TECHNICAL REPORT

IST-MBT-2013-02

Model-based Mutation Testing with Timed Automata

Bernhard K. Aichernig[1], Florian Lorber[1], and
Dejan Ničković[2]

{aichernig, florber}@ist.tugraz.at, dejan.nickovic@ait.ac.at

January 2013

[1] Institute for Software Technology
Graz University of Technology
Graz, Austria

.

[2] AIT Austrian Institute of Technology
Department of Safety and Security
Vienna, Austria

Abstract. Model-based testing is a popular technology for automatic and systematic test case generation (TCG), where a system-under-test (SUT) is tested for conformance with a model that specifies its intended behavior. Model-based mutation testing is a specific variant of model-based testing that is fault-oriented. In mutation testing, the test case generation is guided by a *mutant*, an intentionally altered version of the original model that specifies a common modelling error.

In this paper, we propose a mutation testing framework for real-time applications, where the model of the SUT and its mutants are expressed as a variant of timed automata. We develop an algorithm for mutation-based real-time test case generation that uses symbolic bounded model checking techniques and incremental solving. We present an implementation of our test case generation technique and illustrate it with a non-trivial car alarm example, providing experimental results.

1 Introduction

A common practice to show that a system meets its requirements and works as expected consists in *testing* the system. Historically, testing has been predominantly a manual activity, where a human designs a test experiment, by choosing inputs that are fed to the SUT, and observing its reactions and outputs. Traditional testing suffers from two pitfalls: (1) due to a finite number of experiments, testing activity can only reveal presence of safety errors in the system, but not their absence; and (2) the testing process is manual, hence ad-hoc, time and human resource consuming and error-prone.

The first short-coming of testing was addressed by formal verification and theorem proving, which consist in providing rigorous evidence in the form of a mathematical proof that a system always behaves according to its specification. The automation of the verification technology was enabled with *model checking* [29, 12], a technique that consists in exhaustive exploration of the system's underlying state space. Although model checking resolves in theory the issues present in classical testing, it suffers from the so-called state-space explosion problems. In the past decades, large part of the effort invested by the verification research community went into developing methods that fight the state-space explosion problem (see for example [11, 7, 13]).

Model-based testing [32] was introduced as a pragmatic compromise between the conceptual simplicity of classical testing, and automation and exhaustiveness of model checking. In model-based testing, test suites are automatically generated from a mathematical *model* of the SUT. The main advantage of this technique is the full test automation that provides effective means to catch errors in the SUT. The aim of model-based testing is to check conformance of the SUT to a given specification, where the SUT is often seen as a "black-box" with unknown internal structure, but observable input/output interface. Model-based testing is commonly combined with some coverage criteria, with the aim to generate test cases that cover most possible use cases of the SUT.

Model-based mutation testing is a specific type of model-based testing, in which faults are deliberately injected into the specification model. The aim of

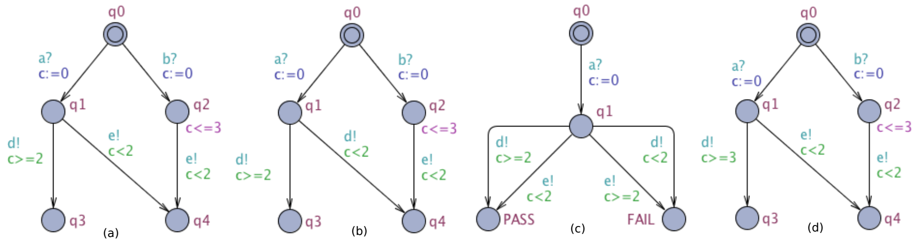

Fig. 1. Timed mutants example: (a) TA model A; (b) mutant M_1; (c) a test case generated from M_1 and (d) mutant M_2

mutation-based testing techniques is to generate test cases that can detect the injected errors. This means that a generated test case shall fail if it is executed on a (deterministic) system-under-test that implements the faulty model. The power of this testing approach is that it can guarantee the absence of certain specific faults. In practice, it will be combined with standard techniques, e.g. with random test-case generation [1]. Mutation-based testing was studied in [2, 30] in the context of UML models, and in [9, 16] in the context of Simulink models. Model-based mutation testing is also known as specification-based mutation testing. A recent survey by Jia and Harman [17] documents the growing interest in mutation testing and points out the open problem of generating test cases by means of mutation analysis. The present work contributes to this line of research.

Classical model-based testing is not always adapted to study embedded system models, which often contain real-time requirements. In the past decade, there was a number of attempts to extend model-based testing to the real-time context, requiring the usage of timed models for the specification of the SUT and adaptation of the classical conformance relations to real-time. A comprehensive overview and comparison of real-time conformance relations is presented in [31]. A framework for black-box conformance testing based on the model of partially-observable, non-deterministic timed automata is proposed in [18]. Two types of tests are considered: (1) analog-clock tests which are generated either offline with a restricted set of resources (clocks) or on-the-fly without that restriction; and (2) digital-clock tests that can measure time only with finite precision. In a similar work [23], a test case generation technique is developed for non-deterministic, but determinizable timed automata. Another extension of model-based testing to the real-time context is introduced in [10], where the authors provide an operational interpretation of the notion of quiescence for real-time behavior. In [20], an online testing tool for real-time systems is proposed, based on symbolic techniques of Uppaal [19]. Testing real-time models expressed as UML models is studied in [27], where the focus is given on the usage of bounded model checking techniques combined with abstract interpretation for the test case generation.

In this paper, we propose a framework for mutation testing of real-time systems modeled using a deterministic class of timed automata (TA) [3]. Given a

TA specification, we first propose a number of *mutation operators* which mimic common modeling errors. Given a specification model of an SUT and its mutant, we develop a technique for automatic generation of a real-time test case which exactly tries to "drive" the SUT to the error inserted by the mutation operator, as illustrated by the following example.

Example 1. Consider the TA model A from Figure 1 (a) and its timed mutants M_1 and M_2, shown in Figure 1 (b,d). The mutant M_1 alters the original specification by changing the output action of the transition $q_1 \rightarrow q_4$ from $e!$ to $d!$. In model-based mutation testing, we generate a test case leading to an error introduced by the mutant, resulting in the test case shown in Figure 1 (c). Note that not every mutant of a real-time specification introduces errors. The mutant M_2 is such an example.

In contrast to [18, 23, 10, 20], we propose a *symbolic* test generation procedure based on bounded model checking (BMC) techniques. We believe that using SMT solvers to generate test cases is a promising technique in our context because mutation testing is fault oriented, and focuses on finding finite traces which expose faults resulting from mutating a specification. In addition, SMT solvers provide support for future extensions such as handling unbounded data domains.We are not aware of other work which applies BMC techniques for testing TA models.

The survey of Fraser et al. [15] gives a great overview of different previous approaches to generate test cases via model checking and shows the progress made in the past years. [8] investigates problems and solutions for test case generation via model checkers for non-deterministic specifications. The work by Nilsson et al. [26, 24] proposes a mutation-based testing framework for real-time systems using Timed Automata with Tasks. In contrary to our work, he generates test cases for mutants that violate task deadlines, while we check the conformance between mutants and specification with regard to a timed input/output conformance relation. While some of the mutation operators we propose are specific to timed automata, most of them can be related to the state chart operators introduced in [14].

We first introduce in Section 2 a TA variant for real-time SUT specification. In Section 3, we propose several mutation operators adapted to the TA model. We recall the timed input/output conformance relation tioco in Section 4 and discuss how we handle quiescence in our framework. In Section 5, we propose an encoding of TA k-reachability problem into an SMT problem and develop an algorithm based on BMC for automatic test-case generation from a TA specification and its timed mutant. In Section 6 we present an implementation of our mutation testing for real-time systems. Section 7 describes a car alarm system case study used to illustrate our approach, and presents experimental results. In Section 8, we conclude the paper.

2 Timed Automata with Inputs and Outputs

The time domain that we consider is the set $\mathbb{R}_{\geq 0}$ of non-negative reals. We denote by Σ the finite set of actions, partitioned into two disjoint sets Σ_I and Σ_O of input and output actions, respectively. A *time sequence* is a finite or infinite non-decreasing sequence of non-negative reals. A *timed trace* σ is a finite alternating sequence of actions and time delays of the form $t_1 \cdot a_1 \cdots t_k \cdot a_k \cdots$, where for all $i \geq 1$, $a_i \in \Sigma$ and $(t_i)_{i \geq 1}$ is a time sequence.

Let $\mathcal{C}$ be a finite set of *clock* variables. Clock *valuation* $v(c)$ is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$ assigning a real value to every clock $c \in \mathcal{C}$. We denote by $\mathcal{H}$ the set of all clock valuations and by $\mathbf{0}$ the valuation assigning 0 to every clock in $\mathcal{C}$. Let $v \in \mathcal{H}$ be a valuation and $t \in \mathbb{R}_{\geq 0}$, we then have $v + t$ defined by $(v + t)(c) = v(c) + t$ for all $c \in \mathcal{C}$. For a subset ρ of $\mathcal{C}$, we denote by $v[\rho]$ the valuation such that for every $c \in \rho$, $v[\rho](c) = 0$ and for every $c \in \mathcal{C} \backslash \rho$, $v[\rho](c) = v(c)$.

A *clock constraint* φ is a conjunction of predicates over clock variables in $\mathcal{C}$ defined by the grammar

$$\varphi ::= c \circ k \mid \varphi_1 \wedge \varphi_2,$$

where $c \in \mathcal{C}$, $k \in \mathbb{N}$ and $\circ \in \{<, \leq, =, \geq, >\}$. Given a clock valuation $v \in \mathcal{H}$, we write $v \models \varphi$ when v satisfies the clock constraint φ. We are now ready to formally define *input/output* timed automata (TAIO):

Definition 1. *An* input/output timed automaton[1] *A is a tuple* $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$, *where:*

- *Q is a finite set of* locations*;*
- *$\hat{q} \in Q$ is the* initial *location;*
- *Σ_I is a finite set of* input actions *and Σ_O is a finite set of* output actions*, such that $\Sigma_I \cap \Sigma_O = \emptyset$. We denote by Σ the set of actions $\Sigma_I \cup \Sigma_O$;*
- *$\mathcal{C}$ is a finite set of* clock *variables;*
- *I is a finite set of location* invariants*, that are conjunctions of constraints of the form $c < d$ or $c \leq d$, where $c \in \mathcal{C}$ and $d \in \mathbb{N}$. Each invariant is bound to its specific location;*
- *Δ is a finite set of* transitions *of the form (q, a, g, ρ, q'), where*
  - *$q, q' \in Q$ are the* source *and the* target *locations;*
  - *$a \in \Sigma$ is the transition action*
  - *g is a* guard*, a conjunction of constraints of the form $c \circ d$, where $\circ \in \{<, \leq, =, \geq, >\}$ and $d \in \mathbb{N}$;*
  - *$\rho \subseteq \mathcal{C}$ is a set of clocks to be* reset*.*

[1] Our TAIO are very similar to UPPAAL timed automata, which we use to illustrate our examples. One difference is that for simplicity of presentation we do not have *urgent* and *committed* locations. However, these types of locations are just syntactic sugar to make modelliing easier, and can be expressed with standard timed automata.

We say that a TAIO A is *deterministic* if for all (q, a, g_1, ρ_1, q_1) and (q, a, g_2, ρ_2, q_2) in Δ, $q_1 \neq q_2$ implies that $g_1 \wedge g_2 = \emptyset$. We denote by $\mathcal{A}$ the set of all TAIO and by $\mathrm{Det}(\mathcal{A}) \subset \mathcal{A}$ the set of all deterministic TAIO. We denote by $\Delta_O \subseteq \Delta$ the set $\{\delta = (q, a, g, \rho, q') \mid \delta \in \Delta \text{ and } a \in \Sigma_O\}$ of transitions in Δ labeled by an output actions and by $\Delta_I = \Delta \backslash \Delta_O$ the set of transitions in Δ labeled by an input action. We also define $|\mathcal{G}|$ as the number of basic constraints that appear in all the guards of all the transitions in A, i.e. $|\mathcal{G}| = \Sigma_{\delta \in \Delta} |J_g|$, where $\delta = (q, a, g, \rho, q')$ and g is of the form $\bigwedge_{j \in J_g} c_j \circ d_j$. We similarly define $|\mathcal{I}|$ as the number of basic constraints that appear in all the invariants of all the locations in A.

The *semantics* of a TAIO $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$ is given by the *timed input/output transition system* (TIOTS) $[[A]] = (S, s_0, \mathbb{R}_{\geq 0}, \Sigma, T)$, where

- $S = \{(q, v) \in Q \times \mathcal{H} \mid v \models I(q)\}$;
- $s_0 = (q_0, \mathbf{0})$;
- $T \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ is the transition relation consisting of *discrete* and *timed* transitions such that:
  - **Discrete transitions:** $((q, v), a, (q', v')) \in T$, where $a \in \Sigma$, if there exists a transition (q, a, g, ρ, q') in Δ, such that: (1) $v \models g$; (2) $v' = v[\rho]$ and (3) $v' \models I(q')$; and
  - **Timed transitions:** $((q, v), t, (q, v + t)) \in T$, where $t \in \mathbb{R}_{\geq 0}$, if $v + t \models I(q)$.

A *run* r of a TAIO A is the sequence of alternating timed and discrete transitions of the form

$$(q_1, v_1) \xrightarrow{t_1} (q_1, v_1 + t_1) \xrightarrow{\delta_1} (q_2, v_2) \xrightarrow{t_2} \cdots ,$$

where $q_1 = \hat{q}$, $v_1 = \mathbf{0}$ and $\delta_i = (q_i, a_i, g_i, \rho_i, q_{i+1})$. The run r of A induces the timed trace $\sigma = t_1 \cdot a_1 \cdot t_2 \cdots$. We denote by $L(A)$ the set of timed traces induced by all runs of A.

3 Mutation of TAIOs

Mutation of a specification consists in altering the model in a small way, mimicking common modelling errors. In our setting, a mutation is a function $\mu_{\mathrm{m}} : \mathrm{Det}(\mathcal{A}) \to 2^{\mathcal{A}}$ parametrized by a mutation operator m which maps a deterministic TAIO A into a finite set $\mu_{\mathrm{m}}(A)$ of possibly non-deterministic TAIOs , where each $M \in \mu_{\mathrm{m}}(A)$ is called an m-mutant of A. We now introduce and define specific mutation operators which are relevant to the TAIO model.

Definition 2. *Given a* TAIO $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$, *its mutants are defined by the following* mutation operators*:*

1. ***Change action** (μ_{ca}) generates from A a set of $|\Delta_I|(|\Sigma_O|) + |\Delta_O|(|\Sigma_O| - 1)$ mutants, where every mutant changes a single transition in A by replacing the action labeling the transition by a different output label. A* TAIO *$M \in \mu_{ca}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a_m, g, \rho, q')$, $a_m \in \Sigma_O$ and $a_m \neq a$;*

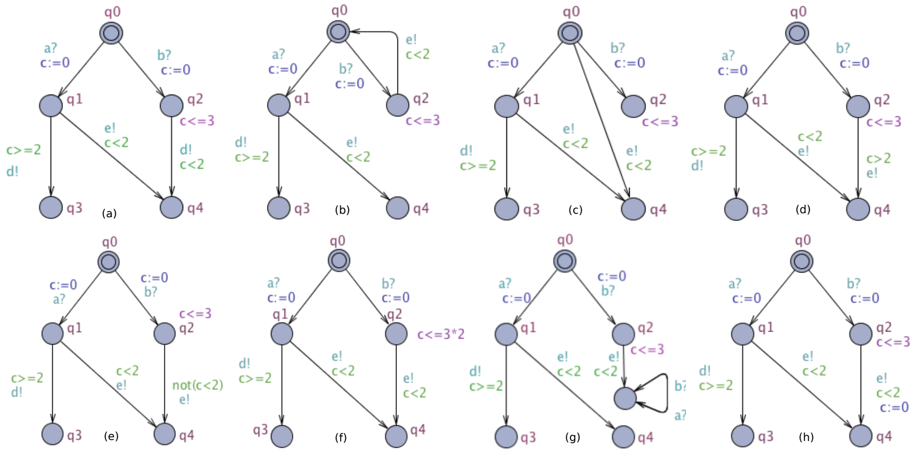

Fig. 2. Mutant M of model A resulting from: (a) $\mu_{ca}(A)$; (b) $\mu_{ct}(A)$; (c) $\mu_{cs}(A)$; (d) $\mu_{cg}(A)$; (e) $\mu_{ng}(A)$; (f) $\mu_{ci}(A)$; (g) $\mu_{sl}(A)$; (h) $\mu_{ir}(A)$;

2. ***Change target** (μ_{ct}) generates from A a set of $|\Delta|(|Q|-1)$ mutants, where every mutant replaces the target location of a transition in A, by another location in A. A* TAIO *$M \in \mu_{ct}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g, \rho, q'_m)$, $q'_m \in Q$ and $q'_m \neq q'$;*
3. ***Change source**[2] (μ_{cs}) generates from A a set of $|\Delta|(|Q|-1)$ mutants, where every mutant replaces the source location of a transition in A, by another location in A. A* TAIO *$M \in \mu_{cs}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q_m, a, g, \rho, q')$, $q_m \in Q$ and $q_m \neq q$;*
4. ***Change guard** (μ_{cg}) generates from A a set of $4|\mathcal{G}|$ mutants, where every mutant replaces a transition in A with another one which changes the original guard by altering every equality/inequality sign appearing in the guard by another one. A* TAIO *$M \in \mu_{cg}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\}))$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g_m, \rho, q')$, $g = \bigwedge_{i \in I} c_i \circ_i d_i$, $g_m = \bigwedge_{i \in I} c_i \circ_i^m d_i$, $\circ, \circ_i^m \in \{<, \leq, =, \geq, >\}$, $\circ_i \neq \circ_i^m$ for some $i \in I$ and $\circ_j = \circ_j^m$ for all $j \neq i$;*
5. ***Negate guard** (μ_{ng}) generates from A a set of $|\Delta|$ mutants, where every mutant replaces the guard in a transition in A, by its negation. A* TAIO

[2] Note that change source and target mutation operators also generate mutants where a self-loop transition is created, hence there is no need for a separate "self-loop" mutation operator.

$M \in \mu_{ng}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$ and $\delta_m = (q_m, a, \neg g, \rho, q')$[3];

6. ***Change invariant (μ_{ci})*** *generates from A a set of $|\mathcal{I}|$ mutants, where every mutant replaces the invariant of a location with another invariant which multiplies all original constants by a factor of 2. A* TAIO *$M \in \mu_{ci}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I_m, \Delta)$, and there exists $q \in Q$ such that $I(q) = \bigwedge_{i \in I} c_i \circ d_i$, $\circ \in \{<, \leq\}$, $I_m(q) = \bigwedge_{i \in I} c_i \circ d_i^m$, $d_i^m = 2d_i$ for some $i \in I$, $d_j^m = d_j$ for all $j \neq i$ and $I(q') = I_m(q')$ for all $q' \neq q$;*
7. ***Sink location (μ_{sl})*** *generates from A a set of $|\Delta|$ mutants, where every mutant replaces the target location of a transition in A, by a newly created sink location which models a don't care location which accepts all inputs. A* TAIO *$M \in \mu_{sl}(A)$, if M is of the form $(Q \cup \{sink\}, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\} \cup \Delta_{sink})$, such that $\Delta_{sink} = \{(sink, a, true, \{\}, sink) \mid a \in \Sigma_I\}$, $\delta = (q, a, g, \rho, q') \in \Delta$ and $\delta_m = (q, a, g, \rho, sink)$;*
8. ***Invert reset (μ_{ir})*** *generates from A a set of $|\Delta||\mathcal{C}|$ mutants, where every mutant replaces a transition in A, by another transition with the occurence of one clock flipped compared to the original set of clocks. A* TAIO *$M \in \mu_{cs}(A)$, if M is of the form $(Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, (\Delta \backslash \{\delta\}) \cup \{\delta_m\})$, such that $\delta = (q, a, g, \rho, q') \in \Delta$, $\delta_m = (q, a, g, \rho_m, q')$, and for some $c \in \mathcal{C}$ either $\rho_m = \rho \cup \{c_m\}$ if $c_m \notin \rho$, or $\rho_m = \rho \backslash \{c_m\}$ if $c_m \in \rho$.*

Figure 2 illustrates mutants resulting from applying the above mutation operators to the model A from Figure 1(a). The effectiveness of the mutation operators is analysed and evaluated in more details in Section 7. For more complex models there might rise the need to reduce the amount of mutants. Here we refer to the survey by Jia and Harmann[17], that describes multiple ways of reducing mutants for mutation testing, which can in gerneal also be applied to model-based mutationt testing.

Several different approaches to model mutation have already been published, using Finite State Machines [28, 14], Kripke structures [8] or Event Sequence Graphs [6]. [25] introduces mutation operators for Timed Automata with Tasks, yet his mutation operators concentrate on tasks and timeliness and not the core essence of timed automata. Our mutation operators 6 and 8 are specific to timed automata, the other mutation operators described in this section are similar or closly related to the operators described in [14].

4 Conformance Relations for Timed Automata

Different real-time extensions of the input-output conformance relation ioco were studied and compared in [31]. We consider the timed input-output conformance relation introduced in [18] and inspired by ioco. Intuitively, $\mathcal{A}_I$ conforms to $\mathcal{A}_S$ if for each observable behavior specified in $\mathcal{A}_S$, the possible outputs of $\mathcal{A}_I$ after this

[3] For the sake of simplicity, we represent δ_m as a single transition even though $\neg g$ may also have disjunctions. The guard $\neg g$ can be represented in DNF and every disjunction of the guard can be used as a guard of a separate transition.

behavior is a subset of the possible outputs of $\mathcal{A}_S$. In contrast to ioco, tioco does not use the notion of quiescence, but requires explicit specification of timeouts. Since we consider TAIO without silent (τ) transitions, all actions are observable. Hence, we present a simplified version of the tioco definition from [18].

Given a TAIO A and $\sigma \in \mathrm{RT}(\Sigma)$, $A\,\mathrm{after}\,\sigma$ is the set of all states of A that can be reached by the sequence σ, i.e.

$$A\,\mathrm{after}\,\sigma = \{s \in S \mid \hat{s} \xrightarrow{\sigma} s\}.$$

Given a state $s \in S$, elapse(s) is the set of all delays that can elapse from s without A making any action:

$$\mathrm{elapse(s)} = \{t > 0 \mid s \xrightarrow{t}\}$$

Given a state $s \in S$, $\mathrm{out}(s)$ is the set of all output actions or time delays that can occur when the system is at state s.

$$\mathrm{out}(s) = \{a \in \Sigma_O \mid s \xrightarrow{a}\} \cup \mathrm{elapse(s)}$$

This definition naturally extends to a set of states S, i.e. $\mathrm{out}(S) = \bigcup_{s \in S} \mathrm{out}(s)$.

Definition 3. *The* timed input-output conformance relation*, denoted tioco, is defined as*

$$A_I\ \mathit{tioco}\ A_S\ \mathit{iff}\ \forall \sigma \in\ L(A_S)\ :\ \mathit{out}(A_I\ \mathit{after}\,\sigma) \subseteq \mathit{out}(A_S\ \mathit{after}\,\sigma)$$

In [18], the authors develop a number of theoretical results about the tioco relation. In particular, they establish that given two TAIO A_I and A_S, if A_I tioco A_S, then the set of observable traces of A_I is included in the set of observable traces of A_S, while the converse is not true in general. However, if A_S is input-enabled, then the set inclusion between observable traces of A_I and A_S also implies the tioco conformance of A_I to A_S.

Specification automaton A_S has often intentionally under-specified inputs in order to model assumptions about the environment in which the SUT is designed to operate correctly. Hence, the input-enabledness is not a desired requirement for A_S in this context. In [32], [18], the notion of *demonic completion* was introduced to transform automatically a model A_S and make it input-enabled. In essence, all non-specified inputs in all locations of A_S lead to a new *sink* "don't care" location, from which any behavior is possible. Figure 3 illustrates demonic completion $d(A)$ of a TAIO A.

Formally, given a deterministic TAIO $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$, its demonic completion $d(A)$ is the input-enabled TAIO $d(A) = (Q \cup \{\mathrm{sink}\}, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I_d, \Delta_d)$, where

- $I_d(q) = I(q)$ and $I_d(\mathrm{sink}) = \mathrm{true}$
- $\Delta_d = \Delta \cup \{(\mathrm{sink}, a, \mathrm{true}, \{\}, \mathrm{sink}) \mid a \in \Sigma\} \cup \{(q, a, \neg g, \{\}, \mathrm{sink}) \mid q \in Q \wedge a \in \Sigma_I\}$, such that for each $q \in Q$ and $a \in \Sigma_I$, $g = (g_1 \vee \ldots \vee g_k) \wedge I(q)$, where $\{g_i\}_i$ are guards of the outgoing transitions of q labeled by a.

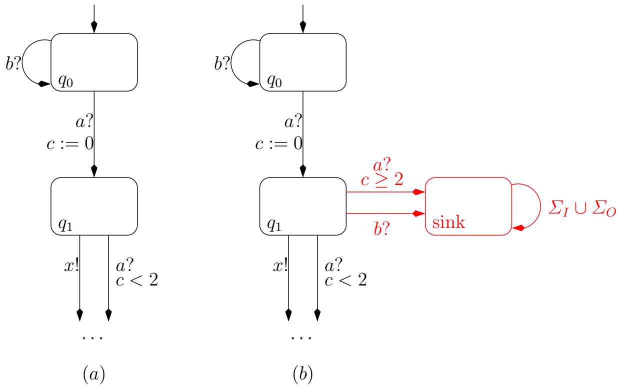

Fig. 3. Demonic completion of TAIO: (a) A; and (b) $d(A)$.

It is not hard to see that

$$L(d(A)) = L(A) \cup \{\sigma \cdot a \cdot (\mathbb{R}_{\geq 0} \cdot \Sigma)^* \mid a \in \Sigma_I, \sigma \in L(A) \wedge \sigma \cdot a \notin L(A)\}$$

Given an arbitrary TAIO A_I and a deterministic specification TAIO A_S, considering the demonic completion $d(A_S)$ instead of A_S does not affect the conformance relation. Formally, we have the following proposition, proved in [18].

Proposition 1. *Given a deterministic* TAIO A_S *and its demonic completion* $d(A_S)$*, for any* TAIO A_I*,* A_I *tioco* A_S *if and only if* A_I *tioco* $d(A_S)$*.*

It turns out that given two TAIO A_S and A_I, by applying demonic completion $d(A_S)$ to A_S, checking tioco of A_I to A_S is equivalent to checking the language inclusion $L(A_I) \subseteq L(d(A_S))$, as shown in [18].

Proposition 2. *Given a* TAIO A_I *and a deterministic* TAIO A_S*,* A_I *tioco* A_S *if and only if* $L(A_I) \subseteq L(d(A_S))$*.*

By Proposition 2, it follows that the tioco conformance of A_I to A_S when A_S is deterministic can be replaced by language inclusion check between A_I and $d(A_S)$. Finally, the problem of checking $L(A_I) \subseteq L(d(A_S))$ is decidable when A_S is deterministic [3].

Remark: *Quiescence* was introduced in the ioco conformance relation to distinguish states which do not accept any output actions, and thus prevent the system to autonomously proceed without external stimuli. In practice, testing an SUT in a quiescent state consists in waiting for some predetermined timeout to expire, ensuring that the SUT does not generate output actions. After timeout expiration, it is assumed that the SUT will not generate output actions. A timed extension of ioco from [10] introduces the notion of M-quiescence which makes the timeout an explicit parameter of the definition, resulting in a family of conformance relations. In contrast, tioco does not use quiescence, but rather expects timeouts to be part of the specification model. We believe that the tioco approach is more natural since it exposes timeouts in an explicit way and gives more flexibility to the engineer, while resulting in a more elegant definition of the conformance relation. However, we do not put restrictions on our TAIO model, and allow true invariants and guards. As a consequence, we add an additional global timeout, but defer it to the test driver, as explained in Section 7.

5 Symbolic Test Case Generation from Timed Mutants

In classical model-based testing, the SUT is often not known in advance and can be seen as a black box. The conformance relations such as ioco and tioco serve to establish soundness of the TCG algorithms, but are not actually computed between two models. In fact, only the specification model is explored in order to generate test cases, and the conformance relation is used to decide which tests pass and which fail.

In contrast, mutation testing requires effective conformance checking of the mutated model to the original specification model. Mutation testing is a particular instance of fault oriented testing where the test cases are generated in a way that tries to "steer" the SUT towards failure, due to a common modeling error if one exists. Hence, the rational behind this approach is that if the mutated model does not conform to its original version, the mutation introduces traces which were not in the original model, and the non-conformance witness trace serves as the basis to generate a test case. In case that the mutated model conforms to its original version, the mutation does not introduce new behaviors with respect to the original specification, hence no useful test case is generated. It follows that in mutation-based testing, test cases are generated only if the mutated model does not conform to its original version. We propose a TCG algorithm which can be summarized as follows:

1. Given a deterministic TAIO A, a mutation operator m and a mutation function μ_m, generate the mutant $M \in \mu_m(A)$;
2. Generate $d(A)$ by demonic completion of A;
3. Check M tioco A, by effectively checking the language inclusion $L(M) \subseteq L(d(A))$;
4. If $L(M) \not\subseteq L(d(A))$, generate a test case based on the trace which witnesses non conformance of M to A.

The steps 1 and 2 were already presented in Section 3 and 4, respectively. In this section, we detail the steps 3 and 4 of our test case generation framework.

5.1 *k*-Bounded Language Inclusion

We have seen that mutation-based testing is fault-oriented, i.e. test cases are generated only if the mutated model does not conform to its original version. Consequently, symbolic techniques based on BMC are well-adapted to solve this type of problems. BMC was used in [4, 22] for the reachability analysis of TA, and in [5] for checking the language inclusion between two timed automata. We now show how the language inclusion problem from 3 can be encoded as a k-bounded language inclusion SMT problem. Intuitively, given two TAIO A_I and A_S such that A_S is deterministic and an integer bound k, we have $L(A_I) \not\subseteq^k L(A_S)$ if there exists a timed trace $\sigma = t_1 \cdot a_1 \cdots t_i \cdot a_i$ such that $i \leq k$, $\sigma \in L(A_I)$ and $\sigma \notin L(A_S)$. We now show how to construct a formula $\varphi^k_{A_I,A_S}$ that is satisfiable if and only if $L(A_I) \not\subseteq^k L(A_S)$.

Let $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$ be a TAIO. We denote by $\mathrm{loc}_A : Q \rightarrow \{1, \ldots, |Q|\}$ and $\mathrm{act}_A : \Sigma \rightarrow \{1, \ldots, |\Sigma|\}$ functions assigning unique integers to locations and actions in A, respectively. Given A and a constant k, we denote by X the set of variables $\{x^1, \ldots, x^{k+1}\}$ that range over the domain $\{1, \ldots, |Q|\}$, where x^i encodes the location of A after the i^{th} step. Similarly, let $\mathcal{A} = \{\alpha^1, \ldots, \alpha^k\}$ be the set of variables ranging over $\{1, \ldots, |\Sigma|\}$, where α^i encodes the action in A applied in the i^{th} discrete step. We denote by $D = \{d^1, \ldots, d^k\}$ the set of real-valued variables, where d^i encodes the delay action applied in the i^{th} time step. Let C^i denote the set of real variables obtained by renaming every clock $c \in \mathcal{C}$ by c^i. We denote by $C = \bigcup_{i=1}^{k+1} C^i \cup \bigcup_{i=1}^{k+1} C^{*,i}$ the set of real (clock valuation) variables, where $c^{*,i} \in C^{*,i}$ and $c^i \in C^i$ encode the valuation of the clock $c \in \mathcal{C}$ after the i^{th} timed and step, respectively.

We express the effect of applying Reset_ρ in the i^{th} step of a run to the set $\mathcal{C}$ of clocks in A as follows:

$$\mathrm{doReset}^i_{A,\rho}(C) \equiv \bigwedge_{c \in \rho} c^{i+1} = 0 \wedge \bigwedge_{c \notin \rho} c^{i+1} = c^{*,i}$$

We express the i^{th} passage of time in A as follows:

$$\mathrm{tDelay}^i_A(D, C) \equiv \bigwedge_{c \in \mathcal{C}} (c^{*,i} - c^i) = d^i$$

The i^{th} time step in a location $q \in Q$ is expressed with:

$$\mathrm{tStep}^i_{A,q}(D, X, C) \equiv x^i = \mathrm{loc}_A(q) \wedge \mathrm{tDelay}^i_A(D, C) \wedge I(q)[\mathcal{C} \backslash C^{*,i}],$$

where $I(q)[\mathcal{C} \backslash C^{*,i}]$ is the invariant of q, with every clock $c \in \mathcal{C}$ substituted by $c^{*,i}$. The formula for the i^{th} discrete step is:

$$\begin{aligned}\mathrm{dStep}^i_{A,\delta}(\mathcal{A}, X, C) \equiv x^i = \mathrm{loc}_A(q) \wedge \alpha^i = \mathrm{act}_A(a) \wedge g[\mathcal{C} \backslash C^{*,i}] \wedge \\ \mathrm{doReset}^i_{A,\rho}(C) \wedge x^{i+1} = \mathrm{loc}_A(q')\end{aligned}$$

where $g[\mathcal{C} \backslash C^{*,i}]$ denotes the guard of δ, where every clock $c \in \mathcal{C}$ is substituted by $c^{*,i}$. We express the segment of a path in TAIO A from j to k with the following formula:

$$\mathrm{path}^{j,k}_A(\mathcal{A}, D, X, C) \equiv \bigwedge_{i=j}^{k} (\bigvee_{q \in Q} \mathrm{tStep}^i_{A,q}(D, X, C) \wedge \bigvee_{\delta \in \Delta} \mathrm{dStep}^i_{A,\delta}(\mathcal{A}, X, C))$$

The initial state of TAIO A is expressed as follows:

$$\text{init}_A(X, C) \equiv x^1 = \text{loc}_A(\hat{q}) \wedge \bigwedge_{c \in \mathcal{C}} (c^1 = 0)$$

Let $A_I = (Q_I, \hat{q}_I, \Sigma_I, \Sigma_O, \mathcal{C}, I_I, \Delta_I)$ and $A_S = (Q_S, \hat{q}_S, \Sigma_I, \Sigma_O, \mathcal{C}, I_S, \Delta_S)$ be two TAIOs such that A_S is deterministic. The general formula $\varphi^k_{A_I,A_S}(i, \mathcal{A}, D, X_I, X_S, C_I, C_S)$ which specifies the negation of k-language inclusion is expressed as follows:

$$\begin{aligned}\varphi^k_{A_I,A_S} \equiv & \textstyle\bigwedge_{i=1}^{k}(d^i \geq 0 \wedge \alpha^i \geq 1 \wedge \alpha^i \leq |\Sigma|) \wedge i \geq 1 \wedge i \leq k \quad \wedge \\ & \text{init}_{A_I}(X_I, C_I) \wedge \text{init}_{A_S}(X_S, C_S) \wedge \text{path}^{1,i}_{A_I}(\mathcal{A}, D, X_I, C_I) \wedge \\ & \text{path}^{1,i-1}_{A_S}(\mathcal{A}, D, X_S, C_S) \wedge \neg\text{path}^{i,i}_{A_S}(\mathcal{A}, D, X_S, C_S)\end{aligned}$$

5.2 Test Case Generation

Given a specification model A and its mutant M, our test case generation algorithm creates a *test* only if M does not conform to A. The generated test follows a *test purpose* which is in our case the timed trace σ which witnesses the non conformance of M to A and exposes the error caused by the mutation in M. We denote a test by A_T and give it in a form of a deterministic TAIO.

The test A_T specifies the execution of real-time traces and provides a *verdict* after observing at most k combined (timed/discrete) steps of a trace. The verdict can be:

- *Pass* (**pass**) - if the test purpose was successfully reached and the error introduced by the mutant was not exposed by the SUT during the test execution;
- *Inconclusive* (**inc**) - if the test purpose covering the fault introduced by the mutant could not be reached by the SUT during the test execution;
- *Fail* (**fail**) - if the fault introduced by the mutant as part of the test purposed was exposed by the SUT during the test execution.

The skeleton of A_T consists of the sequence $q_1 \cdot \delta_1 \cdots q_k \cdot \delta_k$ of locations and transitions in A which are executed while observing the witness trace $\sigma = t_1 \cdot a_i \cdots t_k \cdot a_k$. This skeleton corresponds effectively to the test purpose described above. In addition, A_T is completed in order to satisfy a number of properties described next. After observing a prefix $\sigma' = t_1 \cdot a_1 \cdots t_i \cdot a_i$ of σ, A_T is in location q_i, where $i < k$, and can do one of the following:

- Wait if the invariant of q_i allows a positive time delay;
- Emit action a if a is an input action equal to a_i and the transition δ_i is enabled, and move to location q_{i+1};
- Accept action a if a is an output action equal to a_i and the transition δ_i is enabled, and move to location q_{i+1};
- Accept action a if a is an output action different from a_i and there exists an enabled transition δ in A with source location q_i and labeled with a, and move to the **inc** verdict location;

- Refuse action a if a is an output action and there is no transitions in A with the source location q_i which is both labeled by a and enabled, and move to the **fail** verdict location.

Finally, when A_T is in location q_k, it accepts all output actions a such that there exists an enabled transition δ in A with the source location q_k and labeled by a, moving to the **pass** verdict location, and it rejects all other output actions, moving to the **fail** verdict locations. The test case generation procedure is formalized in Algorithm 1.

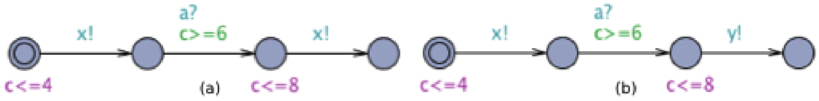

Fig. 4. Necessity of symbolic constraints on inputs in a test.

Note that our test A_T follows a fixed qualitative sequence of actions, defined by the witness σ. In particular, it stops following a valid output according to the specification A if it differs from the one defined in the witness σ, by returning the **inc** verdict. It means that the test is not pursued when the SUT deviates from the test purpose. On the other hand, A_T is time adaptive, and the witness σ defines a class of timing constraints which are allowed by the test. In fact, it is highly unlikely that an expected output action is preceded by the exact time delay as defined by the witness trace. Hence, we need the test to be more flexible and to accept the expected output in a larger time range defined by the specification model. In addition, if we allow time flexibility for output actions, we cannot use the strict time delay, defined by the witness trace σ, to precede an input action either, since it may violate input assumptions of the specification during some test executions. We illustrate this observation in Figure 4, which depicts model A (Figure 4 (a)), and its mutant M (Figure 4 (b)). The trace $\sigma = 4 \cdot x! \cdot 2 \cdot a? \cdot 2 \cdot y!$ witnesses the non-conformance of M to A and can be used to develop the skeleton of the test A_T. During the test execution, the test may observe the prefix $\sigma' = 2 \cdot x!$ which is allowed by the specification. In that case, if A_T requires exactly 2 time units to elapse between observing $x!$ and emitting $a?$, the assumptions expressed by A are violated. Hence the need to keep the timing constraints symbolic, defining the elapse of time between $x!$ and $a?$ dependent on the previous observations.

6 Implementation

In this section, we present the tool that implements the test case generation framework described in Section 5. The implementation of the algorithms is done in Scala (v2.9.1). We use standard Uppaal TA XML format to model TAIO

Algorithm 1 Test case generation algorithm.

```
Input: $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$ and $q_1 \cdot \delta_1 \cdots q_k \cdot \delta_k \cdot q_{k+1}$
Output: Test automaton $A_T$
$Q_T \leftarrow \bigcup_{i=1}^{k} \{q_i\} \cup \{\mathbf{pass}, \mathbf{fail}, \mathbf{inc}\}$
$\Delta_T \leftarrow \bigcup_{i=1}^{k-1} \{\delta_i\}$
for $i = 1$ to $k$ do
    for all $(q_i, a, g, \rho, q') \in \Delta \backslash \{\delta_i\}$ st. $a \in \Sigma_O$ do
        if $i < k$ then
            $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, g, \{\}, \mathbf{inc})\}$
        else
            $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, g, \{\}, \mathbf{pass})\}$
        end if
    end for
    for all $a \in \Sigma_O$ st. $\exists (q_i, a, g, \rho, q') \in \Delta$ do
        $g_T \leftarrow (g_1 \vee \ldots \vee g_j) \wedge I(q)$ st. $\{g_i\}$ are guards of outgoing transitions from $q_T$
        labeled by $a$
        $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, \neg g_T, \{\}, \mathbf{fail})\}$
    end for
    for all $a \in \Sigma_O$ st. $\nexists (q_i, a, g, \rho, q') \in \Delta$ do
        $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, \mathrm{true}, \{\}, \mathbf{fail})\}$
    end for
end for
return $A_T \leftarrow (Q_T, \hat{q}, \Sigma_O, \Sigma_I, \mathcal{C}, I, \Delta_T)$
```

specifications. The (bounded) language inclusion between two TAIOs is computed using the Z3 (v4.0) SMT solver [21]. The communication between our implementation in Scala and the Z3 solver relies on the Scala^Z3 API. Figure 5 gives an overview of different tool component interactions.

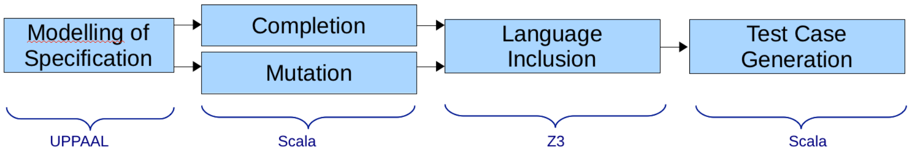

Fig. 5. Test case generation framework.

The test case generation framework, depicted in Figure 5, consists of four main steps:

1. Parsing and demonic completion of the TAIO model;
2. Mutation of the TAIO model;
3. Language inclusion between the original model and its mutant;
4. Test case generation.

In what follows, we present more details about these steps.

Specification Parsing and Demonic Completion: The TAIO model specified in the Uppaal XML format is parsed with Scala's parser generator. We require the following restrictions on the Uppaal automata in order to guarantee their compliance with the TAIO model: (1) one automaton per file; (2) no urgent nor committed locations. Of course urgent locations can still be expressed via invariants. We note that the actual modelling can have important impact on the number and effectiveness of consecutive generation of mutants and test cases.

We implemented the demonic completion of the model by straighforward application of the procedure from Section 4.

Mutation of Models: Our tool supports all mutation operators introduced in Definition 2. We store each mutant as a separate Uppaal XML model.

Language Inclusion: The language inclusion check between a model and its mutant is at core of the TCG framework. We translate an Uppaal model and its mutant to a bounded language inclusion problem expressed as an SMT-LIB2 formula, following the procedure described in Section 5.1. The formula is then fed to the Z3 solver, which checks whether there exists a satisfying assignment to the variables which represents a witness trace violating the language inclusion property.

Apart from this standard procedure, we also implemented the same TCG algorithm using Z3's incremental solving feature, with the aim to improve the computation time of the bounded language inclusion check. Given an SMT formula expressing the k-bounded language inclusion problem, we first feed the Z3 solver with the sub-formula for the i-bounded language inclusion problem, for some i smaller than k. Z3 checks the satisfiability of the sub-formula, and if a satisfying assignment is found, the procedure can stop. Otherwise, we pop the sub-formula from the Z3 stack and push the sub-formula expressing the step from i to $i+1$. The procedure is iterated until a witness is found or the k bound is reached.

Test case generation: If Z3 generates a counter-example which witnesses violation of language inclusion between the specification and its mutant, we use this counter-example together with the specification model in order to generate a test case. The test case generation implementation closely follows Algorithm 1.

7 Case Study and Experimental Results

In this section we illustrate our TCG approach with the Car Alarm System (CAS) [2, 30] and evaluate the framework, presenting the evaluation results.

The car alarm system (CAS) is a model inspired by the Ford's demonstrator developed in the EU FP7 project MOGENTES[4]. The model is developed from the following requirements provided by Ford:

1. *Arming:* The system is armed $20s$ after the vehicle is locked and the bonnet, luggage compartment and all doors are closed;

[4] http://www.mogentes.eu/

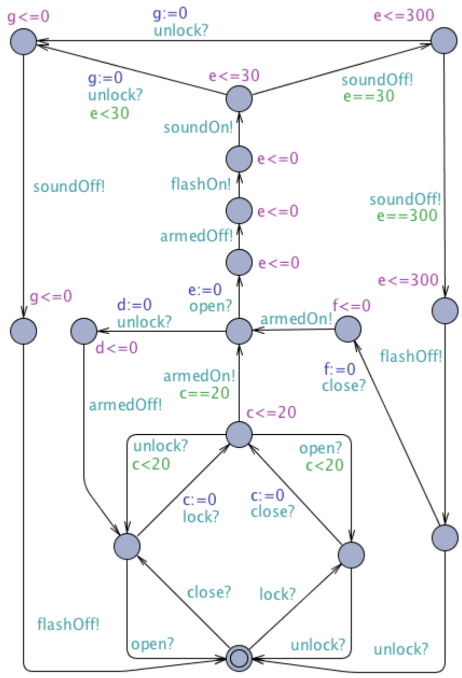

Fig. 6. A TAIO model of the car alarm system.

2. *Alarm:* The alarm sounds for $30s$ if an unauthorized person opens the door, the luggage compartment or the bonnet. The hazard flasher lights flashes for $5min$;
3. *Deactivation:* The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

We developed the TAIO model of the CAS, shown in Figure 6, following the above requirements.

We applied our mutation testing tool to the CAS example. We first generated all the mutants (1099) and for each mutant checked whether it tioco-conforms to the original CAS model, by effectively doing the k-bounded language inclusion test. We set the maximal k bound to 20 for the k-bounded language inclusion test. We generated tests from all the non-conformant mutants. The whole procedure took 62.3 minutes and produced a total of 628 test cases. 471 mutants are tioco conform to the specification and therefore did not produce any test cases. Table 1 shows the run time of the standard and incremental approaches for the language inclusion applied on the CAS and a single equivalent mutant, indicating the efficiency of the incremental solving.

In order to evaluate our mutation testing framework, we used an existing implementation of CAS [1], developed in Java. The implementation consists of

k	5	10	15	20
Std Solving	$0.1s$	$40.1s$	$115.2s$	$279.5s$
Inc Solving	$0.1s$	$0.3s$	$0.6s$	$1.0s$

Table 1. Computation time for solving the k-bounded language inclusion between CAS model and its equivalent mutant.

	Mutants	Equiv.	Pairwise	Different
			Equiv.	Faults
SetState	6	0	1	5
Close	16	2	6	8
Open	16	2	6	8
Lock	12	2	4	6
Unlock	20	2	8	10
Constr.	2	0	1	1
Total	72	8	26	38

Table 2. Injected faults into the CAS implementation.

4 public methods, *open*, *close*, *lock* and *unlock*, and 2 internal methods, *setState* and the constructor. The CAS implementation simulates time elapse with a *tick* function. We used μJava[5] to mutate the above functions (except *tick*), resulting in 72 mutants, some of them equivalent to the original implementation or to other mutants. The total of 38 unique faulty implementations were derived, as summarized in Table 2. Both the correct and the 38 faulty CAS implementations were used to evaluate the effectiveness of the test cases we generated.

We developed a test driver in order to be able to execute tests that we generated on the CAS implementation. We integrated quiescence into the test driver, which is responsible to detect prolonged absence of output actions. We set the maximal timeout that the driver is allowed to wait for an output action to 400 time units. If the timeout is reached without observing an action, the test outputs a verdict **pass** if the test is in the last location with the true invariant or **inc** otherwise. The test driver emits an input action as soon as the associated transition becomes enabled. If the timeout is reached before the transition labeled by the input action becomes enabled, the test driver returns the **inc** verdict. Note that the test execution was conducted against a known Java implementation which models time passage as discrete ticks. Our test driver can be interfaced to any SUT which is a simulated implementation model (for instance a Simulink model). The test driver can handle arbitrary (variable) passages of time, as long as the time passage is simulated and provided in form of a time stamp. However, our current test driver does not fully support interfacing with physical SUT implementations, where the real passage of time cannot be controlled. In order to allow such support, we would need to model, possibly elaborated, interfacing delays between the SUT and the test driver (see [18] for a detailed discussion

[5] http://cs.gmu.edu/ offutt/mujava/

	Action	Target	Source	Guard	Not-Guard	Invariant	Sink State	Clock	**Total**
Model Mutants [#]	139	375	375	24	25	11	25	125	1099
TCs [#]	139	267	165	6	3	11	25	12	628
av. Kills per TC [#]	12.5	13.2	12.4	16.3	16.3	17.8	17.8	13.8	13
Mutation Score [%]	71	94.7	92.1	57.9	47.4	60.5	89.5	57.9	**100**

Table 3. Mutation analysis on each of the mutation operators evaluated on 38 faulty implementations of the CAS using tioco as conformance relation.

on test execution and "delay automata"). We postpone the extension of our test drivers to physical real-time SUT implementation for future work.

We say that a faulty implementation is *killed* if at least one test case reaches the verdict **fail** during a test execution. We analyzed the effectiveness of our mutation operators with respect to their ability to kill faulty implementations. Table 3 summarizes the results on effectiveness of mutation operators, where each row provides the number of mutants, the number of resulting test cases, the average number of faulty implementations killed per test case and the *mutation score* of a mutation operator. Mutation score is the measure which gives the percentage of faulty implementations killed by mutants resulting of a single mutation operator. We achieved a total of 100% mutation score for the combined mutation operators. The highest mutation score is achieved by the "change target" operator, however at the price of generating 375 mutants and 267 test cases. Evaluation results also showed that most of the faulty implementations were killed by "change target" mutants in which self-loops were created. Finally, we observed that 3 faulty implementations were only killed by mutants resulting from "sink state" mutations.

Following the above observations, we conducted another experiment in which we only applied "sink state" and "self-loop" mutations, resulting in only 50 mutants. All mutants were shown to be non tioco-conformant to the original models, generating 50 test cases in just $56s$. In addition, the combination of these two operators achieved 100% mutation score on their own. These results indicate that a smart choice of a small number of mutation operators and their ordering can achieve high mutation scores while considerably reducing test case generation and execution times.

8 Conclusion

We proposed a novel mutation testing framework for real-time systems. Our TCG technique relies on symbolic BMC and uses incremental SMT solving. We illustrated our testing approach on a Car Alarm System example and presented promising experimental results, showing that we were able to kill all the faulty implementations efficiently.

In the next step, we will apply our framework to other case studies, and study mutation operators effectiveness in more detail, with the aim to identify a small set of operators which achieve high mutation scores for a larger class of problems.

We will also consider extending our test driver to allow test execution on physical real-time implementations. We also plan to extend the expressiveness of timed automata model with data variables, non-determinism and silent transitions. We will finally extend our current framework by providing support for incremental test case generation for real-time systems consisting of multiple components.

References

1. Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. Efficient mutation killers in action. In *ICST*, pages 120–129, 2011.
2. Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011.
3. Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.
4. Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In *FORTE*, pages 243–259, 2002.
5. Bahareh Badban and Martin Lange. Exact incremental analysis of timed automata with an SMT-solver. In *FORMATS*, pages 177–192, 2011.
6. Fevzi Belli, Christof J. Budnik, and W. Eric Wong. Basic operations for generating behavioral mutants. In *Proceedings of the Second Workshop on Mutation Analysis*, MUTATION '06, pages 9–, Washington, DC, USA, 2006. IEEE Computer Society.
7. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.
8. Sergiy Boroday, Alexandre Petrenko, and Roland Groz. Can a model checker generate tests for non-deterministic systems? *Electronic Notes in Theoretical Computer Science*, 190(2):3 – 19, 2007. ¡ce:title¿Proceedings of the Third Workshop on Model Based Testing (MBT 2007)¡/ce:title¿.
9. Angelo Brillout, Nannan He, Michele Mazzucchi, Daniel Kroening, Mitra Purandare, Philipp Rümmer, and Georg Weissenbacher. Mutation-based test case generation for simulink models. In *FMCO*, pages 208–227, 2009.
10. Laura Brandán Briones and Ed Brinksma. A test generation framework for *quiescent* real-time systems. In *FATES*, pages 64–78, 2004.
11. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439, 1990.
12. Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.
13. Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.
14. S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, and P.C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 210 –219, 1999.
15. Gordon Fraser, Franz Wotawa, and Paul E. Ammann. Testing with model checkers: a survey. *Softw. Test. Verif. Reliab.*, 19(3):215–261, September 2009.
16. Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded simulink via formal concept analysis. In *DAC*, pages 224–229, 2011.

17. Yue Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649 –678, Sept.-Oct. 2011.
18. Moez Krichen and Stavros Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.
19. Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *STTT*, 1(1-2):134–152, 1997.
20. Marius Mikucionis, Kim Guldstrand Larsen, and Brian Nielsen. T-uppaal: Online model-based testing of real-time systems. In *ASE*, pages 396–397, 2004.
21. Leonardo Moura and Nikolaj Bjrner. Z3: An efficient smt solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.
22. Peter Niebert, Moez Mahfoudh, Eugene Asarin, Marius Bozga, Oded Maler, and Navendu Jain. Verification of timed automata via satisfiability checking. In *FTRTFT*, pages 225–244, 2002.
23. Brian Nielsen and Arne Skou. Automated test generation from timed automata. *STTT*, 5(1):59–77, 2003.
24. R. Nilsson, J. Offutt, and S.F. Andler. Mutation-based testing criteria for timeliness. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 306 –311 vol.1, sept. 2004.
25. R. Nilsson, J. Offutt, and S.F. Andler. Mutation-based testing criteria for timeliness. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 306 –311 vol.1, sept. 2004.
26. Robert Nilsson. Mutation-based testing criteria for timeliness, in. In *In Proceedings of the 28th Annual Computer Software and Applications Conference (COMPSAC*, pages 306–312. IEEE Computer Society, 2004.
27. Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods*, pages 298–312, 2011.
28. S.C. Pinto Ferraz Fabbri, M.E. Delamaro, J.C. Maldonado, and P.C. Masiero. Mutation analysis testing for finite state machines. In *Software Reliability Engineering, 1994. Proceedings., 5th International Symposium on*, pages 220 –229, nov 1994.
29. Jean-Pierre Queille and Joseph Sifakis. Iterative methods for the analysis of petri nets. In *Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets*, pages 161–167, 1981.
30. Rupert Schlick, Wolfgang Herzner, and Elisabeth Jöbstl. Fault-based generation of test cases from UML-models - approach and some experiences. In *SAFECOMP*, pages 270–283, 2011.
31. Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In *FORMATS*, pages 250–264, 2008.
32. Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.