination of operator and faulty system we denote if the corresponding tests resolved in at least one primary fail ( ). Otherwise, we denote if they resolved in at least one other fail ( ). Each PIT operator creates a set of faulty systems that we divide with vertical lines ( ).

Ecdar detected faults in our initial implementation. We used Ecdar to locate and fix the faults. For this case study, we use the final version of the implementation. It has no faults according to Ecdar.

We perform our tests on a Windows 10 Pro v. 1803 computer with an AMD Ryzen 7 1690 CPU and two Kingston HyperX SH103S3/120G SSDs in Raid 0. We mutate using all Ecdar mutation operators. We define 1 time unit as 1 s, the number of concurrent SUT instances as 5, the maximum wait time as 420 time units, and the step bound as 40 (see Section 6).

It took 0.61 s to generate all 1173 mutants and 30 s to generate all 772 test-cases. A test execution using real-time took 3 h 52 m, while an execution using simulated time took 26 s, which is a significant speed-up.

To evaluate our integration of Ecdar we use the mutation testing tool PIT[2] to generate variations of the car alarm system. We used all mutation operators from PIT and mutated the `CarAlarm` class (see Section 6) to generate 278 systems.

We use the Ecdar extension to test the systems generated with PIT. In order to speed up test execution, we test using simulated time. Ecdar reports no fails for 47 of the systems. Through inspection of these systems, we found that they are all equivalent to the original car alarm system. Another 91 systems crashed while testing. When a SUT crashes, Ecdar provides the stack trace, which allows developers to locate the fault. However, it does not provide us with an output with which we can determine the verdict. This leaves us with 140 failing systems that do not crash. We denote these as *faulty systems*. The results from testing the faulty systems can be seen in Table 2.

We observe that every faulty system has a failed test-case. This shows that *Ecdar detected all the generated faults*. In three systems (e.g. number 4 in set 1) $\mathcal{M}_{gc}$ and $\mathcal{M}_{gov}$ are the only ones that could detect the fault. These operators are new and defined in this paper. This shows that the new operators improve the ability to detect faults.

As mentioned in Section 4.2 Ecdar includes primary fails. For three systems (e.g. number 5 in set 1) $\mathcal{M}_{goc}$ is the only one to achieve a primary fail. This operator is new and defined in this paper. This

[2] `http://pitest.org/` - version 1.3.2

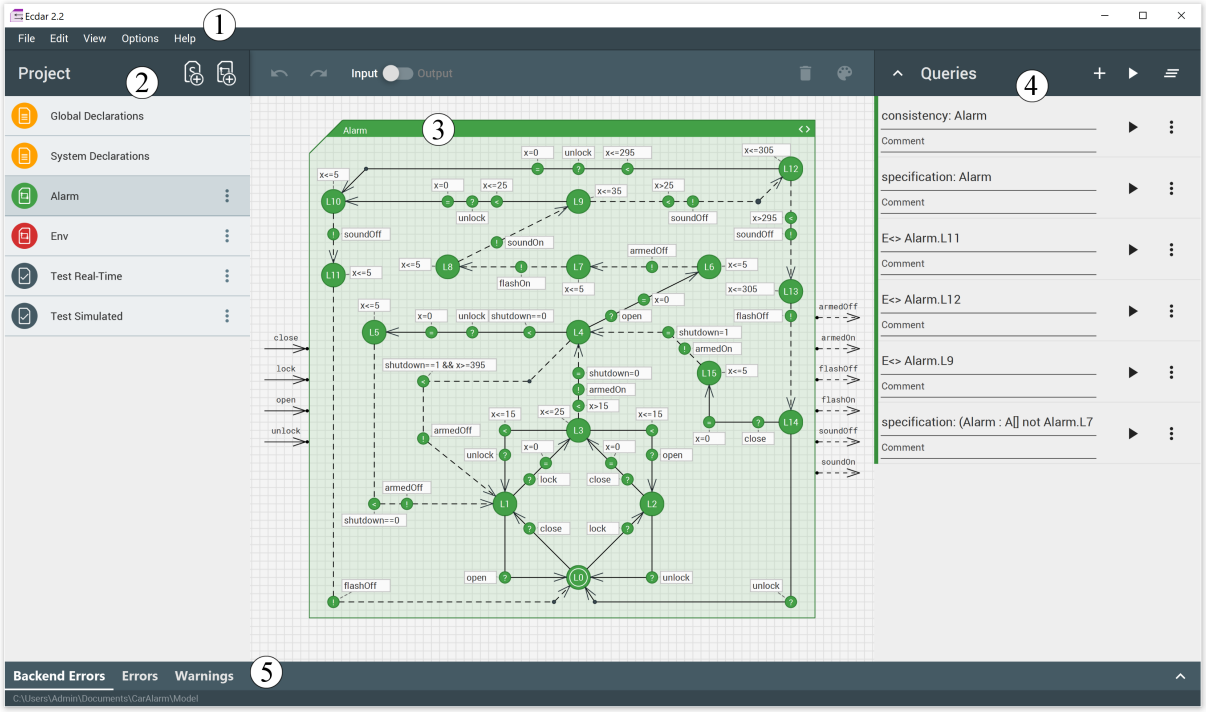


Figure 5: A screenshot of the Ecdar UI overlayed with numbered circles.

shows that the new operators improve the ability to locate faults.

Two faulty systems (e.g. number 8 in set 1) have no primary fails. Both change an output to one not defined in the test model. In order to generate primarily failed test-cases for such faults, their mutants would have to guess these non-defined outputs. As there is an infinite number of non-defined outputs, it is in practice impossible to guarantee a primary fail for every such fault.

6 User Interface

The Ecdar extension is implemented as a program that works as an IDE. Figure 5 shows a screenshot of Ecdar, where:

① from the menu bar, we can create new documents and change options;

② from the project pane, we can manage the declarations of the system, the TIOA, and the test plans;

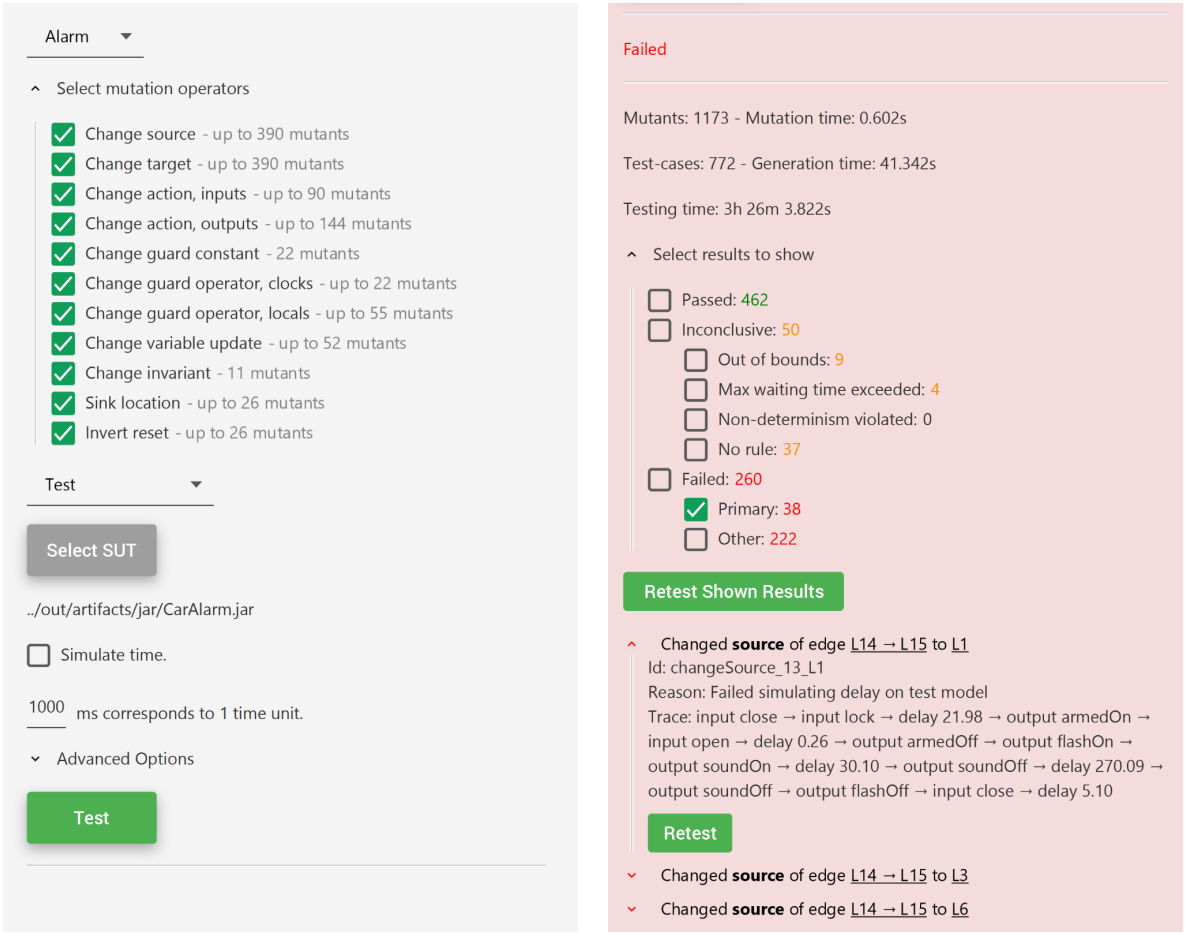
③ on the canvas, we can edit TIOA. For instance, the TIOA in Figures 1 and 4 are modelled this way;

④ on the query pane, we can use the Ecdar engine to verify TCTL properties; and

⑤ on the error pane, errors and warnings automatically appear.

From the menu bar ①, we can create a new test plan. In order to test, we simply need to choose our test model, select the path to our SUT or a program interfacing it, and (if testing using real-time) define what 1 time unit in the model corresponds to in real-time (see Figure 6a). The current version of Ecdar only works with a SUT that is a JAR file or interfaced by one.

Testers can adjust a test plan by changing:



(a) A test plan.

(b) Results of a failed test run.

Figure 6: Partial screenshots of an Ecdar UI.

- what mutation operators to test with,
- whether to test using simulated time,
- the maximum number of concurrent threads used for generation,
- the maximum number of concurrent instances of the SUT,
- the maximum wait time, and
- the step bound.

For each executed test-case, Ecdar displays a description of the mutant, the id of the test-case, the verdict, reason for the verdict, and the trace performed by the SUT (see Figure 6b). The tester should use this information to locate faults.

Ecdar supports retesting selected test-cases without redoing mutation or test-case generation. It can also export mutants for custom use. The extension and sample projects (including the car alarm presented in Section 5) are licensed under MIT[3] and are available at `http://ulrik.blog.aau.dk/ecdar/ecdar-2-2/`.

[3] `https://opensource.org/licenses/MIT`

7 Conclusion

In this paper, we present an extension of Ecdar that integrates conformance testing into the tool in order to improve productiveness and reliability. It uses Model-Based Mutation Testing (MBMT) that – contrary to similar approaches – is fault-based, proving the absence of certain types of faults. It also generates test-cases solely based on the test model. Thus, the tester does not need to provide any other constructs unlike what is required for some other methods.

Ecdar mutates the test model through 11 mutation operators and uses the Ecdar engine to generate strategies that we use for test-case generation. The tool then executes the test-cases using either real-time or simulated time. Testing using real-time enables testing of physical systems. Testing using simulated time allows for a significant speed-up. To further speed up testing, we parallelise test-case generation and test execution.

Ecdar is an open-source IDE using the Ecdar engine. It can model Timed I/O Automata (TIOA), verify them, and test a system based on a TIOA.

We conduct a case study using the independent mutation testing tool PIT to generate 140 faulty systems. *Testing with Ecdar detected all faults*. We introduce new mutation operators that improve the ability to detect and locate faults.

Future Work Ecdar can combine models with conjunction, composition, and quotient. It does, however, only support testing of one uncombined model at a time. This makes it tedious to test integrated systems and makes it unusable for integration testing. Future work could include support for testing integrated systems. The challenge is to simulate combined automata. We could take inspiration from UPPAAL TRON [9], as it solves a similar problem; it does not know the state of the System Under Test (SUT), so it keeps track of multiple potential states. When testing integrated systems, we could check refinements using compositional verification [5] in order to speed up test-case generation.

We presented in Section 5 a small case study to demonstrate MBMT with Ecdar. However, future work could include an industrial case study.

Ecdar only supports testing using first-order mutants. Future work could include using higher-order mutants to detect faults not detectable by first-order mutants. Future work could also include reducing generation and execution time. An approach is to prioritise test-cases so that we only generate and execute test-cases with a higher probability of detecting a fault. Alternatively, Lorber et al. [15] and Devroey et al. [8] propose other approaches for speeding up, as presented in Section 2.

Finally, future work could include adding a command-line interface to test with Ecdar. We can, for instance, use this to conformance test during continuous integration.

Acknowledgements We thank Ulrik Nyman for supervision, and Florian Lorber and Ulrik for fruitful discussions throughout the project period.

References

[1] B.K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick & S. Tiran (2015): *Killing Strategies for Model-based Mutation Testing*. *Softw. Test. Verif. Reliab.* 25(8), pp. 716–748, doi:10.1002/stvr.1522.

[2] B.K. Aichernig, K. Hörmaier & F. Lorber (2014): *Debugging with Timed Automata Mutations*. In A. Bondavalli & F. Di Giandomenico, editors: *Computer Safety, Reliability, and Security*, Springer International Publishing, Cham, pp. 49–64, doi:10.1007/978-3-319-10506-2_4.

[3] B.K. Aichernig, F. Lorber & D. Ničković (2013): *Time for Mutants — Model-Based Mutation Testing with Timed Automata*. In M. Veanes & L. Viganò, editors: *Tests and Proofs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 20–38, doi:10.1007/978-3-642-38916-0_2.

[4] M. van der Bijl, A. Rensink & J. Tretmans (2004): *Compositional Testing with ioco*. In A. Petrenko & A. Ulrich, editors: *Formal Approaches to Software Testing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 86–100, doi:10.1007/978-3-540-24617-6_7.

[5] A. David, K.G. Larsen, A. Legay, M.H. Møller, U. Nyman, A.P. Ravn, A. Skou & A. Wąsowski (2012): *Compositional verification of real-time systems using Ecdar*. *International Journal on Software Tools for Technology Transfer* 14(6), pp. 703–720, doi:10.1007/s10009-012-0237-y.

[6] A. David, K.G. Larsen, A. Legay, U. Nyman & A. Wąsowski (2010): *ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems*. In A. Bouajjani & W. Chin, editors: *Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 365–370, doi:10.1007/978-3-642-15643-4_29.

[7] A. David, K.G. Larsen, A. Legay, U. Nyman & A. Wasowski (2010): *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, ACM, New York, NY, USA, pp. 91–100, doi:10.1145/1755952.1755967.

[8] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P. Schobbens & P. Heymans (2016): *Featured Model-based Mutation Analysis*. In: *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, ACM, New York, NY, USA, pp. 655–666, doi:10.1145/2884781.2884821.

[9] A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson & A. Skou (2008): *Testing Real-Time Systems Using UPPAAL*, pp. 77–117. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-78917-8_3.

[10] A. Hessel & P. Pettersson (2007): *Cover-a test-case generation tool for timed systems*. Available at `http://hessel.nu/CoVer/`.

[11] R.M. Hierons & M.G. Merayo (2007): *Mutation Testing from Probabilistic Finite State Machines*. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pp. 141–150, doi:10.1109/TAIC.PART.2007.20.

[12] Y. Jia & M. Harman (2011): *An Analysis and Survey of the Development of Mutation Testing*. *IEEE Transactions on Software Engineering* 37(5), pp. 649–678, doi:10.1109/TSE.2010.62.

[13] J.H. Kim, K.G. Larsen, B. Nielsen, M. Mikučionis & P. Olsen (2015): *Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools*. In M. Núñez & M. Güdemann, editors: *Formal Methods for Industrial Critical Systems*, Springer International Publishing, Cham, pp. 47–61, doi:10.1007/978-3-319-19458-5_4.

[14] K.G. Larsen, F. Lorber, B. Nielsen & U.M. Nyman (2017): *Mutation-Based Test-Case Generation with Ecdar*. In: *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 319–328, doi:10.1109/ICSTW.2017.60.

[15] F. Lorber, K.G. Larsen & B. Nielsen (2018): *Model-Based Mutation Testing of Real-Time Systems via Model Checking*. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. To appear.

[16] Object Management Group (2015): *OMG Unified Modeling Language*. `http://www.omg.org/spec/UML/2.5/PDF`.

[17] J. Tretmans (2008): *Model Based Testing with Labelled Transition Systems*, pp. 1–38. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-78917-8_1.