MUPPAAL: Reducing and Removing Equivalent and Duplicate Mutants in UPPAAL

Jaime Cuartas
Universidad del Valle
Cali, Colombia
jaime.cuartas@correounivalle.edu.co

Jesús Aranda
Universidad del Valle
Cali, Colombia
jesus.aranda@correounivalle.edu.co

Maxime Cordy
SnT, University of Luxembourg
Luxembourg, Luxembourg
maxime.cordy@uni.lu

James Ortiz
FOCUS/NaDI, University of Namur
Namur, Belgium
james.ortizvega@unamur.be

Gilles Perrouin
PReCISE/NaDI, University of Namur
Namur, Belgium
gilles.perrouin@unamur.be

Pierre-Yves Schobbens
PReCISE/NaDI, University of Namur
Namur, Belgium
pierre-yves.schobbens@unamur.be

***Abstract*—Mutation Testing (MT) is a test quality assessment technique that creates mutants by injecting artificial faults into the system and evaluating the ability of tests to distinguish these mutants. We focus on MT for safety-critical Timed Automata (TA). MT is prone to equivalent and duplicate mutants, the former having the same behaviour as the original system and the latter other mutants. Such mutants bring no value and induce useless test case executions. We propose MUPPAAL, a tool that: (1) offers a new operator reducing the occurrence of mutant duplicates; (2) an efficient bisimulation algorithm removing remaining duplicates; (3) leverages existing equivalence-avoiding mutation operators. Our experiments on four UPPAAL case studies indicate that duplicates represent up to 32% of all mutants and that the MUPPAAL bisimulation algorithm can identify them more than 99% of the time.**

***Index Terms*—Model-Based Testing, Timed Automata, Mutation Testing, UPPAAL**

I. Introduction

Timed systems (TS) are systems in which strict time constraints are essential for reliability and correctness. They appear in planes, trains, and a variety of safety-critical systems. Ensuring Quality Assurance (QA) of TS is essential. Model-Based Testing (MBT) [41], [44] exploits (timed) specifications to generate test cases assessing the system's behaviour and avoids scalability issues induced by exhaustive verification. Yet, one must ensure its ability to find bugs. Mutation Testing (MT) [24] creates *mutants* of the system by injecting artificial defects via predefined *mutation operators*. Tests can then *distinguish* (or *kill*) mutants if they behave differently on the mutant than on the original system. The *mutation score* is the ratio of killed mutants to the total number of mutants. Though MT has long focused on code [38], [16], [34], Model-Based Mutation Testing (MBMT) helps in the automatic identification of defects related to missing functionality and misinterpreted specifications [9] that are difficult to identify via code-based testing [22], [43]. Yet, not all mutants are relevant. Some may be *equivalent*, i.e., they exhibit the same behaviour as the original system despite their syntactic difference [37]. Therefore, no test case can distinguish such mutants. Similarly, *duplicate* mutants exhibit the same behaviour as other mutants [36], [37]. Preventing and removing such *useless* mutants reduces the computation costs of (MB)MT and builds more trust in mutation scores. Recently, Basile *et al.* tackled the equivalent mutant problem for Timed Automata with Input and Output (TAIO): they defined mutation operators preventing mutants from refining the original system [6], [5]. However, this technique does not address duplicate mutants: in our experiments, up to 32% of all generated mutants were duplicates. We propose MUPPAAL, a mutation approach that addresses this challenge for Timed Automata (TA) specified in UPPAAL [7]. The contributions of this paper are the following:

1) We introduce a *novel timed mutation operator*, SMI-NR, that we *proved to prevent duplicate mutants* by design;
2) We *detect duplicate mutants* using a timed bisimulation algorithm [35] to assess behavioural equivalence between two mutants. When duplicate mutants are detected, we keep only one of them;
3) We provide a random simulation baseline to compare to timed bisimulation: If, for a mutant, we can find a trace that the other mutant cannot execute, then we can conclude that the two mutants are not duplicates. The heuristic suggests a pair of mutants as duplicates otherwise;
4) We implemented MUPPAAL using the UPPAAL execution engine for TAs and UPPAAL-TRON [30] for timed trace generation and checking. In addition to the above contributions, MUPPAAL supports mutation operators for TA from [1], [6], [35] avoiding equivalent mutants using refinement prevention [5], [6];
5) We assessed MUPPAAL[1]. Our results on four cases indicate that timed bisimulation offers the *best trade-off* between performance and accuracy of detection. In contrast, the random baseline suggests many false duplicates (up to 10 times compared to bisimulation). Our

[1]MUPPAAL implementation and full results of its evaluation are available: https://anonymous.4open.science/r/Muppaal-91A7

novel mutation operator effectively reduces the number of mutants while perfectly capturing the initial mutation operator behaviour.

The remainder of this paper is as follows. Section II introduces the formalisms we use and the equivalent and duplicate mutant problem. Section III presents our new mutation operator and duplicate removal algorithms. Section IV describes MUPPAAL and reports on our experiments. Section V presents related work, and Section VI wraps up with concluding remarks and future work.

II. Background

A. Clocks and Timed Automata

To model the continuous time domain, we use non-negative real-valued variables: *clocks*. Clocks are variables that increase at the same rate (i.e., synchronously). TA are one of the most studied formalisms for modelling TS [3]. Several model checkers such as UPPAAL [7], KRONOS [8], and HYTECH [19] rely on TA. TA are an extension of Finite State Automata (FSA) with a set of clocks increasing at the same rate. Resetting TA clock means updating the clock value to zero. TA allows enabling or disabling transitions using *clock constraints*, and we take transition actions if all other conditions are satisfied. We use an extension of TA called Timed Automata with Inputs and Outputs (TAIO) [2]. TAIO partition the actions into two disjoint sets for inputs and outputs [12][2]. Here, we use the extension of TAIO proposed by Aichernig *et al.* [2]

B. Timed Automata with Inputs and Outputs

A TAIO is a refined TA where we model the interaction between a system and its environment by using output and input actions [2]. The *clock constraints* are defined below.

Definition 1 (Clock constraints)**.** *Let X be a finite set of clock variables ranging over $\mathbb{R}_{\geq 0}$ (non-negative real numbers). Let $\Phi(X)$ be a set of clock constraints over X. A clock constraint $\phi \in \Phi(X)$ can be defined by the following grammar:*

$$\phi ::= true \mid x \sim c \mid \phi_1 \ \wedge \ \phi_2$$

where $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, >, \leq, \geq, =\}$.

Definition 2 (Clock Invariants)**.** *Let X be a finite set of clock variables ranging over $\mathbb{R}_{\geq 0}$. Let $\Delta(X)$ be a set of clock invariants over X. Clocks invariants are clock constraints of the following form:*

$$\delta ::= true \mid x < c \mid x \leq c \mid \phi_1 \ \wedge \ \phi_2$$

where $x \in X$, $c \in \mathbb{N}$.

Definition 3 (Clock valuations)**.** *Given a finite set of clocks X, a clock valuation function, $\nu : X \to \mathbb{R}_{\geq 0}$ assigning to each clock $x \in X$ a non-negative value $\nu(x)$. We denote $\mathbb{R}^X_{\geq 0}$ the set of all valuations. For a clock valuation $\nu \in \mathbb{R}^X_{\geq 0}$ and a time value $d \in \mathbb{R}_{\geq 0}$, $\nu + d$ is the valuation satisfied by $(\nu + d)(x) = \nu(x) + d$ for each $x \in X$. Given a clock subset $Y \subseteq X$, we denote $\nu[Y \leftarrow 0]$ the valuation defined as follows: $\nu[Y \leftarrow 0](x) = 0$ if $x \in Y$ and $\nu[Y \leftarrow 0](x) = \nu(x)$ otherwise.*

In TAIO, the transitions can have a *guard* that will allow the transitions to be taken or not, performing actions and resetting clocks. In TAIO, one classifies actions (or alphabet) into two disjoint subsets: input actions (suffixed with ?) and output actions (suffixed with !) [2]. The output actions of a TAIO $\mathcal{A}$ can be input actions of a TAIO $\mathcal{B}$. We adapt the definition of [26], where the initial location is unique, and discrete variables and internal actions are not allowed. We formally define TAIO as:

Definition 4 (TAIO)**.** *A TAIO is a tuple $(L, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T, I)$, where:*

- *L is a finite set of locations,*
- *$l_0 \in L$ is an initial location,*
- *X is a finite set of clocks,*
- *Σ_I is a finite set of input actions (?),*
- *Σ_O is a finite set of output actions (!),*
- *$\Sigma = \Sigma_I \cup \Sigma_O$, is a finite set of input and output actions, such that $\Sigma_I \cap \Sigma_O = \emptyset$,*
- *$T \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a finite set of transitions,*
- *$I : L \to \Delta(X)$ is a function that associates to each location a clock invariant.*

For a transition $(l, a, \phi, Y, l') \in T$, we classically write $l \xrightarrow{a,\phi,Y} l'$ and call l and l' the source and target location, ϕ is the guard, a the action (or alphabet), Y the set of clocks to reset. The semantics of a TAIO is a Timed Input/Output Transition System (TIOTS) where a *state* is a pair $(l, \nu) \in L \times \mathbb{R}^X_{\geq 0}$, where l denotes the current location with its accompanying clock valuation ν, starting at (l_0, ν_0) where ν_0 maps each clock to 0. The transitions can be of types: Delay transitions only let time pass without changing location. We only consider *legal* states, i.e. states satisfying the current state invariant $\nu \models$ Discrete transitions occur instead between a source and a target location. The transition can only occur if the current clock values satisfy both the guard of the transition and the invariant of the target location.

Definition 5 (Semantics of TAIO)**.** *Let $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T, I)$ be a TAIO. The semantics of TAIO $\mathcal{A}$ is given by a TIOTS$(\mathcal{A}) = (S, s_0, \Sigma_I, \Sigma_O, \rightarrow)$ where:*

- *$S \subseteq L \times \mathbb{R}^X_{\geq 0}$ is a set of states,*
- *$s_0 = (l_0, \nu_0)$ with $\nu_0(x) = 0$ for all $x \in X$ and $\nu_0 \models I(l_0)$,*
- *$\Sigma_\Delta = \Sigma \uplus \mathbb{R}_{\geq 0}$,*
- *$\rightarrow \subseteq S \times \Sigma_\Delta \times S$ is a transition relation defined by the following two rules:*
  - ***Discrete transition:*** *$(l, \nu) \xrightarrow{a} (l', \nu')$, for $a \in \Sigma$ iff $l \xrightarrow{a,\phi,Y} l'$, $\nu \models \phi$, $\nu' = \nu[Y \leftarrow 0]$ and $\nu' \models I(l')$ and,*
  - ***Delay transition:*** *$(l, \nu) \xrightarrow{d} (l, \nu + d)$, for some $d \in \mathbb{R}_{\geq 0}$ iff $\nu + d \models I(l)$.*

A *path* in TIOTS$(\mathcal{A})$ is a finite sequence of consecutive delays and discrete transitions. A finite execution frag-

ment of $\mathcal{A}$ is a path in TIOTS($\mathcal{A}$) starting from the initial state $s_0 = (l_0, \nu_0)$, with delay and discrete transitions alternating along the path: $\rho = (l_0, \nu_0) \xrightarrow{d_0} (l_0, \nu_0') \xrightarrow{a_0} (l_1, \nu_1) \dots (l_{n-2}, \nu_{n-2}') \xrightarrow{a_{n-1}} (l_{n-1}, \nu_{n-1} \xrightarrow{d_n} (l_n, \nu_n)$ where $\nu_0(x) = 0$ for every $x \in X$. A *path* of TIOTS($\mathcal{A}$) is *initial* if $s_0 = (l_0, \nu_0) \in S$, where $l_0 \in L$, ν_0 assign 0 to each clock, and *maximal* if it ends in a location without outgoing edges.

A *timed trace* [3] over Σ is a finite sequence $\theta = ((\sigma_1, t_1), (\sigma_2, t_2) \dots (\sigma_n, t_n))$ of actions paired with non-negative real numbers (i.e., $(\sigma_i, t_i) \in \Sigma \times \mathbb{R}_{\geq 0}$) such that the timestamped sequence $t = t_1 \cdot t_2 \cdots t_n$ is non-decreasing (i.e., $t_i \leq t_{i+1}$).

Example 1. *Let $\mathcal{A}$ be the TAIO depicted in Fig 1. $\mathcal{A}$ contains two locations: l_0 (initial) and l_1. We denote input actions ($?$) and output actions ($!$). In particular, l_0 is the only location to define an invariant not trivially true: $I(l_0) = (x < 7)$, forcing the TAIO to exit l_0 before x becomes 7. Location l_1 has a true invariant (thus not drawn), allowing it to stay in l_1 forever. Suppose the current location is l_1. The transition $l_1 \xrightarrow{b?,(y=9),\{x:=0;y:=0\}} l_0$ specifies that when the input action $b?$ occurs and the guard $y = 9$ holds, this enables the transition, leading to a new current location l_0, while resetting clock variables x and y. Note that using a location invariant (which specifies the time limit to stay in a given location) differs from using a guard (specifying when the transition is enabled). The automaton in Fig 1 is nondeterministic because location l_1 has two outgoing transitions on the same input action ($b?$).*

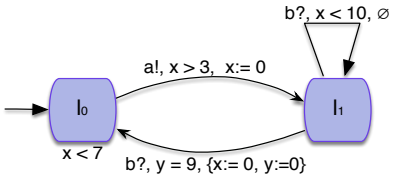

Fig. 1: A nondeterministic TAIO with two clocks x and y.

Definition 6 (Deterministic TAIO)**.** *A deterministic TAIO is a tuple $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T, I)$ such that: for every $l \in L$, for all actions $a \in \Sigma$, for every pair of different edges of the form $(l, a, \phi_1, Y_1, l_1') \in T$ and $(l, a, \phi_2, Y_2, l_2') \in T$, imply $\phi_1 \cap \phi_2 = \emptyset$ and $l_1' = l_2'$. (2) For every $l \in L$, for all actions $a \in \Sigma$, and every valuation ν there is an edge (l, a, ϕ, Y_1, l') such that $\nu \models \phi$.*

C. Timed Bisimulation and Trace Simulation

1) Traces: Simulation is a widely used technique to test software systems. However, *simulation* is insufficient to prove the absence of errors in safety-critical systems because of its non-exhaustiveness.

A *simulation trace* is a *timed trace* collected during the simulation execution.

2) Timed Bisimulation: To reason about the behavioural equivalence of mutants, we use the classical notion of timed bisimulation [10].

Definition 7 (**Timed Bisimulation** [10])**.** *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two TIOTS over the set of actions $\Sigma = (\Sigma_I \cup \Sigma_O)$. Let $S_{\mathcal{D}_1}$ (resp., $S_{\mathcal{D}_2}$) be the set of states of $\mathcal{D}_1$ (resp., $\mathcal{D}_2$). A timed bisimulation over TIOTS $\mathcal{D}_1$, $\mathcal{D}_2$ is a binary relation $\mathcal{R} \subseteq S_{\mathcal{D}_1} \times S_{\mathcal{D}_2}$ such that, for all $s_{\mathcal{D}_1} \mathcal{R}\ s_{\mathcal{D}_2}$, the following holds:*

1) *For every discrete transition $s_{\mathcal{D}_1} \xrightarrow{a}_{\mathcal{D}_1} s'_{\mathcal{D}_1}$ with $a \in \Sigma$, there exists a matching transition $s_{\mathcal{D}_2} \xrightarrow{a}_{\mathcal{D}_2} s'_{\mathcal{D}_2}$ such that $s'_{\mathcal{D}_1} \mathcal{R} s'_{\mathcal{D}_2}$ and symmetrically.*
2) *For every delay transition $s_{\mathcal{D}_1} \xrightarrow{d}_{\mathcal{D}_1} s'_{\mathcal{D}_1}$ with $d \in \mathbb{R}_{\geq 0}$, there exists a matching transition $s_{\mathcal{D}_2} \xrightarrow{d}_{\mathcal{D}_2} s'_{\mathcal{D}_2}$ such that $s'_{\mathcal{D}_1} \mathcal{R} s'_{\mathcal{D}_2}$ and symmetrically.*

$\mathcal{D}_1$ and $\mathcal{D}_2$ are timed bisimilar, written $\mathcal{D}_1 \sim \mathcal{D}_2$, if there exists a timed bisimulation relation $\mathcal{R}$ over $\mathcal{D}_1$ and $\mathcal{D}_2$ containing the pair of initial states.

D. Mutation Operators and Equivalence Problem

1) TA and Mutation Operators.: Nilsson *et al.* [33] were among the first to extend TA with a task model. A task model consists of a set of n (real-time) tasks, and they give each task a period T_i, a worst-case execution time C_i, and a relative deadline D_i and mutation operators. Nilsson *et al.* proposed six mutation operators: *execution time* (ET) affects the execution time of a task; *hold time shift* (HTS) and *lock/unlock time* (LUT) operators either shift the whole lock/unlock time interval for a resource or only one of its bounds; *precedence constraints* (PC) operators change precedence relations between pairs of tasks. The authors also define automata operators that affect both invariant and guard constraints either for a given location (*inter-arrival time* (IAT)) or for the initial location (*pattern offset* (PO)). Aboutrab *et al.* [21] and Aichernig *et al.* [2] also proposed some mutation operators for UPPAAL to test the behaviour of TS. Three of them are not time-related: *change action* (CA), *change source* (CS)/*target* (CT), and *sink location* (SL). The time-related operators are: *change guard* (CG) alters the inequality within the guard constraint, *negate guard* (NG) operator replaces a transition's Boolean guard by its logical negation, *invert reset* (IR) selects one clock variable and either adds it to the list of clocks to be reset during the transition if it is absent or removes it from the list if it is present, *change invariant* (CI) adds one time-unit to the invariant constraint in an automaton location. Basile *et al.* [6] proposed six mutation operators on TAIO, designed to avoid the generation of mutants subsumed by construction. Three of six mutation operators in [6] are time-independent: *Transition MIssing* (TMI) removes a transition; *Transition Add* (TAD) adds a transition between two locations; *State MIssing* (SMI) removes an arbitrary location (also called *state*) other than the initial location and all its incoming/outgoing transitions. The other three time-related operators are: *Constant eXchange Larger* (CXL) increases the constant of a clock constraint, *Constant eXchange Smaller* (CXS) decreases the constant of a clock constraint and *Clock Constraint Negation* (CCN) negates a clock constraint. The main idea in [6] is to perform a refinement check between

Nilsson *et al.* [33]		Aichernig *et al.* [2]		Basile *et al.* [6]	
Op	**Description**	**Op**	**Description**	**Op**	**Description**
ET	Execution time	**CA**	Change action	**TMI**	Transition missing
IAT	Inter-arrival time	**CT**	Change target	**TAD**	Transition ADd
PO	Pattern offset	**CS**	Change source	**SMI**	State missing
LT	Lock time	**CG**	Change guard	**CXL**	Constant exchange L
UT	Unlock time	**NG**	Negate guard	**CXS**	Constant exchange S
HTS	Hold time shift	**CI**	Change invariant	**CCN**	Constraint negation
PC	Precedence constraints	**SL**	Sink location	-	-
-	-	**IR**	Invert reset	-	-

TABLE I: Mutation operators for TA.

the mutant and the system model, using ECDAR [29]. Table I shows the mutation operators retrieved from the considered contributions.

2) Equivalent/Duplicate mutation problem.: MT is one of the most effective coverage criteria to evaluate test suite quality [24], [37]. In addition, several recent empirical studies have evaluated the effectiveness and efficiency of MT [36], [25], [37]. However, MT has a high cost. Equivalent and duplicate mutants (i.e., useless mutants [36]) contribute to increasing costs [37]: between 30% - 40% of equivalent mutants [32] and between 20% - 30% of duplicate mutants [37]. MUPPAAL decrease MBMT costs by eliminating useless mutants from the analysis.

E. UPPAAL and UPPAAL-TRON

UPPAAL is a tool for the modelling, simulation, and verification of networks of TA extended with data types, user functions, clocks, and synchronous communication channels [7]. UPPAAL-TRON [17] is a testing tool, based on UPPAAL, suited for online black-box conformance testing of TS. UPPAAL-TRON is used for testing the Implementation Under Test (IUT). UPPAAL-TRON can use a randomized online testing algorithm, an extension of the UPPAAL model checker [7]. UPPAAL-TRON can generate and execute tests event by event in real-time by stimulating and monitoring the IUT. UPPAAL-TRON performs these two operations, computing the possible set of symbolic states based on the *timed trace* observed so far. A *timed trace* in UPPAAL-TRON consists of a sequence of input or output actions and time delays [30].

III. Overcoming Equivalent and Duplicate Mutants Problem

Three strategies target the equivalent (and duplicate) mutant problem [32] : (1) avoid (2) detect, and (3) suggest equivalent (and duplicate) mutants. We describe in this section how MUPPAAL implements them.

A. Avoiding Equivalent and Duplicate Mutants

Mutation operator design is essential for an effective MBMT tool. It must generate as few mutants as possible without losing efficiency, *i.e.*, avoiding useless mutants. However, most MBMT tools [33], [21], [1] do not avoid the generation of useless mutants. Basile *et al.* [6] avoid equivalent mutants by proposing operators guaranteeing mutants do not refine the original system's behaviour. We have implemented them for UPPAAL. In addition, we offer a new duplicate-avoiding mutation operator.

1) Mutation Operators and Non-Equivalent Mutants.: Here, we use the guidelines and the six mutation operators of [6] (see Table I). We rely on them to avoid equivalent mutants.

2) A new duplicate-avoiding Mutation Operator.: A duplicate mutant has the same behaviour as another mutant and is thus useless. Basile *et al.* refinement technique ensures non-equivalence [6], [5] but does not avoid mutant duplicates. Hence, we introduce a new mutation operator (SMI-NR), avoiding first-order duplicate mutants between SMI and TMI.

Example 2. *Fig. 2 illustrates a base system modelled as a non-duplicate TAIO. Applying TMI, i.e. removing the second transition* $(l_1, a?, true, \emptyset, l_2)$ *gives Fig. 3, while applying SMI, i.e. removing location* l_2*, gives Fig. 4. Both have the same behaviour: they are thus mutant duplicates.*

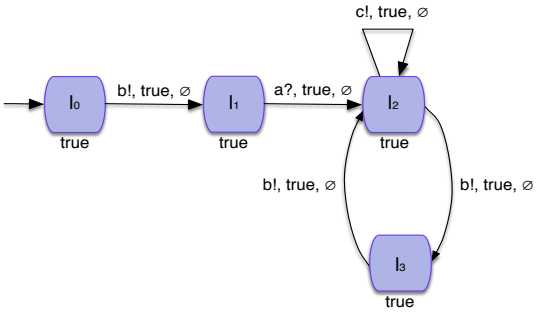

Fig. 2: An original model (TAIO).

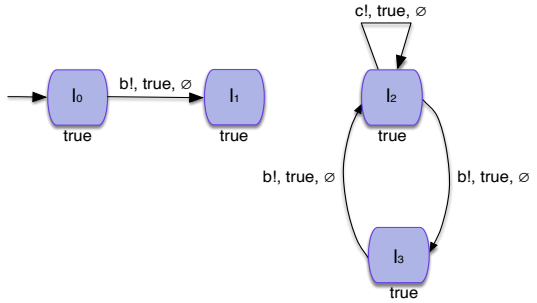

Fig. 3: A mutant generated by TMI operator from Figure 2.

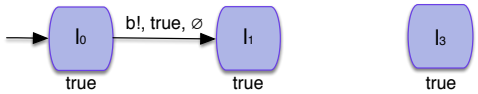

Fig. 4: A mutant generated by SMI operator from Figure 2.

To formally specify SMI-NR, we first note that a mutation operator is a function $\mathcal{M}_\mu$ that generates a set of mutants from a TAIO. We use μ to refer to each specific operator presented in [6]. The following theorem and proposition consider the case of a TMI mutant and a SMI mutant being timed bisimilar.

Theorem 1 (TMI and SMI duplicate mutants)**.** *Let $\mathcal{A}$ be a TAIO and the mutants $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ where:*

$$\mathcal{A}_{tmi} \in \mathcal{M}_{tmi}(\mathcal{A}) \text{ such that } \mathcal{A}_{tmi} = (L, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T \setminus \{t_{tmi}\}, I),\ t_{tmi} = (l_1, a_{tmi}, \phi, Y, l_2) \in T \text{ and } a_{tmi} \in \Sigma_I, \text{ and } \mathcal{A}_{smi} \in \mathcal{M}_{smi}(\mathcal{A}) \text{ such that } \mathcal{A}_{smi} = (L \setminus \{l_{smi}\}, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T_{smi}, I),\ l_{smi} \in L, l_{smi} \neq l_0, T_{smi} = \{(l_1, a, \phi, Y, l_2) \in T \mid l_{smi} \neq l_1 \text{ and } l_{smi} \neq l_2\}$$

Then, $\mathcal{A}_{tmi} \sim \mathcal{A}_{smi}$ iff every initial and finite execution fragment of $\mathcal{A}$ ending in the location $l_{smi} \in L$, takes the same discrete transition, with the same transition $t_{tmi} = (l, a, \phi, Y, l_{smi}) \in T$ for some occurrence of the location l_{smi}.

Verifying the condition in Theorem 1 to prevent that TMI and SMI induce duplicates is costly. Therefore, we define a relaxed condition in Proposition 1 (see proofs in companion website) permitting to avoid some duplicate mutants using a breadth-first search algorithm in polynomial time.

Proposition 1 (Duplicate mutants)**.** *Let $\mathcal{A}$ be a TAIO and the mutants $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ where:*

$$\mathcal{A}_{tmi} \in \mathcal{M}_{tmi}(\mathcal{A}) \text{ such that } \mathcal{A}_{tmi} = (L, l_0, X, \Sigma_I, \Sigma_O, T \setminus \{t_{tmi}\}, I),\ t_{tmi} = (l_1, a_{tmi}, \phi, Y, l_2) \in T \text{ and } a_{tmi} \in \Sigma_I, \text{ and } \mathcal{A}_{smi} \in \mathcal{M}_{smi}(\mathcal{A}) \text{ such that } \mathcal{A}_{smi} = (L \setminus \{l_{smi}\}, l_0, X, \Sigma_I, \Sigma_O, T_{smi}, I),\ l_{smi} \in L, l_{smi} \neq l_0, T_{smi} = \{(l_1, a, \phi, Y, l_2) \in T \mid l_{smi} \neq l_1 \text{ and } l_{smi} \neq l_2\}$$

If every possible initial finite execution fragment of $\mathcal{A}$ ending in location l_{smi} has the same previous location $l'_{smi} \neq l_{smi}$ for some l_{smi} occurrence and l'_{smi} only has one edge $t_{tmi} = (l'_{smi}, a, \phi, Y, l_{smi})$ to l_{smi}, then $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ are duplicates.

Since Proposition 1 is not a sufficient condition, we cannot prevent some duplicates with this condition. We depict in Fig 5 and 6 examples where Proposition 1 cannot prevent duplicates.

B. Detecting Duplicate Mutants

Mutant duplicates are a well-known issue in mutation testing: empirical studies report that between 20% and 30% of all generated mutants are duplicates [36], [32], which affects mutation testing effectiveness [28]. In addition, to be computationally tractable, the SMI-NR is incomplete. Therefore, we present an approach to detect and remove duplicate mutants after mutant generation by using a *timed bisimulation algorithm* [10], [35]. MUPPAAL uses Ortiz *et al.*' timed bisimulation algorithm [35]. Because timed bisimulation's complexity is

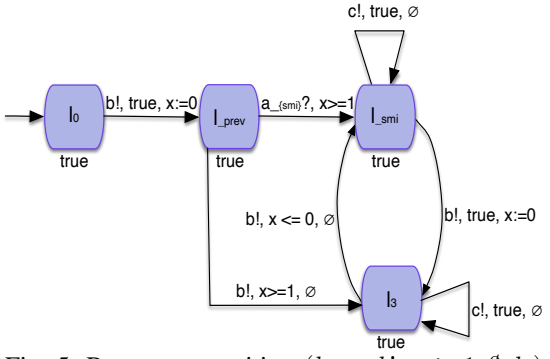

Fig. 5: Removes transition $(l_{prev}, b!, x \geq 1, \emptyset, l_3)$.

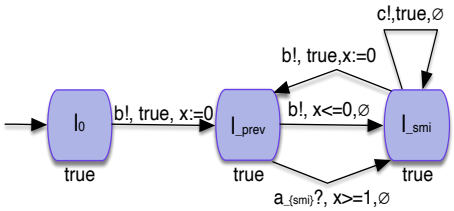

Fig. 6: Removes location l_3 from Fig 5.

EXPTIME [10], if the bisimulation process takes longer than the specified time to analyse a pair of mutants, our algorithm will stop the bisimulation process.

Algorithm 1 describes how MUPPAAL detects duplicates using timed bisimulation [35]. It works as follows. At the first iteration, it checks if there is no timeout involved when comparing the pair of two mutants and if these mutants are bisimilar using the BisimilarAlgo (line 9). Then, it updates $\mathcal{MU}^b$ with one of the two bisimilar mutants (line 11). For each pair of mutants in $\mathcal{LM}$, it assesses whether the two mutants are bisimilar and updates $\mathcal{MU}^b$ accordingly. Otherwise, if there is a timeout involved in the comparison of the pair of two mutants (line 19), then it updates $\mathcal{MU}^{tm}$ (line 14). At the second iteration, it checks if the pair of the two following mutants are bisimilar, up to the n_{th} iteration. Finally, algorithm 1 returns a pair with two sets, where $\mathcal{MU}$ is a set of non-duplicate mutants and $\mathcal{MU}^{tm}$ is a set of mutants whose analysis ended with a timeout.

C. Suggesting Duplicate Mutants

As stated before, timed bisimulation is computationally costly (EXPTIME [10]). To assess this complexity in practice, we present a baseline approach based on a simulation that *suggests* (it is not exact) mutants as potential duplicates. MUPPAAL uses the tools UPPAAL and UPPAAL-TRON to automatically suggest duplicate mutants.

1) Random Simulation.: To suggest duplicates, we take a pair of mutants, generate a random set of traces from one of the two mutants and run them on the other mutant model and reciprocally [13]. We check whether the mutants accept these traces (i.e., whether the mutants can simulate the actions and delays). If a simulation trace fails to run on one of the models, we deduce that the mutants cannot be bisimilar. However, if

```
1  Input: A list of mutants LM and a set of
       mutants MU
2  Output: A pair with two sets (no duplicate and
       timeout mutants ended).
3  MU^b ={}; A set of bisimilar mutants
4  MU^tm ={}; A set of mutants ended with timeout
5  for(i=0; i<LM.size()-1; i++){
6   for(j=i+1; j<LM.size(); j++){
7    //Execute timed bisimulation algorithm
8       if((!timeout()) && (BisimilarAlgo(LM[i],
       LM[j]))){
9         // Add any of the two bisimilar mutants
10        MU^b.add(LM[i]);
11         }else if(timeout()){
12        // Add the i-th mutant ended with timeout
13         MU^tm.add(LM[i],LM[j]);
14         }else{
15          skip; }
16   }
17 }
18 MU = MU - MU^b - MU^tm;
19 return pair(MU, MU^tm);
```

Algorithm 1: Bisimulation Process.

```
1  Input: A mutant M_1 ∈ MU^tm, number of traces to
       generate N, and a simulation time k
2  Output: An array with UPPAAL-TRON traces
3  T=[N];   // The set of N UPPAAL traces
4  T'=[N];  // The set of N UPPAAL-TRON traces
5  for(i=0;i<N; i++){
6      //The seed for the pseudo-random generator
7      r = random();
8      //Get random trace from UPPAAL
9      T[i]=VerifyTA(M_1, k, r);
10     //Translate trace to UPPAAL-TRON format
11     tree=parser(lexer(T[i]));
12     T'[i] = tree.format();
13 }
14 return T';
```

Algorithm 2: Trace generation.

```
1  Input: A set of mutants MU^tm, a number of traces
       to generate N and a simulation time k
2  Output: A set of not-bisimilar mutants
3  // The set of N UPPAAL-TRON traces per mutant
4  T'=[MU^tm.size()][N];
5  for(i=0; i<MU^tm.size(); i++){
6      T'[i]=TraceGeneration(MU^tm[i], N, k);
7      }
8  NBM = [()] // List of no bisimilar mutants
9  for(i=0; i<MU^tm.size()-1; i++){
10     for(j=i+1; j<MU^tm.size(); j++){
11         for(k=0; k<N; k++){
12             Pass1=Tron.check(MU^tm[i], T'[j,k]);
13             Pass2=Tron.check(MU^tm[j], T'[i,k]);
14             if(!(Pass1∧Pass2))
15               NBM=NBM.add((MU^tm[i]));
16         }
17     }
18 }
19 return NBM;
```

Algorithm 3: Random Trace Simulation Algorithm.

all simulation traces are accepted, we consider mutants as probably bisimilar (i.e., we cannot guarantee the existence of the bisimilarity relation).

We use the query *simulate [<=**k**; **N**] 1* using UPPAAL to get traces which simulate **k** units of time and getting **N** traces. Then, we use UPPAAL-TRON to check the validity of the traces of one mutant into the other [17]. To perform trace simulations on UPPAAL-TRON, our translator tool uses ANTLR [39] to parse and translate the traces from UPPAAL into a *preamble* file and a *trace* file. UPPAAL-TRON needs these two files to monitor an execution: (1) the *preamble* file provides the required definitions to configure and prepare UPPAAL-TRON for test execution of the trace, and (2) the *trace* file, which is a sequence of actions and delays to check if the model can execute. Given two mutants, *Automaton* $\mathcal{A}$ and *Automaton* $\mathcal{B}$, our tool proceeds as follows (for **N** random traces): (1) it takes the *Automaton A* to generate traces using UPPAAL. Then the tool reads them and parses trace t_i, builds the preamble, and per each t_i it makes a t_i' file with the UPPAAL-TRON format. (2) once traces are created, the tool uses UPPAAL-TRON to check if the *Automaton* $\mathcal{B}$ can execute the trace, returning as output Passed or Failed. Hence, all pairs not accepting random traces are not bisimilar.

2) Random Simulation Algorithms.: Algorithm 2 describes the random trace generation process of MUPPAAL. The algorithm works as follows: at the first iteration, it computes the N random traces for the first mutant present in $\mathcal{MU}^{tm}$ (line 7) with a simulation time k and a random number generator r (line 9). In addition, we translate the random traces generated from UPPAAL format to UPPAAL-TRON format (lines 13-14). The algorithm repeats until it produces N random traces for all mutants in $\mathcal{MU}^{tm}$ with a simulation time k and a random number generator r.

Algorithm 3 describes the random trace simulation process and uses Algorithm 2 to compute their UPPAAL-TRON traces. The algorithm works as follows. At the first iteration, it checks the N random traces generated by the second mutant $\mathcal{MU}^{tm}[j]$ on the first mutant $\mathcal{MU}^{tm}[i]$ (lines 10-12). In addition, if a pair of mutants are not bisimilar, $\mathcal{PBM}$ is updated with the ith mutants (line 16). At the second iteration, it checks the N random traces generated by the other two mutants in $\mathcal{MU}^{tm}$ and so on up to the n_{th} iteration.

IV. Evaluation

A. MUPPAAL Tool

MUPPAAL automates the whole mutation testing process on top of the UPPAAL verification tools. The tool is written in Java 8 and supports all the operators proposed by Basile *et al.* [6] plus the SMI-NR operator and is easily extendable to new ones. It uses the ANTLR library to parse the model and generate syntactically correct and non-equivalent mutants (thanks to the operators). It then proceeds to duplicate mutants analysis. MUPPAAL is available on our companion website.

B. Case Studies

Our studies stem from UPPAAL specifications of these cases and are available at https://github.com/farkasrebus/XtaBenchmarkSuite. For each case study, we consider the biggest and principal automaton (or process in UPPAAL) from the automata network.

Gear Control (GC). The GC models a simple gear controller for vehicles [31]. The GC model contains 24 states, of which 10 have invariants. All invariants are of the form $x \leq c$ for a clock x and constant c. There are 30 transitions, of which two have guards of the form $x < c$ and two have guards of the form $x \geq c$, for some clock x and constant c.

Collision Avoidance (CA). The CA case models a protocol where different agents want to get access to Ethernet through a shared channel [23]. The CA model has six states and 12 transitions, of which nine have guards of the form $x == c$ and four have guards of the form $x < c$, for some clock x and constant c.

Train Gate Controller (TGC). The TGC models a railway system that controls access to a bridge for several trains [4]. The bridge is a shared resource accessible by only one train at a time. The TGC model has 14 states, all of which have invariants. All invariants are of the form $x < c$ for a clock x and constant c. There are 18 transitions, of which four have guards of the form $x < c$, and four have guards of the form $x > c$, for a clock x and constant c.

A combined Gear control (CGC). The CGC models a (manually) combined gear controller for vehicles [31]. The CGC model contains 85 states, of which 20 have invariants. All invariants are of the form $x \leq c$ for a clock x and constant c. There are 120 transitions, of which ten have guards of the form $x < c$, and 10 have guards of the form $x \geq c$, for some clock x and constant c.

	GC	CA	TGC	CGC
TMI	13	9	14	36
TAD	501	26	179	1,625
SMI	12	2	12	27
SMI-NR	3	0	2	5
CXL	0	1	4	4
CXS	2	1	4	6
CCN	2	2	8	10
Total	533	41	222	1713

TABLE II: Number of generated mutants per operator

C. Research Questions

To evaluate the MUPPAAL workflow depicted in Fig 7, we consider the following research questions:

- **RQ1:** How does random trace simulation compare to timed bisimulation to identify duplicates?
- **RQ2:** What is the scalability and performance of timed bisimulation compared to random trace simulation?
- **RQ3:** How does our novel SMI-NR operator compare to the original SMI operator?

In Fig 7 on our four cases (see Section IV-B). For each case, we first generate (step 1) a set of non-equivalent mutants

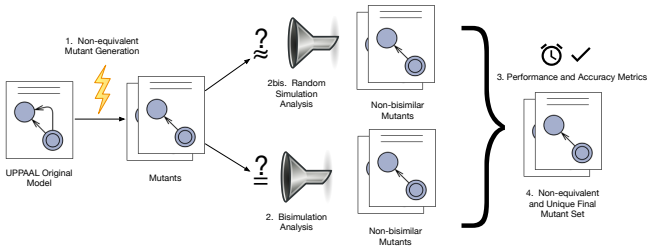

Fig. 7: Experimentation Workflow

($\mathcal{MU}$) using the operators presented in Table I (Basile *et al.* [6]) and our novel operator SMI-NR. This results in 2509 mutants as presented in Table II. Then, we independently apply our timed bisimulation and random trace simulation algorithms on this set (steps 2, 2bis). For both cases: we fix the maximum computation budget to 2100 seconds per pair of mutants or 12GB of RAM, whichever limit the analysis reached first. Regarding random trace simulation, we have three settings: 1) two traces per model and 100-time units; 2) 10 traces per model and 1000 time units; 3) 100 traces per model and 10000-time units. We run each setting ten times to mitigate randomness effects. In step 3, we collect the execution times and the number of duplicates and likely duplicates for analysis (**RQ1** & **RQ2**). We use timed bisimulation to compare SMI-NR and SMI mutants (**RQ3**). We ran our experiments on a UBUNTU 21.10 $\times$ 86_64 GNU/Linux machine with 16 cores, 2.2 GHz, 32GB RAM.

D. Results and Discussion

Case	GC	CA	TGC	CGC
ratio Bisimulation	41/533	12/41	71/222	373/1,713
ratio Trace (**N**=2,**k**=100,**E**=10)	432/533 (**st**=32.2)	38/41 (**st**=9.2)	152/222 (**st**=14.7)	1327/1,713 (**st**=38.3)
ratio Trace (**N**=10, **k**=1000)	247/533 (**st**=18.1)	38/41) (**st**=3.8)	119/222 (**st**=20.7)	774/1,713 (**st**=32.8)
ratio Trace (**N**=100, **k**=10000)	206/533 (**st**=37.4)	27/41) (**st**=10.0)	108/222 (**st**=13.1)	664/1,713 (**st**=57.0)

TABLE III: Proportion of mutant duplicates. For random trace simulation, we report the average with standard deviation (**st**)

1) Answering RQ1.: Table III reveals that mutant duplicates represent up to 32% of the total number of mutants, justifying the need for duplicate prevention and removal techniques. In general, random trace simulation overestimates the number of duplicates up to an order of magnitude. Drastically increasing the number of traces and time units yield only limited improvements. We conclude that *random trace simulation suggests too many duplicates*.

2) Answering RQ2.: Timed bisimulation is EXPTIME-complete, implying that some comparisons could exceed our computation budget. Table IV reports that only 34 mutant comparisons (amongst more than 1.5 million) timed out for CGC when running bisimulation. All settings of the random baseline scaled up. Yet, the largest one is up to 19 times slower for 3 out of the 4 cases. *Timed bisimulation has*

Case	TT	TB	TR (s)	BI (s)
GC	0	0	0.153 (**N**=2, **k**=100, **E**=10, st=0.007) 0.548 (**N**=10, **k**=1000, **E**=10, **st**=0.07) 6.38 (**N**=100, **k**=10000, **E**=10, **st**=0.08)	3.94 (st=0.54)
CA	0	0	0.040 (**N**=2, **k**=100, **E**=10, st=0.006) 1.16 (**N**=10, **k**=1000, **E**=10, **st**=0.07) 10.5 (**N**=100, **k**=10000, **E**=10, **st**=0.23	2.31 (st=0.71)
TGC	0	0	0.018 (**N**=2, **k**=100, **E**=10, st=0.002) 0.13 (**N**=10, **k**=1000, **E**=10, **st**=0.014) 1.69 (**N**=100, **k**=10000, **E**=10, **st**=0.20	2.83 (st=1.05)
CGC	0	34	0.803 (**N**=2, **k**=100, **E**=10, st=0.10) 1.91 (**N**=10, **k**=1000, **E**=10, **st**=0.17) 337.2 (**N**=100, **k**=10000, **E**=10, **st**=10)	17.58 (**st**=3.9)

TABLE IV: The total number of pairs of mutants ended by timeout of traces (TT), ended by timeout of bisimulation (TB), the average execution time(s) using traces (TR), using bisimulation (BI), the number of traces (**N**), of runs (**E**), the units of time (**k**), and the standard deviation (st).

	GC	CA	TGC	CGC
# SMI mutants	12	2	12	27
# SMI duplicates	9	2	10	22
# SMI-NR mutants	3	0	2	5
# Bisimilar pairs SMI-NR-SMI	3	0	2	5

TABLE V: SMI-NR and SMI Operators Comparison

*good scalability overall. Considering **RQ1**, it offers the best compromise between accuracy and performance.*

3) Answering RQ3.: Table V compares the SMI and SMI-NR mutants. SMI can generate a large proportion of duplicates (up to 83%) while SMI-NR by design does not produce any duplicate. The two last rows of Table V allow observing that SMI-NR produce unique mutants while preserving the behaviour of the SMI operator. We conclude that the SMI-NR operator *offers a viable alternative to SMI, introducing the same faults while preventing duplicates.*

E. Threats to Validity

Internal validity. We selected four cases of different natures: a gear controller, a network communication model avoiding collisions, and a train gate controller. These models have different sizes and numbers of clock constraints. They enabled us to observe differences in detecting and removing duplicate mutants. **Construct validity.** We chose our baseline settings to expose diverse tradeoffs between performance and accuracy concerning timed bisimulation. We did not explore larger values of **N** and **k** since accuracy only marginally improved for even higher execution times. We ran each comparison ten times to mitigate randomness effects.
External validity. We cannot guarantee that our results extend to all timed systems expressed in UPPAAL. Our cases were enough to assess diversity regarding mutants types and their analysis times.

V. Related Work

Several works cover the long-standing equivalent mutant problem [34], [13], [32], [36], [24]. Interest in the mutant duplicate problem is more recent [36], mostly at the code level [28]. MBMT gained traction more recently [14], [13], [15], [1]. Researchers applied MBMT for timed specifications [33], [21], [2], [6], [29], [42]. In [33], the authors present six mutation operators for TA, but do not guarantee the absence of equivalent or duplicate mutants. Aichernig *et al.* [2] design eight mutation operators for TA based on [21]. Again, these operators do not prevent generating equivalent or duplicate mutants. Basile *et al.* introduced six mutation operators for TS [6]. These mutation operators follow the same construction as those defined in [2], [21]. However, Basile *et al.* use a timed refinement technique to avoid the generation of equivalent mutants but do not address mutant duplicates [5], [6]. Larsen *et al.* defined an MBMT technique [29] on top of the UPPAAL-ECDAR verification tool [11]. It also uses refinement checking to eliminate equivalent mutants but does not address duplicates. Aichernig *et al.* designed an MBMT tool called MoMuT::TA [1]. MoMuT::TA maps TA to formal semantics and performs a conformance check between mutants and the original model to generate test cases automatically. The tool UPPAL-TRON [17] is an addition to the UPPAAL environment. One can also use it to handle conformance tests on TS. UPPAL-TRON simulates the IUT with input deemed relevant by the model, monitors the outputs, and checks the conformance of these against the behaviour specified in the model. Hessel and Pettersson proposed an MBMT tool called Cover [20]. Cover generates test-cases based on TA and Timed Computation Tree Logic (TCTL). One uses properties written in TCTL to verify the test model. Similar approaches exist [27], [18]. μUTA introduces a test generation method to derive mutants from the specification and executes them via online testing. It focuses on robustness testing of web services [40].

VI. Conclusion

In this paper, we proposed MUPPAAL, a mutation tool suite for model-based timed systems. It integrates equivalence-avoiding operators and focuses on alleviating the mutant duplicate problem (up to 32% of all mutants). MUPPAAL implements a novel duplicate reduction operator and a timed bisimulation algorithm. Our tool offers the best compromise between performance and accuracy compared to a random baseline. In the future, we will design more duplicate-avoiding operators and extend mutations to networks of timed automata.

Acknowledgment

Gilles Perrouin is an FNRS (Fonds National de la Recherche Scientifique) Research Associate. Jaime Cuartas received support from ERASMUS+ while at the University of Namur. Maxime Cordy obtained funding from FNR Luxembourg (grant INTER/FNRS/20/15077233/Scaling Up Variability/Cordy). Work partially funded by ERDF project IDEES. We thank Paul Temple for the early discussions on this work.

References

[1] B. K. Aichernig, J. Auer, E. Jöbstl, R. Korosec, W. Krenn, R. Schlick, and B. V. Schmidt. Model-based mutation testing of an industrial measurement device. In *Tests and Proofs*, volume 8570 of *LNCS*, pages 1–19. Springer, 2014.

[2] B. K. Aichernig, F. Lorber, and D. Nickovic. Time for mutants - model-based mutation testing with timed automata. In M. Veanes and L. Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013.

[3] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[4] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC*, pages 592–601. ACM, 1993.

[5] D. Basile, M. H. t. Beek, S. Lazreg, M. Cordy, and A. Legay. Static detection of equivalent mutants in real-time model-based mutation testing. *Empirical Software Engineering*, 27(7):160, 2022.

[6] D. Basile, M. H. ter Beek, M. Cordy, and A. Legay. Tackling the equivalent mutant problem in real-time systems: the 12 commandments of model-based mutation testing. In R. E. Lopez-Herrejon, editor, *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, pages 30:1–30:11. ACM, 2020.

[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In Hu, A. J. Vardi, and M. Y., editors, *Computer Aided Verification 10th International Conference, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–549, Vancouver, BC, Canada, June 1998.

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

[10] K. Cerāns. Decidability of bisimulation equivalences for parallel timer processes. In G. von Bochmann and D. K. Probst, editors, *Proceedings of the 4th International Workshop on Computer Aided Verification (CAV'92)*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1993.

[11] A. David, K. Larsen, A. Legay, U. Nyman, and A. Wasowski. Ecdar: An environment for compositional design and analysis of real time systems. In *Lecture Notes in Computer Science*, volume 6252/2010, Germany, 2010.

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

[13] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, and P. Heymans. Model-based mutant equivalence detection using automata language equivalence and simulations. *Journal of Systems and Software*, 2018.

[14] X. Devroey, G. Perrouin, M. Papadakis, P.-Y. Schobbens, and P. Heymans. Featured Model-based Mutation Analysis. In *International Conference on Software Engineering, ICSE*, Austin, TX, USA, 2016.

[15] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, ISSRE '99, pages 210–, Washington, DC, USA, 1999.

[16] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, pages 1–30, 2014.

[17] K. Guldstrand Larsen, M. Mikucionis, and B. Nielsen. Uppaal tron user manual - docs.uppaal.org, Jun 2017.

[18] T. R. Gundersen, F. Lorber, U. Nyman, and C. Ovesen. Effortless fault localisation: Conformance testing of real-time systems in ecdar. *Electronic Proceedings in Theoretical Computer Science*, 277:147–160, Sep 2018.

[19] T. A. Henzinger, P.-H. Ho, and H. Wong-toi. Hytech: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:460–463, 1997.

[20] A. Hessel and P. Pettersson. Cover-a test-case generation tool for timed systems. *Testing of software and communicating systems*, pages 31–34, 2007.

[21] R. M. Hierons, S. Counsell, and M. AbouTrab. Specification mutation analysis for validating timed testing approaches based on timed automata. In *2013 IEEE 37th Annual Computer Software and Applications Conference*, pages 660–669, Los Alamitos, CA, USA, jul 2012.

[22] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, 1976.

[23] H. Jensen, K. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using spin and uppaal. *BRICS Report Series*, 3, 01 2002.

[24] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649–678, Sept. 2011.

[25] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *FSE 2014: Proceedings of the ACM SIGSOFT 22nd Symposium on the Foundations of Software Engineering*, pages 654–665, Hong Kong, Nov. 2014.

[26] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. *The Theory of Timed I/O Automata, Second Edition*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[27] J. H. Kim, K. G. Larsen, B. Nielsen, M. Mikucionis, and P. Olsen. Formal analysis and testing of real-time automotive systems using UPPAAL tools. In M. Núñez and M. Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *Lecture Notes in Computer Science*, pages 47–61. Springer, 2015.

[28] B. Kurtz, P. Ammann, J. Offutt, and M. Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*, pages 142–151. IEEE Computer Society, 2016.

[29] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 319–328, March 2017.

[30] K. G. Larsen, M. Mikucionis, and B. Nielsen. Testing real-time embedded software using uppaal-tron: an industrial case study. In *the 5th ACM international conference on Embedded software*, pages 299 – 306. ACM Press New York, NY, USA, September 18–22 2005.

[31] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer*, 3(3):353–368, Aug 2001.

[32] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Trans. Software Eng.*, 40(1):23–42, 2014.

[33] R. Nilsson, J. Offutt, and S. F. Andler. Mutation-based testing criteria for timeliness. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01*, COMPSAC '04, pages 306–311, Washington, DC, USA, 2004.

[34] J. Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, Oct. 2011.

[35] J. J. Ortiz, M. Amrani, and P. Schobbens. Multi-timed bisimulation for distributed timed automata. In C. Barrett, M. Davies, and T. Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 52–67, 2017.

[36] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple fast and effective equivalent mutant detection technique. In *International Conference on Software Engineering, ICSE*, pages 936–946. IEEE, 2015.

[37] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 112, 2018.

[38] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *ISSRE*, pages 121–130. IEEE, 2010.

[39] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, Raleigh, NC, 2 edition, 2013.

[40] F. Siavashi, J. Iqbal, D. Truscan, and J. Vain. *Testing Web Services with Model-Based Mutation*, page 45–67. Springer, 2017.

[41] J. Tretmans. *Formal Methods and Testing – An Outcome of the FORTEST Network (Revised Papers Selection)*, chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer-Verlag, 2008.

[42] J. J. O. Vega, G. Perrouin, M. Amrani, and P.-Y. Schobbens. Model-based mutation operators for timed systems: A taxonomy and research

agenda. *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018.

[43] J. M. Voas and G. McGraw. *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, Inc., 1997.

[44] J. Zander, I. Schieferdecker, and P. J. Mosterman. *Model-Based Testing for Embedded Systems*. CRC Press, 2017.