Empirical Software Engineering (2022) 27:160
https://doi.org/10.1007/s10664-022-10149-y

Static detection of equivalent mutants in real-time model-based mutation testing

An Empirical Evaluation

Davide Basile[1] · Maurice H. ter Beek[1] · Sami Lazreg[2] · Maxime Cordy[2] · Axel Legay[3]

Accepted: 3 March 2022
© The Author(s) 2022

Abstract
Model-based mutation testing has the potential to effectively drive test generation to reveal faults in software systems. However, it faces a typical efficiency issue since it could produce many mutants that are equivalent to the original system model, making it impossible to generate test cases from them. We consider this problem when model-based mutation testing is applied to real-time system product lines, represented as timed automata. We define novel, time-specific mutation operators and formulate the equivalent mutant problem in the frame of timed refinement relations. Further, we study in which cases a mutation yields an equivalent mutant. Our theoretical results provide guidance to system engineers, allowing them to eliminate mutations from which no test case can be produced. Our empirical evaluation, based on a proof-of-concept implementation and a set of benchmarks from the literature, confirms the validity of our theory and demonstrates that in general our approach can avoid the generation of a significant amount of the equivalent mutants.

Keywords Software product line · mutation-based testing · real-time system

1 Introduction

Testing a real-time system against safety-critical requirements is a difficult problem due to the time-sensitiveness of its behaviour. To help in this task, model-based testing methods

---

Communicated by: Philippe Collet, Sarah Nadi, Christoph Seidl, and Leopoldo Motta Teixeira

This article belongs to the Topical Collection: *Software Product Lines and Variability-rich Systems (SPLC)*

CRediT author statement: D. Basile (first author): Conceptualization, Writing - Original Draft, Software, Validation, Data Curation, Investigation. M.H. ter Beek: Writing - Original Draft, Visualization, Investigation, Project administration, Supervision. S. Lazreg: Writing - Review & Editing, Software, Validation, Data Curation, Investigation. M. Cordy: Writing - Review & Editing, Software, Data Curation, Project administration. A. Legay: Project administration.

✉ Davide Basile
davide.basile@isti.cnr.it

Extended author information available on the last page of the article.

automate the generation of test cases by using a formal model of the system (Utting et al. 2012). The model drives the generation of test cases according to different criteria, such as classical state or branch coverage (Masri and Zaraket 2016), or feature combination coverage in the specific context of software product lines (Lee et al. 2020). Testing a formal model rather than source code allows to detect, among others, misinterpretations of requirements or systemic issues arising from time-dependent interactions of the system with its environment. Such detections would be harder at source code level.

Mutation testing (Aichernig et al. 2015; Brillout et al. 2009) is a technique commonly used to evaluate the thoroughness of test cases or to support their generation (Andrews et al. 2006; Offutt 2011). It can be applied both to the implementation (source code) and to the specification (model). A set of *mutation* operators, simulating possible faults in the system, are applied to the model, obtaining a so-called *mutant*. Thus, given a set of mutants, the effectiveness of a set of test cases can be evaluated according to the number of mutants it detects (i.e., mutants that produce different output than the original system). Test cases generated from a mutant are capable to detect bugs mimicked by that mutation. The fundamental underlying assumption is the existence of a coupling effect, i.e. the fact that "*simple faults are coupled to more complex faults in such a way that a test suite that detects simple faults is sensitive enough to likely detect complex faults as well*" (Petrovic et al. 2021). It has been shown (Andrews et al. 2006) that mutation-based testing is more effective in finding real faults than other techniques (Offutt 2011; Baker and Habli 2013; Aichernig et al. 2013).

Scalability of this approach is of paramount importance, because a large number of mutations is required in order to build effective test cases. However, many of these mutations are useless because they generate mutants that have no behaviour that the original system had not. In such cases, no test case can be generated to differentiate the mutant from the original system, leading to useless analyses and waste of computational resources. Code-based mutation testing research has worked on methods to detect and avoid *equivalent* mutants that are semantically equivalent to the original program (Madeyski et al. 2014). In model-based mutation testing, this problem generalizes to that of detecting *subsumed* mutants, which have less (or equal) behaviour than the original system model.

One viable method is to organise the mutants as a product line of mutations, in the *featured mutant model* (Devroey et al. 2016). Such a product line enables the effective generation and validation of mutants against given test cases. However, an efficient featured mutant model should be built upon a set of effective mutations (i.e., those producing useful mutants), rather than from random mutations. This constitutes an important contribution to avoiding the equivalent/subsumed mutant problem.

In this paper, we tackle the problem of testing real-time systems effectively and efficiently. We adopt the model-based mutation testing approach for real-time systems presented in Larsen et al. (2017). We augment the set of existing mutations with a few mutation operators that affect the timing of the system behaviour (e.g., one such operator delays the execution of an action by the system), first introduced in Basile et al. (2020a). Then, we address the subsumed mutant problem: we formally prove the conditions under which mutations inherently (i.e., by construction) produce subsumed mutants. We achieve this on the basis of *refinement relations*, which can be used to show that a model (the system) subsumes another (the mutant). Our endeavour yields clear guidelines for real-time system engineers, which they can follow in order to reduce their testing effort by ignoring equivalent mutants.

This paper builds on results from Basile et al. (2020a); more precisely, we extend it in the following way. We prove novel auxiliary theoretical results concerning non-subsumed mutants (in Section 4.2), while we refer to Basile et al. (2020a) for proofs of earlier

theoretical results reported here for the sake of completeness. Moreover, we add a thorough empirical evaluation (in Section 5), considering more case studies retrieved from the literature, and extending the experiments to consider second-order mutations, for all case studies. Finally, the experiments protocol has been fully automatised (cf. Section 5.2) and the implementation of mutant generation and checking is open source and available online, thus allowing reproducibility of the experiments.

Summarising, our contributions are as follows.

1. We propose novel time-specific mutation operators for real-time models.
2. We study and formally prove under which conditions mutation operators, including the time-specific ones, yield equivalent (or subsumed) mutants, from which no test case can be generated, and we provide guidelines that can be used to prevent the generation of such mutants.
3. We study and formally prove under which conditions mutation operators, including the time-specific ones, yield non-equivalent (or non-subsumed) mutants. These results must also consider when a mutation produces a non-redundant mutant, and auxiliary results not presented before address this issue formally.
4. We formalise our theoretical results using product lines of mutants. We use the featured mutant model (Devroey et al. 2016) to model the variability of the different mutations that can be applied, using a feature-aware extension of timed graphs (the mathematical structure used to check refinement relations), in a similar way that other formalisms have been extended with variability (Cordy et al. 2012b; Cordy et al. 2012a; Classen et al. 2013; Ter Beek et al. 2016; Basile et al. 2020b; Ter Beek et al. 2020; Ter Beek et al. 2021).
5. We implement our approach in a proof-of-concept tool and validate the soundness and effectiveness of the guidelines, based on an industrial system from the automotive domain and several other case studies from the literature, for first-order as well as second-order mutants.
6. The experiments protocol is completely automatised based on software for (i) mutant generation, (ii) mutant checking with the provided tool, and (iii) automatic refinement checking using the off-the-shelf tools Uppaal TIGA (Behrmann et al. 2007) and Ecdar (David et al. 2010b), to validate the proposed approach empirically.

Outline In Section 2, we discuss related work, followed by background material on (featured) timed games and the featured mutant model in Section 3, where we also introduce the novel mutation operators. Our main contributions are presented in Section 4, where we classify mutation operators and present guidelines for selecting effective mutations, and in Section 5, where we report the results of an empirical evaluation of our guidelines for both first- and second-order mutants. In Section 6, finally, we conclude the paper and provide some ideas for future work. Due to their size, the results of the aforementioned empirical evaluation for second-order mutants are reported in Appendix A.

2 Related Work

This paper, as an extension of Basile et al. (2020a), mainly builds upon two recent results on mutation-based testing (Devroey et al. 2016; Larsen et al. 2017). Featured mutant models were introduced in Devroey et al. (2016) for efficiently validating test cases against different possible mutations. Indeed, a single execution on the generated featured transition

system (Classen et al. 2013) suffices to check all mutants at once. However, in contrast to our approach, no guidelines are provided on how to select the mutations to generate the featured mutant model, that is, the mutations are selected randomly.

While Devroey et al. (2016) studies the problem of checking given test cases, Larsen et al. (2017) considers the problem of generating valid test cases for real-time system models. Basically, a test case generated through mutation-based testing is guaranteed by construction to distinguish certain mutants from the system model.

Compared to Devroey et al. (2016), in Larsen et al. (2017) the mutants are not organised as a product line and thus have to be checked one by one to generate the test cases. Moreover, both approaches generate random mutations that may result ineffective for generating/validating the test cases. Our approach improves on this by providing clear guidelines that allow to establish upfront which mutations can safely be ignored since no test case can be produced from them.

In Luthmann et al. (2017) and Luthmann et al. (2019), an approach to the generation of non-subsumed mutants is proposed, using Configurable Parametric Timed Automata (CoPTA) models, which analyses constraints of the generated zone graph. We do not generate zone graphs but instead statically identify mutations to be discarded based on the fact that we know from our theoretical results that they will generate subsumed mutants.

Earlier, in Aichernig et al. (2013), mutation-based testing for timed automata was introduced, extending standard mutation operators presented in Fabbri et al. (1999) with new mutations tailored for timed automata. We use some of those mutations, but also some of the new ones we introduced in Basile et al. (2020a). Compared to Larsen et al. (2017), a k-bounded language inclusion test between the mutant and the system model is used rather than refinement checking with Ecdar.

In Aichernig et al. (2013), a Car Alarm System (CAS) model of Ford is used as case study for experiments and evaluation. In Basile et al. (2020a), we used the same case study; in this paper, we consider five further case studies from the literature (Hune et al. 2001; Feo-Arenis et al. 2014; Hoxha et al. 2015; André et al. 2019; Basile et al. 2020c). Similar to Larsen et al. (2017), the approach in Aichernig et al. (2013) comes without a procedure or guidelines for selecting effective mutations, and no product line is used either. In particular, 471 out of a total of 1099 generated mutants are tested and subsequently discarded, because they cannot be used for generating test cases. We present a technique that allows to avoid the generation of ineffective mutants.

Mutation-based test-case generation is also discussed in Aichernig et al. (2015), for the case of UML state machine diagrams. The technique for comparing the mutant with the system model is similar to the one in Aichernig et al. (2013), and the same CAS case study is used for experiments. Mutations are applied randomly and ineffective mutants (i.e., mutants subsumed by the system model) are discarded subsequent to their generation.

Finally, the survey in Jia and Harman (2011) points out that “one barrier to wider application of mutation testing centers on the problems associated with equivalent mutants”. Our paper is an effort in the direction of reducing the generation of ineffective mutants upfront, within the framework proposed by Larsen et al. (2017) and adopting the featured mutant model construction of Devroey et al. (2016).

3 Background

In this section, we provide some background needed for the sequel.

3.1 Timed Games

Timed games (TG) are transition systems which can remain in a certain state or location only a specific amount of time, can execute a transition only within a certain time interval, and distinguish between controllable and uncontrollable actions. TG are based on timed (game) automata (Alur and Dill 1994; Asarin et al. 1998) and form the underlying behavioral structure of featured timed game (automata) (Cordy et al. 2012b; Cordy et al. 2013).

In reactive systems, one usually distinguishes between uncontrollable and controllable actions, that are assigned to inputs and outputs, respectively, if the environment is uncontrollable and vice versa otherwise.

Time is represented by clocks whose values evolve continuously. Clocks can be regarded as chronometers: their value can be inspected and reset, but not modified arbitrarily. Conditions over clock values are called *clock constraints*.

Definition 1 (Clock constraints) A clock constraint over a set C of clocks is formed according to the grammar $g \ ::= \ \top \mid n \sim c \mid g \wedge g$, with $n \in \mathbb{N}$, $c \in C$, and $\sim \in \{<, \leq, \geq, >\}$.

We denote by $CC(C)$ the set of clock constraints over C. In TG, a clock constraint can label either a state or a transition. In case it labels a state, the constraint is a location *invariant*, which defines the interval of time in which the system can be in the state. In case it labels a transition, it is a transition *guard* specifying the interval of time during which the system can execute the transition. Note that the domain of the numeric constants in clock constraints is limited to natural numbers. Without loss of generality, we could use real numbers. However, natural numbers facilitate the implementation of clock constraints by allowing efficient data structures.

Definition 2 (Timed games) Let $(Loc, Act, C, Trans, \ell_0, Inv, AP, L)$ be a timed game (TG) where

- Loc is a finite set of locations;
- Act is a finite set of actions, partitioned into controllable actions Act^c and uncontrollable actions Act^u;
- C is a finite set of clocks;
- $Trans \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ is a transition relation;
- $\ell_0 \in Loc$ is the initial location;
- $Inv : Loc \to CC(C)$ is a total function associating locations with invariants;
- AP is a set of atomic propositions; and
- $L : Loc \to 2^{AP}$ is a total function associating locations with atomic propositions satisfied in those locations.

For a transition $t = (\ell, g, \alpha, R, \ell')$, ℓ is the starting location, g is the transition guard, α is the action triggering the transition, R is the subset of clocks to reset, and ℓ' is the target location. We may also write t as $\ell \xrightarrow{g,\alpha,R} \ell'$ and omit g and/or R when immaterial, and instead of $\{x\}$ for a reset of clock x, we may also write $x := 0$.

Example 1 In Fig. 1(left), a TG model of a soda vending machine is depicted. From its initial state s_0, the insertion of a euro coin (€) results in the clock being (re)set to zero and a move to state s_1. This input action is modelled as a controllable transition (drawn as a solid

arc). The vending machine can remain in this state for at most 5 time units but only within 2 time units it can deliver a soda bottle (🍾), returning to its initial state. The latter action is modelled as an uncontrollable transition (drawn as dotted arc). Note that we may speak of (un)controllable transitions when their action labels are (un)controllable. A TG model of a tea vending machine is depicted in Fig. 1(right).

The semantics of a TG is commonly defined as an infinite transition system (TS) whose states consist of a location and a valuation of the clocks. The transitions can be categorised into two types. *Delay transitions* do not change the location of the system, but only represent the passing of time. They may occur only if the invariant of the current location is still satisfied after the delay modelled by the transition. *Discrete transitions* instead occur when the system moves from one location to another. They may occur only if the current clock values satisfy both the guard of the executed transition and the invariant of the target location. After the execution of such transitions, clock values can be reset.

Definition 3 (TG semantics) We define the semantics of a timed game $tg = (Loc, Act, C, Trans, \ell_0, Inv, AP, L)$ as the semantics of the TS $(Loc \times Val(C), Act \cup \mathbb{R}_{\geq 0}, Trans', (\ell_0, v_0), AP \cup CC(C), L')$, denoted by $[\![tg]\!]_{TG}$, and such that $Val(C)$ is the set of clock evaluations, i.e., the set of total functions $v : C \to \mathbb{R}^+$ that assign a non-negative real value to every clock; $v_0 = \{\, v_0(c) = 0 \mid c \in C \,\}$; $L'(\ell, v) = L(\ell) \cup \{\, cc \in CC(C) \mid v \models cc \,\}$; and

$$[\![tg]\!]_{TG} = \{\, L(\ell_0), L(\ell_1), \ \ldots \in (2^{AP \cup CC(C)}) \mid \forall i \in \mathbb{N} \bullet \exists \alpha_i \in Act \cup \mathbb{R}_{\geq 0} \bullet ((\ell_i, v) \xrightarrow{\alpha_i} (\ell_{i+1}, v')) \,\}$$

3.2 Featured Timed Games

Featured timed games (FTG) extend TG with variability in the same way that featured transition systems (FTS) (Classen et al. 2013) extend (labelled) transition systems (LTS). FTS concisely model the behaviour of all products of a product line in a single superimposed LTS through the annotation of transitions with feature expressions, i.e., conditions expressing their existence in products, based on a feature model.

We assume products to be represented by sets of Boolean features and a feature model to be defined as a pair $(F, P \subseteq 2^F)$, where F is a set of features and P is the set of valid products. The semantics of a feature model φ, denoted by $[\![\varphi]\!]_{FM}$, is then its set of valid products. It can be represented by either a propositional formula or by the usual feature diagram. Let $\mathbb{B} = \{\top, \bot\}$ denote the Boolean constants true ($\top$) and false ($\bot$), and let $\mathbb{B}(F)$ denote the set of Boolean expressions over F (i.e., using features as propositional variables). The elements of $\mathbb{B}(F)$ are also called *feature expressions*. Formally, a feature expression χ is a total function $\{\top, \bot\}^{|F|} \to \{\top, \bot\}$ that associates every combination of features with a truth value. A feature expression can be interpreted as a set of products $[\![\chi]\!] \subseteq 2^F$ defined

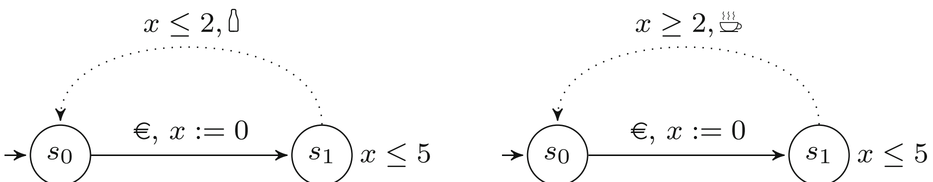

Fig. 1 TG models of a soda vending machine (left) and a tea vending machine (right)

as all products p for which the induced truth assignment ($\top$ for $f \in p$, $\bot$ for $f \notin p$, for features $f \in F$) validates χ. Feature expressions and clock constraints allow modelling the behaviour of real-time variable-intensive systems.

Definition 4 (Featured timed games) Given a timed game $(Loc, Act, C, Trans, Loc_0, Inv, AP, L)$, the decuple $(Loc, Act, C, Trans, Loc_0, Inv, AP, L, \varphi, \gamma)$ is a featured timed game (FTG) where

- φ is a feature model over a finite set F of features; and
- $\gamma : (Trans \cup (Loc \rightarrow CC(C))) \rightarrow \mathbb{B}(F)$ is a total function associating feature expressions to transitions and invariants.

As for FTS, the function γ associates a feature expression χ to some transition $t = (\ell, g, \alpha, R, \ell')$ such that $\gamma(t) = \chi$ encodes the set of products able to execute t. We may also write t as $\ell \xrightarrow{[\chi]g,\alpha,R} \ell'$ and omit g and/or R when immaterial. The function γ moreover associates a feature expression χ to a location invariant $Inv(\ell) = g$, for some $\ell \in Loc$, such that $\gamma(g) = \chi$, which we may also write as $[\chi]g$, encodes the set of products with the invariant g in location ℓ. Note that $[\top]$ stands for a feature expression that is always satisfied (by any product).

Example 2 In Fig. 2(left), an FTG *ftg* of a product line of vending machines is depicted. The feature model is $s \vee t$, with features s for soda and t for tea. From the initial state s_0, the insertion of a euro coin (€), which is always possible (the feature expression is always true) and which results in the clock being (re)set to zero, leads to state s_1. This is a controllable (input) action. A vending machine can remain in this state for at most 5 time units. Vending machines with feature s can deliver a soda bottle (🧴) before 2 time units have passed. Vending machines with feature t can deliver a cup of tea (☕) after at least 2 time units have passed (producing tea takes more time). Note that in the presence of both features, after precisely 2 time units have passed, a choice occurs. Both (output) actions are uncontrollable.

FTG model real-time behaviour of a product line. Moreover, from an FTG we can derive TG modelling behaviour of specific products. This is achieved by *projection* of an FTG onto

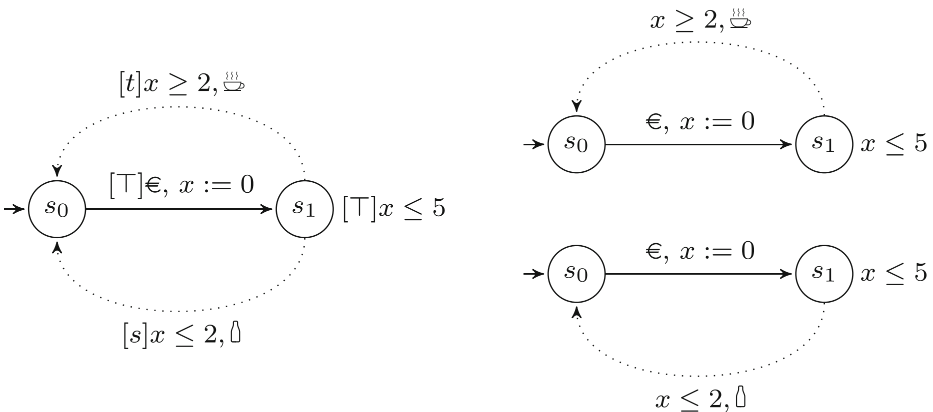

Fig. 2 FTG model of a product line of vending machines (left) and TG models of two of its products (right) reproduced from Fig. 1

a product p obtained in much the same way as an LTS is obtained from an FTS (Classen et al. 2013): all transitions and invariants unavailable in product p are removed.

Definition 5 (FTG projections) The projection of an FTG $ftg = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, \varphi, \gamma)$ onto a valid product $p \in [\![\varphi]\!]_{FM}$ is the TG $ftg_{|p} = (Loc, Act, C, Trans', Loc_0, Inv', AP, L)$ where

$$Trans' = \{ t = (\ell, g, \alpha, R, \ell') \mid t \in Trans \wedge p \models \gamma(t) \}; \text{ and}$$
$$Inv'(\ell) = Inv(\ell)_{|p}, \; \forall \ell \in Loc \text{ and the projection of an invariant } g \text{ onto a product } p \text{ is recursively defined as}$$

$$g_{|p} = \begin{cases} (g_1)_{|p} \wedge (g_2)_{|p} & \text{if } g = g_1 \wedge g_2 \\ g' & \text{if } (g = [\chi]g') \wedge p \in [\![\chi]\!] \\ \top & \text{if } (g = [\chi]g') \wedge p \notin [\![\chi]\!] \end{cases}$$

Example 3 In Fig. 2(right), products $ftg_{|\{s\}}$ and $ftg_{|\{t\}}$ of the FTG ftg are depicted. The TG $ftg_{|\{s\}}$ in Fig. 2(bottom-right) is a model of the vending machine that can only deliver soda bottles, whereas the TG $ftg_{|\{t\}}$ in Fig. 2(top-right) is a model of the vending machine that can only deliver tea. Product $ftg_{|\{s,t\}}$ is not shown.

The semantics of an FTG model of a product line is defined as a function that associates every valid product with the semantics of its projection.

Definition 6 (FTG semantics) The semantics of an FTG $ftg = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, \varphi, \gamma)$ is defined as the function $[\![ftg]\!]_{FTG}$ such that

$$\forall p \in [\![\varphi]\!]_{FM} \bullet [\![ftg]\!]_{FTG}(p) = [\![ftg_{|p}]\!]_{TG}$$

3.3 Featured Mutant Model

The idea underlying model-based mutation testing is to guide the test-case generation by mutants, which are typically obtained through random mutations of the original model. Organising the mutants as a product line of mutations, a family of variations of the system under test (SUT), coined the featured mutant model (FMM) in Devroey et al. (2016), enables the efficient generation, configuration, and execution of mutants. Each feature in the FMM corresponds to a single application of one mutant operator on the original model.

Like Devroey et al. (2016), we use a selection of the operators proposed by Fabbri et al. (1999), based on Chow (1978) and Weyuker et al. (1994), to generate mutants from a TS:

`TMI` Transition MIssing operator removes a transition;
`TAD` Transition ADd operator adds a transition between two states;
`SMI` State MIssing operator removes a state (other than the initial state) and all its incoming/outgoing transitions.

Additionally, we introduce the following operators specific to timed models, which change the constant in clock constraints, which we recall to be either a transition guard or a location invariant:

`CXL` Constant eXchange L operator increases the constant of a clock constraint;
`CXS` Constant eXchange S operator decreases the constant of a clock constraint;
`CCN` Clock Constraint Negation operator negates a clock constraint.

The CCN operator is inspired by the μ_{ng} operator from Aichernig et al. (2013), where only clock constraints appearing as transition guards are negated.

Each operator can be used to generate mutants using either the enumerative approach or the FMM approach. In the enumerative approach, each mutation transforms an FTG model *ftg*, representing the SUT behaviour, into a mutant ftg_m.

Example 4 The FTG in Fig. 3(right) has been obtained from the FTG in Fig. 3(left) by applying the mutation operators `TMI`, `CXL`, and `CXS`. The transition labelled with a soda bottle was removed (`TMI`). Moreover, constant 2 in the clock constraint that acts as transition guard was increased to **4** to model that producing a tea takes more time (`CXL`). Instead, constant 5 in the clock constraint that acts as location invariant was decreased to **4** to model that the vending machine takes less time to produce a drink (`CXS`). Thus, the transition from s_1 to s_0 that models the delivery of a cup of tea now occurs (instantaneously) precisely when $x = 4$. The feature model was not changed.

In the FMM approach, each mutation operator is added as a feature to the existing feature model. When considering first-order mutation (only one mutation can be applied to the original system), the features/mutations are mutually exclusive. For higher order mutations, disjunction is used instead.

Example 5 Adding the `TMI`, `CXL`, and `CXS` operators to the FTG in Fig. 4(left), results in the FTG ftg_{fmm} depicted in Fig. 4(right) with feature model φ_{fmm} depicted in Fig. 5. We now explain this.

To begin with, the `TMI` operator removes the transition $t_1 = s_1 \xrightarrow{[s]x \leq 2,} s_0$ of the base model in the following way:

1. The feature expression $\neg tmi$ is added to the feature expression of t_1, resulting in transition $s_1 \xrightarrow{[s \wedge \neg tmi]x \leq 2,} s_0$, meaning that this transition may be fired only if the *tmi* mutation is deactivated (and if s is true);
2. The feature *tmi* is added to the feature model φ_{fmm} representing the application of the mutation operator (cf. Figure 5).

Moreover, the `CXL` operator increases the constant 2 to 4 in the clock constraint that acts as guard on the transition $t_2 = s_1 \xrightarrow{[t]x \geq 2,} s_0$ of the base model, in the following way:

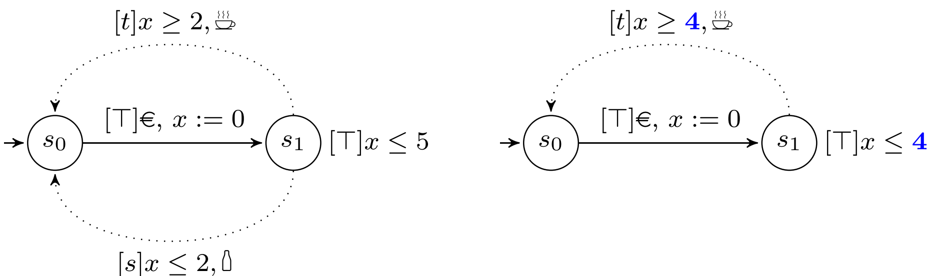

Fig. 3 FTG (right) resulting from the application of mutation operators `TMI`, `CXL`, and `CXS` to the FTG (left) reproduced from Fig. 2(left)

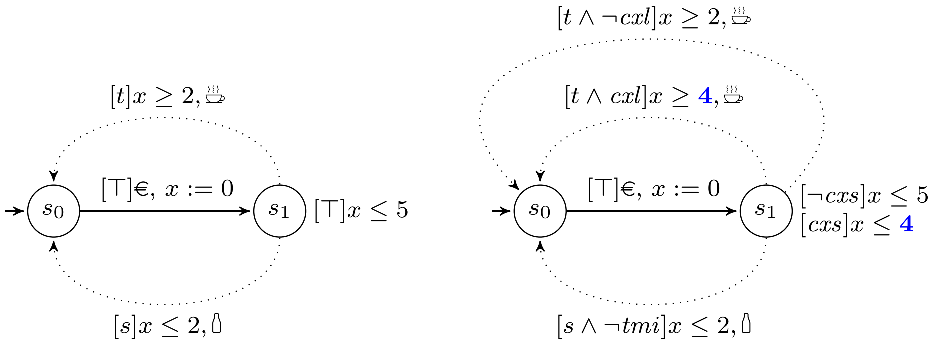

Fig. 4 FTG ftg_{fmm} (right) resulting from the addition of the mutation operators `TMI`, `CXL`, and `CXS` to the FTG (left) reproduced from Fig. 2(left); its associated feature model is depicted in Fig. 5

1. The feature expression $\neg cxl$ is added to the feature expression of t_2, resulting in transition $s_1 \xrightarrow{[t \wedge \neg cxl]x \geq 2, ☕} s_0$, meaning that this transition may be fired only if the *cxl* mutation is deactivated (and if *t* is true);
2. The transition $s_1 \xrightarrow{[t \wedge cxl]x \geq 4, ☕} s_0$ is added, meaning that this transition with feature expression $t \wedge cxl$ and clock constraint $x \geq 4$ may be fired only if the *cxl* mutation is activated (and if *t* is true);
3. The feature *cxl* is added to the feature model φ_{fmm} representing the application of the mutation operator (cf. Figure 5).

Finally, the `CXS` operator decreases the constant 5 to 4 in the featured clock constraint $[\top]x \leq 5$, which acts as invariant of the state s_1 of the base model, in the following way:

1. The feature expression $\neg cxs$ is added to the featured clock constraint of state s_1, meaning that the updated featured clock constraint $[\neg cxs]x \leq 5$ acts as invariant $x \leq 5$ of s_1 only if the *cxs* mutation is deactivated;
2. The feature expression *cxs* is added to the featured clock constraint of state s_1, meaning that the updated featured clock constraint $[cxs]x \leq 4$ acts as invariant $x \leq 4$ of s_1 only if the *cxs* mutation is activated;
3. The feature *cxs* is added to the feature model φ_{fmm} representing the application of the mutation operator (cf. Figure 5).

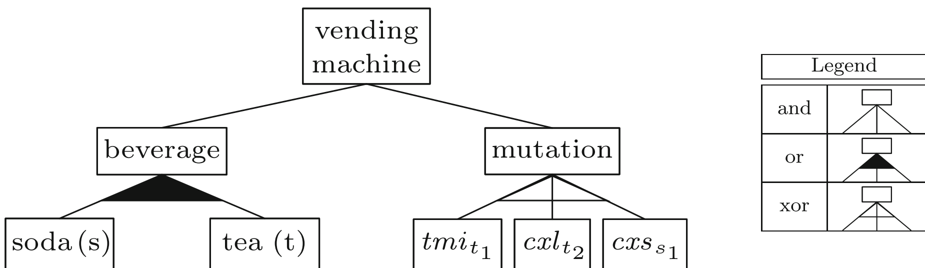

Fig. 5 Feature model φ_{fmm} of the FTG ftg_{fmm} depicted in Fig. 4(right)

Hence, mutation operators are added to the FMM under construction.

4 Classifying Mutations

Our main theoretical contribution is a classification of mutations to identify those that are effective (i.e., can be used to generate test cases). Our key idea from Basile et al. (2020a) is that, *by construction*, some mutations produce mutants that have the same (or a subset of the) behaviour of the SUT. Discarding them will speed-up the mutation testing process, as we would avoid fruitless attempts to generate test cases. Thus, we aim to characterise these mutations by formally proving under which conditions (i.e., mutation operator and the elements of the model to which it is applied) the produced mutant is subsumed by the SUT.

Recall that a test case generated from a mutant provides a sequence of inputs that makes the mutant behave differently than the SUT (in terms of accepted inputs, produced outputs, or execution time). Thus, the goal of the test case is to distinguish whether the system on which it is executed is the original one or the mutant. For a mutant to remain "live" (as it is named in the jargon), there must be no test case that can distinguish it from the SUT. This is equivalent to proving that the mutant is a *refinement* of the SUT (Larsen et al. 2017). Refinement checking is solved as a two-player timed game, where one player (playing the "whenever" transitions of the forthcoming Definition 7) wins if the mutant is not a refinement of the system (the mutant is killed) and the other player (playing the "then" transitions of Definition 7) wins if the mutant is a refinement (the mutant is alive). If the mutant is not a refinement, then the counterexample represents the test case that distinguishes the mutant from the SUT.

In what follows, we consider the mutation operators mentioned in Section 3.3 and state under which conditions their application results in a refinement of the original model. First-order mutations were shown to offer a higher fault-revealing ability (Papadakis and Malevris 2010). Our theoretical results hold for first-order and also higher order mutations. As such, when proving refinement relations, we consider the general case where mutations are applied to mutants of the SUT (either subsumed or not). Similarly, our work generalises to the case where the original model represents the behaviour of not only one system, but of a whole product line of systems. Thus, our theoretical developments are defined over FTG rather than single TG. To summarise, all results described hereafter apply to (1) **any-order mutations** and (2) **families of systems**.

4.1 Subsumed Mutants

To begin with, we formalise the notion of refinement between TG, adapted from David et al. (2010a) and Larsen et al. (2017). In Larsen et al. (2017), real-time systems are modelled as timed I/O automata, in which input actions are defined controllable and output actions are defined uncontrollable. The main idea is to perform a refinement check between the mutant and the system model, using Ecdar (David et al. 2010b), which is a tool built on top of Uppaal TIGA (Behrmann et al. 2007) that implements the timed interface theory from David et al. (2010a). Basically, a refinement model (i.e., a live mutant) must be able to mimic all controllable transitions of the original system model, while the original model must be able to mimic all uncontrollable transitions of the refinement. In our case, controllable transitions correspond to inputs (since a live mutant must accept all inputs that the original system accepts), whereas uncontrollable transitions correspond to outputs and delays (since a live mutant should not exhibit any behaviour that does not belong to the system). Note that this

is the opposite of the standard notion of modal refinement, where the inputs are seen as sent by an uncontrolled environment (Larsen et al. 2007). In other words, here the viewpoint is switched to the environment (David et al. 2010a; Larsen et al. 2017).

Definition 7 (Refinement) A TG $tg_1 = (Loc_1, Act_1, C_1, Trans_1, \ell_{01}, Inv_1, AP_1, L_1)$ is a refinement of a TG $tg_2 = (Loc_2, Act_2, C_2, Trans_2, \ell_{02}, Inv_2, AP_2, L_2)$, denoted as $tg_1 \preceq tg_2$, if there exists a binary relation $R \subseteq (Loc_1, Val(C_1)) \times (Loc_2, Val(C_2))$ that contains $s = ((\ell_{01}, v_{01}), (\ell_{02}, v_{02}))$ and is such that for each pair of locations and clocks values $((\ell_1, v_1), (\ell_2, v_2)) \in R$, it holds:

- whenever $(\ell_2, v_2) \xrightarrow{\alpha} (\ell'_2, v_2)$ for some ℓ'_2 and $\alpha \in Act^c_2$, then $(\ell_1, v_1) \xrightarrow{\alpha} (\ell'_1, v_1)$ for some ℓ'_1, $\alpha \in Act^c_1$ and $((\ell_1, v_1), (\ell'_2, v_2)) \in R$
- whenever $(\ell_1, v_1) \xrightarrow{\alpha} (\ell'_1, v_1)$ for some ℓ'_1 and $\alpha \in Act^u_1$, then $(\ell_2, v_2) \xrightarrow{\alpha} (\ell'_2, v_2)$ for some ℓ'_2, $\alpha \in Act^u_2$ and $((\ell'_1, v_1), (\ell'_2, v_2)) \in R$
- whenever $(\ell_1, v_1) \xrightarrow{\delta} (\ell_1, v'_1)$ for some v'_1 and $\delta \in \mathbb{R}_{\geq 0}$, then $(\ell_2, v_2) \xrightarrow{\delta} (\ell_2, v'_2)$ for some v'_2 and $((\ell_1, v'_1), (\ell_2, v'_2)) \in R$

We now provide a definition of *subsumed* mutant, where Op_{fmm} is the set of mutations. Basically, after applying an additional mutation the resulting mutant is a refinement of the former one on which the additional mutation was not applied.

Definition 8 (Subsumed mutant) Let *ftg* be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$. We say that m differs from m' by op iff $m = m' \cup op$ for some $op \in Op_{fmm}$. Moreover, we say that m is *subsumed* by m' iff $ftg_{|m} \preceq ftg_{|m'}$, and we say that it is *non-subsumed* otherwise.

Example 6 Consider the FTG ftg_{fmm} of Example 5, reproduced in Fig. 6(left), and its mutants $m_1 = \{s, tmi_{t_1}\}$ and $m'_1 = \{s\}$, depicted in Figs. 6(top-right) and 6(bottom-right), respectively, i.e., with $t_1 = s_1 \xrightarrow{x \leq 2, \lozenge} s_0$.

In this case, m_1 differs from m'_1 by tmi_{t_1}. Moreover, let $tg_1 = ftg_{fmm|m_1}$ and $tg_2 = ftg_{fmm|m'_1}$. It holds that $tg_1 \preceq tg_2$, i.e., tg_1 is subsumed by tg_2. Indeed, for all values v of x in the interval $[0, 5]$, the three points of Definition 7 hold for $((s_{1_{tg_1}}, v_{tg_1}), (s_{1_{tg_2}}, v_{tg_2}))$, and there is no configuration (s_1, v) with $v > 5$ because it would violate the invariant of s_1.

We only consider deterministic TG, as usual (Larsen et al. 2017; Aichernig et al. 2015; Aichernig et al. 2013). The following proposition identifies conditions under which a mutant is subsumed.

Proposition 1 (Basile et al. 2020a) *Let ftg be an FTG, let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$, and let m differ from m' by op. Then m is a subsumed mutant of m' iff op has introduced either less uncontrollable or more controllable behaviour (or trivially if the behaviour is unchanged).*

In the remainder of this section, we present several results for identifying mutations that generate subsumed mutants by construction. Proof (sketches) can be found in Basile et al. (2020a). We start with those operations that were proposed by Fabbri et al. (1999), followed by the novel ones introduced in this paper.

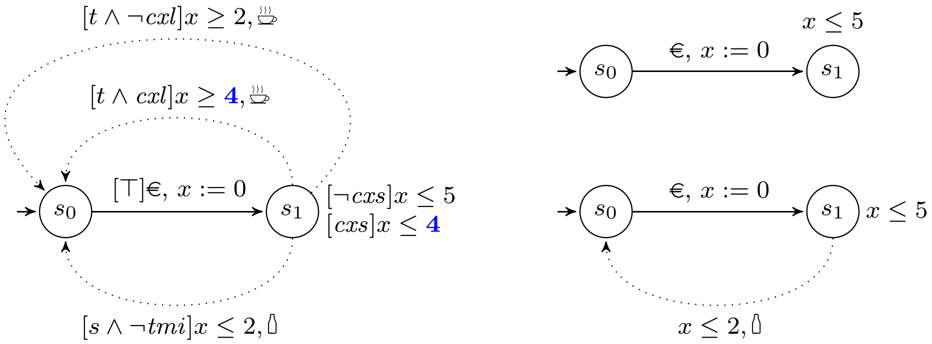

Fig. 6 FTG ftg_{fmm} (left) reproduced from Fig. 4 (right) and its mutants $ftg_{fmm|m_1}$ (top-right) and $ftg_{fmm|m'_1} = ftg_{|m'_1}$ (bottom-right) reproduced from Fig. 2(bottom-right)

`TMI` **mutation** The `TMI` mutation is used to remove a transition from the system. The following lemma shows that removing an uncontrollable transition from a mutant, by construction the resulting mutant is subsumed by the original one.

Lemma 1 (TMI Subsumed) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{tmi_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *with action in* Act^u*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

Example 7 We illustrate the usefulness of this result. Recall that test-case generation is more effective if the number of subsumed mutants is minimised. Continuing the previous example, since t_1 is an uncontrollable transition, Lemma 1 implies that $ftg_{fmm|m_1}$ is subsumed by $ftg_{fmm|m'_1}$, i.e., this is not a good candidate mutation for the configuration m'_1.

`TAD` **mutation** The `TAD` mutation is used to add a transition to the system. The next lemma shows that by adding a controllable transition to a mutant, the obtained mutant is subsumed by the original one.

Lemma 2 (TAD Subsumed) *Let ftg be an FTG and* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{tad_t\} \cup m'$ *for some t with action in* Act^c*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

`SMI` **mutation** The state missing `SMI` mutation removes a location from the system (not the initial location however). This is equivalent to making the location unreachable, i.e., removing all its incoming transitions. Hence, the results on `TMI` can be applied. The following lemma shows when this mutation produces a subsumed mutant.

Lemma 3 (SMI Subsumed) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{smi_\ell\} \cup m'$ *for some* $\ell \in Loc_{ftg_{|m'}}$*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$ *if there exists no transition t with target location* ℓ *and action* $\alpha \in Act^c$*.*

We continue with the mutation operators that were firstly introduced in Basile et al. (2020a).

`CXL` **mutation** We first turn our attention to the mutation `CXL`, that increases the constant of a clock constraint. The next lemma shows when the mutation operator `CXL` applied on a transition produces a mutant that is subsumed.

Lemma 4 (CXL Subsumed Transitions) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{cxl_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *with source* ℓ *and either (i) action in* Act^c *and guard* $g = x \leq k$ *or (ii) action in* Act^u*, guard* $g == k$ *and* $Inv(\ell) = x \leq k$ *or (iii) action in* Act^u *and guard* $g = x \geq k$*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

Example 8 Recall from Example 5 the mutation operator CXL applied to the FTG ftg_{fmm} that is reproduced in Fig. 7(left), and now consider its mutants $m_1 = \{t, cxl_{t_2}, cxs_{s_1}\}$ and $m'_1 = \{t, cxs_{s_1}\}$, depicted in Figs. 7(top-right) and 7(bottom-right), respectively, i.e., with $t_2 = s_1 \xrightarrow{x \geq 2, ☕} s_0$. Since t_2 is an uncontrollable transition, Lemma 4(iii) implies that $ftg_{fmm|m_1}$ is subsumed by $ftg_{fmm|m'_1}$, i.e., this is not a good candidate mutation for the configuration m'_1.

Finally, the next lemma identifies the conditions under which applying CXL on an invariant yields a subsumed mutant.

Lemma 5 (CXL Subsumed Invariants) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{cxl_\ell\} \cup m'$ *for some location* $\ell \in Loc_{ftg_{|m'}}$ *with* $Inv(\ell) = x \geq k$ *and for all valuations* v *of clock* x *such that* $k < v \leq k'$ *for* k' *mutation,* (ℓ, v) *can only be reached through a transition with action in* Act^u *and target* ℓ*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

Example 9 Consider Fig. 8, assume that $tg_1 = ftg_{|m}$ and $tg_2 = ftg_{|m'}$, for some ftg and $m = \{cxl_{s_1}\} \cup m'$, where cxl increases by one unit the clock constraint of location s_1. By Lemma 5, it holds that tg_1 is subsumed by tg_2.

CXS **mutation** We now turn our attention to the mutation CXS that decreases the constant of a clock constraint. The next lemma shows that the mutation operator CXS applied to a guard of the form $x \leq k$ of an uncontrollable transition or to a guard of the form $x \geq k$ of a controllable transition produces a mutant that is subsumed.

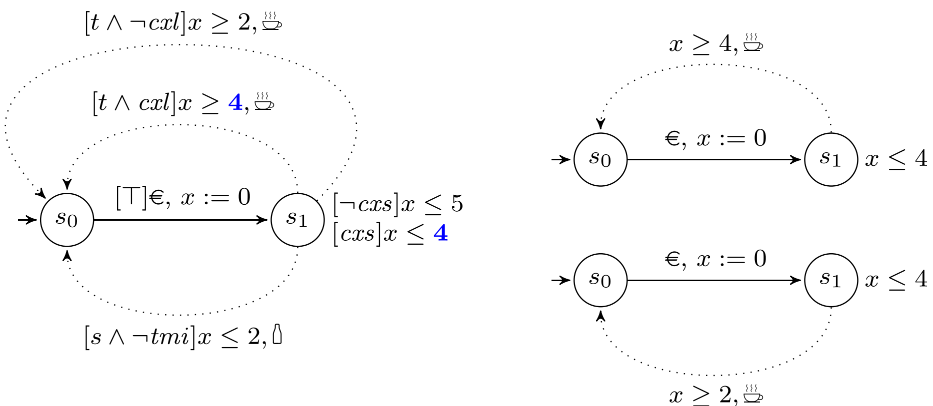

Fig. 7 FTG ftg_{fmm} (left) reproduced from Fig. 4 (right) and its mutants $ftg_{fmm|m_1}$ (top-right) and $ftg_{fmm|m'_1}$ (bottom-right)

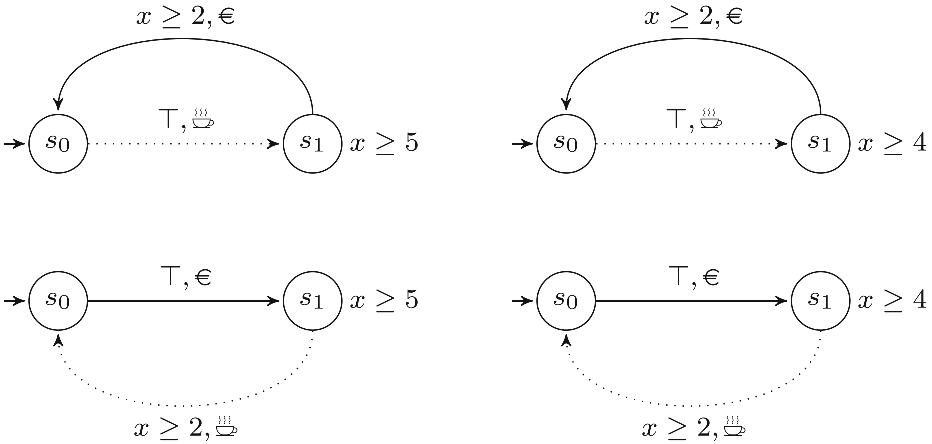

Fig. 8 Four TG used in Examples 9 and 10: tg_1 (top-left), tg_2 (top-right), tg_3 (bottom-left), and tg_4 (bottom-right)

Lemma 6 (CXS Subsumed Transitions) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{cxs_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *with either (i) action in* Act^u *and guard* $g = x \leq k$*; or (ii) action in* Act^c *and guard* $g = x \geq k$*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

Finally, the next lemma identifies the conditions under which the application of `CXS` on an invariant yields a subsumed mutant.

Lemma 7 (CXS Subsumed Invariants) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{cxs_\ell\} \cup m'$ *for some location* $\ell \in Loc_{ftg_{|m'}}$ *and either (i)* $Inv(\ell) = x \leq k$ *and for all valuations* v *of clock* x *such that* $k' \leq v < k$ *for* k' *mutation,* (ℓ, v) *can only be reached through a transition with action in* Act^u *and target* ℓ *or (ii)* $Inv(l) = x \geq k$ *and for all valuations* v *of clock* x *such that* $k' \leq v < k$ *for* k' *mutation,* (ℓ, v) *can only be reached through a transition with action in* Act^c *and target* ℓ*. Then* $ftg_{|m}$ *is subsumed by* $ftg_{|m'}$*.*

Example 10 Consider again Fig. 8, now assuming that $tg_3 = ftg_{|m'}$ and $tg_4 = ftg_{|m}$, for some ftg and $m = \{cxs_{s_1}\} \cup m'$, where cxs decreases by one unit the constant of the clock constraint of location s_1. By Lemma 7, it holds that tg_3 is subsumed by tg_4.

4.2 Auxiliary Results on Non-subsumed Mutants

Although our focus is on detecting subsumed mutants, the developed theory is also helpful in spotting when a specific mutation yields by construction a non-subsumed mutant. The following auxiliary results target non-subsumed mutants and complement the results stated so far (from Basile et al. 2020a). At this point, it is important to note that the experiments in Section 5 will only exploit results on subsumed mutants to discard ineffective mutations. However, the auxiliary results presented in this section could be exploited to perform a different evaluation: instead of discarding subsumed mutants, only generate (statically known) non-subsumed ones. This evaluation is harder, because only in specific cases it is possible to statically detect when a mutation is *non-redundant*. This is left for future work. A TG is

said to be *non-redundant* if every location ℓ is reachable in at least one trace, it is not time-locked (i.e., delay is possible), and every transition is executable in at least one trace. For the non-subsumed lemmata, we will only consider non-redundant TG. Indeed, mutating a redundant element may produce a subsumed mutant. Note that redundant specifications are ill-defined and should be amended prior to any application, be it testing, model checking or any other.

We introduce the auxiliary definition of non-redundancy preserving mutation. This will be useful for the forthcoming results about non-redundant higher order mutations, whose hypothesis is that the mutated mutant is non-redundant.

Proof (sketches) of the results already reported earlier can be found in Basile et al. (2020a).

Definition 9 (Non-redundancy preserving mutation) Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and with m that differs from m' by the application of some mutation $op \in Op_{fmm}$. Then, m is a *non-redundancy preserving mutation* of m' iff $ftg_{|m'}$ is non-redundant implies $ftg_{|m}$ is non-redundant.

Below follow two generic results on non-redundancy preserving on location or transition for any mutation. Note that the results below are operating at the syntactic level (i.e., statically). This means that the information on the specific values of clocks is missing. This information is only known at the semantics level (i.e., during the execution), where states are pairs of locations and clock evaluations. However, under specific hypothesis, it is possible to infer (statically) the values of clocks. Intuitively, if all clocks are reset when entering a location, the clocks evaluation when entering the location is statically known to be zero, and it is possible to predict whether guards and invariants will be satisfied. This allows to provide a result on non-redundancy preserving of a generic mutation involving a location that can be checked statically.

Proposition 2 (Non-redundancy preserving mutation on location) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$, and with m that differs from m' by the application of some mutation $op \in Op_{fmm}$ mutating a location ℓ, with invariant $Inv(\ell)$.*

If there exists a transition $t' = (\ell_{t'}, g_{t'}, \alpha_{t'}, R_{t'}, \ell)$, for some $\ell_{t'}, g_{t'}, \alpha_{t'}, R_{t'}$ and with $R_{t'} = C$, such that $v_0 \models Inv(\ell)$, and for all transitions $\hat{t}$ with source ℓ, it holds that $Inv(\ell) \wedge g_{\hat{t}} \not\models false$ and $R_{\hat{t}} = C$, then $ftg_{|m'}$ is non-redundant implies $ftg_{|m}$ is non-redundant.

Proof (sketch) By assumption t' is non-redundant, so ℓ is reachable through t'. By assumption, when reaching ℓ through t', $Inv(\ell)$ is satisfied. Moreover, all guards of outgoing transitions of ℓ at some point are satisfied by hypothesis. By the fact that all variables are reset in each transition, it holds that the behaviour of the underlying transition system is unchanged, thus $ftg_{|m}$ is non-redundant. □

We also provide a result on non-redundancy preserving of a generic mutation involving a transition that can be checked statically.

Proposition 3 (Non-redundancy preserving mutation on transition) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$, and with m that differs from m' by the application of some mutation $op \in Op_{fmm}$ mutating a transition $t = (\ell, g_t, \alpha_t, R_t, \ell'_t)$ for some ℓ, g_t, α_t, R_t, ℓ'_t such that $R_t = C$. If $Inv(\ell) \wedge g_t \not\models false$, then $ftg_{|m'}$ is non-redundant implies $ftg_{|m}$ is non-redundant.*

Proof (sketch) By assumption ℓ is reachable, and at some point transition t can be fired. By the fact that all variables are reset in t, it holds that the behaviour of the underlying transition system is unchanged, thus $ftg_{|m}$ is non-redundant. □

`TMI` **mutation** In case the location target of the removed transition is target of another transition, the mutation is non redundancy-preserving. This result (and the others) builds on the hypothesis of non-redundancy.

Proposition 4 (Non redundancy-preserving `TMI` mutation) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and with* $m = \{tmi_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *and let* $t' \in Trans_{ftg_{|m}}$ *such that* $t \neq t'$ *and* t, t' *have the same target location. Then,* $ftg_{|m'}$ *is non-redundant implies* $ftg_{|m}$ *is non-redundant.*

Proof Since ℓ is reachable by a transition $t' \neq t$ and by the fact that the projection removes all syntactically redundant elements, it follows that ℓ and t' are non-redundant and thus $ftg_{|m}$ is non-redundant. □

The following lemma shows that the application of a mutation `TMI` on a transition t (tmi_t in the following) of a mutant m', i.e., removing such a transition from m', produces by construction a mutant m that is non-subsumed by m', in case t is controllable.

Lemma 8 (TMI Non-subsumed) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{tmi_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *with action in* Act^c*. Then* $ftg_{|m}$ *is non-subsumed by* $ftg_{|m'}$.

Example 11 We illustrate the usefulness of this result by means of an example. Consider the FTG ftg'_{fmm}, depicted in Fig. 9(left), and its mutants $m_2 = \{tmi_{t_2}\}$ and $m'_2 = \emptyset$, depicted in Figs. 9(middle) and 9(right), respectively, i.e., $t_2 = s_0 \xrightarrow{€, x:=0} s_1$. Since t_2 is a controllable transition, Lemma 8 implies that $ftg'_{fmm|m_2}$ is non-subsumed by $ftg'_{fmm|m'_2}$, i.e., this is a good candidate mutation for the configuration m'_2.

`TAD` **mutation** The non-redundancy condition for transition adding mutation is trivial.

Proposition 5 (Non redundancy-preserving `TAD` mutation) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and with* $m = \{tad_t\} \cup m'$ *for some t non-redundant in* $ftg_{|m}$ *Then,* $ftg_{|m'}$ *is non-redundant implies* $ftg_{|m}$ *is non-redundant.*

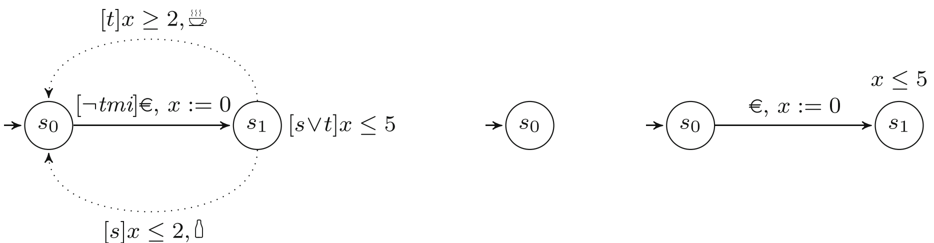

Fig. 9 FTG ftg'_{fmm} (left) and its mutants $ftg'_{fmm|m_2}$ (middle) and $ftg'_{fmm|m'_2} = ftg_{fmm|m_1}$ (right) reproduced from Fig. 6(top-right)

Proof Trivially, by non-redundancy of t. □

We note that a non-redundant transition to add could simply have a guard trivially *true* and an empty set of reset clocks (this will not be the case for the experiments in Section 5). Under the (assumed) hypothesis that the added transition is executable in at least one trace, such a mutation produces a non-subsumed mutant if an uncontrollable transition is added and a subsumed mutant if a controllable transition is added.

The next lemma shows that the application of a mutation TAD on a transition t (tad_t in the following) of a mutant m', i.e., adding such a transition to m', produces by construction a mutant m that is non-subsumed by m', in case t is uncontrollable. Note that the added transition is non-redundant in the mutant.

Lemma 9 (TAD Non-subsumed) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{tad_t\} \cup m'$ *for some t with action in* Act^u. *Then* $ftg_{|m}$ *is non-subsumed by* $ftg_{|m'}$.

SMI mutation The next lemma is about non-redundancy preserving state-missing mutation. Basically, the hypothesis of this lemma checks that no redundant locations are created by the mutation.

Proposition 6 (Non redundancy-preserving SMI mutation) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and with* $m = \{smi_\ell\} \cup m'$ *for some* $\ell \in Loc_{ftg_{|m'}}$ *where* $\forall \ell' \neq \ell$ *such that* ℓ' *is target of t and* ℓ *is source of t,* $\exists t' \in Trans_{ftg_{|m}}$ *with target* ℓ' *and source* $\ell'' \neq \ell, \ell'$. *Then,* $ftg_{|m'}$ *is non-redundant implies* $ftg_{|m}$ *is non-redundant.*

Proof The condition $\forall \ell' \neq \ell$ such that ℓ' is target of t and ℓ is source of t, $\exists t' \in Trans_{ftg_{|m}}$ with target ℓ' and source $\ell'' \neq \ell, \ell'$ ensures that the removal of such transitions t do not cause any redundant location, using the same argument used in Proposition 4. □

CXL mutation The condition for non-redundancy preserving CXL mutation is based on a relaxation of the clock constraint.

Proposition 7 (Non redundancy-preserving CXL mutation) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and with* $m = \{cxl_x\} \cup m'$ *for some* $x \in Trans_{ftg_{|m'}} \cup Loc_{ftg_{|m'}}$ *with clock constraint* $x \leq k$. *Then,* $ftg_{|m'}$ *is non-redundant implies* $ftg_{|m}$ *is non-redundant.*

Proof This mutation introduces a relaxation of the clock constraint of transition t. As such, the previous behaviour is still available and by the hypothesis of non-redundancy of $ftg_{|m'}$ it follows that $ftg_{|m}$ is non-redundant. □

Let c be a clock and let k be some constant. Then such a mutation does not generate a subsumed mutant when applied to a guard of the form $x \leq k$ of an uncontrollable transition or to a guard of the form $x \geq k$ of a controllable transition, under conditions discussed in the next lemma.

Lemma 10 (CXL Non-subsumed Transitions) *Let ftg be an FTG and let* $[\![\varphi]\!]$ *be the set of mutants with* $m, m' \in [\![\varphi]\!]$ *and* $m = \{cxl_t\} \cup m'$ *for some* $t \in Trans_{ftg_{|m'}}$ *with source* ℓ *and*

either (i) action $\alpha \in Act^u$, guard $g = x \leq k$ and there exists a valuation of clock $k < v \leq k'$ for k' mutation such that $(\ell, v) \xrightarrow{\alpha}_{ftg_{|m}}$ or (ii) action $\alpha \in Act^c$, guard $g = x \geq k$ and there exists a valuation of clock $k \leq v < k'$ for k' mutation such that $(\ell, v) \xrightarrow{\alpha}_{ftg_{|m'}}$. Then $ftg_{|m}$ is non-subsumed by $ftg_{|m'}$.

The next lemma provides the conditions under which the application of the mutation operator `CXL` on an invariant of a location ℓ (written as cxl_ℓ), yields a non-subsumed mutant.

Lemma 11 (CXL Non-subsumed Invariants) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and $m = \{cxl_\ell\} \cup m'$ for some location $\ell \in Loc_{ftg_{|m'}}$ and either (i) $Inv(\ell) = x \geq k$ and there exists a valuation v of clock x such that $k < v \leq k'$ for k' mutation such that (ℓ, v) is reached through a transition with action in Act^c and target ℓ or (ii) $Inv(\ell) = x \leq k$. Then $ftg_{|m}$ is non-subsumed by $ftg_{|m'}$.*

`CXS` mutation Also the condition for non-redundancy preserving `CXS` mutation is based on a relaxation of the clock constraint.

Proposition 8 (Non redundancy-preserving `CXS` mutation) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and with $m = \{cxs_x\} \cup m'$ for some $x \in Trans_{ftg_{|m'}} \cup Loc_{ftg_{|m'}}$ with clock constraint $x \geq k$. Then, $ftg_{|m'}$ is non-redundant implies $ftg_{|m}$ is non-redundant.*

Proof This mutation introduces a relaxation of the clock constraint. Thus, the previous behaviour is still available and by the hypothesis of non-redundancy of $ftg_{|m'}$ it follows that $ftg_{|m}$ is non-redundant. □

Again, let c be a clock and let k be some constant. Then such a mutation produces a non-subsumed mutant when applied to a guard of the form $x \leq k$ of a controllable transition or to a guard of the form $x \geq k$ of an uncontrollable transition, as the next lemma shows.

Lemma 12 (CXS Non-subsumed Transitions) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and $m = \{cxs_t\} \cup m'$ for some $t \in Trans_{ftg_{|m'}}$ with source ℓ and either (i) action in Act^c, $g = x \leq k$ and there exists a clock valuation $k' < v \leq k$ with k' mutation such that $(\ell, v) \xrightarrow{\alpha}_{ftg_{|m'}}$ or (ii) action in Act^u, $g = x \geq k$ and there exists a clock valuation $k' \leq v < k$ with k' mutation such that $(\ell, v) \xrightarrow{\alpha}_{ftg_{|m}}$. Then $ftg_{|m}$ is non-subsumed by $ftg_{|m'}$.*

The next lemma provides the conditions under which the application of the mutation operator `CXS` on an invariant of a location ℓ (written as cxs_ℓ) produces a non-subsumed mutant.

Lemma 13 (CXS Non-subsumed Invariants) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and $m = \{cxs_\ell\} \cup m'$ for some location $\ell \in Loc_{ftg_{|m'}}$ and either (i) $Inv(\ell) = x \geq k$ and there exists a valuation v of clock x such that $k' \leq v < k$ for k' mutation such that (ℓ, v) is reached through a transition with action in Act^u and target ℓ or (ii) $Inv(\ell) = x \leq k$ and there exists a valuation v of clock x such that $k' \leq v < k$ for*

k' mutation such that (ℓ, v) is reached through a transition with action in Act^c and target ℓ. Then $ftg_{|m}$ is non-subsumed by $ftg_{|m'}$.

Example 12 Recall from Example 5 the mutation operator CXS applied to the FTG ftg_{fmm} reproduced in Fig. 10(left), and consider its mutants $m_1 = \{t, cxs_{s_1}\}$ and $m'_1 = \{t\}$, depicted in Figs. 10(top-right) and 10(bottom-right), respectively. Since the clock constraint $x \leq 5$ acting as an invariant of s_1 is reached through the controllable transition $s_0 \xrightarrow{€,\, x:=0} s_1$, Lemma 13(ii) implies that $ftg_{fmm|m_1}$ is non-subsumed by $ftg_{fmm|m'_1}$, i.e., this is a good candidate mutation for the configuration m'_1.

CCN mutation We turn our attention to the CCN operator that negates a clock constraint of a transition. For this mutation, no non-redundancy preserving properties have been identified. Indeed, the negation of a clock constraint requires to explore the zone graph to check if the corresponding transition is non-redundant, as well as other locations and transitions only reachable by that transition. For all non-redundant TG, this mutation always generates a non-subsumed mutant.

Lemma 14 (CCN Non-subsumed) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and $m = \{ccn_t\} \cup m'$ for some $t \in Trans_{ftg_{|m'}}$. Then $ftg_{|m}$ is non-subsumed by $ftg_{|m'}$.*

4.3 Classifying Mutations

The following main theorem sums up all the results presented in this section. The specific additional conditions that need to hold for each mutation operator can be found in the corresponding lemmata.

Theorem 1 (Classifying mutations) *Let ftg be an FTG and let $[\![\varphi]\!]$ be the set of mutants with $m, m' \in [\![\varphi]\!]$ and with m that differs from m' by the application of some mutation $op \in Op_{fmm}$. Table 1 summarises when $ftg_{|m}$ is (non-)subsumed by $ftg_{|m'}$ based on the applied mutation.*

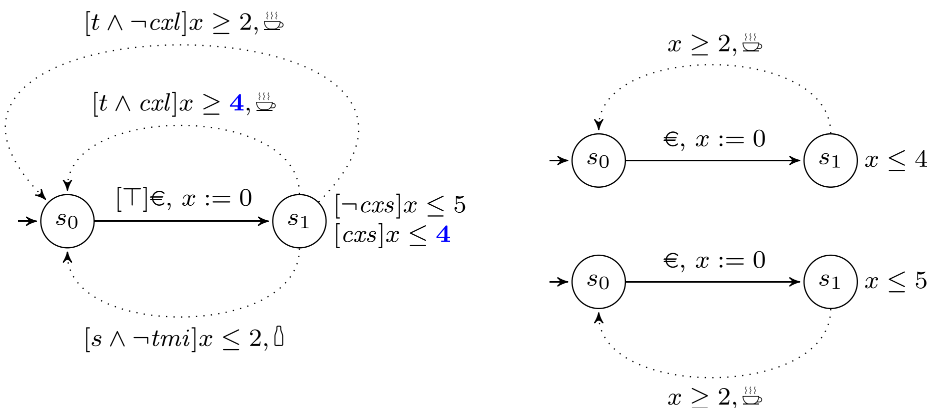

Fig. 10 FTG ftg_{fmm} (left) reproduced from Fig. 4 (right) and its mutants $ftg_{fmm|m_1}$ (top-right) and $ftg_{fmm|m'_1}$ (bottom-right) reproduced from Fig. 2(top-right)

Table 1 Subsumption relation for mutation operators, where ✗ stands for subsumed and ✓ for non-subsumed, while their subscripts x refer to the corresponding Lemmata x for the specific additional conditions that must hold for each mutation operator

subject \ operator	TMI	TAD	SMI	CXL	CXS	CCN
controllable transition	✓$_{8}$	✗$_{2}$				
+ clock constraint ≤				✗$_{4}$	✓$_{12}$	✓$_{14}$
+ clock constraint ≥				✓$_{10}$	✗$_{6}$	✓$_{14}$
uncontrollable transition	✗$_{1}$	✓$_{9}$				
+ clock constraint ≤				✓$_{10}$	✗$_{6}$	✓$_{14}$
+ clock constraint ==				✗$_{4}$		
+ clock constraint ≥				✗$_{4}$	✓$_{12}$	✓$_{14}$
location						
$\not\to$ controllable transition			✗$_{3}$			
invariant clock constraint						
→ controllable transition						
+ clock constraint ≤				✓$_{11}$	✓$_{13}$	
+ clock constraint ≥				✓$_{11}$	✗$_{7}$	
invariant clock constraint						
→ uncontrollable transition						
+ clock constraint ≤				✓$_{11}$	✗$_{7}$	
+ clock constraint ≥				✗$_{5}$	✓$_{13}$	

Proof The proof is obtained by cases, applying the lemmata discussed so far. □

4.4 Generating Effective Mutations

Based on our results for detecting subsumed mutants (i.e., Lemmata 1–7), we provide guidelines for generating effective FMM and their corresponding FTG. The FTG in Fig. 3 (right) is an example of a "bad" model. This is because two out of three mutants of the model are subsumed, and a subsumed mutant cannot be used to generate effective test cases (Larsen et al. 2017). Hence, while building the FMM and the corresponding FTG (cf. Section 3.3), ideally one wants to minimise the number of subsumed mutants, thus maximising the effectiveness of the test-case generation phase. To do so, we select from Table 1 those results for detecting subsumed mutants that are applicable by only checking the syntax of the original model, rather than those based on the semantics.

Note that in Basile et al. (2020a), we reported two more commandments. The seventh commandment in Basile et al. (2020a) was distilled from the current Lemma 11, and it is a condition ensuring that the generated mutant is non-subsumed. As stated in Section 4.2, Lemma 11 also requires the original model to be non-redundant. This was the first commandment in (Basile et al. 2020a), in fact a pre-requisite on the used models. Since our focus is on subsumption detection and to improve the separation of concerns, all results about non-subsumed detection have now been rendered as auxiliary and are thus no longer part of the guidelines (and neither is the non-redundant requirement).

The 10 Commandments of Model-Based Mutation Testing:

(1) `TMI` shall not be applied to uncontrollable transitions;
(2) `TAD` shall not be applied to controllable transitions;
(3) `SMI` shall not be applied if all incoming transitions are uncontrollable;
(4) `CXL` shall not be applied to controllable transitions with guards of the form $x \leq k$;
(5) `CXL` shall not be applied to uncontrollable transitions with guards of the form either (i) $x \geq k$ or (ii) $x == k$, and source invariant $x \leq k$;
(6) `CXL` shall not be applied to invariants of the form $x \geq k$ whenever all incoming transitions are uncontrollable;
(7) `CXS` shall not be applied to controllable transitions with guards of the form $x \geq k$;
(8) `CXS` shall not be applied to uncontrollable transitions with guards of the form $x \leq k$;
(9) `CXS` shall not be applied to invariants of the form $x \leq k$ whenever all incoming transitions are uncontrollable;
(10) `CXS` shall not be applied to invariants of the form $x \geq k$ whenever all incoming transitions are controllable.

Finally, we note that such guidelines could be implemented directly as constraints in the feature model of the FMM (cf. Figure 5), such that subsumed mutants are prevented from being generated. In the next section, we provide an empirical evaluation of the results presented in this section, i.e., certain subsets of mutations are guaranteed to produce mutants that are a refinement of the SUT and thus there is no need to use them.

5 Evaluation

To further validate our theoretical results and their benefits, we conduct a completely automatised empirical evaluation based on a proof-of-concept tool we developed, extending the preliminary experiments performed in Basile et al. (2020a).

5.1 Research Questions and Methodology

The objective of our empirical evaluation is to identify the mutation operators and the conditions under which a non-effective (i.e., subsumed) mutant is generated. We already addressed this by formally proving that mutants resulting from specific operators are subsumed under specific conditions, as reported in Theorem 1 and Table 1. To raise confidence in our results, we confront our theory with a practical implementation. Thus, we ask:

RQ1: Are our guidelines sound, i.e., are all mutants rejected by the guidelines indeed subsumed mutants?

Our next question concerns the benefits of avoiding the generation of mutants that are subsumed by construction. In practice, the saved computation time is dependent on the concrete test-case generation and execution platform. Instead, we measure these benefits in

a relative way, as the percentage of subsumed mutants that our guidelines can detect. Thus, we ask:

RQ2: How complete are our guidelines in detecting subsumed mutants?

To answer these questions, we apply the mutation operators to a given original model to produce first-order mutants. To produce second-order mutants we apply second mutations on first-order mutants. Then, we check whether those mutants violate guidelines and whether they are subsumed by the original system model, using the refinement check implemented in the Uppaal or Ecdar tools (David et al. 2010b).

5.2 Implementation

A proof-of-concept software has been implemented to perform the evaluation. The development of this software has been made necessary due to the high number of mutants to check for each case study. Indeed, in Basile et al. (2020a) the experiments were only considering first-order mutations of a single case study and were only partially automatised, requiring manual intervention to collect the data, parse the logs, and to produce the output tables. As a side-product, the software can be used as a tool to automatically check the guidelines on given mutants of Uppaal/Ecdar models, as well as for mutant generation with the mutation operators discussed in Section 3.3.

It is important to state at this point that the organisation of TG into the framework of FTG and FMM has been used to obtain a clean formalisation of the theoretical results discussed in Section 4. However, the guidelines for detecting subsumed mutants are independent from the way in which mutants are generated, i.e., they are not constrained to use FMM or FTG.

The experiments protocol is modular and has been divided into two main activities, depicted in Fig. 11, which led to the (parallel) production of two applications called AppMutant (developed by the third, fourth, and first authors) and AppEcdar (developed by the first author).

The experiment protocol is organised into a control part and an experimental part. The control part uses the run-time refinement checking offered by Uppaal (originally introduced

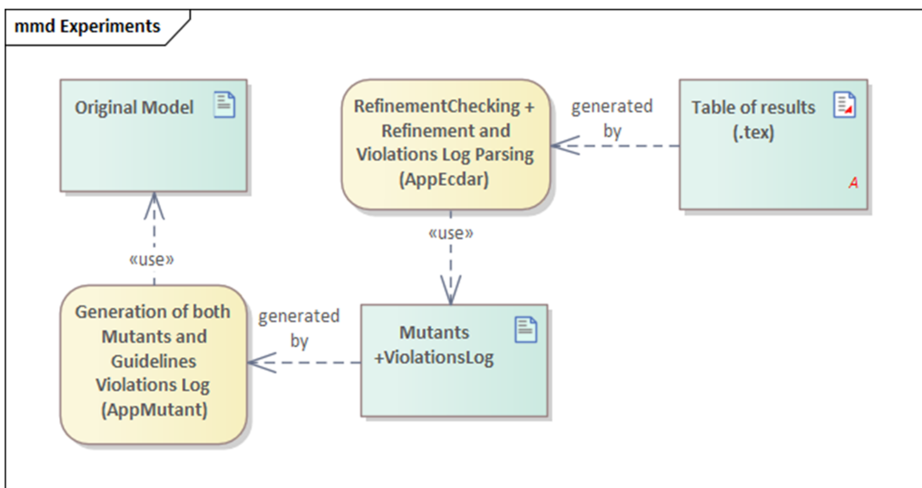

Fig. 11 Experiments mind mapping

in Ecdar) to check, for each mutant, whether or not it is a refinement. The experimental part statically analyses the mutants to check if they are violating the guidelines presented in this paper.

We note that whilst the theory presented in Section 4 only distinguishes between subsumed and non-subsumed mutants, the refinement checking performed by Uppaal may also return *inconsistent* as output. An inconsistent model is such that it cannot be refined to one satisfying two additional requirements, called *output urgency* and *independent progress* (cf. David et al. 2015, Def. 4). Output urgency states that an automaton cannot delay an enabled output. Independent progress is not satisfied by an automaton if it cannot internally (i.e., by only delaying or firing uncontrollable (output) actions) prevent an invariant from being violated in a location, against all possible inputs that may or may not arrive from the environment.

When answering a refinement query `Sm <= S`, Uppaal firstly refines `Sm` to be consistent. This means that the uncontrollable behaviour of `Sm` violating the requirements of consistency is pruned before performing the refinement checking (after pruning, the model can still be inconsistent because the controllable (input) behaviour is not pruned). Since, as for a subsumed mutant, an inconsistent model is not usable, we consider both subsumed and inconsistent mutants as "bad" mutants. More specifically, an inconsistent model is subsumed by any other model (similarly to how from false premises any conclusion can be reached), thus an inconsistent model is also a subsumed model.

As we will see in Section 5.4, this additional pruning operation of Uppaal that introduces further redundant mutated behaviour may cause subsumed or inconsistent mutants to be undetected by the commandments (for which this behaviour is not considered redundant).

Initially, the original models have been collected with only few adjustments (cf. Section 5.3) to make them consistent and usable by Uppaal. Indeed, we assume the *competent programmer hypothesis* (DeMillo et al. 1978) (i.e., developers produce initial models close to being correct). Thus, we assume that developers make only small mistakes (e.g., we will only mutate the clock constants by one time unit, see below). These simple mistakes (simulated by small mutations) can be put in cascade or coupled to form other emergent faults using higher-order mutants, according to the *coupling effect* (DeMillo et al. 1978).

The first activity is the generation of mutants which uses these original models to generate their mutants, in particular first-order and second-order (i.e., two mutations are applied) mutants. Each mutant operator is systematically applied to each relevant element (location or transition) of each model. The second-order mutants generation can be performed exhaustively or on a sample of each mutation ($\leq$10%, see below). Sampling can be useful when the number of second-order mutants is huge, due to a combinatorial explosion. We generated all mutants for all case studies, and sampling had been used earlier to quickly validate the code and models. Moreover, the first activity also performs the experimental part on the generated mutants: for each mutant it is checked whether or not it violates some guideline.

AppMutant is the software that has been developed (in Java) to perform both the generation of mutants and the guidelines checking. The class diagrams of AppMutant are displayed in Fig. 12.

Each mutation is represented as a class instantiating the interface `Mutation`, offering facilities to compute the number of possible applications of the mutation to a specific model, as well as applying the mutation to a model (given as a parameter). Which of the identified possible mutants is to be generated (and stored) is a second parameter to the method `apply`. The guideline checking is also performed in the method `apply`: a Boolean flag is returned by this method with value *true* in case the generated mutant is violating a guideline, and value *false* otherwise.

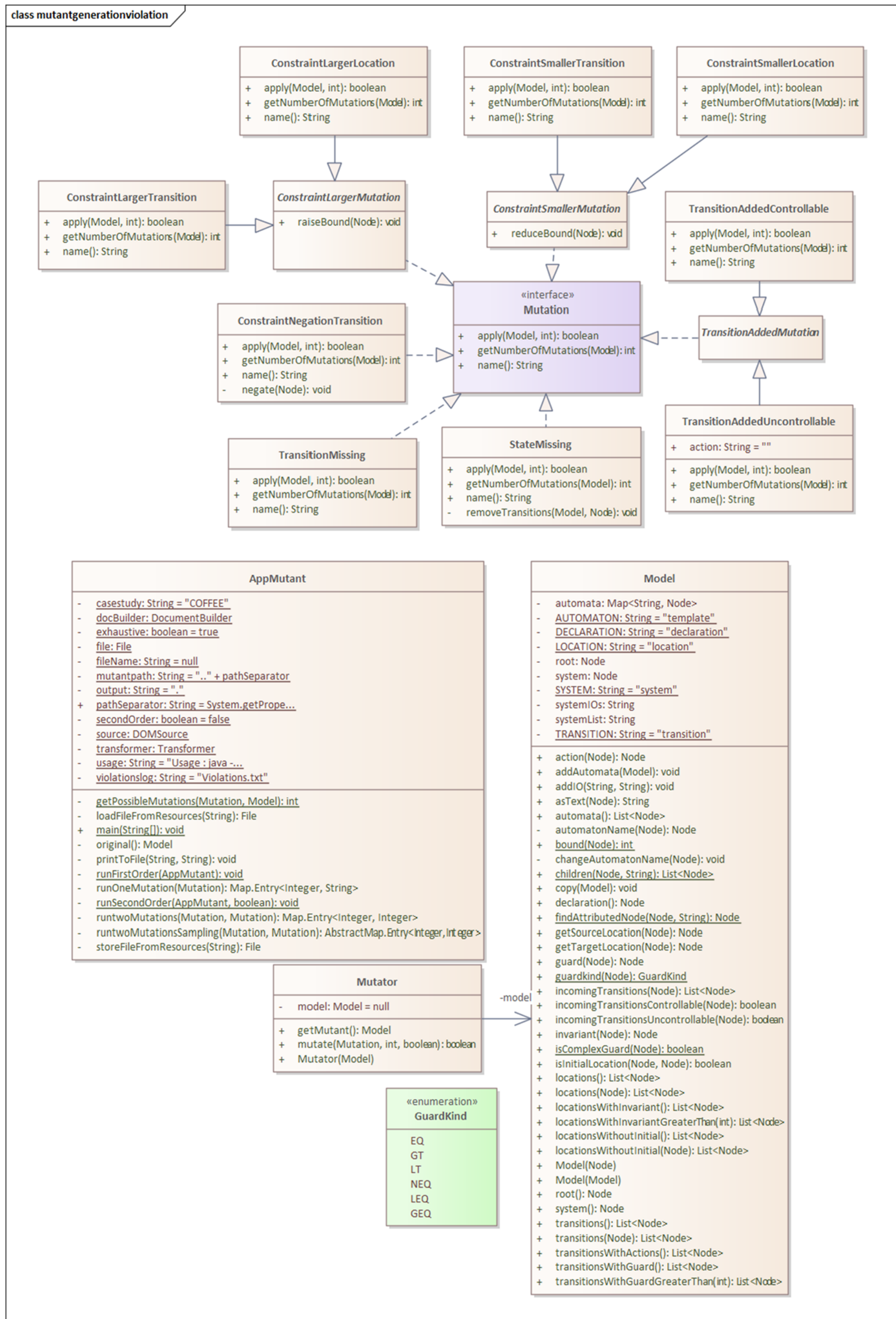

Fig. 12 Class diagrams of AppMutant (for readability, dependencies are not drawn as arrows)

For example, the abstract class `ConstraintSmallerMutation` implements `Mutation` and represents the mutation operator `CXS`. Two implementations of this abstract class are available, depending on whether the mutation is applied to a transition or to a location. In both cases, a method `reduceBound` is available to reduce the constant of the clock constraint. In the performed experiments, the bound is always reduced (resp., incremented) by one unit in `CXS` (resp., `CXL`). Note that `CCN` is only applied to transitions. The results presented in Section 4 are applicable to constraints with no conjunctions nor disjunctions. Accordingly, only these constraints can be mutated by the mutation operators. The `TAD` mutation operator is implemented by the abstract class `TransitionAddedMutation`. This abstract class is implemented by `TransitionAddedControllable` and `TransitionAddedUncontrollable`. The implementation of this mutation operator simply clones a transition (the first one encountered in the model), then changes the cloned transition source and target states and its action, and adds it to the model. In particular, a unique dummy action is used to avoid non-determinism.

The `Mutator` class is used to apply a `Mutation` to a `Model`. The class `Model` offers facilities to manage a Uppaal model (stored in `.xml` format). AppMutant is the executable class. It features methods for running first-order and second-order experiments.

Sampling is implemented in the `runTwoMutationsSampling` utility method. More in detail, this method takes as parameters the two mutation operators to apply, and the original model is retrieved with the method `original`. The method `getPossibleMutations` of the retrieved model returns the number of possible mutations that can be applied to that model for a given mutation operator (as parameter). By calling this method for the two mutation operators and multiplying the returned values (say, $n1$ and $n2$) the total number of possible mutations to apply is computed. The sampled number of mutants to generate is computed as $max(\frac{n1 \times n2}{10}, 1000)$. For each mutant, the selected mutations to apply are chosen uniformly in the interval $[0, n1]$ for the first mutation and $[0, n2]$ for the second mutation.

The executable AppMutant has options for generating either first-order or second-order mutants, with or without sampling. The outputs of the execution are the created mutants (stored as Uppaal models) and a comma separated values file containing for each mutation operator the number of violations. Moreover, for each mutation a file listing the mutants that are violating guidelines is provided. Each generated model comes equipped with two or three automata: the original model (`Spec`), the first-order mutant (`Spec_mutant`), and the second-order mutant (`Spec_mutant_mutant`).

After computing both mutants and the violations log, the second activity is carried out using the AppEcdar application. AppEcdar has two concerns: execute for each mutant model the refinement checking and parse the logs.

During the refinement checking the queries evaluated by Uppaal are as follows (for first- and second-order mutants, respectively):

```
refinement   : Spec_mutant <= Spec
refinement   : Spec_mutant_mutant <= Spec
```

The evaluations of these queries for each mutant model produce logs that are grouped by their mutation operators and stored.

Concerning log parsing, the logs of violations and refinement checking are processed to provide as output the LaTeX tables showed in this paper. Moreover, for validation purposes,

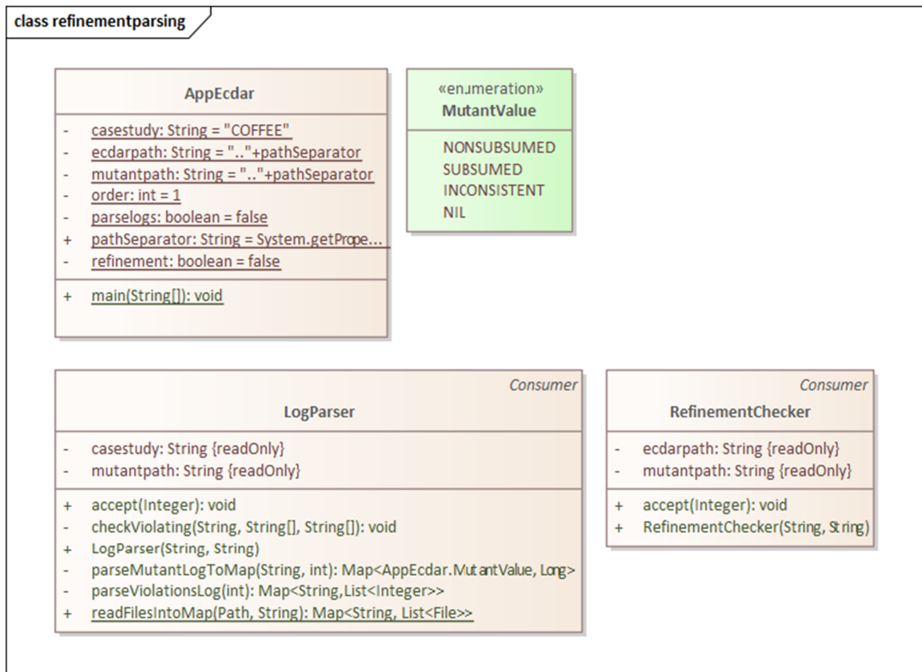

Fig. 13 Class diagrams of AppEcdar (for readability, dependencies are not drawn as arrows)

a text file (called false negatives) is used to store all mutants violating a guideline whose refinement checking output is non-subsumed. The class diagrams of AppEcdar are displayed in Fig. 13. The two main concerns of this activity are separated into two classes extending the Consumer functional interface of Java. The RefinementChecker class overrides the method apply(Integer) of Consumer<Integer> to execute the refinement checking, where the parameter indicates whether first-order or second-order mutants are processed. This class basically launches the verifytga Uppaal process, and can be executed on either Windows, Linux or Mac OS X. Similarly, LogParser extends the same interface and is used to perform the parsing: the mutants and violations logs parsing are decomposed into two separate methods. The class AppEcdar is the executable class. It offers options to select which case study to analyse, whether to perform refinement checking (indicating whether for first or second order), and whether to perform logs parsing (indicating whether for first or second order).

To enable the replication of our experiments, the source code, models, binaries and a video tutorial are publicly available.[1] Note that, for refinement checking, a distribution of Ecdar or Uppaal TIGA is required.[2] The experiments have been executed on a Windows machine with Java version 9.0.4 and a Linux Machine with OpenJDK 1.8, both using Uppaal TIGA 4.1.4 (rev. 5535), March 2014 academic version.

[1] Available at https://doi.org/10.5281/zenodo.5749732. The sources are also available at https://bitbucket.org/davidebasile84/timed_mutation/src/master/

[2] Available at https://uppaal.org/downloads/other/

5.3 Subject Systems

In this section, we describe the six case studies we analysed and provide references to the literature. While some case studies (i.e., the Car Alarm System) were already provided as TG in the literature, others required minor adaptations to be usable by Uppaal. Examples of such minor adaptations are: declarations of inputs as controllable and outputs as uncontrollable, and declaration of internal transitions as uncontrollable.

We also faced three issues. The first is that some case studies were parametric, whilst Uppaal TIGA cannot process parametric timed automata (i.e, it is not possible to have a transition using variables in the guard and channel synchronisation). Parametric constraints may also be mutually exclusive such that each set-up of parameters makes some transitions redundant. We solved this by instantiating each parameter to a specific value, taken from the literature when possible, and removing redundant behaviour.

The second issue is that Uppaal TIGA and timed interfaces do not support shared memory, i.e., no global variables are allowed in refinement checking. Since we analysed automata in isolation, all variables were made local.

The third issue is that Uppaal TIGA requires that each model is consistent (cf. Section 5). In some of the case studies the automata had originally been designed to be consistent when composed with the other automata of the same case study (with specific parameter assignments), but not for any possible environment, as instead is required by the consistency checking of Uppaal TIGA. We solved this third issue by fixing the automata not satisfying these properties for the specific set-up of parameters. Indeed, performing refinement checking of inconsistent models would be useless.

We specify below the changes we applied to the models in order to deal with these three issues. We evaluated the guidelines on six case studies retrieved from the literature that are detailed next.

5.3.1 CAS

The Car Alarm System (CAS) automaton we used in Basile et al. (2020a) stems from Ford's automotive demonstrator in the MOGENTES project, and is depicted in Fig. 14 (for readability, we omitted the sink state with incoming transitions from all states receiving unexpected inputs). The original model accounts six automata that are composed synchronously. We applied the mutations on the main automaton of the model, called system. This automaton has already been used for experiments in Aichernig et al. (2015), Larsen et al. (2017), and Aichernig et al. (2013). The system model allows as inputs the unlocking, locking, closing, and opening of a car's door. The outputs are the signals for arming, unarming, and turning the sound and flash alarms on and off. We mainly used the adaption of the automaton to Uppaal in Larsen et al. (2017), in which all input transitions are marked as controllable and all output transitions as uncontrollable.

5.3.2 Coffee

The coffee machine (COFFEE) model stems from an introduction to parametric timed automata (André et al. 2019), for didactic purposes. The model is composed of only one automaton, depicted in Fig. 15, and allows a user to press a button for coffee and possibly add sugar, upon which a cup of coffee is prepared and eventually delivered.

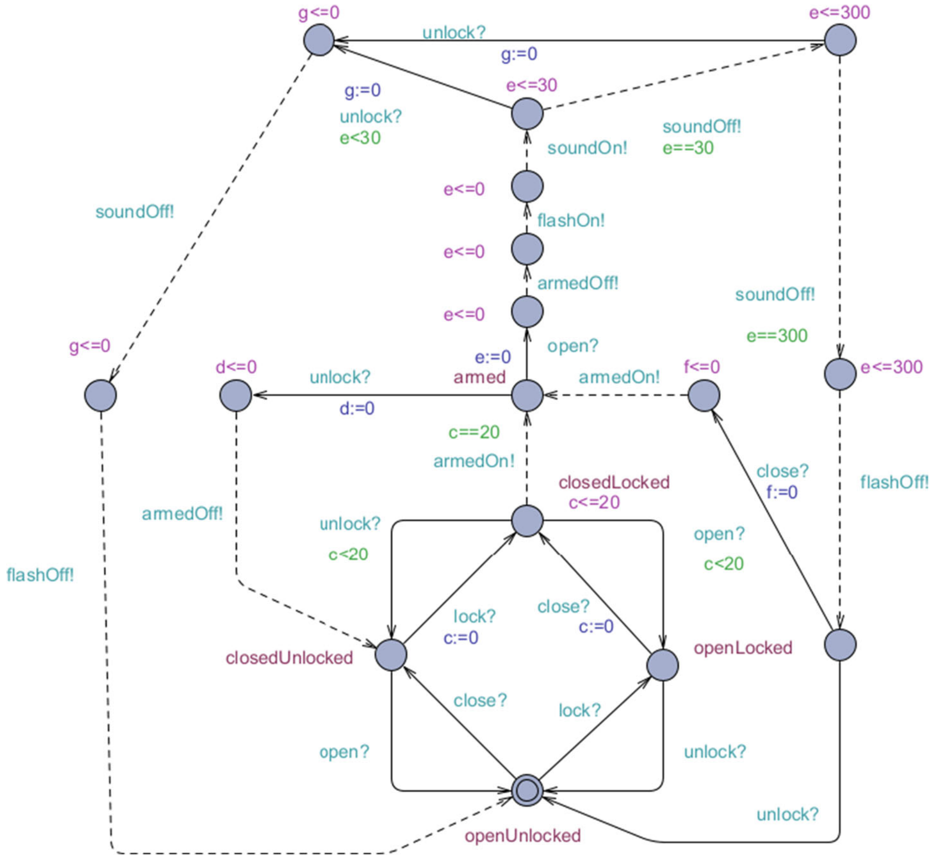

Fig. 14 The CAS automaton from Larsen et al. (2017) (for readability, the sink state is not depicted)

5.3.3 Accel

This is another parametric timed automata model from the automotive domain, stemming from Waga et al. (2017) and André et al. (2018), where it was presented as generated from a Simulink model of a scenario of monitoring the gear change of an automatic transmission system (Hoxha et al. 2015). The automaton that has been used for the experiments is depicted in Fig. 16. It has been slightly modified from its original version. In particular, we abstracted away the clock variable `xabs`. It models a basic automatic transmission requirement from Hoxha et al. (2015) (through a pattern that matches a part of a timed word that violates it): when shifting into any gear, there should be no shift from that gear to any other gear within a given time threshold.

5.3.4 Hotel

This is a timed service contract automata model from Basile et al. (2020c), depicted in Fig. 17. It models a hotel booking system that offers different types of rooms, requests payment either in cash or by card (of different duration) from clients, and possibly emits a receipt.

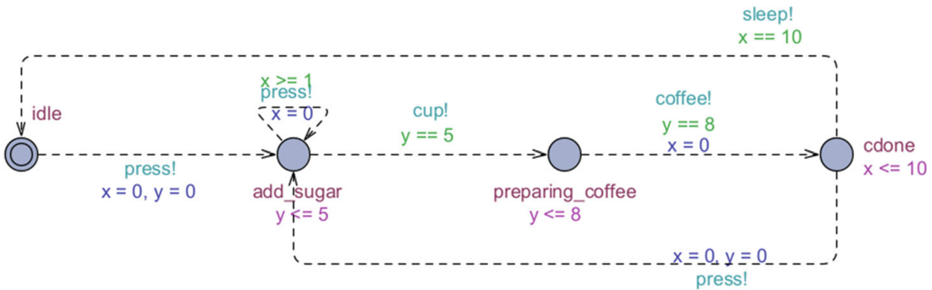

Fig. 15 The COFFEE automaton from André et al. (2019)

5.3.5 WFAS

The Wireless Fire Alarm System (WFAS) model is a parametric timed automaton model from Benes et al. (2015). It stems from Feo-Arenis et al. (2014), where it was presented as a formalisation of the requirements specified by a company specialised in radio technology, with the aim of verifying that the WFAS design passes the conformance tests put forward by European standards. It models two wireless sensors communicating with an alarm controller over a communication channel. The controller synchronises the two sensors and uses a clock x.

As for ACCEL, the automaton that has been used for the experiments, depicted in Fig. 18, has been slightly modified from its original version. We retrieved the values of the

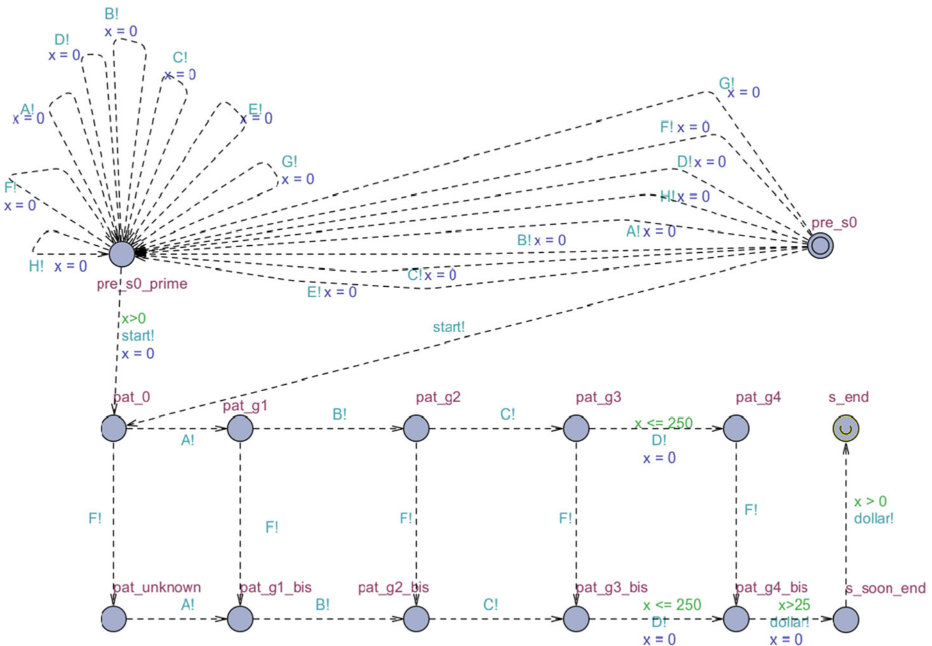

Fig. 16 The adapted ACCEL automaton of the parametric timed pattern matching benchmark from Waga et al. (2017) and André et al. (2018)

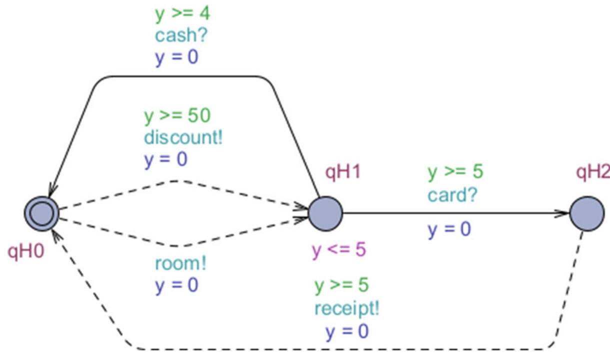

Fig. 17 The HOTEL automaton from Basile et al. (2020c)

parameters from Benes et al. (2015). We abstracted away the clock variable y from the clock invariant and removed redundant transitions.

We added two additional uncontrollable transitions from states cont_1 and cont_3 to the timeout state to satisfy independent progress and make the model consistent.

5.3.6 Mutex

This a parametric timed automaton model from Hune et al. (2001) of Fischer's mutual exclusion protocol (Lamport 1987) (cf. Ter Beek and Kleijn 2012 for an untimed token-based solution). Its purpose is to guarantee mutually exclusive access to a critical section among competing processes. Also this automaton, depicted in Fig. 19, has been slightly modified. The original model used a global variable lock shared with the other (replicated) automata. We pruned two transitions that were never enabled (and redundant) since this automaton is analysed in isolation.

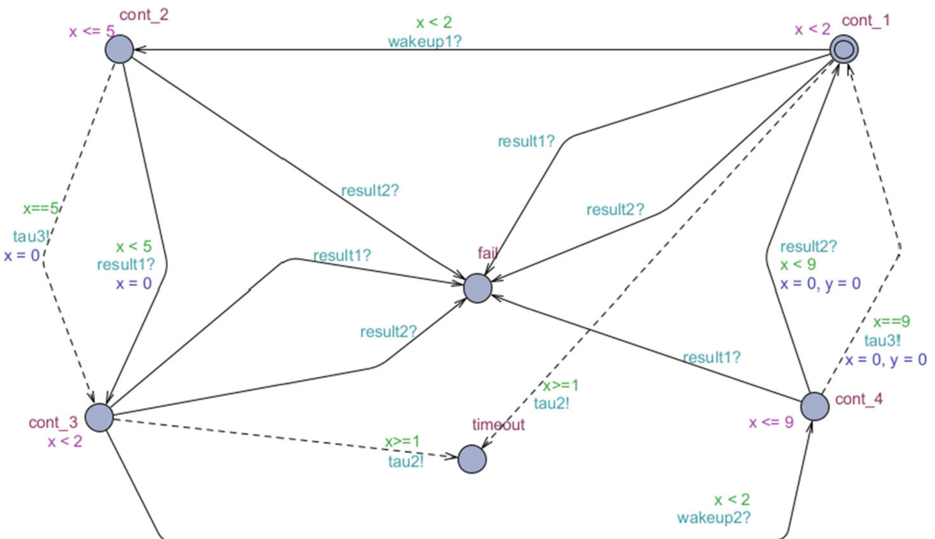

Fig. 18 The WFAS automaton adapted from Benes et al. (2015)

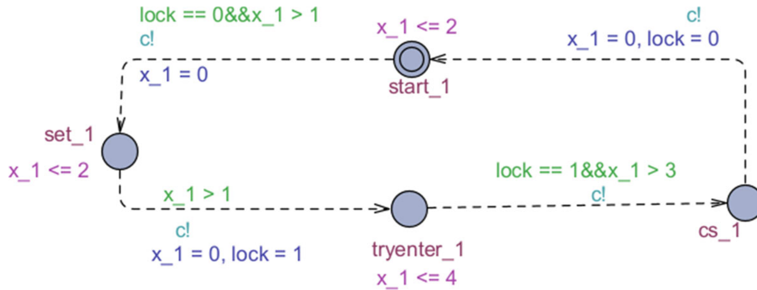

Fig. 19 The MUTEX automaton adapted from Hune et al. (2001)

5.4 Validation Results

Here we interpret the results of our experiments for RQ1 and RQ2 for all the case studies. We applied all mutation operators discussed in the paper, namely `TMI`, `TAD`, `SMI`, `CXL`, `CXS`, and `CCN`. Mutations are applied singularly for first-order mutants or in pairs for second-order mutants. For each case study, we provide a table for first-order mutations and a table for second-order mutations (for reasons of readability, the latter are reported in Appendix A). Each table reports the total number of mutants, that of non-subsumed mutants, of subsumed mutants, of inconsistent mutants, and of mutants violating the guidelines, as well as—in the last column—the ratio of mutants that are statically detected to be subsumed by our guidelines, computed as follows:

$$\frac{\text{(models) violating (guidelines)}}{\text{subsumed + inconsistent (models)}}$$

5.4.1 First Order

We start by discussing the results for first-order experiments. Firstly, no false negatives are detected: none of the mutants violating guidelines is non-subsumed, thus validating the theoretical results presented in the initial part of Section 4 (i.e., Lemmata 1–7).

CAS Table 2 contains the results for first-order mutants. We note that we refined the mutant generation phase, to reduce the number of inconsistent mutants generated. Thus, the results in Table 2 are not based on the same mutants as those in Basile et al. (2020a). A total of 721 first-order mutants were generated. A total of 77% of the first-order subsumed mutants and inconsistent mutants generated for this case study are detected by both Uppaal and the guidelines.

As showed in Fig. 14, many locations of this automaton have upper-bound invariants (i.e., $\leq$) and only one outgoing transition. This means that whenever one of these transitions is removed, the mutant may become inconsistent. For example, from the initial location `openUnlocked`, the sequence `close? lock?` of inputs leads to the location `closedLocked`. A mutation that removes the only uncontrollable outgoing transition of `closedLocked` produces an inconsistent mutant. Indeed, if no input arrives the system can only delay for a finite amount of time before the invariant $\texttt{c} \leq \texttt{20}$ is violated. This is the case for `TMI` and `SMI` mutations, the two mutation operators that remove transitions.

We also note two locations with invariant $\texttt{e} \leq \texttt{300}$ and incoming uncontrollable transition with guard $\texttt{e} = \texttt{300}$. In this case, increasing the constant of the clock guard or

Table 2 CAS first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	88	71	6	11	17	100%
TAD	578	197	381	0	289	76%
SMI	16	7	0	9	8	89%
CXL	20	8	9	3	6	50%
CXS	13	6	3	4	6	86%
CCN	6	6	0	0	0	-
totals	721	295	399	27	326	77%

decreasing the constant of the clock invariant makes the uncontrollable transition redundant. This in turn makes the source state of the (only outgoing uncontrollable) redundant transition not satisfying the independent progress property: the corresponding mutant is inconsistent. This is the case for mutations CXL and CXS. Moreover, the worst performance of the guidelines are for mutation CXL. This is due to the fact that enlarging a clock constraint constant in this specific automaton is likely to make an element redundant (as in the previous example), and thus make the corresponding mutant subsumed, even if no guideline is violated. Since the original model is consistent, TMI applied to a controllable transition (and thus not violating a guideline) never produces a redundant element. Indeed, in this automaton, 100% of subsumed or inconsistent mutants produced with TMI are detected to be violating a guideline.

COFFEE Table 3 contains the results for first-order mutants. A total of 55 first-order mutants were generated. A total of 91% of the first-order subsumed mutants and inconsistent mutants generated for this case study could have been avoided, because they are statically detected to be violating the guidelines.

This automaton shows the best performances of the guidelines when compared to the other case studies. Indeed, 100% of subsumed or inconsistent mutants for mutation operators TMI, SMI, and CXS are violating guidelines, whilst TAD has also a higher percentage of mutants detected by the guidelines. Operator CXL has six subsumed mutants, of which four are detected by the guidelines. These two subsumed mutants not detected by the guidelines result from the application of CXL to either location add_sugar or preparing_coffee (cf. Figure 15). In both cases, the constant is incremented by one.

We note that both models (original and mutant) can reach a configuration where $\ell =$ add_sugar and $y = 5$. From this configuration, if the mutant delays by one time unit then the original model cannot also delay. According to Definition 7 this proves that the mutant is not a refinement. However, in Uppaal this additional delay of the mutant is not allowed. The reason is that the reached configuration ($\ell =$ add_sugar and $y = 6$) is "bad": it violates the independent progress property (neither delays nor uncontrollable moves are allowed), which is required by Uppaal to hold for the mutant. Such transitions leading to "bad" configurations are pruned by Uppaal *before* performing the refinement checking (David et al. 2015). Accordingly, Uppaal detects this mutant to be subsumed. Basically, the increment of the constant adds redundant uncontrollable behaviour that does not modify the original specification.

Table 3 COFFEE first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	6	0	6	0	6	100%
TAD	32	15	17	0	16	94%
SMI	3	0	3	0	3	100%
CXS	6	3	3	0	3	100%
CXL	7	1	6	0	4	67%
CCN	1	1	0	0	0	-
totals	55	20	35	0	32	91%

The same reasoning applies to the case where CXL is applied to the location preparing_coffee. On the converse, this is not the case when applying CXL to the location cdone: the added uncontrollable behaviour is not redundant because of a further outgoing uncontrollable transition. Indeed, this mutant is detected by Uppaal to be non-subsumed.

ACCEL Table 4 contains the results for first-order mutants. A total of 451 first-order mutants were generated. A total of 86% of the first-order subsumed mutants and inconsistent mutants generated for this case study are statically detected by the guidelines. In this case, the percentage of detected mutants is 100% for the mutation operators TMI, SMI, CXS, and CXL.

The only inconsistent mutant is generated by TAD when adding a controllable transition from the initial state pre_s0 to state s_end. Indeed, the system cannot block inputs, and from the target state s_end no delay is allowed (the location is urgent) and no outgoing uncontrollable transition is present. This makes the mutant inconsistent (and correctly spotted as violating the second commandment). We note that the ACCEL model has no invariant on locations, and thus all locations are satisfying independent progress (except for the urgent one).

There are 16% TAD mutants detected by Uppaal as subsumed but not detected by the guidelines. These added uncontrollable transitions have no guard but only a dummy output.

Table 4 ACCEL first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	33	0	33	0	33	100%
TAD	392	158	233	1	196	84%
SMI	13	0	13	0	13	100%
CXS	3	1	2	0	2	100%
CXL	5	2	3	0	3	100%
CCN	5	4	1	0	0	0%
totals	451	165	285	1	247	86%

For example, the mutation adding an additional uncontrollable transition from the initial state to itself is detected as being subsumed, although it is not according to Definition 7. Since all locations are reachable, satisfy independent progress, and no constraints on locations are present, we argue that these 16% TAD subsumed mutants may be false positives (although we have no details on the internal implementation of Uppaal). However, due the high number of such subsumed mutants (i.e., 233), it is not possible to manually inspect each one of them. Notwithstanding false positives (which deteriorate the percentage of subsumed mutants detected by the guidelines), the guidelines in this case study are still efficient, by statically detecting 86% of the subsumed mutants.

Finally, in this case study we also note the presence of a subsumed mutant for the mutation operator CCN. This mutation is negating the clock guard of the transition with source s_soon_end and target s_end. This would apparently be a counterexample to Lemma 14. However, this is not the case because the generated mutant is redundant, thus violating the hypothesis of Section 4.2. Indeed, this specific mutation is not preserving non-redundancy of the mutant. This is because the negated clock guard $x \leq 0$ can never be enabled, since the target state is only reached when $x > 25$. This last example emphasises the hardness of statically detecting non-redundancy preserving mutations.

HOTEL Table 5 contains the results for first-order mutants. A total of 39 first-order mutants were generated.

A total of 51% of the first-order subsumed mutants and inconsistent mutants generated for this case study are statically detected by the guidelines, this being the worst performances when compared to the other case studies. This is because Hotel represents a "bad" model for the guidelines. We remark that the guidelines do not consider additional constraints imposed by Uppaal on the mutants, causing the increment of redundancy in the mutants. Indeed, since location qH1 (cf. Figure 17) does not satisfy independent progress, Uppaal prunes the only two outgoing transitions from the initial state, prior to start refinement checking. Thus, all elements (apart from the initial location) are redundant in the model when checking for refinement. Indeed, all mutants generated by TMI, SMI, CXL, CXS, and CCN are mutating redundant elements and thus are all subsumed.

For example, the four subsumed mutants of CCN (negating the 4 clock guards) are all cases of mutations adding redundant uncontrollable behaviour. Now consider the mutation negating the guards of the transition with source state qH0 and target state qH1. By Definition 7, from the initial configuration $(\texttt{qH0}, y = 0)$ the mutant could fire this mutated uncontrollable transition (and indeed this mutation is not violating any guideline). However,

Table 5 HOTEL first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	5	0	5	0	3	60%
TAD	18	4	13	1	9	64%
SMI	2	0	2	0	1	50%
CXL	5	0	5	0	2	40%
CXS	5	0	5	0	3	60%
CCN	4	0	4	0	0	0%
totals	39	4	34	1	18	51%

since the reached location qH1 does not satisfy independent progress, this uncontrollable behaviour is pruned by Uppaal prior to the refinement checking (i.e., the added behaviour is redundant). Therefore, the mutant is detected as subsumed.

The only mutant detected as inconsistent by Uppaal is a TAD mutation adding a controllable transition (qH0, $y \geq 4$, dummy?, $y = 0$, qH1). Indeed, as previously stated, location qH1 does not satisfy independent progress, and the system cannot prevent to reach this "bad" location due to the presence of the added dummy input transition. Note that this mutant is also violating a guideline: if this mutant were not inconsistent, it would anyway be subsumed.

Finally, concerning TAD applied to uncontrollable transitions, there are four non-subsumed mutants and five subsumed mutants (not detected by the guidelines). The four non-subsumed mutants are obtained by adding transitions from location qH1 to either of the locations of the model (3 in total). These mutants are non-subsumed because the mutations make location qH1 satisfy independent progress (and thus non-redundant). The last case is adding a transition from the initial state to qH2. This transition is not redundant because it does not involve state qH2, and the corresponding mutant is non-subsumed.

WFAS Table 6 contains the results for first-order mutants. A total of 121 first-order mutants were generated. A total of 74% of the first-order subsumed mutants and inconsistent mutants generated for this case study are detected by the guidelines.

Mutation TMI produces four inconsistent mutants that are all detected by the guidelines. Indeed, removing one of such four uncontrollable transitions makes the source state inconsistent (i.e., not satisfying independent progress).

One application of SMI to the state timeout causes the violation of the third commandment. The corresponding mutant is inconsistent. Its mutation operator corresponds to removing two uncontrollable transitions (similar to TMI). Removing state cont_3 (and all its incident transitions) makes state cont_2 not satisfying independent progress, and thus inconsistent. Since cont_3 has one incoming controllable transition, it does not violate the third commandment. The application of SMI to locations fail, cont_4, and cont_2 produces non-subsumed mutants.

Mutation TAD adds 36 controllable transitions (producing 36 mutants), all detected as violating guidelines and all subsumed. Of the 36 uncontrollable transitions added, 26 produce non-subsumed mutants, while ten produce subsumed mutants (not detected by the guidelines). These ten subsumed mutants all have an additional uncontrollable transition

Table 6 WFAS first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	14	10	0	4	4	100%
TAD	72	26	46	0	36	78%
SMI	5	3	0	2	1	50%
CXS	12	8	0	4	0	0%
CXL	12	2	6	4	8	80%
CCN	6	6	0	0	0	-
totals	121	55	52	14	49	74%

with guard x ≥ 2 and a dummy output. Two with source cont_1 and target cont_1 (resp., cont_3) and two with source timeout and target fail (resp., timeout) are not redundant, and their target state is not violating independent progress. These four mutants are false positives. The remaining added transitions are one with source cont_2 and target cont_2, two with source cont_3 and target cont_1 (resp., cont_3), one with source cont_4 and target cont_4, two with source fail and target fail (resp., timeout). From these source states the added transitions are never enabled, and thus redundant.

The application of CXS to any clock guard produces a non-subsumed mutants (for a total of 8 mutants), whereas its application to a clock invariant produces an inconsistent mutant. Of these four inconsistent mutants none is detected to be violating a guideline. Indeed, these are all cases of mutations that deactivate an uncontrollable transition, and similar to TMI this in turns causes the mutated state to not satisfy independent progress.

There are only two non-subsumed CXL mutants, namely the mutations applied to either location cont_1 or cont_3. For example, the mutation on cont_1 allows to delay by three time units before firing the transition to timeout. This behaviour is not allowed in the original model and thus according to Definition 7 the mutant is non-subsumed. The same happens with the mutation on cont_3. The other two applications of CXL to locations cont_2 and cont_4 are subsumed and are not detected by the guidelines. This is because Uppaal prunes the transitions delaying of one further time unit in both locations prior to refinement, since this (uncontrollable) behaviour leads to a configuration not satisfying independent progress. The application of CXL to transitions produces four inconsistent and four subsumed mutants. These are all detected to be violating the third commandment and point (i) of the fifth commandment. The four inconsistent mutants are mutating the guards of uncontrollable transitions, making them redundant and the source location violating independent progress.

Finally, all CCN mutations are producing non-subsumed mutants. We conclude that this case study is susceptible to violate independent progress when mutated, this being the main reason for the alternating performances of the mutation operators, whereas for TAD this is due to the possibility of adding redundant uncontrollable transitions.

MUTEX Table 7 contains the results for first-order mutants. A total of 47 first-order mutants were generated. A total of 73% of the first-order subsumed mutants and inconsistent mutants generated for this case study are detected to be violating some guideline.

In this case study, mutation operators TMI, SMI, CXL, and CXS detect 100% of subsumed or inconsistent mutants. The only CCN mutant is non-subsumed. Mutation TAD applied to controllable transitions produces 16 mutants, of which two are inconsistent and the others are subsumed. All 16 mutants are violating the second commandment. One of the two inconsistent mutants is generated by adding a controllable transition from start_1 to tryenter_1. In this case, the target state does not satisfy independent progress, because the only outgoing uncontrollable transition becomes disabled. The other inconsistent mutant is similar, in this case however the source state of the added transition is set_1.

Finally, the only subsumed mutants not detected by the guidelines are due to mutation TAD applied to uncontrollable transitions. This produces six non-subsumed mutants and ten subsumed mutants that are not detected by the guidelines. This is because the ten subsumed mutants are adding uncontrollable transitions whose guard is never enabled (due to the implementation of the TAD operator), and are thus redundant. Hence, in this specific case study the ten subsumed mutants are missed by the guidelines because of the way the TAD operator is implemented, i.e., the chosen guard for the added transition may render the new transition redundant.

Table 7 MUTEX first-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI	4	0	1	3	4	100%
TAD	32	6	24	2	16	62%
SMI	3	0	0	3	3	100%
CXS	3	0	0	3	3	100%
CXL	4	3	0	1	1	100%
CCN	1	1	0	0	0	-
totals	47	10	25	12	27	73%

5.4.2 Second Order

We now interpret the outcomes of the second-order experiments. Due to the high number of mutants (hundreds of thousands), it is not feasible to manually inspect each of the generated mutants (as we did for the first-order mutants). However, since second-order mutants are generated by mutating first-order mutants, the insights on the first-order experiments can be extended to the case of second-order mutants. The results of the second-order experiments for the six case studies are reported in Tables 8, 9, 10, 11, 12 and 13 and summarised in Fig. 22.

We observe a deterioration in the percentage of subsumed/inconsistent mutants detected by the guidelines. The average deterioration is $\approx 25\%$, with a minimum of 15% deterioration for ACCEL and a maximum of 34% for WFAS. By looking at the first-order experiments, this variation can be explained by the fact that ACCEL's undetected subsumed mutants are false positives, and none of them is violating independent progress. The converse happens for WFAS, whose mutants are highly susceptible to violate independent progress, and this is amplified in the second-order experiments.

We note that in the second-order experiments it is required that both mutations are violating the guidelines. If only one mutation is violating a guideline but the other is not, then the corresponding second-order mutant is detected as not violating the guidelines. Indeed, although the second mutation is relative to the corresponding first-order mutant (i.e., second-order mutants are obtained by performing an additional mutation on first-order mutants), the "bad" mutants are those that are refinements of the original model, and not of their relative first-order mutant. Indeed, we are interested in detecting all mutants (no matter of what order) that are subsumed by the original model.

More in detail, given a model `Spec`, a first-order mutant `Spec_mutant`, and a second-order mutant `Spec_mutant_mutant`, AppEcdar checks whether `Spec_mutant_mutant <= Spec`. If the first mutation on the original model violates any guideline, by the theoretical results presented in Section 4, it holds that `Spec_mutant <= Spec`. If also the second mutation of the first-order mutant violates any guideline, we know that `Spec_mutant_mutant <= Spec_mutant`. At this point, by the transitivity of `<=`, it is possible to conclude that `Spec_mutant_mutant <= Spec`. However, if one of the two violations is not violating any guideline, this conclusion cannot be reached. This also entails that first-order subsumed mutants cannot be discarded when performing second-order mutations, since otherwise it would not be possible to conclude that `Spec_mutant_mutant <= Spec` using the theoretical results of Section 4.

One of the causes of this deterioration of the guidelines' performances in the second-order mutants is the conjunction of two violation requirements.

Consider the models in Fig. 20. The first-order mutant MUTANTtad introduces controllable behaviour (with TAD, by adding a controllable transition), thus being subsumed by the original model SUT and violating the second commandment. Another first-order mutant MUTANTcxs, mutated with CXS, is subsumed by SUT due to the ninth commandment. Even though the two mutations are separately producing first-order mutants that are subsumed and separately violating guidelines, the application of CXS to MUTANTtad produces a mutant MUTANTtadcxs that is not violating the guidelines, even if MUTANTtadcxs <= SUT.

Indeed, applying the CXS mutation to MUTANTtad reduces the added controllable behaviour (i.e., it restricts the clock invariant target of the added transition), and thus MUTANTtadcxs $\not\leq$ MUTANTtad. Correspondingly, the second mutation is not violating any guideline and the experiments cannot conclude that the second-order mutant is violating the guidelines. However, since this reduced controllable behaviour was not present in SUT, it holds that MUTANTtadcxs <= SUT.

Finally, whilst applying a mutation on the original model is less prone to mutate a redundant element, this is not the case for second-order mutants. Indeed, if one mutation produces redundant behaviour (possibly also due to independent progress), then applying the second mutation on one such redundant element could produce a subsumed mutant not detected by our guidelines. Also, in case of an inconsistent first-order mutant, it is unlikely that the second mutation will fix the inconsistency.

We note that the presented theoretical results only account for a single mutation of a model (being it the original model or a mutant), and the second-order results are obtained by transitivity of <=. Further results about applying more than one single mutation on a model could help to improve the performances of higher-order mutants (e.g., syntactic conditions under which second-order mutants are subsumed by the SUT). However, such results are hard to obtain due to the possible presence of redundant behaviour and interactions between

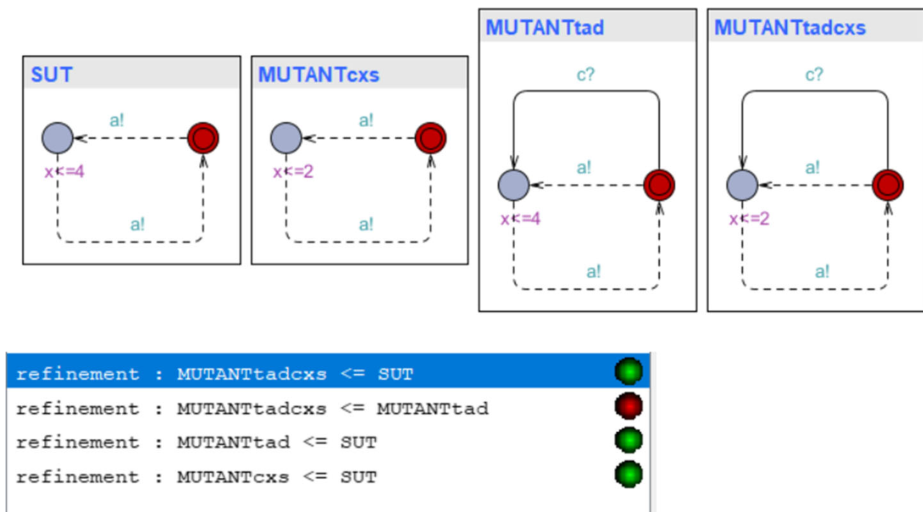

Fig. 20 An example showing a subsumed second-order mutant not detected by the guidelines using Uppaal TIGA

mutations. For example, it is not always true that if two mutations produce two separate subsumed first-order mutants, then their composition produces a second-order subsumed mutant (as occurs in Fig. 20).

To show this, consider Fig. 21. This example shows that the two first-order mutants `MUTANT1` (applying `CXS` on a transition) and `MUTANT2` (applying `CXL` on a location) are both subsumed by the original model `SUT`, because they are mutating redundant elements. However, the application of the two mutations (no matter in what order) produces a second-order mutant that is not subsumed by the `SUT`. This is because an interaction occurs between the two mutations, causing the redundant (uncontrollable) behaviour of the original model to become non-redundant and the second-order mutant to be non-subsumed.

Regardless of the obstacles when applying the guidelines to higher-order mutants, in our experiments we were able to statically detect 71% of the subsumed/inconsistent second-order mutants for the ACCEL case study, and 65% for the COFFEE case study.

5.5 Final Considerations

The experiments confirm that all mutants violating the guidelines were subsumed or inconsistent, as expected, thus providing further confidence in our results and an answer to RQ1. This has been theoretically proved in Section 4, and the correctness of the proofs was confirmed during the experiments. Indeed, AppEcdar performed a false negative checking, to check for the presence of non-subsumed mutants violating the guidelines. No such mutants have been identified in any of the case studies. The correctness of the theoretical results can also be observed by the fact that, for each case study, the number of mutants violating the guidelines is never greater than the number of mutants that are either subsumed or inconsistent. We remark that all subsumed and inconsistent mutants that are violating the guidelines are those that would not be generated if one were to apply the guidelines presented in Section 4.4.

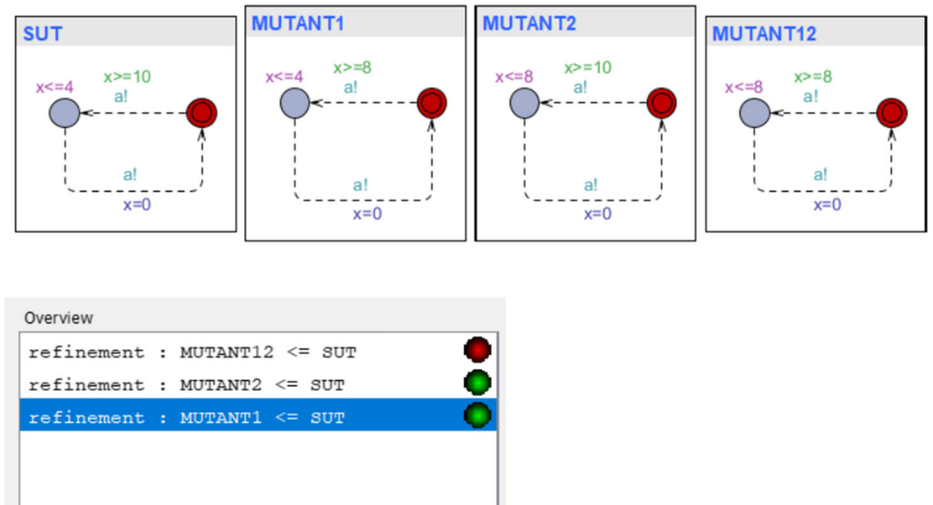

Fig. 21 An example, using Uppaal TIGA, showing two subsumed first-order mutants whose composition of mutations is not subsumed

To answer RQ2, we report a histogram summarising the outcome of the experiments in Fig. 22. The results confirm the gain of our approach: more than half of all subsumed first-order mutants are detected by our guidelines for each case study, detecting 91%, 86%, and 77% subsumed mutants for, respectively, the COFFEE, ACCEL, and CAS case studies. As discussed in Section 5.4.2, this holds also for second-order mutants. Indeed, the guidelines were capable of detecting more than half of the second-order subsumed or inconsistent mutants for the case studies COFFEE, ACCEL, and CAS, and fewer for the remaining three case studies.

We also report a histogram in Fig. 23 showing what is the likelihood of generating a subsumed mutant when applying random mutations, with or without using the guidelines, for both first-order and second-order experiments. We assume that all mutations have the same probability of being applied. For each case study, the probability of generating a subsumed mutant is computed as

$$1 - \frac{\texttt{non-subsumed mutants}}{\texttt{total mutants} - \texttt{mutants violating guidelines}}$$

(the parameter of `mutants violating guidelines` is equal to zero when generating mutants without using the guidelines). Notably, the guidelines can drastically diminish the probability of generating subsumed mutants in some case study. The histogram shows how this probability gets lower when applying two mutations (independently of the usage of the guidelines). Indeed, by augmenting the number of mutations, it is less likely that they will have no effect on the mutated model. We note that HOTEL has the highest probability of generating subsumed mutants. This also explains the worst performances of the guidelines for this particular case study.

Several directions of improvement of the static analysis performed by the guidelines are possible. Different implementations of the mutation operators can be investigated. For example, our implementation of the `TAD` mutation operator adds uncontrollable transition that may be guarded. This is likely to introduce subsumed mutants not detected by our

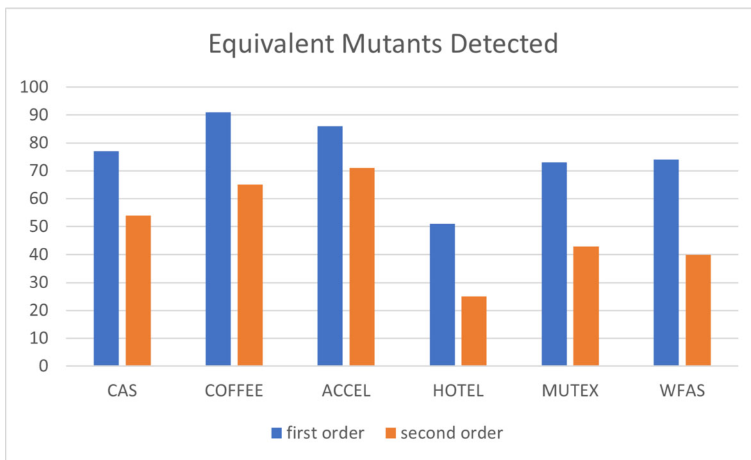

Fig. 22 Percentage of subsumed mutants that are detected by the guidelines

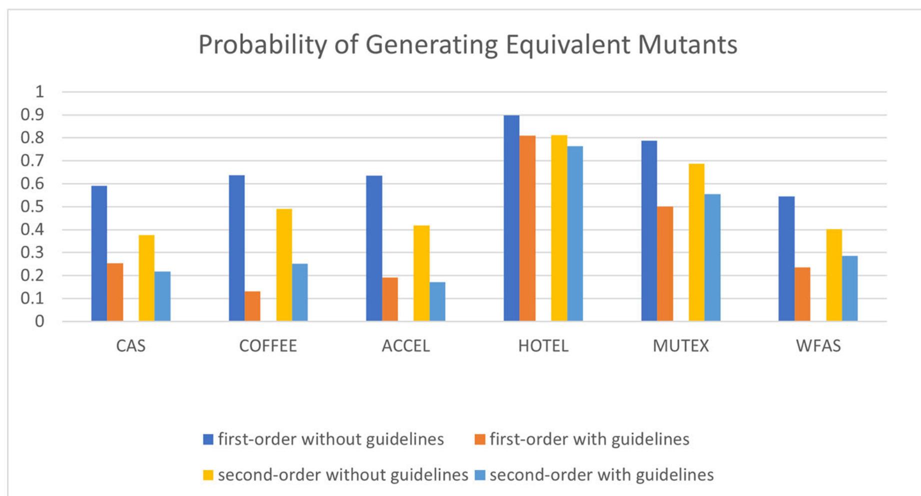

Fig. 23 The probability of randomly generating a subsumed mutant (with or without using the guidelines), assuming that mutants are distributed uniformly

guidelines when such guards are never satisfied. By changing the implementation of TAD such that no guarded transition is added, the generated mutants are less likely to be subsumed. We note that TAD mutation is responsible for many undetected subsumed mutants in all the analysed case studies.

Parametric timed games (Luthmann et al. 2019; Luthmann et al. 2017) can be investigated to perform exhaustive experiments for the mutations CXL and CXS for all possible constant value updates. In the presented experiments, CXL and CXS updated the constants of clock constraints by increasing or decreasing by one unit.

More research is needed for the static detection of redundant elements (e.g., dangling locations). Our experiments have exploited the refinement checking provided by Uppaal, which however performs a pre-processing step to prune uncontrollable behaviour not consistent. For experiments performed with Uppaal, syntactically detecting locations not satisfying independent progress is possible. For example, if a location has a clock invariant of the form $x \leq k$ and no outgoing uncontrollable transition, then this location is not satisfying independent progress. This is the case for location qH1 of the HOTEL case study, which is indeed generating the worst performances for our guidelines. We argue that by statically pre-processing models similar to the HOTEL model to remove such "bad" locations, the performances can be improved.

In Fig. 21, an example of two mutations producing subsumed mutants whose composition is producing a non-subsumed mutant is displayed. However, these two mutations are not violating the guidelines. Indeed, the set of mutants violating the guidelines is a strict subset of the subsumed mutants. Further research is needed to prove or disprove whether the composition of two mutations violating guidelines always yields a subsumed mutant.

Finally, we note that the variation of percentages of detected mutants in each case study

also depends on the number of controllable and uncontrollable transitions, as well as the presence of different types of constraints, which are specific to each case study.

5.6 Threats to Validity

Our empirical results essentially rely on the refinement relation as implemented in Uppaal/Ecdar. Should this implementation deviate (even slightly) from the definition we employ (itself based on the paper introducing Ecdar (David et al. 2010b)), we might witness the occurrence of false positives (mutants erroneously labelled as subsumed). To mitigate this risk, we conducted a non-exhaustive manual analysis of the sampled mutants.

Nevertheless, we did not remove these few false positives, which would actually improve our results since it would increase the percentage of subsumed mutants detected by our guidelines.

6 Conclusion and Future Work

We presented a methodology for discarding ineffective mutations for testing real-time systems. An effective mutant can be used to generate test cases that distinguish the mutant from the original system model. The framework of TG and Ecdar refinement checking of Larsen et al. (2017) was adopted, and mutants were organised as a product line of mutations using the approach of Devroey et al. (2016). Our guidelines to the construction of such a featured mutant model can be encoded as constraints in the feature model, to guarantee that effective mutants will be generated. Our experiments confirmed the soundness of our approach and demonstrated that our actionable guidelines can significantly reduce the number of subsumed mutants.

In future work, we plan to exploit the auxiliary results from Section 4.2 to perform a different evaluation than the one presented in this paper, viz., instead of discarding subsumed mutants, only generate (statically known) non-subsumed ones. As mentioned before, this evaluation is harder because only in specific cases it is possible to detect whether a mutation is non-redundant.

In the future, we also plan to investigate a family-based technique for checking refinements all-at-once directly on the FTG, in order to take further advantage of the product-line approach and of our technique for building effective featured mutant models. This would allow the generation of the smallest set of test cases that can distinguish all killable mutants. One way to do so is to design a feature-aware extension of the refinement checking procedure of David et al. (2010a, 2010b). By associating a feature to each mutant (Devroey et al. 2016), one could then collect the feature expressions identifying all mutants for which the refinement holds, and those for which it does not, in a single play. This problem was studied in the non-timed case (Cordy et al. 2012a), but it remains unaddressed for real-time systems. The addition of time makes this problem challenging, as there is no known efficient way to encode time and variability in a single data structure (Cordy et al. 2012b).

Our work also provides the foundations to evaluate real-time test cases. To this aim, one could apply the approach of Devroey et al. (2016) on a featured timed model to identify which mutants are killed. Again, this would require data structures combining time with variability.

Appendix A: Results for Second-Order Mutations

Table 8 CAS second-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI-TMI	7657	5844	30	1783	272	15%
TMI-TAD	50866	42576	2335	5955	4913	59%
TMI-SMI	1409	573	0	836	136	16%
TMI-CXS	1137	672	18	447	102	22%
TMI-CXL	1753	1248	57	448	99	20%
TMI-CCN	523	456	0	67	0	0%
TAD-TMI	51444	42576	2913	5955	4913	55%
TAD-TAD	334084	188798	145286	0	83521	57%
TAD-SMI	9250	4285	33	4932	2176	44%
TAD-CXS	7517	4168	1161	2188	1683	50%
TAD-CXL	11563	6497	3383	1683	1734	34%
TAD-CCN	3468	3468	0	0	0	-
SMI-TMI	1251	470	0	781	121	15%
SMI-TAD	8194	3823	33	4338	2048	47%
SMI-SMI	241	80	0	161	59	37%
SMI-CXS	188	59	0	129	44	34%
SMI-CXL	293	109	0	184	42	23%
SMI-CCN	85	33	0	52	0	0%
CXS-TMI	1146	678	18	450	102	22%
CXS-TAD	7517	4168	1161	2188	1734	52%
CXS-SMI	210	70	0	140	48	34%
CXS-CXS	172	80	9	83	36	39%
CXS-CXL	263	120	40	103	36	25%
CXS-CCN	79	54	0	25	0	0%
CXL-TMI	1762	1254	57	451	102	20%
CXL-TAD	11563	6497	3383	1683	1734	34%
CXL-SMI	322	124	0	198	48	24%
CXL-CXS	270	120	47	103	36	24%
CXL-CXL	403	206	87	110	36	18%
CXL-CCN	121	96	6	19	0	0%
CCN-TMI	529	462	0	67	0	0%
CCN-TAD	3468	3468	0	0	0	-
CCN-SMI	97	42	0	55	0	0%
CCN-CXS	79	54	0	25	0	0%
CCN-CXL	121	96	6	19	0	0%
CCN-CCN	36	30	6	0	0	0%
totals	519081	323354	160069	35658	105775	54%

Table 9 COFFEE second-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI-TMI	30	0	30	0	30	100%
TMI-TAD	193	61	128	4	96	73%
TMI-SMI	18	0	18	0	18	100%
TMI-CXS	33	8	25	0	18	72%
TMI-CXL	38	2	36	0	20	56%
TMI-CCN	5	4	1	0	0	0%
TAD-TMI	225	61	160	4	96	59%
TAD-TAD	1024	735	289	0	256	89%
TAD-SMI	97	11	82	4	36	42%
TAD-CXS	193	124	65	4	36	52%
TAD-CXL	225	105	116	4	64	53%
TAD-CCN	32	32	0	0	0	-
SMI-TMI	9	0	9	0	9	100%
SMI-TAD	55	11	40	4	27	61%
SMI-SMI	6	0	6	0	6	100%
SMI-CXS	10	0	10	0	6	60%
SMI-CXL	12	0	12	0	6	50%
SMI-CCN	2	2	0	0	0	-
CXS-TMI	36	8	28	0	18	64%
CXS-TAD	193	124	65	4	48	70%
CXS-SMI	18	0	18	0	9	50%
CXS-CXS	36	19	17	0	9	53%
CXS-CXL	42	14	28	0	12	43%
CXS-CCN	6	6	0	0	0	-
CXL-TMI	42	2	40	0	24	60%
CXL-TAD	225	105	116	4	64	53%
CXL-SMI	21	0	21	0	12	57%
CXL-CXS	43	14	29	0	12	41%
CXL-CXL	49	13	36	0	16	44%
CXL-CCN	7	7	0	0	0	-
CCN-TMI	6	4	2	0	0	0%
CCN-TAD	32	32	0	0	0	-
CCN-SMI	3	2	1	0	0	0%
CCN-CXS	6	6	0	0	0	-
CCN-CXL	7	7	0	0	0	-
CCN-CCN	1	0	1	0	0	0%
totals	2980	1519	1429	32	948	65%

Table 10 ACCEL second-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI−TMI	1056	0	1056	0	1056	100%
TMI−TAD	12937	4944	7959	34	6468	81%
TMI−SMI	429	0	429	0	429	100%
TMI−CXS	96	32	64	0	64	100%
TMI−CXL	160	61	99	0	96	97%
TMI−CCN	160	125	35	0	0	0%
TAD−TMI	13329	4936	8359	34	6468	77%
TAD−TAD	153667	97510	55375	782	38416	68%
TAD−SMI	5097	1434	3650	13	2366	65%
TAD−CXS	1177	704	469	4	392	83%
TAD−CXL	1961	1248	707	6	588	82%
TAD−CCN	1961	1709	246	6	0	0%
SMI−TMI	380	0	380	0	380	100%
SMI−TAD	4395	1434	2948	13	2197	74%
SMI−SMI	156	0	156	0	156	100%
SMI−CXS	33	10	23	0	22	96%
SMI−CXL	55	18	37	0	33	89%
SMI−CCN	55	39	16	0	0	0%
CXS−TMI	99	32	67	0	66	99%
CXS−TAD	1177	704	469	4	392	83%
CXS−SMI	39	10	29	0	26	90%
CXS−CXS	9	5	4	0	4	100%
CXS−CXL	15	6	9	0	6	67%
CXS−CCN	15	13	2	0	0	0%
CXL−TMI	165	61	104	0	99	95%
CXL−TAD	1961	1248	707	6	588	82%
CXL−SMI	65	18	47	0	39	83%
CXL−CXS	15	6	9	0	6	67%
CXL−CXL	25	16	9	0	9	100%
CXL−CCN	25	22	3	0	0	0%
CCN−TMI	165	125	40	0	0	0%
CCN−TAD	1961	1709	246	6	0	0%
CCN−SMI	65	39	26	0	0	0%
CCN−CXS	15	13	2	0	0	0%
CCN−CXL	25	22	3	0	0	0%
CCN−CCN	25	20	5	0	0	0%
totals	202970	118273	83789	908	60366	71%

Table 11 HOTEL second-order mutants results

measure / operator	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI-TMI	20	0	20	0	6	30%
TMI-TAD	91	19	66	6	27	38%
TMI-SMI	10	0	10	0	3	30%
TMI-CXS	21	0	21	0	9	43%
TMI-CXL	21	0	21	0	4	19%
TMI-CCN	16	0	16	0	0	0%
TAD-TMI	109	19	84	6	27	30%
TAD-TAD	327	124	169	34	81	40%
TAD-SMI	37	3	32	2	6	18%
TAD-CXS	110	24	78	8	33	38%
TAD-CXL	110	24	78	8	18	21%
TAD-CCN	91	20	65	6	0	0%
SMI-TMI	4	0	4	0	1	25%
SMI-TAD	17	3	12	2	4	29%
SMI-SMI	2	0	2	0	1	50%
SMI-CXS	4	0	4	0	0	0%
SMI-CXL	4	0	4	0	1	25%
SMI-CCN	3	0	3	0	0	0%
CXS-TMI	25	0	25	0	9	36%
CXS-TAD	92	20	65	7	27	38%
CXS-SMI	10	0	10	0	3	30%
CXS-CXS	25	0	25	0	9	36%
CXS-CXL	25	0	25	0	6	24%
CXS-CCN	20	0	20	0	0	0%
CXL-TMI	25	0	25	0	6	24%
CXL-TAD	92	20	65	7	18	25%
CXL-SMI	10	0	10	0	2	20%
CXL-CXS	25	0	25	0	6	24%
CXL-CXL	25	0	25	0	4	16%
CXL-CCN	20	0	20	0	0	0%
CCN-TMI	20	0	20	0	0	0%
CCN-TAD	73	16	52	5	0	0%
CCN-SMI	8	0	8	0	0	0%
CCN-CXS	20	0	20	0	0	0%
CCN-CXL	20	0	20	0	0	0%
CCN-CCN	16	0	16	0	0	0%
totals	1548	292	1165	91	311	25%

Table 12 WFAS second-order mutants results

operator \ measure	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI-TMI	183	98	0	85	12	14%
TMI-TAD	1010	738	6	266	144	53%
TMI-SMI	71	35	0	36	4	11%
TMI-CXS	162	80	0	82	0	0%
TMI-CXL	162	80	0	82	28	34%
TMI-CCN	79	56	0	23	0	0%
TAD-TMI	1082	738	78	266	144	42%
TAD-TAD	5184	3068	2116	0	1296	61%
TAD-SMI	362	221	0	141	30	21%
TAD-CXS	939	609	46	284	36	11%
TAD-CXL	939	330	337	272	288	47%
TAD-CCN	505	446	48	11	0	0%
SMI-TMI	49	21	0	28	2	7%
SMI-TAD	277	180	0	97	25	26%
SMI-SMI	21	8	0	13	0	0%
SMI-CXS	47	20	0	27	0	0%
SMI-CXL	47	20	0	27	6	22%
SMI-CCN	22	14	0	8	0	0%
CXS-TMI	170	84	0	86	0	0%
CXS-TAD	867	581	2	284	0	0%
CXS-SMI	62	28	0	34	0	0%
CXS-CXS	143	68	0	75	0	0%
CXS-CXL	147	60	12	75	0	0%
CXS-CCN	73	50	0	23	0	0%
CXL-TMI	170	84	0	86	32	37%
CXL-TAD	867	316	273	278	288	52%
CXL-SMI	62	28	0	34	8	24%
CXL-CXS	147	60	12	75	0	0%
CXL-CXL	147	36	36	75	64	58%
CXL-CCN	73	50	0	23	0	0%
CCN-TMI	85	60	0	25	0	0%
CCN-TAD	433	427	0	6	0	0%
CCN-SMI	31	18	0	13	0	0%
CCN-CXS	73	50	0	23	0	0%
CCN-CXL	73	50	0	23	0	0%
CCN-CCN	36	30	6	0	0	0%
totals	14800	8842	2972	2986	2407	40%

Table 13 MUTEX second-order mutants results

measure / operator	mutants	non-subsumed	subsumed	inconsistent	violating guidelines	violating / (subsumed + inconsistent)
TMI-TMI	13	0	0	13	12	92%
TMI-TAD	130	20	24	86	64	58%
TMI-SMI	13	0	0	13	12	92%
TMI-CXS	13	0	0	13	12	92%
TMI-CXL	17	3	0	14	3	21%
TMI-CCN	4	1	0	3	0	0%
TAD-TMI	162	19	56	87	64	45%
TAD-TAD	1027	344	576	107	256	37%
TAD-SMI	98	10	0	88	36	41%
TAD-CXS	98	7	0	91	36	40%
TAD-CXL	131	95	0	36	16	44%
TAD-CCN	33	31	0	2	0	0%
SMI-TMI	7	0	0	7	6	86%
SMI-TAD	56	8	0	48	27	56%
SMI-SMI	7	0	0	7	6	86%
SMI-CXS	8	0	0	8	7	88%
SMI-CXL	10	0	0	10	1	10%
SMI-CCN	2	0	0	2	0	0%
CXS-TMI	13	0	0	13	12	92%
CXS-TAD	98	7	0	91	48	53%
CXS-SMI	10	0	0	10	9	90%
CXS-CXS	10	0	0	10	9	90%
CXS-CXL	14	0	3	11	3	21%
CXS-CCN	4	1	0	3	0	0%
CXL-TMI	18	3	0	15	4	27%
CXL-TAD	131	95	0	36	16	44%
CXL-SMI	14	0	0	14	3	21%
CXL-CXS	15	0	4	11	3	20%
CXL-CXL	19	11	0	8	1	12%
CXL-CCN	4	4	0	0	0	-
CCN-TMI	5	1	0	4	0	0%
CCN-TAD	33	31	0	2	0	0%
CCN-SMI	4	0	0	4	0	0%
CCN-CXS	4	1	0	3	0	0%
CCN-CXL	4	4	0	0	0	-
CCN-CCN	1	0	1	0	0	0%
totals	2230	696	664	870	666	43%

Acknowledgements We thank the anonymous reviewers for useful comments and suggestions that helped us to improve the presentation. Davide Basile and Maurice H. ter Beek acknowledge funding from the national MIUR-PRIN 2020TL3X8X project T-LADIES (Typeful Language Adaptation for Dynamic, Interacting and Evolving Systems). Maxime Cordy and Sami Lazreg are supported by FNR Luxembourg (grant INTER/FNRS/20/15077233/Scaling Up Variability/Cordy).

Declarations

Conflict of Interests The authors have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Aichernig BK, Brandl H, Jöbstl E, Krenn W, Schlick R, Tiran S (2015) Killing strategies for model-based mutation testing. Softw Test Verif Reliab 25(8):716–748. https://doi.org/10.1002/stvr.1522

Aichernig BK, Lorber F, Nickovic D (2013) Time for mutants: Model-based mutation testing with timed automata. In: Veanes M, Viganò L (eds) Proceedings of the 7th international conference on tests and proofs (TAP'13), LNCS, vol 7942. Springer, pp 20–38. https://doi.org/10.1007/978-3-642-38916-0_2

Alur R, Dill DL (1994) A theory of timed automata. Theoret Comput Sci 126(2):183–235. https://doi.org/10.1016/0304-3975(94)90010-8

André É, Hasuo I, Waga M (2018) Offline timed pattern matching under uncertainty. In: Proceedings of the 23rd international conference on engineering of complex computer systems (ICECCS'18). IEEE, pp 10–20. https://doi.org/10.1109/ICECCS2018.2018.00010

André É, Knapik M, Lime D, Penczek W, Petrucci L (2019) Parametric verification: an introduction. In: Koutny M, Pomello L, Kristensen LM (eds) Transactions on petri nets and other models of concurrency XIV, LNCS, vol 11790. Springer, pp 64–100. https://doi.org/10.1007/978-3-662-60651-3_3

Andrews JH, Briand LC, Labiche Y, Namin AS (2006) Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans Softw Eng 32(8):608–624. https://doi.org/10.1109/TSE.2006.83

Asarin E, Maler O, Pnueli A, Sifakis J (1998) Controller synthesis for timed automata. IFAC Proc 31(18):447–452. https://doi.org/10.1016/S1474-6670(17)42032-5. Proceedings of the 5th IFAC conference on system structure and control (SSC'98)

Baker R, Habli I (2013) An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Trans Softw Eng 39(6):787–805. https://doi.org/10.1109/TSE.2012.56

Basile D, ter Beek MH, Cordy M, Legay A (2020a) Tackling the equivalent mutant problem in real-time systems: The 12 commandments of model-based mutation testing. In: Proceedings of the 24th ACM conference on systems and software product lines (SPLC'20). ACM, pp 252–262. https://doi.org/10.1145/3382025.3414966

Basile D, ter Beek MH, Degano P, Legay A, Ferrari GL, Gnesi S, Di Giandomenico F (2020b) Controller synthesis of service contracts with variability. Sci Comput Program, 187. https://doi.org/10.1016/j.scico.2019.102344

Basile D, ter Beek MH, Legay A (2020c) Timed service contract automata. Innovations Syst Softw Eng (16), 199–214. https://doi.org/10.1007/s11334-019-00353-3

Behrmann G, Cougnard A, David A, Fleury E, Larsen KG, Lime D (2007) UPPAAL-Tiga: time for playing games! In: Damm W, Hermanns H (eds) Proceedings of the 19th international conference on computer aided verification (CAV'07), LNCS, vol 4590. Springer, pp 121–125. https://doi.org/10.1007/978-3-540-73368-3_14

Benes N, Bezdek P, Larsen KG, Srba J (2015) Language emptiness of continuous-time parametric timed automata. In: Halldórsson MM, Iwama K, Kobayashi N, Speckmann B (eds) Proceedings of the 42nd

international colloquium on automata, languages, and programming (ICALP'15), LNCS, vol 9135. Springer, pp 69–81. https://doi.org/10.1007/978-3-662-47666-6_6

Brillout A, He N, Mazzucchi M, Kroening D, Purandare M, Rümmer P, Weissenbacher G (2009) Mutation-based test case generation for Simulink models. In: De Boer FS, Bonsangue MM, Hallerstede S, Leuschel M (eds) Proceedings of the 8th international symposium on formal methods for components and objects (FMCO'09), LNCS, vol 6286. Springer, pp 208–227. https://doi.org/10.1007/978-3-642-17071-3_11

Chow TS (1978) Testing software design modeled by finite-state machines. IEEE Trans Softw Eng SE-4(3):178–187. https://doi.org/10.1109/TSE.1978.231496

Classen A, Cordy M, Schobbens P, Heymans P, Legay A, Raskin J (2013) Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans Softw Eng 39(8):1069–1089. https://doi.org/10.1109/TSE.2012.86

Cordy M, Classen A, Perrouin G, Schobbens P, Heymans P, Legay A (2012a) Simulation-based abstractions for software product-line model checking. In: Proceedings of the 34th international conference on software engineering (ICSE'12). IEEE, pp 672–682. https://doi.org/10.1109/ICSE.2012.6227150

Cordy M, Schobbens P, Heymans P, Legay A (2012b) Behavioural modelling and verification of real-time software product lines. In: Proceedings of the 16th international software product line conference (SPLC'12). ACM, pp 66–75. https://doi.org/10.1145/2362536.2362549

Cordy M, Legay A, Schobbens P, Traonouez L (2013) A framework for the rigorous design of highly adaptive timed systems. In: Proceedings of the 1st FME workshop on formal methods in software engineering (FormaliSE'13). IEEE, pp 64–70. https://doi.org/10.1109/FormaliSE.2013.6612279

David A, Larsen KG, Legay A, Nyman U, Traonouez L, Wąsowski A (2015) Real-time specifications. Int J Softw Tools Technol Transf 17(1):17–45. https://doi.org/10.1007/s10009-013-0286-x

David A, Larsen KG, Legay A, Nyman U, Wąsowski A. (2010a) Timed I/O automata: a complete specification theory for real-time systems. In: Proceedings of the 13th international conference on hybrid systems: computation and control (HSCC'10). ACM, pp 91–100. https://doi.org/10.1145/1755952.1755967

David A, Larsen KG, Legay A, Nyman U, Wąsowski A (2010b) ECDAR: an environment for compositional design and analysis of real time systems. In: Bouajjani A, Chin WN (eds) Proceedings of the 8th international symposium on automated technology for verification and analysis (ATVA'10), LNCS, vol 6252. Springer, pp 365–370. https://doi.org/10.1007/978-3-642-15643-4_29

DeMillo R, Lipton R, Sayward F (1978) Hints on test data selection: Help for the practicing programmer. IEEE Comp 11(4):34–41. https://doi.org/10.1109/C-M.1978.218136

Devroey X, Perrouin G, Papadakis M, Legay A, Schobbens P, Heymans P (2016) Featured model-based mutation analysis. In: Proceedings of the 38th international conference on software engineering (ICSE'16). ACM, pp 655–666. https://doi.org/10.1145/2884781.2884821

Fabbri S, Maldonado JC, Sugeta T, Masiero PC (1999) Mutation testing applied to validate specifications based on statecharts. In: Proceedings of the 10th international symposium on software reliability engineering (ISSRE'99). IEEE, pp 210–219. https://doi.org/10.1109/ISSRE.1999.809326

Feo-Arenis S, Westphal B, Dietsch D, Muñiz M, Andisha AS (2014) The wireless fire alarm system: ensuring conformance to industrial standards through formal verification. In: Jones C, Pihlajasaari P, Sun J (eds) Proceedings of the 19th international symposium on formal methods (FM'14), LNCS, vol 8442. Springer, pp 658–672. https://doi.org/10.1007/978-3-319-06410-9_44

Hoxha B, Abbas H, Fainekos G (2015) Benchmarks for temporal logic requirements for automotive systems. In: Frehse G, Althoff M (eds) Proceedings of the 1st and 2nd international workshop on applied verification for continuous and hybrid systems (ARCH'14-'15), EPiC Series in Computing, vol 34. EasyChair, pp 25–30. https://doi.org/10.29007/xwrs

Hune T, Romijn J, Stoelinga M, Vaandrager FW (2001) Linear parametric model checking of timed automata. Tech. Rep. CSI-R0102, University of Nijmegen. http://hdl.handle.net/2066/18941

Jia Y, Harman M (2011) An analysis and survey of the development of mutation testing. IEEE Trans Softw Eng 37(5):649–678. https://doi.org/10.1109/TSE.2010.62

Lamport L (1987) A fast mutual exclusion algorithm. ACM Trans Comput Syst 5(1):1–11. https://doi.org/10.1145/7351.7352

Larsen KG, Lorber F, Nielsen B, Nyman U (2017) Mutation-based test-case generation with Ecdar. In: Proceedings of the 10th IEEE international conference on software testing, verification and validation workshops (ICSTW'17). IEEE, pp 319–328. https://doi.org/10.1109/ICSTW.2017.60

Larsen KG, Nyman U, Wąsowski A (2007) Modal I/O automata for interface and product line theories. In: De Nicola R (ed) Proceedings of the 16th european symposium on programming (ESOP'07), LNCS, vol 4421. Springer, pp 64–79. https://doi.org/10.1007/978-3-540-71316-6_6

Lee J, Kang S, Jung P (2020) Test coverage criteria for software product line testing: Systematic literature review. Inf Softw Technol, 122. https://doi.org/10.1016/j.infsof.2020.106272

Luthmann L, Gerecht T, Lochau M (2019) Sampling strategies for product lines with unbounded parametric real-time constraints. Int J Softw Tools Technol Transf 21(6):613–633. https://doi.org/10.1007/s10009-019-00532-4

Luthmann L, Stephan A, Bürdek J, Lochau M (2017) Modeling and testing product lines with unbounded parametric real-time constraints. In: Proceedings of the 21st international systems and software product lines conference (SPLC'17). ACM, pp 104–113. https://doi.org/10.1145/3106195.3106204

Madeyski L, Orzeszyna W, Torkar R, Józala M (2014) Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans Softw Eng 40(1):23–42. https://doi.org/10.1109/TSE.2013.44

Masri W, Zaraket F (2016) Coverage-based software testing: Beyond basic test requirements. In: Memon AM (ed) Advances in computers, vol 103, chap 4. Elsevier, pp 79–142. https://doi.org/10.1016/bs.adcom.2016.04.003

Offutt J (2011) A mutation carol: past, present and future. Inf Softw Technol 53(10):1098–1107. https://doi.org/10.1016/j.infsof.2011.03.007

Papadakis M, Malevris N (2010) An empirical evaluation of the first and second order mutation testing strategies. In: Proceedings of the 3rd international conference on software testing, verification and validation workshops (ICSTW'10), pp 90–99. https://doi.org/10.1109/ICSTW.2010.50

Petrovic G, Ivankovic M, Fraser G, Just R (2021) Does mutation testing improve testing practices? In: 43rd IEEE/ACM international conference on software engineering (ICSE'21). IEEE, pp 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

Ter Beek, Cledou G, Hennicker R, Proença J (2021) Featured team automata. In: Huisman M, Pasareanu CS, Zhan N (eds) Proceedings of the 24th international symposium on formal methods (FM'21), LNCS, vol 13047. Springer, pp 483–502. https://doi.org/10.1007/978-3-030-90870-6_26

Ter Beek MH, Kleijn J (2012) Vector team automata. Theor Comput Sci 429:21–29. https://doi.org/10.1016/j.tcs.2011.12.020

Ter Beek MH, van Loo S, De Vink EP, Willemse TA (2020) Family-based SPL model checking using parity games with variability. In: Wehrheim H, Cabot J (eds) Proceedings of the 23rd international conference on fundamental approaches to software engineering (FASE'20), LNCS, vol 12076. Springer, pp 245–265. https://doi.org/10.1007/978-3-030-45234-6_12

Ter Beek MH, Fantechi A, Gnesi S, Mazzanti F (2016) Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. J Log Algebr Meth Program 85(2):287–315. https://doi.org/10.1016/j.jlamp.2015.11.006

Utting M, Pretschner A, Legeard B (2012) A taxonomy of model-based testing approaches. Softw Test Verif Reliab 22(5):297–312. https://doi.org/10.1002/stvr.456

Waga M, Hasuo I, Suenaga K (2017) Efficient online timed pattern matching by automata-based skipping. In: Abate A, Geeraerts G (eds) Proceedings of the 15th international conference on formal modeling and analysis of timed systems (FORMATS'17), LNCS, vol 10419. Springer, pp 224–243. https://doi.org/10.1007/978-3-319-65765-3_13

Weyuker E, Goradia T, Singh A (1994) Automatically generating test data from a Boolean specification. IEEE Trans Softw Eng 20(5):353–363. https://doi.org/10.1109/32.286420

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Davide Basile is a permanent researcher at ISTI-CNR (Pisa, Italy) and a member of the Formal Methods and Tools lab. He obtained his Ph.D. in Computer Science at the University of Pisa (Italy). He has authored more than 40 papers in the field of formal methods, software engineering and dependable computing. His research focuses on developing both novel formalisms for emerging computational paradigms and supporting tools, exploring formal verification techniques and applying state-of-the-art formal methods and tools to the design of real-world systems and emerging technologies. He has been part of the organization of workshops and conferences (incl., FORTE, COORDINATION, VaMoS), he has been a PC member of various conferences (incl. SPLC, IFM, FMICS, COORDINATION) and he is a regular reviewer for various journals (incl. TSE, TOSEM, TITS, TOCL, STVR, FAOC, JSS, STTT, SCICO).

Maurice H. ter Beek is senior researcher at ISTI–CNR (Pisa, Italy) and head of the Formal Methods and Tools lab. He obtained his Ph.D. at Leiden University (The Netherlands). He has authored over 150 peer-reviewed papers, edited over 30 proceedings and special issues of journals, and serves on the editorial boards of the journals *Formal Aspects of Computing: Applicable Formal Methods*, *International Journal on Software Tools for Technology Transfer*, *Journal of Logical and Algebraic Methods in Programming*, *PeerJ Computer Science*, *Science of Computer Programming*, and *ERCIM News*. He works on formal methods and model-checking tools for the specification and verification of safety-critical software systems, focusing in particular on applications in service computing, software product line engineering and railway systems. He is member of the Steering Committees of the FMICS, SPLC and VaMoS conference series, and regular PC member of the COORDINATION, FM, FMICS, FormaliSE, SEFM, SPLC and VaMoS conference series, among others.

Sami Lazreg is a research associate at the *Interdisciplinary Centre for Security, Reliability and Trust research lab*, University of Luxembourg. He obtained his Ph.D. at University Cote d'Azur, Sophia Antipolis, France, in collaboration with Visteon Electronics, a world class leader in automotive systems. His main topics are model based design and design space exploration of embedded systems. He works on variability modelling and variability-aware simulation and model-checking methods to model, verify and optimize embedded software and system product lines.

Maxime Cordy is a Research Scientist at the Interdisciplinary Center for Security, Reliability and Trust (SnT), University of Luxembourg, in the domain of Software Engineering (SE), with a focus on software verification and testing, security and quality of machine learning, and data-intensive systems. He has published 70+ peer-review papers in these areas. He is one of the four permanent scientists of the SnT's SerVal group (SEcurity, Reasoning and VALidation). His research is inspired from and applies to several industry partners (BGL BNP-Paribas, Lombard Int'l, CREOS, Enovos, etc.). He is deeply engaged in making Society benefit from results and technologies produced by research through the founding of spin-off companies and the leadership of private-public partnership projects at SnT. He is steering committee member and former PC chair of the VAMOS conference and MaLTeSQuE workshop. He co-organized several SE workshops (MASES '18, MALTESQUE '19 '22, BENEVOL '20). He was co-chair of ICSME '21 NIER track. He has worked as a program committee member for various tracks of SE and AI conferences incl. IJCAI, ESEC/FSE, PLDI, ISSTA, CAISE, SAC, SPLC. He is distinguished reviewer board member of TOSEM and regular reviewer for other journals (incl. TSE, TOSEM, STVR, FAOC, JSS, STTT).

Axel Legay is Professor at UC Louvain. He also used to work at Inria as team leader in cyber security. He received his Ph.D. in Computer Science from the University of Lige, Belgium. His main research interests are in formal verification, testing, and cyber security. He is a founder and major contributor of statistical model checking (a statistical variant of model checking effectively used in industry), proveline (product lines analysis), and malware analysis. He wrote more than 300 publications and he is a referee for top journals and conferences in those areas. He wrote several open source tools, and he has been institution PI for more than 30 projects.

Affiliations

Davide Basile[1] · Maurice H. ter Beek[1] · Sami Lazreg[2] · Maxime Cordy[2] · Axel Legay[3]

Maurice H. ter Beek
maurice.terbeek@isti.cnr.it

Sami Lazreg
sami.lazreg@uni.lu

Maxime Cordy
maxime.cordy@uni.lu

Axel Legay
axel.legay@uclouvain.be

[1] ISTI–CNR, Via G. Moruzzi 1, 56124 Pisa, Italy

[2] SnT, University of Luxembourg, Esch-sur-Alzette, Luxembourg

[3] Université Catholique de Louvain, Ottignies-Louvain-la-Neuve, Belgium