Time for Mutants —Model-Based Mutation Testing with Timed Automata

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

[1] Institute for Software Technology
Graz University of Technology, Austria
{aichernig,florber}@ist.tugraz.at
[2] AIT Austrian Institute of Technology
Vienna, Austria
dejan.nickovic@ait.ac.at

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 modeling 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* [28,12], 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

M. Veanes and L. Viganò (Eds.): TAP 2013, LNCS 7942, pp. 20–38, 2013.

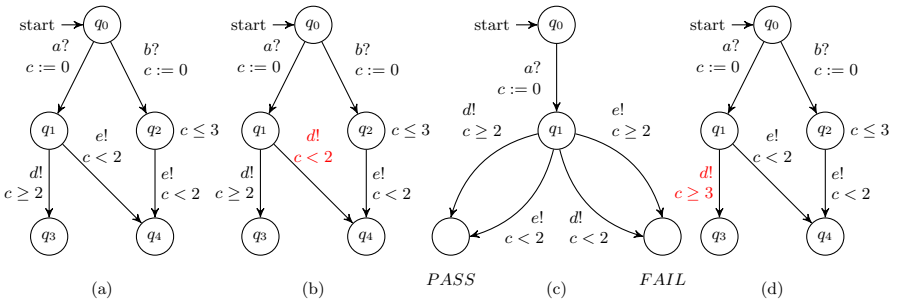

Fig. 1. Timed mutants example: (a) TA model A; (b) mutant M_1; (c) a test case generated from M_1 and (d) mutant M_2

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 [31] 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 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 [2,29] in the context of UML models, and in [9,15] 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 [16] 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.

In embedded systems, models are often derived from real-time requirements, resulting in extensions of model-based testing to the real-time context. This includes usage modeling the SUT with a timed formalism and adaptation of conformance relations to real-time. A comprehensive overview and comparison of real-time conformance relations is presented in [30]. A framework for black-box conformance testing based on the model of partially-observable, non-deterministic

timed automata is proposed in [17]. 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 similar work [22], 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 [19], an online testing tool for real-time systems is proposed, based on symbolic techniques of Uppaal [18]. Testing with real-time UML models is studied in [25], where the focus is given on combining bounded model checking techniques with abstract interpretation for 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 *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: since location q_1 has no invariant, delaying is allowed.

In contrast to [17,22,10,19], we propose a *symbolic* test generation procedure based on bounded model checking (BMC) techniques and which uses a Satisfiability Modulo Theories (SMT) solver to generate test cases. The bounded model checking approach for test case generation is promising in the context of mutation testing, which is fault oriented and focuses on finding finite witnesses exposing faults resulting from mutating a specification. In addition, SMT solvers provide support for future extensions such as handling unbounded data domains. Our test case generation framework combines existing results on TA decision problems [3], real-time conformance relations [17] and symbolic solving of TA decision problems [21,5,4] and applies them in a novel setting of mutation-based test case generation. We are not aware of other work which applies BMC techniques for generating real-time test cases from timed automata.

The survey [14] gives an extensive overview of existing test case generation approaches via model checking. [8] investigates problems and solutions for test case generation via model checkers for non-deterministic specifications. The work in [24,23] proposes a mutation-based testing framework for real-time systems using TA with Tasks. In contrary to our work, they generate test cases for mutants that violate task deadlines, while we check the real-time conformance between a specification and its mutants. 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 [27].

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 non-decreasing sequence of non-negative reals. A *timed trace* σ is a finite alternating sequence of actions and time delays of the form $t_1 \cdot a_1 \cdots t_k \cdot a_k$, where for all $i \in [1, k]$, $a_i \in \Sigma$ and $(t_i)_{i\in[1,k]}$ is a time sequence.

Let $\mathcal{C}$ be a finite set of *clock* variables. Clock *valuation* $v(c)$ is a function $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ 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. *A* TAIO[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* $\Sigma_I \cap \Sigma_O = \emptyset$ *and* Σ *is 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* $d \in \mathbb{N}$ *and each invariant is bound to its specific location, and* 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*.*

We say that a TAIO A is *deterministic* if for all transitions (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 $\mathrm{Det}(\mathcal{A}) \subset \mathcal{A}$ its deterministic subset. We denote by $\Delta_O \subseteq \Delta$

[1] TAIO are similar to UPPAAL TA, 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 modeling easier, and can be expressed with standard timed automata.

the set $\{\delta = (q, a, g, \rho, q') \mid \delta \in \Delta \text{ and } a \in \Sigma_O\}$ of transitions labeled by an output action and by $\Delta_I = \Delta \backslash \Delta_O$ the set of transitions labeled by an input action. We define $|\mathcal{G}|$ to be 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 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) $[[A]] = (S, \hat{s}, \mathbb{R}_{\geq 0}, \Sigma, T)$, where $S = \{(q, v) \in Q \times \mathcal{H} \mid v \models I(q)\}$, $\hat{s} = (\hat{q}, \mathbf{0})$, $T \subseteq S \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S$ 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 $\delta_i = (q_i, a_i, g_i, \rho_i, q_{i+1})$, inducing 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 implementation errors. In our setting, a mutation is a function $\mu_m : \text{Det}(\mathcal{A}) \to 2^{\mathcal{A}}$ parameterized by a mutation operator m which maps a deterministic TAIO A into a finite set $\mu_m(A)$ of possibly non-deterministic TAIOs , where each $M \in \mu_m(A)$ is called an m-mutant of A. For our experiments we only created first-order mutants, i.e., each mutated TAIO covers only one particular mutation.

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. This mimics an implementation fault producing wrong output signals. 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 $a_m \neq a$;*
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. This reflects the behaviour of an implementation fault where a signal leads to a wrong internal state. 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 $q'_m \neq q'$;*

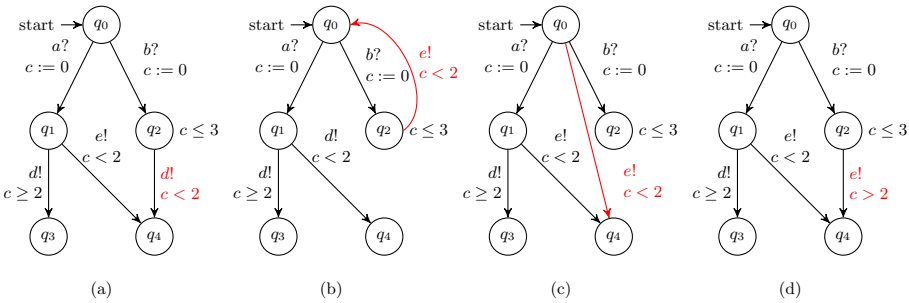

Fig. 2. Mutant M of model A resulting from: (a) $\mu_{ca}(A)$; (b) $\mu_{ct}(A)$; (c) $\mu_{cs}(A)$; (d) $\mu_{cg}(A)$;

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. This expresses an implementation fault where a signal can be triggered from a state where it should be disabled. 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. This covers implementation faults with faulty enabling conditions. 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. This covers implementation faults where the programmer forgot negating a condition. A* TAIO *$M \in \mu_{ng}(A)$, 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 with 1 added to the right side of the invariant. This mimics an "off by one"-fault allowing to stay longer in a state than intended. 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 $I(q) = \bigwedge_{i \in I} c_i \circ d_i$, $\circ \in \{<, \leq\}$, $I_m(q) = \bigwedge_{i \in I} c_i \circ d_i^m$, $d_i^m = d_i + 1$ for some $i \in I$, $d_j^m = d_j$ for all $j \neq i$ and $I(q') = I_m(q')$ for all $q' \neq q$;*

[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.

[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.

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. This expresses a program fault leading to a quiescent state where every input is accepted, but ignored. 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 occurrence of one clock flipped compared to the original set of clocks. This reflects different timing errors, e.g. the incorrect reseting of a timer. 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 analyzed 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[16], that describes multiple ways of reducing mutants for mutation testing, which can in general also be applied to model-based mutation testing.

Several different approaches to model mutation have already been published, using Finite State Machines [26,27], Kripke structures [8] or Event Sequence Graphs [6]. [23] introduces mutation operators for Timed Automata with Tasks, yet the mutation operators there concentrate on tasks and timeliness and not the core essence of TA. Our mutation operators 6 and 8 are specific to TA, while the other ones are similar or closely related to the operators described in [27].

4 Conformance Relations for Timed Automata

Different real-time extensions of the input-output conformance relation ioco were studied and compared in [30]. We consider the timed input-output conformance relation introduced in [17] 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 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 [17], first introducing operators illustrated in Equation 1.

$$\begin{aligned} A \text{ after } \sigma &= \{s \in S \mid \hat{s} \xrightarrow{\sigma} s\} \\ \text{elapse(s)} &= \{t > 0 \mid s \xrightarrow{t}\} \\ \text{out}(s) &= \{a \in \Sigma_O \mid s \xrightarrow{a}\} \cup \text{elapse(s)} \\ \text{out}(S) &= \bigcup_{s \in S} \text{out}(s) \end{aligned} \tag{1}$$

Given a TAIO A and $\sigma \in L(\Sigma)$, A after σ is the set of all states of A that can be reached by the sequence σ. Given a state $s \in S$, elapse(s) is the set of all delays that can elapse from s without A making any action, and out(s) is the set of all output actions or time delays that can occur when the system is at state s, a definition which naturally extends to set of states S.

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

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

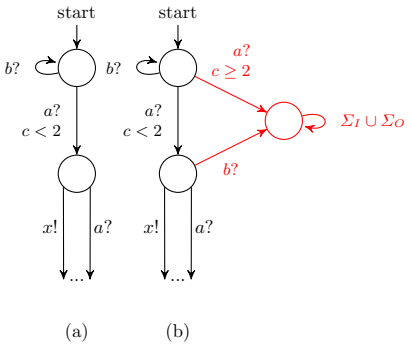

Fig. 3. Demonic completion of TAIO: (a) A; and (b) $d(A)$

In [17], 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 [31,17], the notion of *demonic completion*, illustrated in Figure 3 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.

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 $d(A) = (Q \cup \{\text{sink}\}, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I_d, \Delta_d)$, where $I_d(q) = I(q)$ and $I_d(\text{sink}) = \text{true}$ and $\Delta_d = \Delta \cup \{(\text{sink}, a, \text{true}, \{\}, \text{sink}) \mid a \in \Sigma\} \cup \{(q, a, \neg g, \{\}, \text{sink}) \mid q \in Q \wedge a \in \Sigma_I\}$, 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. Strictly speaking, g contains disjunctions of constraints, and thus cannot be directly used as a guard on a transition in a TAIO. In fact, we would

need to transform g into a disjunctive normal form, and have a separate copy of the transition for each disjunction labeled by the appropriate guard. We omit the details of this transformation. 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 [17].

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 $L(A_I) \subseteq L(d(A_S))$, a result stated in the next Proposition, and which follows from Lemma 3 and Proposition 3 in [17].

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 one can check $L(A_I) \subseteq L(d(A_S))$ instead of checking A_I tioco A_S when A_S is deterministic. In addition, 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 model-based testing, the SUT is often seen as a black box and a conformance relations such as tioco serves to establish soundness of the TCG algorithm, but is not actually computed. In fact, only the specification model is explored in order to generate test cases, and the conformance relation defines the test verdict.

In contrast, mutation testing requires effective conformance checking of the mutated model to the original specification. Mutation testing is a particular instance of fault oriented testing where the test cases are generated in a way that attempts to "steer" the SUT towards failure, due to a common modeling error if one exists. Hence, the rationale 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 behavior with respect to the original specification, hence no useful test case is generated. It follows that test cases are generated only if the mutated model does not conform to its original version. We propose a TCG algorithm, 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 $L(M) \subseteq L(d(A))$;
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. In addition, the language inclusion problem between two timed automata A_I and A_S, where A_S is deterministic is PSPACE-complete, hence computationally expensive. In our setting, we are interested in finding finite counter-example traces witnessing violation of language inclusion. Missing such a witness, due to an insufficient bound, results in generating less test cases and is a trade-off between generating a complete test suite and computing it efficiently.

BMC was used in [4,21] for the reachability analysis of TA, and in [5] for checking the language inclusion between two timed automata. We encode the language inclusion problem 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 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 $\text{loc}_A : Q \to \{1, \ldots, |Q|\}$ and $\text{act}_A : \Sigma \to \{1, \ldots, |\Sigma|\}$ 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 discrete 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:

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

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

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

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

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

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:

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

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:

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

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 $\varphi^k_{A_I, A_S}(i, \mathcal{A}, D, X_I, X_S, C_I, C_S)$ specifies the negation of k-language inclusion:

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

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 according to Algorithm 1 satisfying 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 (Line 7);
- Refuse action a if a is an output action and there are 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 (Lines 12-18).

Finally, when A_T is in location q_k, it accepts all outputs a such that there exists an enabled transition δ in A with source location q_k and labeled by a, moving to the **pass** location (Line 9), and it rejects all other outputs, moving to the **fail** location (Lines 12-18).

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 in the specification A if it differs from the one in the witness σ, and returning **inc** as 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 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 flexible and 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 from the witness trace σ, to precede an input action either, since it may violate input assumptions of the specification during some test execution. We illustrate this observation in Figure 4, which depicts model A and its mutant M. The trace $\sigma = 4{\cdot}x!{\cdot}2{\cdot}a?{\cdot}2{\cdot}y!$ witnesses non-conformance of M to A and is used as the skeleton for the test A_T. During 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 we keep the time constraints symbolic, with an elapse of time between $x!$ and $a?$ dependent on 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 specifications. The (bounded) language inclusion between two TAIOs is computed using the Z3 (v4.0) SMT solver [20]. The communication between our implementation in Scala and the Z3 solver relies on the Scala$^\wedge$Z3 API.

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; and (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. We note that modeling style can have important impact on the number and effectiveness of consecutive generation of mutants and test cases. We implemented demonic completion of the model by direct application of the procedure from Section 4.

Mutation of Models. Our tool supports all mutation operators introduced in Section 3. We store each mutant as a separate Uppaal XML model.

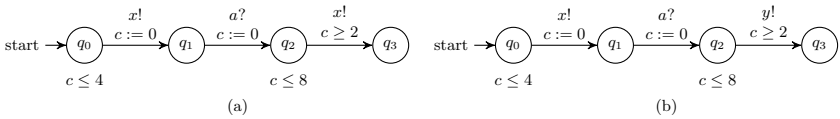

Fig. 4. Necessity of symbolic constraints on inputs in a test

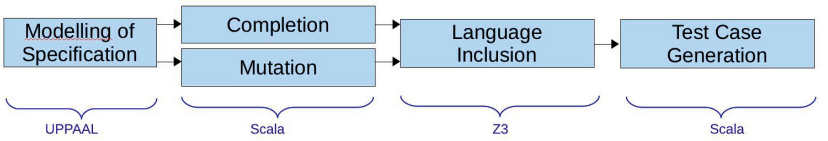

Fig. 5. Test case generation framework

Algorithm 1. Test case generation algorithm

Input: $A = (Q, \hat{q}, \Sigma_I, \Sigma_O, \mathcal{C}, I, \Delta)$ and $\delta_1 \cdots \delta_k$
Output: Test automaton A_T

$\Delta_T \leftarrow \bigcup_{i=1}^{k-1} \{\delta_i\}$
$Q_T \leftarrow \{q_i | (q_i, a, g, p, q') \in \delta_1 \cdots \delta_k\}$
$Q_T \leftarrow Q_T \cup \{\mathbf{pass}, \mathbf{fail}, \mathbf{inc}\}$
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. $\exists (q_i, a, g_j, \rho, q') \in \Delta$ **do**
    $g_T \leftarrow (g_1 \vee \ldots \vee g_n) \wedge I(q)$ st. $\{g_j\}$ 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**
    $\Delta_T \leftarrow \Delta_T \cup \{(q_i, a, \text{true}, \{\}, \mathbf{fail})\}$
  end for
end for
return $A_T \leftarrow (Q_T, \hat{q}, \Sigma_O, \Sigma_I, \mathcal{C}, I, \Delta_T)$

Language Inclusion. 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 fed to the Z3 solver, which looks for the existence of a satisfying assignment to the variables representing a witness trace violating the language inclusion property.

In addition, we 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 stops. 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.

CAS Requirements

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

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 $5min$;

Deactivation: The anti-theft alarm system can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

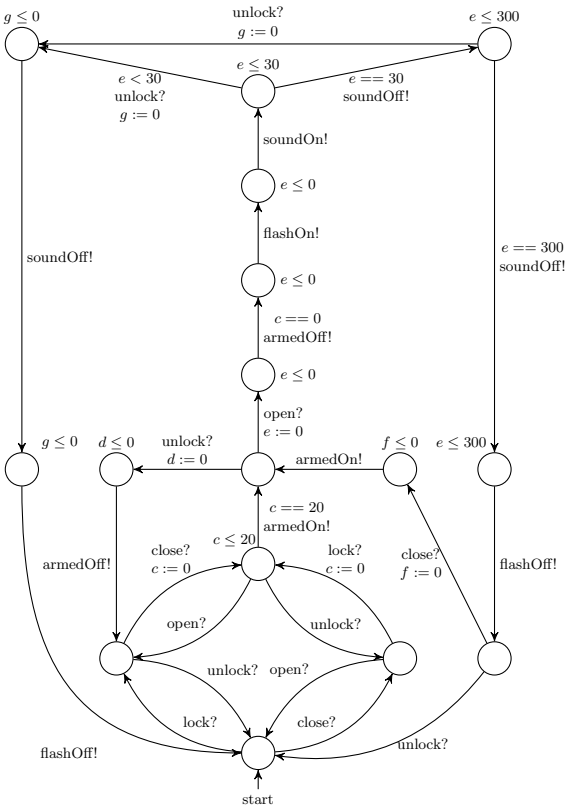

Fig. 6. Car alarm system: requirements and its TAIO model

7 Case Study and Experimental Results

In this section we illustrate our TCG approach with the Car Alarm System (CAS) [2,29] and evaluate the framework. The car alarm system (CAS) is a model inspired by the Ford's demonstrator developed in the EU FP7 project MOGENTES[4]. We developed the TAIO model of the CAS from the requirements provided by Ford, both shown in Figure 6.

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 conformant to the specification and therefore did not produce any test cases. Table 1 shows the run time of the standard and incremental approaches

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

Table 1. Computation time for k-bounded language inclusion check

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

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 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* method. We also used the 38 implementation mutants of the CAS descibed in [1]. They were produced using the Java mutation tool μJava[5]. Applying all mutation operators of μJava to all methods except *tick* resulted originally in 72 mutants.

Some of the mutants were equivalent to the original implementation or to other mutants. After filtering them out, the total of 38 unique faulty implementations were derived. Table 2 shows the total numbers of implementation mutants and equivalent ones. 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 execute generated tests on the CAS implementation. We integrated quiescence in the test driver, which is responsible to detect prolonged absence of outputs. 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 immediately emits an input action when the associated transition becomes enabled. If the timeout is reached before the transition labeled by the input action becomes enabled, the test driver gives the **inc** verdict. Note that we executed tests on a Java implementation which models time passage as discrete ticks. We can currently interface our test driver to any simulated implementation model with arbitrary model of time, as long as time is simulated and communicated in form of time stamps. However, we do not yet support interfacing the test driver to a physical SUT, where the real passage of time cannot be controlled. In order to allow such support, we would need to model elaborated interfacing delays between the SUT and the test driver (see [17] for a detailed discussion on test execution and "delay automata"). We postpone the extension of our test drivers to physical real-time SUT to 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

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

Table 2. Injected faults into the CAS implementation

	Mutants	Equiv.	Pairwise Equiv.	Different 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

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 $M2$, at the price of generating 375 mutants and 267 test cases. Evaluation results also show that most of the faulty implementations are killed by $M2$-mutants which contain self-loops. We also observed that 3 faulty implementations were only killed by "sink state" mutants ($M7$).

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 original models, generating 50 test cases in just $56s$. In addition, combining these two operators resulted in 100% mutation score. These results indicate that a smart choice of a small subset of mutation operators can achieve high mutation scores while considerably reducing test case generation and execution times.

Table 3. Mutation analysis of mutation operators. The list of mutation operators can be seen in Section 3.

		M1	M2	M3	M4	M5	M6	M7	M8	**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**

8 Conclusion

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

In the next step, we will apply our framework to other case studies, and study mutation operators effectiveness in more detail, adding more complex mutation operators and identifying a small set of operators which achieve high mutation scores for a larger class of problems. We will extend our test driver to allow test

execution on physical real-time SUTs. In the current setting, we generate a test case in a form of a simple timed automaton which is extended from the witness trace and the original specification model. We plan to improve witness to test extension, by considering more refined timing constraints in the test contained in the region automaton of the specification model. We also plan to extend the expressiveness of the TA model with data variables, non-determinism and silent transitions. We finally plan to add support for incremental TCG for real-time systems consisting of multiple components.

Acknowledgements. We would like to thank Rupert Schlick from AIT for fruitful discussions and anonymous reviewers for their useful suggestions.

The research leading to these results has received funding from ARTEMIS Joint Undertaking under grant agreement number 269335 (MBAT) and from national fundings (Federal Ministry for Transport, Innovation and Technology and Austrian Research Promotion Agency) under program line Trust in IT Systems, project number 829583 (TRUFAL).

References

1. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: ICST, pp. 120–129 (2011)
2. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: UML in action: a two-layered interpretation for testing. ACM SIGSOFT SEN 36(1), 1–8 (2011)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Audemard, G., Cimatti, A., Kornilowicz, A., Sebastiani, R.: Bounded model checking for timed systems. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 243–259. Springer, Heidelberg (2002)
5. Badban, B., Lange, M.: Exact incremental analysis of timed automata with an SMT-solver. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 177–192. Springer, Heidelberg (2011)
6. Belli, F., Budnik, C.J., Wong, W.E.: Basic operations for generating behavioral mutants. In: MUTATION, pp. 9–18. IEEE Computer Society Press (2006)
7. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 117–148 (2003)
8. Boroday, S., Petrenko, A., Groz, R.: Can a model checker generate tests for non-deterministic systems? ENTCS 190(2), 3–19 (2007); MBT 2007
9. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
10. Briones, L.B., Brinksma, E.: A test generation framework for quiescent real-time systems. In: Grabowski, J., Nielsen, B. (eds.) FATES 2004. LNCS, vol. 3395, pp. 64–78. Springer, Heidelberg (2005)
11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: LICS, pp. 428–439 (1990)
12. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Engeler, E. (ed.) Logic of Programs 1979. LNCS, vol. 125, pp. 52–71. Springer, Heidelberg (1981)

13. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
14. Fraser, G., Wotawa, F., Ammann, P.E.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. 19(3), 215–261 (2009)
15. He, N., Rümmer, P., Kroening, D.: Test-case generation for embedded simulink via formal concept analysis. In: DAC, pp. 224–229 (2011)
16. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), 649–678 (2011)
17. Krichen, M., Tripakis, S.: Conformance testing for real-time systems. Formal Methods in System Design 34(3), 238–304 (2009)
18. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1(1-2), 134–152 (1997)
19. Mikucionis, M., Larsen, K.G., Nielsen, B.: T-uppaal: Online model-based testing of real-time systems. In: ASE, pp. 396–397 (2004)
20. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
21. Niebert, P., Mahfoudh, M., Asarin, E., Bozga, M., Maler, O., Jain, N.: Verification of timed automata via satisfiability checking. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 225–244. Springer, Heidelberg (2002)
22. Nielsen, B., Skou, A.: Automated test generation from timed automata. STTT 5(1), 59–77 (2003)
23. Nilsson, R., Offutt, J., Andler, S.F.: Mutation-based testing criteria for timeliness. In: COMPSAC 2004, vol. 1, pp. 306–311 (2004)
24. Nilsson, R., Offutt, J., Mellin, J.: Test case generation for mutation-based testing of timeliness. ENTCS 164(4), 97–114 (2006); MBT 2006
25. Peleska, J., Vorobev, E., Lapschies, F.: Automated test case generation with SMT-solving and abstract interpretation. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 298–312. Springer, Heidelberg (2011)
26. Fabbri, S.C.P.F., Delamaro, M.E., Maldonado, J.C., Masiero, P.C.: Mutation analysis testing for finite state machines. In: ICSRE, pp. 220–229 (1994)
27. Fabbri, S.C.P.F., Maldonado, J.C., Sugeta, T., Masiero, P.C.: Mutation testing applied to validate specifications based on statecharts. In: Software Reliability Engineering, pp. 210–219 (1999)
28. Queille, J., Sifakis, J.: Iterative methods for the analysis of petri nets. In: Selected Papers from the First and the Second European Workshop on ICATPN, pp. 161–167 (1981)
29. Schlick, R., Herzner, W., Jöbstl, E.: Fault-based generation of test cases from UML-models - approach and some experiences. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 270–283. Springer, Heidelberg (2011)
30. Schmaltz, J., Tretmans, J.: On conformance testing for timed systems. In: Cassez, F., Jard, C. (eds.) FORMATS 2008. LNCS, vol. 5215, pp. 250–264. Springer, Heidelberg (2008)
31. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 1–38. Springer, Heidelberg (2008)