MUPPAAL: Efficient Elimination and Reduction of Useless Mutants in Real-Time Model-based Systems

CUARTAS GRANADA, Jaime; Cortés, David; BETANCOURT ARIAS, Joan Sebastian; ARANDA BUENO, Jesus Alexander; Cordy, Maxime; Ortiz Vega, James Jerson; Perrouin, Gilles; Schobbens, Pierre-Yves

Published in:
Software Testing, Verification and Reliability

DOI:
10.1002/stvr.1907

Publication date:
2024

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
CUARTAS GRANADA, J, Cortés, D, BETANCOURT ARIAS, JS, ARANDA BUENO, JA, Cordy, M, Ortiz Vega, JJ, Perrouin, G & Schobbens, P-Y 2024, 'MUPPAAL: Efficient Elimination and Reduction of Useless Mutants in Real-Time Model-based Systems', *Software Testing, Verification and Reliability*, vol. 35, no. 1, e1907. https://doi.org/10.1002/stvr.1907

Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE PAPER

MUPPAAL: Efficient Elimination and Reduction of Useless Mutants in Real-Time Model-based Systems

Jaime Cuartas[1] | **David Cortés**[2] | **Joan Betancourt**[2] | **Jesús Aranda**[2] | **Maxime Cordy**[3] | **James Ortiz**[4] | **Gilles Perrouin**[5] | **Pierre-Yves Schobbens**[5]

[1]Faculty of Information Technology, Monash University, Melbourne, Autralia

[2]EISC, Universidad del Valle, Cali, Colombia

[3]SnT, University of Luxembourg, Luxembourg, Luxembourg

[4]LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

[5]Faculty of Computer Science, University of Namur, Namur, Belgium

Correspondence
James Ortiz, LTCI, Télécom Paris,
Institut Polytechnique de Paris, Palaiseau, France.
Email: james.ortizvega@telecom-paris.fr

Present address
James Ortiz, LTCI, Télécom Paris,
Institut Polytechnique de Paris 19, place Marguerite
Perey CS 20031,
91123 Palaiseau Cedex, France.

Abstract

To assess test quality, Mutation Testing (MT) creates mutants by injecting artificial faults into the system and evaluates the ability of tests to distinguish these mutants. Tests distinguishing more mutants have also been proven empirically to detect more real faults. MT has been applied to many domains. We focus on MT for timed safety-critical systems modeled as Timed Automata (TA). While powerful, MT usually yields equivalent and duplicate mutants, the former having the same behavior as the original system and the latter other mutants. Such useless mutants bring no value, waste execution time, and can be difficult to detect. We integrate useless mutant detection and removal strategies in our mutation framework MUPPAAL. MUPPAAL leverages existing equivalence-avoiding mutation operators and focuses on detecting mutant duplicates using a scalable bisimulation algorithm and a fast approximate one based on biased simulation. We also demonstrate how to design an operator that reduces the occurrence of mutant duplicates. We evaluate MUPPAAL on six systems, demonstrating that: 1) mutant duplicates account for up to 32% of all generated mutants, 2) our bisimulation approach scales effectively with these systems, and 3) biased simulations further enhance performance. Our heuristic is ten times faster than bisimulation and limits the exploration to two times the number of exact duplicates compared to up to ten times for the baseline.

KEYWORDS
Model-Based Testing, Timed Automata, Mutation Testing, UPPAAL

1 | INTRODUCTION

Timed systems (TS) are systems in which strict time constraints are essential for their reliability and correctness. They appear in several safety-critical applications such as aviation and rail systems. To design TS, engineers specify their behavior using formal models (Timed Automata TA) to analyze them and derive tests assessing the conformance of the running system to its specification. Model-Based Testing (MBT) [1, 2] uses time-bound specifications to generate test cases validating TS behavior and uncovering bugs while mitigating scalability concerns associated with exhaustive verification. Therefore, bug-finding ability is an essential capacity of a good test suite, in addition to structural coverage. Mutation Testing (MT)[3] introduces artificial faults (referred to as *mutations*) into the system using predefined mutation operators in order to evaluate test efficacy in detecting bugs. The result of applying the mutation operator to a system is a *mutant*. Given a set of mutants, tests can *distinguish* (or *kill*) mutants if they behave differently on the mutant than on the original system due to the injected defect's impact: the *mutation score* is the ratio of mutants killed to the total number of mutants. The mutation score can serve as a proxy for a test suite effectiveness [4][5]. Other applications of MT involve test generation [6][7]. Empirical research has shown that tests designed to detect mutants are significantly more likely to detect defects than other test criteria [7][2] and simulate the behavior of real defects [6][7]. Although not as popular as code-based MT [5, 6, 7], Model-based Mutation Testing (MBMT) helps to automatically identify defects related to missing functionality and misinterpreted specifications [8] that are difficult to identify through code-based testing [9, 10]. MBMT is thus well suited for timed systems whose specifications are given as models. The simplicity and versatility of MT in assessing test effectiveness, makes this technique popular in research and industry [11].

However, not all mutants are created equal in terms of utility during test assessment. Some may be *equivalent*, i.e. they exhibit the same behavior as the original system despite their syntactic difference [11]. Therefore, no test case can distinguish such mutants from the original system. Hence they have no purpose for bug detection, waste execution time and effort trying to kill them. Similarly, *duplicate* mutants exhibit the same behavior as other mutants [4, 11]. Preventing and removing such *useless* mutants reduces the computational cost of MBMT and builds more confidence in mutation scores. Therefore, there is a need to prevent and remove useless mutants, especially for timed systems where simulations can be long. Recently, Basile *et al.* tackled the equivalent mutant problem for Timed Automata with Input and Output (TAIO): they defined mutation operators that prevent mutants from refining the original system [12, 13]. However, this technique does not address duplicate mutants: in our experiments, up to 32% of all mutants generated were duplicates motivating the need for a more comprehensive solution to useless mutants. To this end, we present the design and implementation of MUPPAAL, a mutation framework that addresses this challenge for Timed Automata (TA) specified in UPPAAL [14]. Our contributions within the MUPPAAL framework are the following:

1. We introduced a *novel timed mutation operator*, SMI-NR, that we *proved by design to prevent duplicate mutants*;
2. We *detected and removed duplicate mutants* using a parallelized timed bisimulation algorithm [15, 16] to assess behavioural equivalence between two mutants. If any two mutants are duplicates, we keep one of them;
3. We developed two simulation algorithms: one based on random simulations (RS) and one based on biased simulations (BS) [17]. The biased random algorithm targets infected states and transitions [7] that exploit syntactic differences between the original and mutant models. It then leverages the UPPAAL model-checker to generate traces involving infected states to increase the chances of discriminating UPPAAL mutants. The primary goal is to improve our ability to distinguish non-duplicate mutants while keeping a fast heuristic.
4. We implemented MUPPAAL using the UPPAAL execution engine for TAs and UPPAAL-TRON [18] for timed trace generation and checking. In addition to the above contributions, MUPPAAL supports mutation operators for TA from [19, 12, 15] avoiding equivalent mutants using refinement prevention [13, 12];
5. We assessed MUPPAAL[†] on six different UPPAAL models from distinct domains. Timed bisimulation can compute exact (non-)equivalence in less than one second on average for a given pair of mutants. Our biased simulation heuristic is ten times faster but identifies twice as many duplicates as bisimulation. Yet, it can drastically limit the number of potential duplicates to check using the exact bisimulation method, offering a tradeoff between speed and accuracy, and can be combined with timed bisimulation. In contrast, the random baseline suggests many false duplicates (up to 10 times compared to bisimulation) and is not faster than timed bisimulation. Finally, our novel mutation operator effectively reduces the number of mutants while perfectly capturing the initial mutation operator behavior.

This paper builds on our previous work [20] in several ways. First, this paper provides a novel simulation algorithm that exploits the knowledge of infected states to guide timed trace generation [17]. Second, we reimplemented our timed bisimulation algorithm to take advantage of parallel processing [16], yielding scalability and performance improvements. Finally, our evaluation adds two case studies, a tram door model, and a scheduling framework.

The remainder of this paper is as follows. Section 2 introduces the formalisms we use and the equivalent and duplicate mutant problem. Section 3 presents our new mutation operator and duplicate removal algorithms. Section 4 describes MUPPAAL and reports on our experiments. Section 5 presents related work, and Section 6 wraps up with concluding remarks and future work.

2 | BACKGROUND

In this section, we introduce the main formalisms, namely Timed Automata and Timed Automata with Inputs and Outputs. We also describe Timed Bisimulation, Trace Simulation techniques, the UPPAAL tool, and the Equivalent/Duplicate Mutant Problem.

[†] MUPPAAL implementation and full results of its evaluation are available: https://github.com/DavidC0rtes/stvr-muppaal

2.1 | Clocks and Timed Automata

To model the continuous-time domain, we use non-negative real-valued variables, known as *clocks*. Clocks are variables that are incremented synchronously at the same rate. One of the most extensively studied formalisms for modeling TS is the Timed Automata (TA) [21]. Several model checking tools rely on TA for their analysis, including UPPAAL [14], KRONOS [22], and HYTECH [23]. TA extend the capabilities of Finite State Automata (FSA) by incorporating a set of clocks that are incremented at a uniform rate. The reset of a clock in a timed automaton sets its value back to zero. TA allow transitions to be enabled or disabled based on *clock constraints*. A transition occurs if all other conditions are satisfied. Our work uses an extension of TA called Timed Automata with Inputs and Outputs (TAIO) [24]. In TAIO, actions are classified into two distinct sets: inputs and outputs [25][24]. We adopt the TAIO extension of Aichernig *et al.* [24] for our purposes.

2.2 | Timed Automata with Inputs and Outputs

A TAIO is a refined TA in which we model the interaction between a system and its environment by using input and output actions [24]. 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. Let φ_1 and φ_2 be clock constraints. 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 $\nu : X \to \mathbb{R}_{\geq 0}$ assigns 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, transitions can have a *guard* that will allow 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 !) [24]. The output actions of a TAIO $\mathcal{A}$ can be the input actions of a TAIO $\mathcal{B}$. In [26], it is considered that there are multiple initial states, but since we use the UPPAAL tool for modeling and testing our automata, and the automata in UPPAAL have only one state, we will consider only one initial state in the following definition (Definition 4). We formally define TAIO as follows:

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 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, $\nu \models I(l)$. 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 fragment 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: $\rho = (l_0, \nu_0) \xrightarrow{d_0} (l_0, \nu'_0) \xrightarrow{a_0} (l_1, \nu_1) \ldots (l_{n-2}, \nu'_{n-2}) \xrightarrow{a_{n-1}} (l_{n-1}, \nu_{n-1} \xrightarrow{d_n} (l_n, \nu_n)$ 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.

A *timed trace* [21] over Σ is a finite sequence $\theta = ((\sigma_1, t_1), (\sigma_2, t_2) \ldots (\sigma_n, t_n))$ 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 Figure 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 that 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 Figure 1 is nondeterministic because location l_1 has two outgoing transitions on the same input action (b?).

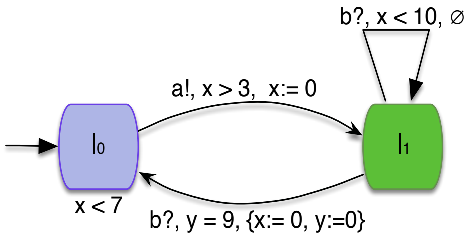

FIGURE 1 A nondeterministic TAIO with two clocks x and y.

Example 2. Figure 2 presents a run of the timed automaton $\mathcal{A}$ presented in Figure 1. A run of $\mathcal{A}$ is a path in TIOTS($\mathcal{A}$) = $(s_0, [x = 0.0, y = 0.0]) \xrightarrow{3.1} (s_0, [x = 3.1, y = 3.1]) \xrightarrow{a!} (s_1, [x = 0.0, y = 3.1]) \xrightarrow{1.9} (s_1, [x = 1.9, y = 5.0]) \xrightarrow{b?} (s_1, [x = 1.9, y = 5.0]) \xrightarrow{4.0} (s_1, [x = 5.9, y = 9.0]) \xrightarrow{b?} (s_0, [x = 0.0, y = 0.0])$ starting from the initial state with each clock set to 0 and the timed trace θ associated with this non-accepting run is $((a, 3.1)(b, 5.0)(b, 9.0))$.

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 $(l, a, \phi_2, Y_2, l'_2) \in T$, implies $\phi_1 \cap \phi_2 = \emptyset$ 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 $\nu \models \phi$.

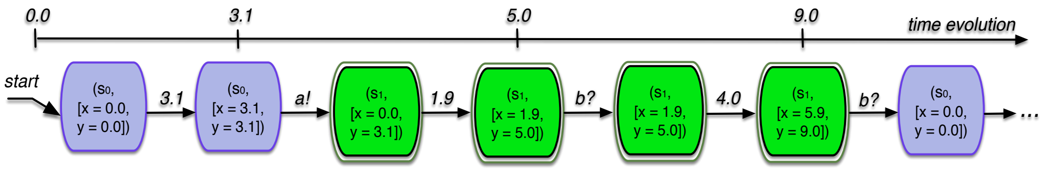

FIGURE 2 The run of the timed automaton $\mathcal{A}$ in Figure 1

2.3 | Timed Bisimulation and Trace Simulation

Timed bisimulation and trace simulation are two important concepts used in systems modeling and verification, especially those with timing constraints and concurrent behavior. Timed bisimulation computes behavioral equivalence in timed systems while considering both observable actions and timing constraints. Trace simulation is a more general technique for checking whether two systems produce similar sequences of observable actions, regardless of timing.

2.3.1 | Trace Simulation

Trace simulation is a technique used to compare and analyze the behavior of two systems by observing sequences of actions or events, called *traces*, generated by each system. The goal of trace simulation is to establish a relationship between the traces of the two systems, and the purpose is to verify that one system behaves like another in terms of observable actions. Trace simulation involves running both systems with the same input, recording their traces, and comparing these traces to check for equivalence. Trace simulation is often used in software testing and verification to ensure that a system or program produces the expected output or behavior for a given input.

Definition 7. A Timed Trace (TT) over Σ is a sequence $\rho = ((\sigma_0, t_0)(\sigma_1, t_1) \ldots (\sigma_n, t_n))$ of actions or propositions paired with non-negative real numbers (i.e., $(\sigma_i, t_i) \in (\Sigma \times \mathbb{R}_{\geq 0})$) such that the timestamp sequence $t = t_0 \cdot t_1 \cdots t_n$ is non-decreasing (i.e., $t_i \leq t_{i+1}$). We sometimes define ρ as the pair $\rho = (\sigma, t)$ with $\sigma \in \Sigma^*$ and t a sequence of timestamps of the same length [21]. Furthermore, a timestamp sequence $t = t_0 \cdot t_1 \cdots t_n$ respects:

1. Initialization: $t_0 = 0$,
2. Monotonicity: for all $i \geq 0$, $t_{i+1} \geq t_i$,
3. Progress: for all $t \in \mathbb{R}_{\geq 0}$, there exists i such that $t_i > t$.

2.3.2 | Timed Bisimulation

Timed bisimulation [27] is a relation between system states that captures behavioral equivalence concerning both functional behavior and timing constraints. Timed bisimulation is used to determine whether two states in a timed system exhibit the same behavior in terms of observable actions, given the timing information. Timed bisimulation is particularly useful for verifying systems modeled as TA or other formalisms with timing constraints.

Definition 8 (Timed Bisimulation [27]**).** 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.

Nilsson *et al.* [28]		Aichernig *et al.* [24]		Basile *et al.* [12]	
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 1 Mutation operators for TA.

Example 3. Consider the two TAIO $\mathcal{A}$ and $\mathcal{B}$ in Figure 3 with the alphabet $\Sigma = \{a?\}$ and the set of clocks $X = \{x, y\}$. $\mathcal{A}$ performs non-deterministically the transition with the guard $x \leq 2$, the action $a?$, resets clock x to 0 and enters location s_1. Similarly, $\mathcal{B}$ is the same but with y. We will show that these TAIO are timed bisimilar (Definition 8). We have $(S_0, [x = 0.0])$ in TIOTS($\mathcal{A}$) and $(T_0, [y = 0.0])$ in TIOTS($\mathcal{B}$). Then, since $\mathcal{A}$ can run the delay transition $(S_0, [x = 0.0]) \xrightarrow{(1.0)} (S_0, [x = 1.0])$ and then $(S_0, [x = 1.0]) \xrightarrow{a?} (s_1, [x = 0.0])$. $\mathcal{B}$ can match this with the delay transition with $(T_0, [y = 0.0]) \xrightarrow{(1.0)} (T_0, [y = 1.0])$ and discrete transition $(T_0, [y = 1.0]) \xrightarrow{a?} (T_1, [y = 0.0])$. So, the given TAIO are bisimilar.

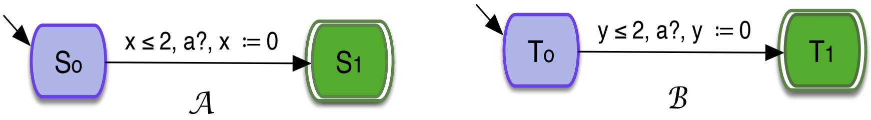

FIGURE 3 Two TAIO $\mathcal{A}$ and $\mathcal{B}$.

2.4 | Mutation Operators and Equivalent Mutant Problem

Mutation operators are an essential component of mutation testing, a software testing technique used to evaluate the effectiveness of a test suite. Mutation operators are responsible for generating these mutations. The equivalent mutation problem highlights the importance of continually improving and extending test coverage to ensure that real defects are not missed during testing.

2.4.1 | TA and Mutation Operators

Nilsson *et al.* [28] 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.* [29] and Aichernig *et al.* [24] also proposed some mutation operators for UPPAAL to test the behavior 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.* [12] proposed six mutation operators in TAIO, designed to avoid the generation of subsumed mutants by construction. Three of the six mutation operators in [12] 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 [12] is to perform a refinement check between the mutant and the system model, using ECDAR [30]. Table 1 shows the mutation operators retrieved from the contributions considered.

2.4.2 | Equivalent/Duplicate mutant problem

MT is one of the most effective coverage criteria for evaluating the quality of the test suite [3, 11]. In addition, several recent empirical studies evaluated the effectiveness and efficiency of MT [4, 31, 11]. The equivalent/duplicate mutant problem is a well-documented challenge in mutation analysis [4][5][32]. It arises when two program variants exhibit identical behavior, making them indistinguishable by test cases. This poses significant problems for both test suite generation and evaluation. In the former case, resources are wasted trying to eliminate mutants that are, in fact, impossible to kill (achieving a 100% mutant kill becomes unattainable if equivalence exists). In the latter case, it distorts the evaluation results because the same (faulty) behavior is assessed several times. Equivalent and duplicate mutants can be depicted as [4]: (*a*) behavioral equivalence between a mutant and the original system, (*b*) behavioral equivalence between two mutants but not with the original system. Mutants that fall under case (*a*) are referred to as *equivalent*, whereas those that fall under case (*b*) are referred to as *duplicates*. In the context of this paper, we focus on both cases, i.e., mutants of case (*a*) and (*b*). The additional costs incurred by these useless mutants are not negligible: between 30% - 40% increase for equivalent mutants [33] and between 20% - 30% for duplicate mutants [11].

2.5 | Useless Mutant Reduction Strategies

2.5.1 | Code-level techniques

To overcome the equivalent mutant problem, the software testing community developed many solutions, mostly at the code level. For source code, this problem is undecidable [33], meaning that there is no exact approach that can remove all equivalent mutants. Therefore, various heuristics have been proposed. Surveys [11, 33] detail these heuristics, they include: search-based approaches [34], constraint-based approaches [5, 35], and compiler optimization [4, 36], the latter having gained popularity because of its simplicity and efficiency [37, 38]. The coming of age of machine learning for software engineering has led some researchers to address the equivalent mutant problem with machine learning and large language models [37] with promising results.

However, these techniques are hardly transferable to our context (timed systems), since they do not work at the same abstraction level and do not consider hard real-time constraints. Also, regarding learning-based techniques, the scarcity of UPPAAL models compared to large repositories of JAVA software is likely to limit training/fine-tuning possibilities to obtain optimized models. Finally, these works concentrate on equivalent mutants and do not specifically address duplicates (though some techniques are applicable in both cases).

2.5.2 | Model-level techniques

One of the advantages of considering systems at the model level is that the equivalent mutant (and its related duplicate) problem becomes decidable depending on the formalism used. This open the way to the use of formal refinement and bisimulation techniques to avoid and get rid of useless mutants in a sound and complete manner. One line of work focus on *avoiding equivalent mutants* in the first place [13, 12]. Basile *et al.* use timed refinement techniques to explore the conditions under which a mutation operator can yield equivalent mutants. By ensuring the non-conformance of the mutant to the original model we can guarantee non-equivalence [12, 13]. An alternative is to *remove equivalent mutants* using timed refinement relations, as proposed by Larsen *et al.* [30] and Aichernig *et al.* [24]. Devorey *et al.* used an efficient bisimulation algorithm for checking equivalence between non-timed automata. While we leverage these strategies in the present paper, we note that, as for code-level mutation, none of them specifically focus on mutant duplicates. Thus, our goal is to apply strategies for avoiding and removing *all* useless mutants, not just equivalent ones. MUPPAAL decreases MBMT costs by eliminating useless mutants from the analysis.

2.6 | UPPAAL and UPPAAL-TRON

UPPAAL is a tool for the modeling, simulation, and verification of networks of TA extended with data types, user functions, clocks, and synchronous communication channels [14]. The UPPAAL tool offers significant advantages, especially when dealing with systems with large and complex dynamics and stochastic behavior. UPPAAL includes a graphical editor and animator/simulator, as well as an efficient model checker.UPPAAL provides an exhaustive symbolic analysis of the model, and either a proof that the model satisfies a property, or a counterexample consisting of a trace of actions and delays illustrating how the property is violated. It has been successfully applied to numerous industrial cases. UPPAAL was extended with components for test generation, scheduling, timed specification and controller synthesis (see Figure 4).

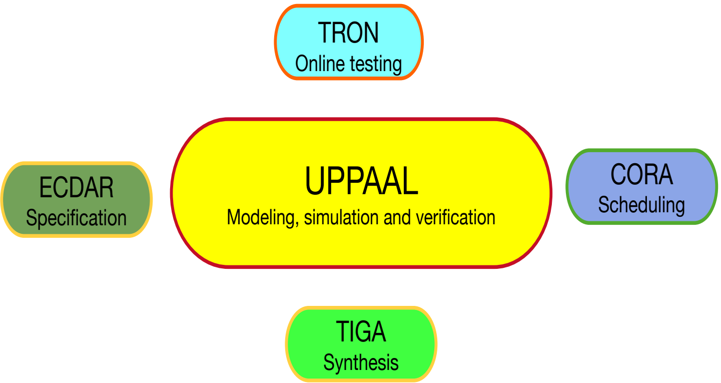

FIGURE 4 UPPAAL Tool and extensions.

Here, we use the component UPPAAL-TRON [39]. TRON is a testing tool, based on UPPAAL, suited for online black-box conformance testing of TS. TRON is used for testing the Implementation Under Test (IUT). TRON can use a randomized online testing algorithm, an extension of the UPPAAL model checker [14]. TRON has well-defined formal semantic and correctness criteria. These criteria define how a correct implementation should behave compared to the modeled behavior. The implementation of the tool allows real-time testing of many real-time systems by extending the efficient algorithms and data structures of the model checker engine. TRON can generate and execute tests event by event in real-time by stimulating and monitoring the IUT. TRON performs these two operations, computing the possible set of symbolic states based on the *timed trace* observed so far. A *timed trace* in TRON consists of a sequence of input or output actions and time delays [18].

2.6.1 | Query Language

In model checking, we must verify that a model complies with the required specifications. Therefore, the properties must be expressed in a formal language, such as a simplified version of TCTL [40] (Timed Computational Tree Logic), which is supported by UPPAAL. TCTL is a specialized version of Computation Tree Logic (CTL) tailored to express real-time properties [40]. These properties find application in systems modeled by TA. Similarly to the CTL properties, the TCTL properties consist of both state and path formulas. A state formula represents an expression defined over the atomic propositions within a model. It can be evaluated separately for each state within a system. A path formula, on the other hand, quantifies the paths within a model regarding either some path ($\exists$) or all paths ($\forall$). Temporal operators are used to specify the states for which the state formula is to be valid. Commonly used operators include $\Box$ for global (indicating that a condition holds now and into the indefinite future) and $\Diamond$ for final (indicating that a condition will hold at some point in the future). Table 2 lists the types of TCTL properties supported by UPPAAL.

As a result, the query language used by UPPAAL to specify properties is less expressive than the TCTL, and not every TCTL formula can be expressed in UPPAAL. The following reachability properties can be written in UPPAAL:

Property 1. `A[] not deadlock`

At every stage, at least one transition will eventually be enabled.

Path formula	UPPAAL	Description
$\exists\Diamond\rho$	E<>ρ	There exists a path where ρ eventually holds.
$\exists\Box\rho$	E[]ρ	There exists a path where ρ is always satisfied.
$\forall\Box\rho$	A[]ρ	For all paths, ρ is always satisfied.
$\forall\Diamond\rho$	A<>ρ	For all paths, ρ is eventually satisfied.
$\rho \leadsto \xi$	ρ--> ξ	ρ implies ξ eventually, always.

TABLE 2 Query language in UPPAAL

Property 2. `E<> A.s`

It is possible to reach the *s* state.

Property 3. `E<> (A.on or B.on)`

It is possible to reach states where *A* is on or *B* is on.

Property 4. `A[] (not A.on or (B.off and C.off))`

For every path, if the state *A* is on, both states *B* and *C* must be off.

3 | OVERCOMING THE EQUIVALENT AND DUPLICATE MUTANTS PROBLEM

Three strategies target the equivalent (and duplicate) mutant problem [33]: (1) avoid (2) detect, and (3) suggest equivalent (and duplicate) mutants. In this section, we describe how MUPPAAL implements them.

3.1 | Avoiding Equivalent and Duplicate Mutants

The design of the mutation operator 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 [28, 29, 19] do not avoid the generation of useless mutants. Basile *et al.* [12] avoid equivalent mutants by proposing operators guaranteeing that mutants do not refine the original system's behavior. We have implemented them for UPPAAL. In addition, we offer a new duplicate-avoiding mutation operator.

3.1.1 | Mutation Operators and Non-Equivalent Mutants.

Here, we use the guidelines and the six mutation operators of [12] (see Table 1). We rely on them to avoid equivalent mutants.

3.1.2 | A new duplicate-avoiding Mutation Operator.

A duplicate mutant has the same behavior as another mutant and is thus useless. The refinement technique of Basile *et al.* ensures non-equivalence [12, 13] 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 4. Figure 5 illustrates a base system modeled as a non-duplicate TAIO. Applying TMI, i.e. removing the second transition $(l_1, a?, true, \emptyset, l_2)$ gives Figure 6, while applying SMI, i.e. removing location l_2, gives Figure 7. Both have the same behavior: they are thus mutant duplicates.

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 [12]. The definition of the SMI-NR operator is as follows:

Definition 9. For a TAIO $\mathcal{A}$. The mutants $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ are defined as follows:

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

$$\mathcal{A}_{smi} \in \mathcal{M}_{smi}(\mathcal{A}) \textit{ such that } \mathcal{A}_{smi} = (L \setminus \{l_{smi}\}, l_0, X, \Sigma_I, \Sigma_O, \Sigma, T_{smi}, I),\ l_{smi} \in L, l_{smi} \neq l_0,$$

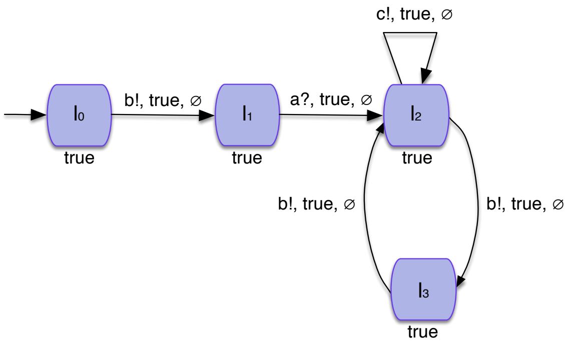

FIGURE 5 An original model (TAIO).

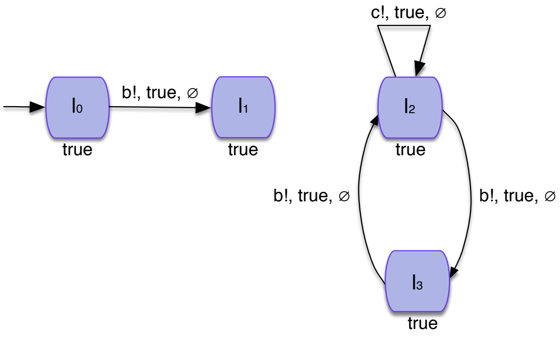

FIGURE 6 A mutant generated by TMI operator from Figure 5.

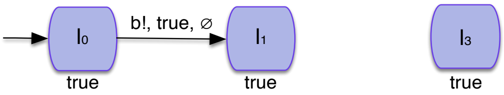

FIGURE 7 A mutant generated by SMI operator from Figure 5.

$$T_{smi} = \{(l_1, a, \phi, Y, l_2) \in T \mid l_{smi} \neq l_1 \text{ and } l_{smi} \neq l_2\}$$

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 and $\mathcal{A}_{smi\text{–}nr}$ is generated.

The conditions for the SMI-NR operator are as follows:

- $l_{smi} \in L \setminus \{l_0\}$, meaning the removed location is not an initial location,
- $\exists t = (l'_{smi}, a, \phi, Y, l_{smi})$, there exists a transition $t \in T$ that has an input action $a \in \Sigma_I$ with l_{smi} as a target location, and some l'_{smi}, ϕ, Y,
- $T_{smi} = \{(l_1, a, \phi, Y, l_2) \in T \mid l_{smi} \neq l_1$ and $l_{smi} \neq l_2\}$, as a consequence of removing a location, the new set of edges does not have the removed location,
- $I \colon L \setminus \{l\} \to \Phi(X)$, where $l \in L$ is from the original model. The locations that are not removed in the mutant keep the same invariant.

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 TMI and SMI induce duplicates is costly. Therefore, we define a relaxed condition in Proposition 1 (see proofs in the companion website) permitting us 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. In Figures 8 and 9 we depict examples where Proposition 1 cannot prevent duplicates. In Figure 8, we can see that removing the transition $(l_{prev}, a_{smi}?, x >= 1, \emptyset, l_{smi})$ leads to a deadlock because the only possible transition is $(l_{prev}, b!, x >= 1, \emptyset, l_3)$. At location l_3, we are stuck in a deadlock and cannot reach location l_{smi}. So here, if we remove such a transition, it is equivalent to the mutant in which we remove the location l_{smi}. In Figure 9, we can see that removing the transition $(l_{prev}, a_{smi}?, x >= 1, \emptyset, l_{smi})$ leads to a deadlock (because we cannot reach the location l_{smi}). Therefore, if we remove such a transition, it is equivalent to the mutant in which we remove the location l_{smi}.

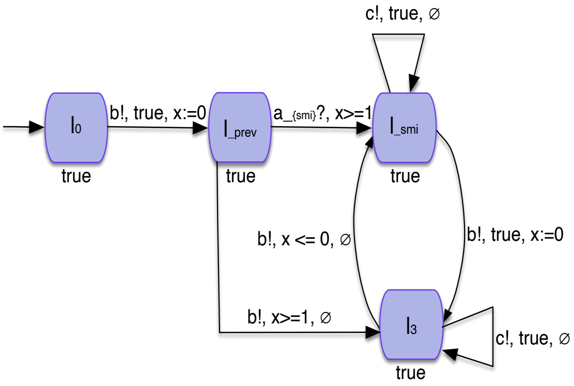

FIGURE 8 Removes transition $(l_{prev}, b!, x \geq 1, \emptyset, l_3)$ could cause a deadlock because the only possible transition is $(l_{prev}, b!, x >= 1, \emptyset, l_3)$.

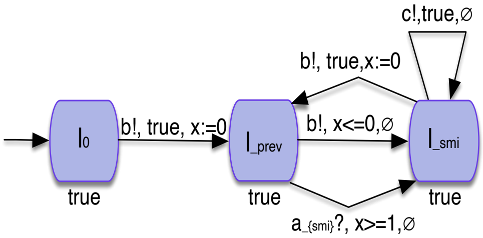

FIGURE 9 Removes location l_3 could cause a deadlock because the only possible location is l_{smi}.

3.2 | 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 [4, 33], which affects mutation testing effectiveness [41]. 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* [27, 15]. MUPPAAL uses Ortiz *et al.* timed bisimulation algorithm [15]. Because timed bisimulation's complexity is EXPTIME [27], if the bisimulation process takes longer than the specified time to analyze a pair of mutants, our algorithm will stop the bisimulation process.

```
Input:  A list of mutants $\mathcal{LM}$
Output: A pair of sets (no duplicate and timeout mutants ended).
$\mathcal{MU}^{tm}$ ={}; A set of pairs of mutants ended with timeout
EQC ={}; A set of equivalence classes
NEQC ={}; A set of no-equivalence classes
for(i=0; i<$\mathcal{LM}$.size()-1; i++){
    for(j=i+1; j<$\mathcal{LM}$.size(); j++){
        // search for an equivalence class in EQC that contains i
        eqc_i = getEquivalenceClass($i$);
        // search for a no-equivalence class in NEQC that contains i
        neqc_i = getNoEquivalenceClass($i$);
        if(eqc_i == null){ // if they do not exist, create them
            EQC.add({$i$});
            NEQC.add({});
        }
        if($j \in$ eqc_i || $j \in$ neqc_i){
        // If the mutant j is in the equivalence class or no-equivalence
        // class of i, timed bisimulation between i and j can be inferred
            continue; }
        // if the result cannot be inferred, compute timed bisimulation
        bisim = BisimilarAlgo($\mathcal{LM}[i]$, $\mathcal{LM}[j]$);
        if(bisim == TIMEOUT){
            // Add the pair of $i-th$, $j-th$ mutants that ended with timeout
            $\mathcal{MU}^{tm}$.add($\mathcal{LM}[i], \mathcal{LM}[j]$);
            continue; }
        // update the equivance classes and no-equivalent classes accordingly
        eqc_j = getEquivalenceClass($j$);
        neqc_j = getNoEquivalenceClass($j$);
        if (bisim == 1){ // if the mutants are equivalent
            merge(eqc_i, eqc_j);
            merge(neqc_i, neqc_j);
        } else if(bisim == 0){ // if the mutants are not equivalent
            neqc_i.extend(eqc_j);
            neqc_j.extend(eqc_i);  }
    }
}
$\mathcal{MU}$ ={};
for(c=0; $c <$ EQC.size(); c++){
    // Take one representative of each equivalence class
    $\mathcal{MU}$.add(EQC[$c$][0]); }
return $pair(\mathcal{MU}, \mathcal{MU}^{tm})$;
```

Algorithm 1 Bisimulation Process taking advantage of timed bisimulation as equivalence relation.

Algorithm 1 describes how MUPPAAL detects duplicates using timed bisimulation [15]. It receives a list of mutants and 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. In addition to the journal version of this article, we present a strategy that can save computing time by taking advantage of the algebraic properties of timed bisimulation.

Timed bisimulation is an equivalence relation [42]. An equivalence relation like timed bisimulation partitions a set of TAIO into disjoint equivalence classes, where the members of each class are duplicates of each other. Then, the problem of detecting duplicates in a set of mutants can be seen as the problem of partitioning it modulo timed bisimulation. Efficiently partitioning a set under an equivalence relation is thoroughly investigated in [43]. We apply these results to save computing time potentially, but first, we need some definitions.

Let timed bisimulation equivalence be denoted as $\sim$. Let X be a set of TAIO. The equivalence class of TAIO $\mathcal{A}$, denoted $[\mathcal{A}]$, is defined as $[\mathcal{A}] = \{x \in X : x \sim \mathcal{A}\}$. A no-equivalence class $[\neg\mathcal{A}]$ is defined as $[\neg\mathcal{A}] = \{x \in X : \exists y \in [\mathcal{A}] \mid x \not\sim y\}$ [42].

With these, we are ready to introduce Algorithm 1, which efficiently constructs the partition induced by timed bisimulation over the input mutant set. It works as follows:

1. Iterate through every pair $\mathcal{A}$, $\mathcal{B}$ of the input mutants (lines 6, 7).
   (a) Get the equivalence class and no-equivalence class of $\mathcal{B}$. If they do not exist, create them (lines 8-15).
   (b) If $\mathcal{A}$ belongs to $[\mathcal{B}]$, then $\mathcal{A}$ is equivalent to $\mathcal{B}$. If $\mathcal{A}$ belongs to $[\neg\mathcal{B}]$, then $\mathcal{A}$ is not equivalent to $\mathcal{B}$. These facts can be inferred without computing timed bisimulation; we may continue with the next pair. (lines 16-20).
   (c) If neither case holds, the timed bisimulation must be computed (line 22).
   (d) If the result is a timeout, then update $\mathcal{MU}^{tm}$ (line 25).
   (e) If the timed bisimulation is true, then $[\mathcal{A}]$ and $[\mathcal{B}]$ are the same, and they must be merged, and this holds for $[\neg\mathcal{A}]$ and $[\neg\mathcal{B}]$ too (line 33).
   (f) If the timed bisimulation is false, then $[\neg\mathcal{A}]$ must include $\mathcal{B}$ and any other mutant equivalent to $\mathcal{B}$. Also, $[\neg\mathcal{B}]$ must include $\mathcal{A}$ and any other mutant equivalent to $\mathcal{A}$ (line 34-37).
2. After the set of mutants was partitioned, construct a new set by taking one mutant of each class. These mutants are guaranteed to be not equivalent. Return the set of non-duplicate mutants and the pairs of mutants that timed out during timed bisimulation (lines 40-45).

This technique was first presented and applied to the equivalent mutant problem in [16], where experimentation and details can be consulted.

3.3 | Suggesting Duplicate Mutants

As stated before, timed bisimulation is computationally costly (EXPTIME [27]). To assess this complexity in practice, we present two baseline approaches (Random and Biased Simulation) 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.

3.3.1 | Random Simulation (RS)

Random Simulation (RS) assumes a uniform distribution of traces over the model; such traces are selected randomly. Therefore, 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 [17]. 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 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 [39]. To perform trace simulations on UPPAAL-TRON, our translator tool uses ANTLR [44] 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

```
1  Input: A mutant $\mathcal{M}_1 \in \mathcal{MU}^{tm}$, number of traces to generate $N$, and a  simulation time $k$
2  Output: An array with UPPAAL-TRON traces
3  $\mathcal{T}$=[$N$];   // The set of N UPPAAL traces
4  $\mathcal{T}'$=[$N$];  // 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      $\mathcal{T}[i]$ =verifyta($\mathcal{M}_1$, k, r);
10     //Translate trace to UPPAAL-TRON format
11     tree=parser(lexer($T[i]$));
12     $\mathcal{T}'[i] = tree.format()$;
13 }
14 return $\mathcal{T}'$;
```

Algorithm 2 Trace generation.

```
Input: A set of mutants $\mathcal{MU}^{tm}$, a number of traces to generate $N$ and a simulation
    time $k$
Output: A list of non-bisimilar mutants
// The set of N UPPAAL-TRON traces per mutant
$\mathcal{T}'$=[$\mathcal{MU}^{tm}$.size()][$N$];
for(i=0; i<$\mathcal{MU}^{tm}.size()$; i++){
    $\mathcal{T}'$[i]=TraceGeneration($\mathcal{MU}^{tm}[i]$, $N$, $k$);
    }
$\mathcal{NBM} = [()]$ // List of no bisimilar mutants
for(i=0; i<$\mathcal{MU}^{tm}.size()$-1; i++){
    for(j=i+1; j<$\mathcal{MU}^{tm}.size()$; j++){
        for(k=0; k<$N$; k++){
            Pass1=Tron.check($\mathcal{MU}^{tm}[i]$, $\mathcal{T}'[j,k]$);
            Pass2=Tron.check($\mathcal{MU}^{tm}[j]$, $\mathcal{T}'[i,k]$);
            if(!(Pass1$\wedge$Pass2))
               $\mathcal{NBM}$=$\mathcal{NBM}$.add(($\mathcal{MU}^{tm}[i]$));
        }
    }
}
return $\mathcal{NBM}$;
```

Algorithm 3 Random Trace Simulation Algorithm.

them, 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 that do not accept random traces are not bisimilar.

3.3.2 | Random Algorithms

Algorithm 2 describes the random trace generation process of MUPPAAL. The algorithm works as follows: a random seed is chosen in line 7, which is then used in line 9 to generate a trace with simulation time k for the mutant $\mathcal{M}_1$ using the Verifyta command-line tool [14]. Later on, we translate the generated trace from UPPAAL format to UPPAAL-TRON format (lines

11-12). 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 12-13). In addition, if mutants are not bisimilar within a pair, $\mathcal{NBM}$ is updated with the *ith* mutants (line 15). 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. Figure 10 depicts the requirements of UPPAAL-TRON (TRON-traces and Mutants).

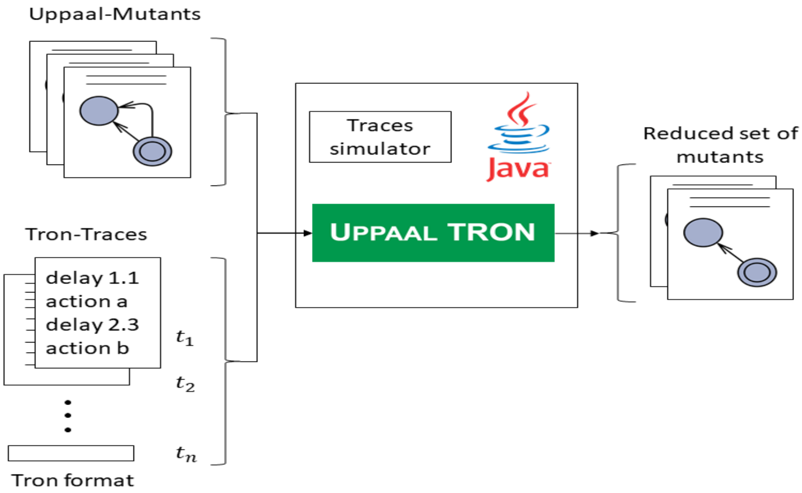

FIGURE 10 Simulation of traces using UPPAAL-TRON

3.3.3 | Biased Simulation (BS)

Biased Simulation (BS) leverages fundamental principles of mutation testing. Mutations are localized and primarily result in behavioral differences. The assumption here is that these differences can be detected through a trace, which, when executed on the original system or its mutant, traverses one of the states affected by the mutation. To achieve this, we employ UPPAAL's verifyta to generate biased traces. These traces are then input into UPPAAL-TRON alongside another mutant to determine if the mutant can faithfully simulate the traces. The presence of a counterexample indicates that the mutants are not equivalent. Otherwise, if no counterexample is found, the result remains inconclusive, prompting a bisimilarity check. The identification of *infected states* is done through a syntactic and semantic comparison between two given models. These models are subsequently utilized to generate the biased traces. This strategy has demonstrated effectiveness in mutation testing at the code level and has proven valuable in identifying equivalent mutants [17, 45, 46, 11]. Our approach focuses on generating biased traces to reduce state exploration, minimize the number of traces required to distinguish one model from another, and consequently, reduce the time needed for this task. More specifically, the set of infected states, denoted as S_{infect}, is constructed by loading them into memory and extending juppaal, a Java API designed to work with UPPAAL model files. Additionally, Google's guava library is employed for performing set computations.

Formally, given $\mathcal{M}_{\mathcal{A}}, \mathcal{M}_{\mathcal{B}}$ two mutants, then, the set of infected locations: $S_{infect} = \mathcal{M}_{\mathcal{A}} - \mathcal{M}_{\mathcal{B}}$ such that:

$$(l_i \in L_{\mathcal{A}}) \in S_{infect} \leftrightarrow l_i \in L_{\mathcal{A}} - L_{\mathcal{B}} \lor l_i \in T_{\mathcal{A}} - T_{\mathcal{B}}$$

A location is different iff it is in the locations difference set or if it is either a source or target of an edge in the edge difference set, where an edge $t \in T$.

$$(t_i \in T_{\mathcal{A}}) \in S_{infect} \leftrightarrow t_i \in T_{\mathcal{A}} - T_{\mathcal{B}}$$

This means that a transition is said to be different from another iff any of the following conditions are met: (*i*) they have different source and/or target, (*ii*) they have different guards, (*iii*) they synchronize over different channels, or (*iv*) they update or select different variables.

In turn, when comparing one location to another, they will be deemed different if they have different types (urgent, committed, normal), if they have different invariants, or if they have different names, in that order. The next step is to generate the traces; since TA deal with real-time, we decided to use verifyta for this task by feeding it the model and a constructed TCTL path formula in the form of `E<> Automata.li` where:

- $l_i \in S_{infect}$

For edges in T_{infect} reachability formulas of the form `E<> reachSource && reachTran && Automata.li` were used, where:

- $l_i \in S_{infect}$
- `reachSource` is a boolean variable set to `true` on all incoming edges to the source location.
- `reachTran` is a boolean variable set to `true` on the specific edge identified as different.

With this, we guarantee that all generated traces will go from the initial location to the desired location and, if necessary, along the desired path. In Figure 11, we show the flow of the biased trace generator.

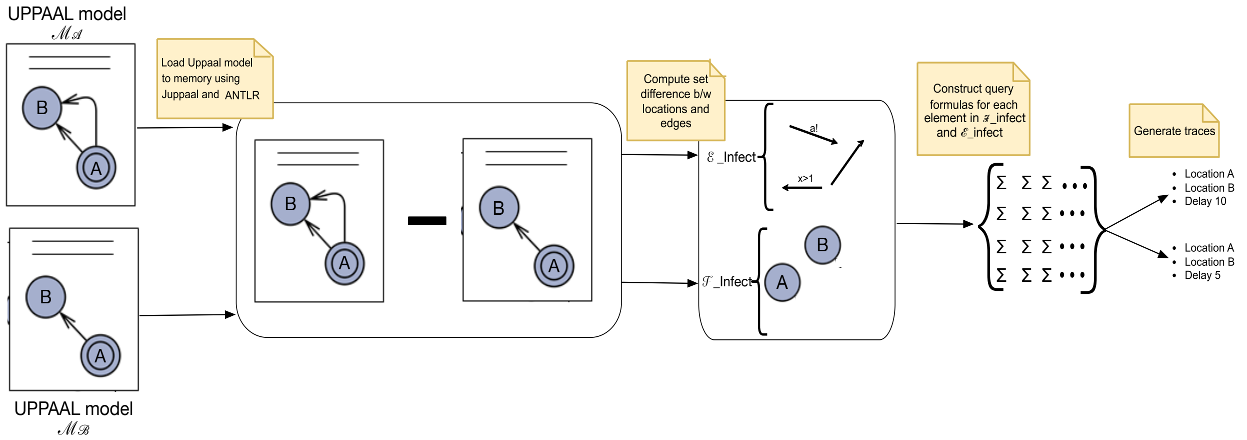
UPPAAL model
$\mathcal{M}_{\mathcal{A}}$
B
A
Load Uppaal model to memory using Juppaal and ANTLR
B
A
B
A
Compute set difference b/w locations and edges
$\mathcal{E}$ _Infect
a!
x>1
$\mathcal{F}$ _Infect
B
A
Construct query formulas for each element in $\mathcal{F}$_infect and $\mathcal{E}$_infect
Σ Σ Σ • • •
Σ Σ Σ • • •
Σ Σ Σ • • •
Σ Σ Σ • • •
Generate traces
Location A
Location B
Delay 10
Location A
Location B
Delay 5
B
A
UPPAAL model
$\mathcal{M}_{\mathcal{B}}$

FIGURE 11 Flow of the biased trace generator.

3.3.4 | Biased Algorithms

Algorithm 4 computes the set of infected locations and edges between any two mutants $\mathcal{M}_{\mathcal{A}}$ and $\mathcal{M}_{\mathcal{B}}$. The algorithm works as follows: in line 12, it computes the set of infected locations ($\mathcal{F}$). In line 15, it computes the set of infected edges ($\mathcal{E}$). At the first iteration in lines 18-27, it adds to $\mathcal{F}$ the source and target locations that are found in $\mathcal{E}$ and do not exist in $\mathcal{F}$.

Algorithm 5 describes the biased trace generation process of MUPPAAL. The algorithm works as follows: at the first iteration, it loops through every infected location, calling the CLI tool verifyta [14] with the generated reachability formula to said location for the mutant $\mathcal{M}_{\mathcal{B}}$ (lines 5-6). The second iteration (lines 9-18) performs the same process for every infected transition, however, the reachability formula is more comprehensive since we want a trace that goes exactly over each transition in $\mathcal{E}$. To do that, and because UPPAAL does not support nested quantifiers e.g. (A[], A<>, E[], E<>), we add to $\mathcal{M}_{\mathcal{B}}$ a boolean variable that is set to false by default and to true on edges going into the source location of *e* (line 12). Finally, the adjusted mutant $\mathcal{M}_{\mathcal{B}'}$ as well as the formula *f* are fed to verifyta, and the result is saved in $\mathcal{T}$ (lines 16-17).

```
Input: Two mutants $\mathcal{M}_\mathcal{A}$ and $\mathcal{M}_\mathcal{B}$
Output: A pair of sets of infected locations ($\mathcal{F}$) and infected edges ($\mathcal{E}$)
$\mathcal{F}$ ={}; // A set of infected locations
$\mathcal{E}$ ={}; // A set of infected edges
$\mathcal{L}_\mathcal{A}$ ={}; // A set of locations of mutant $\mathcal{M}_\mathcal{A}$
$\mathcal{L}_\mathcal{B}$ ={}; // A set of locations of mutant $\mathcal{M}_\mathcal{B}$
$\mathcal{T}_\mathcal{A}$ ={}; // A set of edges of mutant $\mathcal{M}_\mathcal{A}$
$\mathcal{T}_\mathcal{B}$ ={}; // A set of edges of mutant $\mathcal{M}_\mathcal{B}$

$\mathcal{L}_\mathcal{A}$ = $\mathcal{M}_\mathcal{A}$.addLocations();
$\mathcal{L}_\mathcal{B}$ = $\mathcal{M}_\mathcal{B}$.addLocations();
$\mathcal{F}$ = $\mathcal{L}_\mathcal{A}$.difference.removeAll($\mathcal{L}_\mathcal{B}$) // the set of infected locations.
$\mathcal{T}_\mathcal{A}$ = $\mathcal{M}_\mathcal{A}$.addTranstions();
$\mathcal{T}_\mathcal{B}$ = $\mathcal{M}_\mathcal{B}$.addTransitions();
$\mathcal{E}$ = $\mathcal{T}_\mathcal{A}$.difference.removeAll($\mathcal{T}_\mathcal{B}$) // the set of infected edges.

// We expand $\mathcal{F}$ by adding the sources and targets locations of edges in $\mathcal{F}$.
for ($e \in \mathcal{E}$ ) {
   source = $e$.getSource();
   target := $e$.getTarget();
   if (source $\notin \mathcal{F}$) {
      $\mathcal{F}$.add(source);
   }
   if (target $\notin \mathcal{F}$) {
      $\mathcal{F}$.add(target);
   }
}
return $pair(\mathcal{F}, \mathcal{E})$
```

Algorithm 4 Difference between two mutants.

Algorithm 6 describes the biased trace simulation process. The symbolic traces of the input mutant $\mathcal{M}_\mathcal{B}$ are extracted from $\mathcal{T}$ (line 3). After that, we iterate over every trace, translate them to UPPAAL TRON's format using the language recognition tool ANTLR4[44] and execute the tron CLI utility with the other mutant $\mathcal{M}_\mathcal{A}$ to see if it can simulate $\mathcal{M}_\mathcal{B}$'s trace (lines 5-6), according to the verdict either true (can simulate/inconclusive/timeout) or false (cannot simulate) is returned (lines 7-9).

As shown, the biased traces generation and simulation process is unidirectional, since we are computing a set difference ($\mathcal{M}_\mathcal{B} \setminus \mathcal{M}_\mathcal{A}$). To appropriately compare this approach to timed bisimulation one needs to cover both directions, that is why in practice we checked every pair of mutant twice, one for each direction ($\mathcal{M}_\mathcal{B}, \mathcal{M}_\mathcal{A}$), ($\mathcal{M}_\mathcal{A}, \mathcal{M}_\mathcal{B}$), i.e., every combination of two.

4 | EVALUATION

4.1 | 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.* [12] 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 mutant analysis. MUPPAAL is available on our companion website.

```
Input: The set of infected locations ($\mathcal{F}$) and edges ($\mathcal{E}$), and $\mathcal{M}_{\mathcal{B}}$ mutant
Output: An array with UPPAAL symbolic traces
$\mathcal{T} := \{\emptyset\}$    // The initially empty set of UPPAAL traces
for ($s \in \mathcal{F}$) {
    $f$ = "E<>"+$s$.getAutomatonName()+"."+$s$ // reachability formula to infected state
    $t$ = verifyta($\mathcal{M}_{\mathcal{B}}$, $f$)
    $\mathcal{T}$.append($t$)
}
for ($e \in \mathcal{E}$) {
    source = $e$.getSource()
    target = $e$.getTarget()
    $\mathcal{M}_{\mathcal{B}'}$ = addBoolGuardsToPath($\mathcal{M}_{\mathcal{B}}$)
    $f$ = "E<>"+$e$.getAutomatonName()+"."+boolVar+
              "&&" + "+$e$.getAutomatonName()+"."+boolGuard +
              "&&" + "+$e$.getAutomatonName()+"."+target
    $t$ = verifyta($\mathcal{M}_{\mathcal{B}'}$, $f$)
    $\mathcal{T}$.append($t$)
}
return $\mathcal{T}$
```

Algorithm 5 Biased traces generation.

```
Input: The set of symbolic traces ($\mathcal{T}$), mutants $\mathcal{M}_{\mathcal{A}}$, $\mathcal{M}_{\mathcal{B}}$
Output: A boolean ($b$) indicating whether $\mathcal{M}_{\mathcal{A}}$ can simulate all traces of $\mathcal{M}_{\mathcal{B}}$.
$\mathcal{T}_{\mathcal{M}_{\mathcal{B}}} := [N]$    // The set of N UPPAAL traces of $\mathcal{M}_{\mathcal{B}}$
$b$ := true
for ($t \in \mathcal{T}_{\mathcal{M}_{\mathcal{B}}}$) {
    $t'$ := uppaal2Tron($t$)
    s := $Uppaal\ Tron(\mathcal{M}, t')$ // string with UPPAAL-TRON's result.
    if (s == "TEST FAILED") {
        $b$ := false
    }
}

return $b$
```

Algorithm 6 Biased traces simulation.

4.2 | 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 [47]. 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 [48]. 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 [49]. 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 [47]. 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.

Tram Door (TD). The TD model represents the mechanism between a tram door and a retractable bridge (a.k.a. bridge plate) for wheelchair access to trams. The model has 31 locations in total, 7 clock constraints and many more synchronizations.

Scheduling Framework (SF). The SF model is part of a series of models written by some of the core UPPAAL maintainers, presenting modeling patterns for the schedulability analysis problem [50].

	GC	CA	TGC	CGC	TD	SF
TMI	13	9	14	36	10	6
TAD	501	26	179	1,625	566	168
SMI	12	2	12	27	14	5
SMI-NR	3	0	2	5	3	1
CXL	0	1	4	4	1	1
CXS	2	1	4	6	1	1
CCN	2	2	8	10	1	0
Total	533	41	222	1713	596	182

TABLE 3 Number of generated mutants per operator

4.3 | Research Questions

To evaluate the MUPPAAL workflow depicted in Figure 12, we consider the following research questions:

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

Figure 12 presents the experimentation workflow for our six systems (see Section 4.2). For each case, we first generate (step 1) a set of non-equivalent mutants ($\mathcal{MU}$) using the operators presented in Table 1 (Basile *et al.* [12]) and our novel operator SMI-NR. This results in 3287 mutants as presented in Table 3. Then, we independently apply our timed bisimulation, biased trace simulation, and random trace simulation algorithms on this set (steps 2.1, 2.2, and 2.3). In this journal version, we used an improved version of the timed bisimulation algorithm implementation presented in [15], which takes advantage of parallelism and includes other improvements in stability and performance. As a result, no comparison reached a timeout, and average execution times are greatly reduced compared to previous results [20, 16]. The details of this solution for timed bisimulation can be found in [16]. 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 × 86_64 GNU/Linux machine with 8 cores, 2.2 GHz, 32GB RAM.

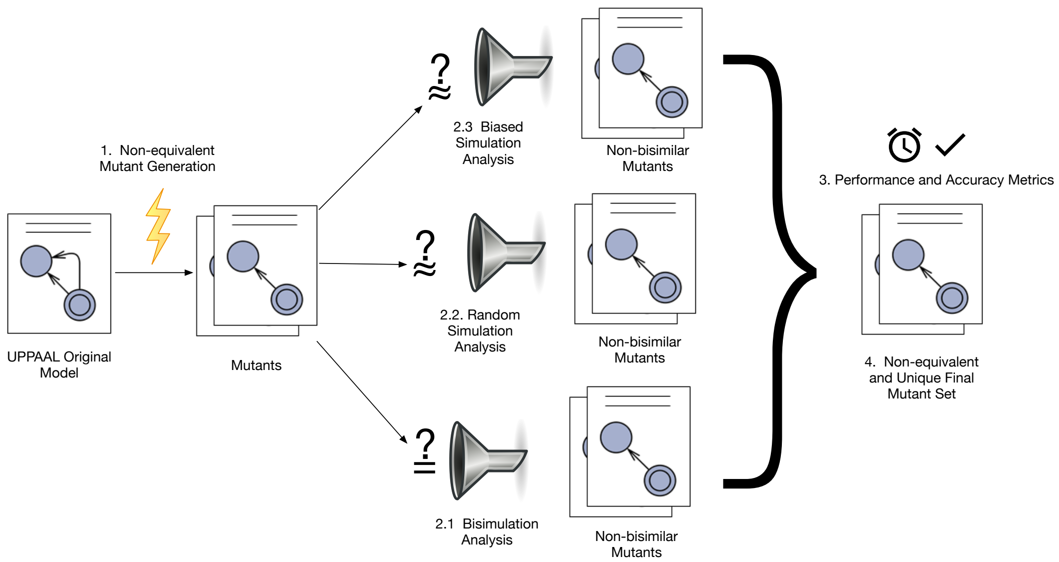
1. Non-equivalent Mutant Generation
UPPAAL Original Model
Mutants
2.3 Biased Simulation Analysis
2.2. Random Simulation Analysis
2.1 Bisimulation Analysis
Non-bisimilar Mutants
Non-bisimilar Mutants
Non-bisimilar Mutants
3. Performance and Accuracy Metrics
4. Non-equivalent and Unique Final Mutant Set

FIGURE 12 Experimentation Workflow

Case	GC	CA	TGC	CGC	TD	SF
	41/533	12/41	71/222	373/1,713	43/596	19/182
ratio Random 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)	514/596 (**st**=4.87)	163/182 (**st**=11.1)
ratio Random 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)	501/596 (**st**=2.75)	159/182 (**st**=10.3)
ratio Random 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)	497/596 (**st**=5.67)	156/182 (**st**=12.79)
ratio Biased Trace	93/533	25/41	87/222	450/1713	103/596	36/182

TABLE 4 Proportion of detected mutant duplicates. For random trace simulation, we report the average with standard deviation (**st**)

4.4 | Results and Discussion

4.4.1 | Answering RQ1.

Table 4 reveals that mutant duplicates represent up to 32% of the total number of mutants (for the TGC case), justifying the need for mutant 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*. In contrast, our biased simulation heuristic *is much more accurate than the random baseline*, identifying 1.8 times as many duplicates than the timed bisimulation algorithm (ground truth).

Furthermore, Figure 13 illustrates the accuracy of biased trace simulation in predicting whether a mutant was a duplicate. It was correct more than 60% of the time for all case studies and more than 80% of the time for 5 out of the 6 case studies considered. We believe this is a respectable result for an approach that is inherently incomplete and will likely never achieve full accuracy. Nevertheless, it demonstrates sufficient potential to justify testing with other case studies and within other formalisms. In the next section, we will examine the cost associated with achieving this level of accuracy.

4.4.2 | Answering RQ2.

Timed bisimulation is EXPTIME-complete, implying that some comparisons could exceed our computation budget. Thanks to our improved bisimulation algorithm, all the comparisons for all our studied systems could run without timeouts. Figure 16

Case	**BT (s)**	**BI (s)**	**TR (s)**
GC	0.038 (**st** = 0.008)	0.215 (**st**=0.17)	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)
CA	0.031 (**st** = 0.009)	0.178 (**st**=0.14)	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)
TGC	0.035 (**st** = 0.007)	0.199 (**st**=0.15)	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)
CGC	0.007 (**st** = 0.548)	0.633 (**st**=0.51)	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)
TD	0.004 (**st** = 0.001)	0.152 (**st**=0.10)	0.06 (**N**=2, **k**=100, **E**=10, st=0.02) 7.3 (**N**=10, **k**=1000, **E**=10, **st**=0.534) 31.7 (**N**=100, **k**=10000, **E**=10, **st**=3.54)
SF	0.003 (**st** = 0.002)	0.048 (**st**=0.11)	0.025 (**N**=2, **k**=100, **E**=10, st= 0.004) 0.034 (**N**=10, **k**=1000, **E**=10, **st**=0.001) 0.531 (**N**=100, **k**=10000, **E**=10, **st**=0.3)

TABLE 5 Average comparison time (s) and standard deviation (**st**) using bisimulation (BI), biased traces (BT), and traces (TR) with the number of traces (**N**), of runs (**E**), the units of time (**k**).

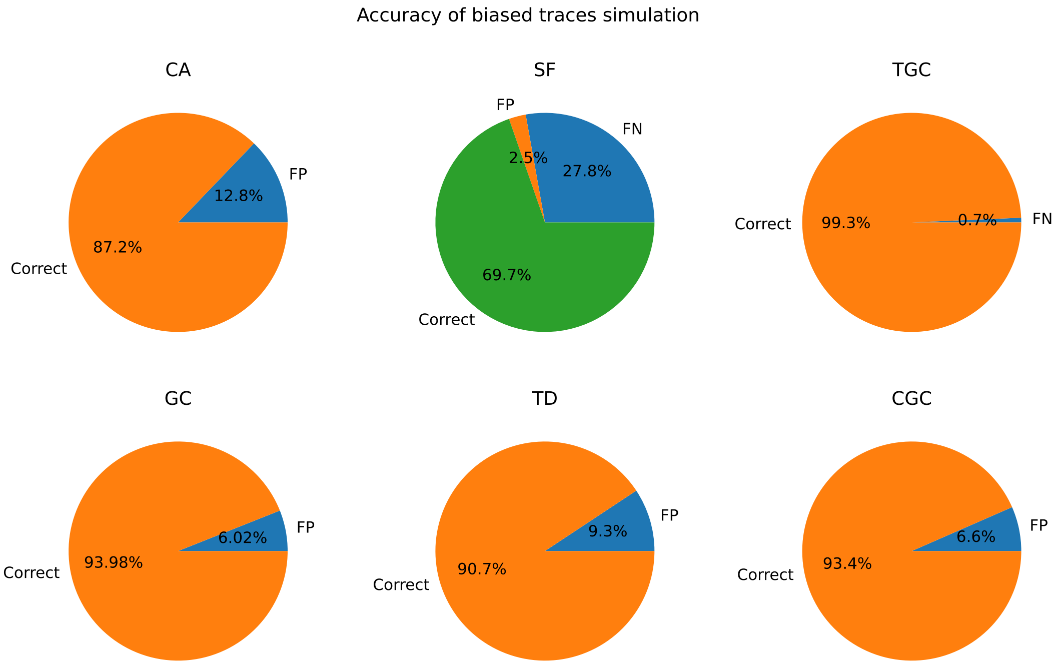

FIGURE 13 Accuracy of the biased traces simulation approach, using timed bisimulation as the ground truth. Showing false negatives (FN), false positives (FP), and true positives and negatives under the Correct label, for each case study.

indicates that the highest execution time to compare two mutants, which occurred for the CGC system is less than 1.5 seconds. We can observe that timed bisimulation is faster than the two largest configurations of the random simulation algorithm (N=10, N=100 in Table 5) for four out of the six systems. It is slower for the smallest configuration in five out of the six cases, where the random simulation only produces two traces. The biased simulation heuristic is faster than timed bisimulation for all the studied systems. By computing the ratio of average execution times between biased simulation and timed bisimulation, we can deduce

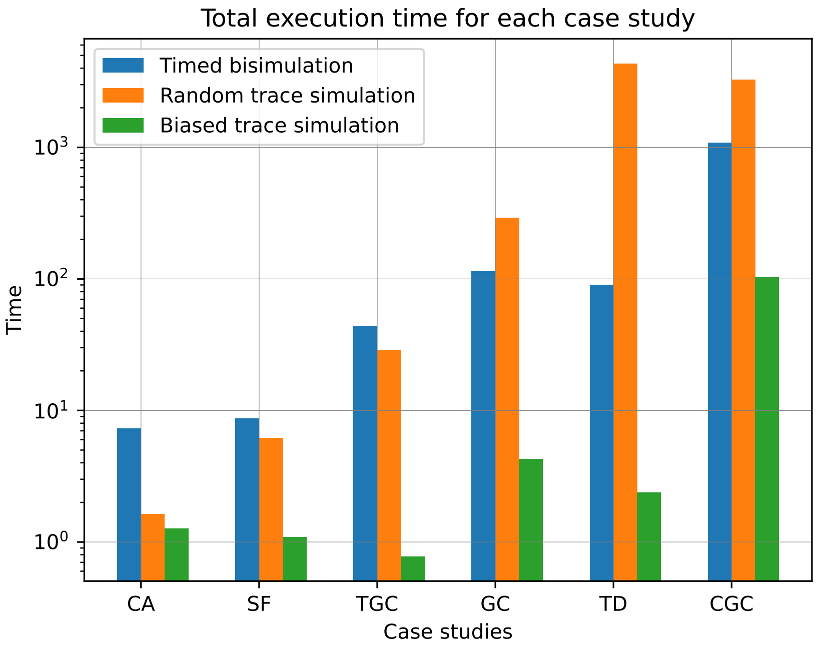

FIGURE 14 Log scale of the total time to generate and detect useless mutants per each case study and technique.

that the worst case (GC, see Table 5), the biased heuristic is 5.6 times faster, and in the best case (CGC) 90 times faster.

To better understand the benefit associated with the approaches shown here, we computed the total cost (in time) of our experimentation workflow per each case study and per each technique (Figure 14) ordered from smallest to biggest case study. Therefore, these numbers include both mutant generation and useless mutant detection costs. Because of these, and strategies to avoid some unnecessary comparisons (see Section 3.2), we cannot directly deduce such costs from pair evaluations shown in Table 5. Additionally, we use our data points to perform a linear regression and compute the graphs depicted on Figure 15. This figure extrapolates the values into three linear curves, expressed as functions $y = f(n)$ where n is the number of mutants, and y the total execution time. A logarithmic scale was used due to the significant variation between approaches and their respective time ranges (see Table 5 and Figure 16). Interestingly, random trace simulation starts showing diminishing returns after the number of mutants exceeds approximately 250, and becomes even more expensive than timed bisimulation. Conversely, biased trace simulation proves to be the least expensive of the three approaches. However, it is noteworthy that each curve exhibits two valleys at roughly the same data points. This suggests that although there is a natural trend for execution time to increase with the input size, the complexity, and behavior of the mutants themselves can also impact the total cost.

If we combine these observations with those of RQ1, we draw the following conclusions. First random simulation *performs poorly in all settings*, since it is ineffective at identifying duplicates and does not have impressive execution times compared to timed bisimulation. The biased simulation heuristic offers *speedy execution times and is much more effective at identifying duplicates than random simulations*. Timed bisimulation is slower, but being an exact algorithm, it is the only one that can give a *definitive verdict on the duplicates*. Thus, biased simulation and timed bisimulation *can work in synergy*. One can first use biased simulation to filter trivial duplicates very quickly and use timed bisimulation only on the *most difficult cases*. For example, for the gear control system, one can focus on the 93 duplicates to run bisimulations on instead of 206 or 432.

4.4.3 | Answering RQ3.

Table 6 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 6 show that SMI-NR produces unique mutants

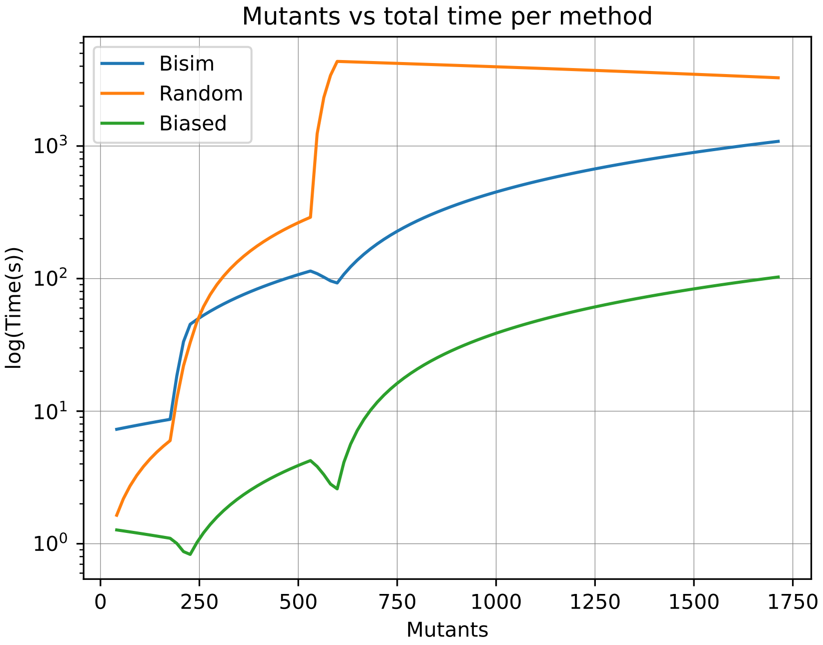

FIGURE 15 Extrapolation of the total time taken as a function of the number of mutants, per each technique.

	GC	CA	TGC	CGC	TD	SF
# SMI mutants	12	2	12	27	14	5
# SMI duplicates	9	2	10	22	11	4
# SMI-NR mutants	3	0	2	5	3	1
# Bisimilar pairs SMI-NR-SMI	3	0	2	5	3	1

TABLE 6 SMI-NR and SMI Operators Comparison

while preserving the behavior of the SMI operator. We conclude that the SMI-NR operator *offers a viable alternative to SMI, introducing the same faults while preventing duplicates*.

4.5 | Threats to Validity

4.5.1 | Internal Validity

To ensure the reliability of our mutation tool, MUPPAAL, we manually verified that each generated mutant conformed to the intended operator definitions. We validated these mutants within UPPAAL and, when necessary, employed bisimulation and refinement checks using external tools such as verifyta, UPPAAL-TRON [14], and TimBrCheck [51] to assess their equivalence to other models. Although this approach significantly reduced errors, we acknowledge that the tool may not be entirely free of bugs. Nonetheless, our method effectively identified and addressed numerous issues.

4.5.2 | 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.

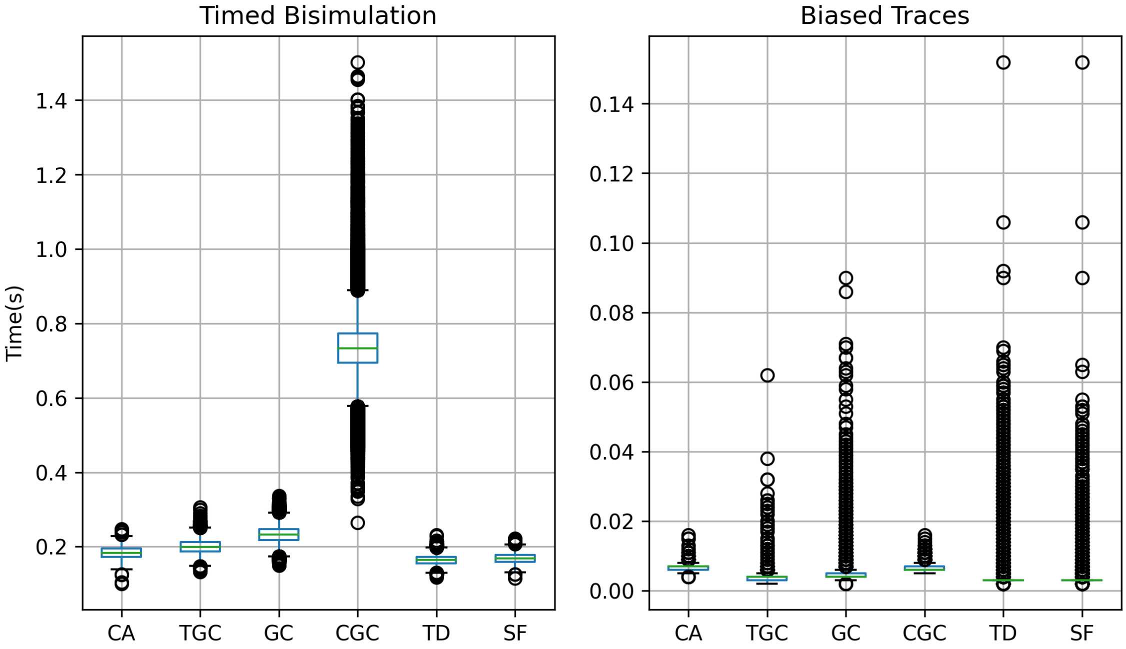

FIGURE 16 Distribution of the time(s) spent comparing any two mutants using timed bisimulation (left) and biased traces (right), for all case studies, over 10 runs.

4.5.3 | External validity

We cannot guarantee that our results extend to all timed systems expressed in UPPAAL. We selected six cases of different natures: a gear controller, a network communication model avoiding collisions, a train gate controller, a tram door, and a scheduling framework. These models have different sizes and numbers of clock constraints. They enabled us to observe differences in detecting and removing duplicate mutants. Our cases were enough to assess diversity regarding mutant types and their analysis times. Additionally, we cannot guarantee either that our results regarding the proportion of duplicates apply to all possible mutation operators. There are different proposals for timed operators [28], especially when they deal with another formalism (timed automata with tasks). For example, when considering operators for networks of timed automata [52], the number of duplicates can reach up to 95% for a certain combination of UPPAAL model and operator. However, the set of operators we have considered in this study are quite generic.

4.5.4 | Conclusion validity

We chose standard ways of measuring MUPPAAL's effectiveness, reporting means, and standard deviation for algorithms involving randomness regarding mutant detection rates and execution costs. We also provided accuracy charts covering false positives and false negatives compared to the ground truth obtained via our exact bisimulation algorithm.

5 | RELATED WORK

Several works cover the long-standing equivalent mutant problem [7, 17, 33, 4, 3]. Interest in the mutant duplicate problem is more recent [4], mostly at the code level [41]. MBMT gained traction more recently [53, 17, 54, 19]. Researchers applied MBMT for timed specifications [28, 29, 24, 12, 30, 55]. In [28], the authors present six mutation operators for TA, but do not guarantee the absence of equivalent or duplicate mutants. Aichernig *et al.* [24] design eight mutation operators for TA based on [29]. Again, these operators do not prevent generating equivalent or duplicate mutants. Basile *et al.* introduced six mutation operators for TS [12]. These mutation operators follow the same construction as those defined in [24, 29]. However, Basile *et al.* used a timed refinement technique to avoid the generation of equivalent mutants but do not address mutant duplicates [13, 12]. Larsen *et al.* defined a MBMT technique [30] on top of the UPPAAL-ECDAR verification tool [56]. It also uses refinement checking to eliminate equivalent mutants but does not address duplicates. Aichernig *et al.* designed a MBMT tool called MoMuT::TA [19]. 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 [39] is an addition to the UPPAAL environment. One can also use it to handle conformance tests on TS. UPPAL-TRON simulates the IUT with inputs deemed relevant by the model, monitors the outputs and checks the conformance of these against the behavior specified in the model. Hessel and Pettersson proposed a MBMT tool called Cover [57]. 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 [58, 59]. 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 [60].

6 | CONCLUSION

This paper presents the implementation and evaluation of MUPPAAL, a mutation framework for timed systems specified as UPPAAL automata. MUPPAAL focuses on getting rid of useless mutants, leveraging refinement techniques to avoid mutant equivalence [12, 13] and offers both reduction (via a dedicated mutation operator) and removal via parallelized bisimulation and fast biased simulation techniques for duplicate mutants. This framework allowed us to assess the duplicate mutant problem, largely ignored by previous studies, and to demonstrate that mutant duplicates can represent up to 32% of the total number of mutants for the six systems we considered. We also showed that bisimulation can analyze a pair of mutants in less than one second and that our biased simulation algorithm offers a compromise between speed and accuracy for five out of the six studied systems. We are continuously extending MUPPAAL, notably in designing mutation operators for timed automata networks [52].

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. Thanks to the multilateral agreement between the University of Namur and Universidad del Valle.

References

[1] Tretmans J. Model Based Testing with Labelled Transition Systems. In: Formal Methods and Testing – An Outcome of the FORTEST Network (Revised Papers Selection). Springer-Verlag; 2008. p. 1-38. Available from: http://dl.acm.org/citation.cfm?id=1806209.1806210.

[2] Zander J, Schieferdecker I, Mosterman PJ. Model-Based Testing for Embedded Systems. CRC Press; 2017.

[3] Jia Y, Harman M. An Analysis and Survey of the Development of Mutation Testing. IEEE Trans Softw Eng. 2011 Sep;37(5):649-78. Available from: http://dx.doi.org/10.1109/TSE.2010.62.

[4] Papadakis M, Jia Y, Harman M, Le Traon Y. 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. IEEE; 2015. p. 936-46.

[5] Papadakis M, Malevris N. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In: ISSRE. IEEE; 2010. p. 121-30.

[6] Fraser G, Arcuri A. Achieving scalable mutation-based generation of whole test suites. Empirical Software Engineering. 2014:1-30.

[7] Offutt J. A mutation carol: Past, present and future. Information and Software Technology. 2011 Oct;53(10):1098-107.

[8] Budd TA, Gopal AS. Program testing by specification mutation. Computer Languages. 1985 Jan;10(1):63-73.

[9] Howden WE. Reliability of the Path Analysis Testing Strategy. IEEE Transactions on Software Engineering. 1976;2(3):208-15.

[10] Voas JM, McGraw G. Software Fault Injection: Inoculating Programs Against Errors. New York, NY, USA: John Wiley & Sons, Inc.; 1997.

[11] Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M. Mutation Testing Advances: An Analysis and Survey. Advances in Computers. 2019:275-378.

[12] Basile D, ter Beek MH, Cordy M, Legay A. Tackling the equivalent mutant problem in real-time systems: the 12 commandments of model-based mutation testing. In: Lopez-Herrejon RE, editor. SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A. ACM; 2020. p. 30:1-30:11. Available from: https://doi.org/10.1145/3382025.3414966.

[13] Basile D, Beek MHt, Lazreg S, Cordy M, Legay A. Static detection of equivalent mutants in real-time model-based mutation testing. Empirical Software Engineering. 2022;27(7):160. Available from: https://doi.org/10.1007/s10664-022-10149-y.

[14] Behrmann G, David A, Larsen KG. A Tutorial on Uppaal. In: Bernardo M, Corradini F, 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. No. 3185 in LNCS. Springer–Verlag; 2004. p. 200-36.

[15] Ortiz JJ, Amrani M, Schobbens P. Multi-timed Bisimulation for Distributed Timed Automata. In: Barrett C, Davies M, Kahsai T, editors. NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings. vol. 10227 of Lecture Notes in Computer Science; 2017. p. 52-67. Available from: https://doi.org/10.1007/978-3-319-57288-8.

[16] Betancourt J, Ortiz J, Aranda J. Applying parallelism to a bisimulation algorithm to improve efficiency in software testing of time-critical systems. Ingeniería y Competitividad. 2023 sep;25(Suplemento):e–20713144. Available from: https://revistaingenieria.univalle.edu.co/index.php/ingenieria_y_competitividad/article/view/13144.

[17] Devroey X, Perrouin G, Papadakis M, Legay A, Schobbens PY, Heymans P. Model-based mutant equivalence detection using automata language equivalence and simulations. Journal of Systems and Software. 2018. Available from: https://www.sciencedirect.com/science/article/pii/S0164121218300475.

[18] Larsen KG, Mikucionis M, Nielsen B. Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: the 5th ACM international conference on Embedded software. ACM Press New York, NY, USA; 2005. p. 299 306. Available from: http://doi.acm.org/10.1145/1086228.1086283.

[19] Aichernig BK, Auer J, Jöbstl E, Korosec R, Krenn W, Schlick R, et al. Model-Based Mutation Testing of an Industrial Measurement Device. In: Tests and Proofs. vol. 8570 of LNCS. Springer; 2014. p. 1-19.

[20] Cuartas J, Aranda J, Cordy M, Ortiz J, Perrouin G, Schobbens PY. MUPPAAL: Reducing and Removing Equivalent and Duplicate Mutants in UPPAAL. In: 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 2023. p. 52-61.

[21] Alur R, Dill DL. A Theory of Timed Automata. Theor Comput Sci. 1994;126(2):183-235. Available from: https://doi.org/10.1016/0304-3975(94)90010-8.

[22] Bozga M, Daws C, Maler O, Olivero A, Tripakis S, Yovine S. Kronos: a model-checking tool for real-time systems. In: Hu, Vardi AJ, Y M, editors. Computer Aided Verification 10th International Conference, CAV'98. vol. 1427 of Lecture Notes in Computer Science. Vancouver, BC, Canada; 1998. p. 546-9. Available from: https://hal.archives-ouvertes.fr/hal-00374784.

[23] Henzinger TA, Ho PH, Wong-toi H. HyTech: A Model Checker for Hybrid Systems. Software Tools for Technology Transfer. 1997;1:460-3.

[24] Aichernig BK, Lorber F, Nickovic D. Time for Mutants - Model-Based Mutation Testing with Timed Automata. In: Veanes M, Viganò L, editors. Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings. vol. 7942 of Lecture Notes in Computer Science. Springer; 2013. p. 20-38.

[25] David A, Larsen KG, Legay A, Nyman U, Wasowski A. 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. New York, NY, USA: ACM; 2010. p. 91-100. Available from: http://doi.acm.org/10.1145/1755952.1755967.

[26] Kaynar DK, Lynch NA, Segala R, Vaandrager FW. The Theory of Timed I/O Automata, Second Edition. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers; 2010. Available from: https://doi.org/10.2200/S00310ED1V01Y201011DCT005.

[27] Cerāns K. Decidability of Bisimulation Equivalences for Parallel Timer Processes. In: von Bochmann G, Probst DK, editors. Proceedings of the 4th International Workshop on Computer Aided Verification (CAV'92). vol. 663 of Lecture

Notes in Computer Science. Springer-Verlag; 1993. p. 302-15.

[28] Nilsson R, Offutt J, Andler SF. Mutation-Based Testing Criteria for Timeliness. In: Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01. vol. 01 of COMPSAC '04. Washington, DC, USA; 2004. p. 306-11. Available from: http://dl.acm.org/citation.cfm?id=1025117.1025515.

[29] Hierons RM, Counsell S, AbouTrab M. Specification Mutation Analysis for Validating Timed Testing Approaches Based on Timed Automata. In: 2013 IEEE 37th Annual Computer Software and Applications Conference. Los Alamitos, CA, USA; 2012. p. 660-9. Available from: https://doi.ieeecomputersociety.org/10.1109/COMPSAC.2012.93.

[30] Larsen KG, Lorber F, Nielsen B, Nyman UM. Mutation-Based Test-Case Generation with Ecdar. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 2017. p. 319-28.

[31] Just R, Jalali D, Inozemtseva L, Ernst MD, Holmes R, Fraser G. 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. Hong Kong; 2014. p. 654-65.

[32] Hierons RM, Bowen JP, Harman M, editors. Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers. vol. 4949 of Lecture Notes in Computer Science. Springer; 2008. Available from: https://doi.org/10.1007/978-3-540-78917-8.

[33] Madeyski L, Orzeszyna W, Torkar R, Józala M. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. IEEE Transactions on Software Engineering. 2014;40(1):23-42.

[34] Adamopoulos K, Harman M, Hierons RM. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: Genetic and Evolutionary Computation–GECCO 2004: Genetic and Evolutionary Computation Conference, Seattle, WA, USA, June 26-30, 2004. Proceedings, Part II. Springer; 2004. p. 1338-49.

[35] Offutt AJ, Pan J. Automatically detecting equivalent mutants and infeasible paths. Software testing, verification and reliability. 1997;7(3):165-92.

[36] Kintis M, Papadakis M, Jia Y, Malevris N, Le Traon Y, Harman M. Detecting Trivial Mutant Equivalences via Compiler Optimisations. IEEE Transactions on Software Engineering. 2018;44(4):308-33.

[37] Tian Z, Shu H, Wang D, Cao X, Kamei Y, Chen J. Large Language Models for Equivalent Mutant Detection: How Far Are We? In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24); 2024. Available from: https://arxiv.org/abs/2408.01760.

[38] Kushigian B, Kaufman S, Featherman R, Potter H, Madadi A, Just R. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24); 2024. .

[39] Guldstrand Larsen K, Mikucionis M, Nielsen B. Uppaal Tron user Manual - docs.uppaal.org; 2017. Available from: https://docs.uppaal.org/extensions/tron/manual.pdf.

[40] Henzinger TA, Nicollin X, Sifakis J, Yovine S. Symbolic Model Checking for Real Time Systems. Information and Computation. 1994;111(2):193-244.

[41] Kurtz B, Ammann P, Offutt J, Kurtz M. 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. IEEE Computer Society; 2016. p. 142-51. Available from: https://doi.org/10.1109/ICSTW.2016.41.

[42] Baier C, Katoen JP. Principles of Model Checking. Mit Press. MIT Press; 2008.

[43] Jayapaul V, Munro JI, Raman V, Satti SR. Sorting and Selection with Equality Comparisons. In: Dehne F, Sack JR, Stege U, editors. Algorithms and Data Structures. Cham: Springer International Publishing; 2015. p. 434-45.

[44] Parr T. The Definitive ANTLR 4 Reference. 2nd ed. Raleigh, NC: Pragmatic Bookshelf; 2013.

[45] Bardin S, Delahaye M, David R, Kosmatov N, Papadakis M, Le Traon Y, et al. Sound and quasi-complete detection of infeasible test requirements. In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE; 2015. p. 1-10.

[46] Krenn W, Schlick R. Mutation-driven Test Case Generation Using Short-lived Concurrent Mutants–First Results. arXiv preprint arXiv:160106974. 2016.

[47] Lindahl M, Pettersson P, Yi W. Formal design and analysis of a gear controller. International Journal on Software Tools for Technology Transfer. 2001 Aug;3(3):353-68. Available from: https://doi.org/10.1007/s100090100048.

[48] Jensen H, Larsen K, Skou A. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. BRICS

Report Series. 2002 01;3.

[49] Alur R, Henzinger TA, Vardi MY. Parametric real-time reasoning. In: STOC. ACM; 1993. p. 592-601.

[50] David A, Illum J, Larsen KG, Skou A. Model-based framework for schedulability analysis using UPPAAL 4.1. In: Model-based design for embedded systems. CRC Press; 2018. p. 117-44.

[51] Luthmann L, Göttmann H, Bacher I, Lochau M. Checking Timed Bisimulation with Bounded Zone-History Graphs – Technical Report; 2020.

[52] Cortés D, Ortiz J, Basile D, Aranda J, Perrouin G, Schobbens PY. Time for Networks: Mutation Testing for Timed Automata Networks. In: Proceedings of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE). FormaliSE '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 44–54. Available from: https://doi.org/10.1145/3644033.3644378.

[53] Devroey X, Perrouin G, Papadakis M, Schobbens PY, Heymans P. Featured Model-based Mutation Analysis. In: International Conference on Software Engineering, ICSE. Austin, TX, USA; 2016. .

[54] Fabbri SCPF, Maldonado JC, Sugeta T, Masiero PC. Mutation Testing Applied to Validate Specifications Based on Statecharts. In: Proceedings of the 10th International Symposium on Software Reliability Engineering. ISSRE '99. Washington, DC, USA; 1999. p. 210. Available from: http://dl.acm.org/citation.cfm?id=851020.856195.

[55] Vega JJO, Perrouin G, Amrani M, Schobbens PY. Model-Based Mutation Operators for Timed Systems: A Taxonomy and Research Agenda. 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS). 2018.

[56] David A, Larsen K, Legay A, Nyman U, Wasowski A. ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In: Lecture Notes in Computer Science. vol. 6252/2010. Germany; 2010. .

[57] Hessel A, Pettersson P. Cover-a test-case generation tool for timed systems. Testing of software and communicating systems. 2007:31-4.

[58] Kim JH, Larsen KG, Nielsen B, Mikucionis M, Olsen P. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In: Núñez M, Güdemann M, editors. Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings. vol. 9128 of Lecture Notes in Computer Science. Springer; 2015. p. 47-61. Available from: https://doi.org/10.1007/978-3-319-19458-5_4.

[59] Gundersen TR, Lorber F, Nyman U, Ovesen C. Effortless Fault Localisation: Conformance Testing of Real-Time Systems in Ecdar. Electronic Proceedings in Theoretical Computer Science. 2018 Sep;277:147–160. Available from: http://dx.doi.org/10.4204/EPTCS.277.11.

[60] Siavashi F, Iqbal J, Truscan D, Vain J. In: Testing Web Services with Model-Based Mutation. Springer; 2017. p. 45–67.