MUPPAAL: Reducing and Removing Equivalent and Redundant Mutants in UPPAAL

Anonymous Author(s)

ABSTRACT

Mutation Testing (MT) is a test quality assessment technique that create mutants by injecting artificial faults to the system and evaluate the ability of tests to detect these mutants. We focus on the application of MT for safety-critical Timed Automata (TA), a special flavor of Model-based Mutation Testing (MBMT). As for other MT flavors, MBMT is prone to equivalent and duplicate mutants, respectively having the same behavior as the original system or one or more mutants. Such mutants are undesirable since they skew mutation scores and induce useless test case executions. We propose MUPPAAL, a holistic approach to useless mutant identification and removal, which combines three approaches: (1) it *avoids equivalent mutants* before generation through refinement and offers one new equivalence-avoiding mutation operator, (2) it *suggests duplicate mutants* using a timed simulation heuristic and (3) it *detects duplicate mutants* by using a novel timed bisimulation technique. Our evaluation on three cases shows efficient tradeoffs by combining strategies.

ACM Reference Format:
Anonymous Author(s). 2022. MUPPAAL: Reducing and Removing Equivalent and Redundant Mutants in UPPAAL. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Our society is relying on many safety-critical Timed Systems (TS) such as planes, trains, cars, or nuclear plants. TS are systems in which strict time constraints are essential for reliability and correctness. TS involve several components and must meet stringent quality requirements with limited computing resources. Quality assurance (QA) is therefore an essential activity in TS development. Indeed, disastrous consequences could result from insufficient QA.

Model-Based Testing MBT [46, 50] is an established QA method for complex systems, which attempts to reunite formal specification benefits (automated analyses at the model level) and conventional testing (uncovering bugs through predefined executions of the system) ones. MBT typically produces test cases from system requirements and an abstract model of the system behaviour. MBT enables controlled exploration of the system real-time behaviour while avoiding scalability issues induced by exhaustive verification. As for any testing technique, though, one must ensure that the test cases produced using MBT are effective at uncovering bugs.

Mutation Testing (MT) [26] is a technique for improving test suites by: (i) evaluating the effectiveness of the test suites [4, 18, 39]; and (ii) supporting the generation of additional tests to improve effectiveness [17, 39, 43]. It creates variants (named *mutants*) of the tested system by injecting one or more artificial simple defects (called mutations) into the system code or model. These mutations are obtained by applying predefined transformations – *mutation operators* – to code/model elements. Then, engineers can run tests on the system and on a mutant, and compare the obtained outputs: when different, the mutant is said to be *killed* (or *distinguished*) by these tests. The effectiveness of tests is then measured by the *mutation score*, i.e. the percentage of killed mutants over the total number of generated mutants.

Model-Based Mutation Testing (MBMT) combines the strengths of both MBT and MT. MBMT complements code-based mutation [17, 39, 43]. Aichernig *et al.*[1] report that model mutants lead to tests that can reveal implementation faults that were neither found by manual tests, nor by the actual operation of an industrial system. MBMT helps in the automatic identification of defects related to missing functionality and misinterpreted specifications [9] that are difficult to identify via code-based testing [24, 48].

However, a downside of MBMT (and MT) is that not all mutants are useful. Some of them may be *equivalent*, i.e. they exhibit the exact same behaviour as the original system in spite of their syntactic difference [42]. These mutants cannot be killed and therefore cannot lead to new test cases. Similarly, *duplicate* mutants are mutants that exhibit the same behaviour as other mutants [41, 42]. Preventing and removing such *useless* mutants reduce computation costs of MT and build more trust in mutation scores.

MUPPAAL addresses this challenge for TS specified in UPPAAL [7]. MUPPAAL combines three approaches that form the contributions of this paper:

(1) We *avoid equivalent mutants* before generation by ensuring that mutants do not refine the original system [6]. We have implemented a set of mutation operators for Timed Automata (TA) [1, 6, 40] which automatically create a non-equivalent mutants set. We also designed a novel timed mutation operator, guaranteeing non-duplicate mutants.
(2) We *suggest duplicate mutants* by using automated behavioural testing. More precisely, we randomly explore the behaviour space of the mutants – i.e. we randomly generate timed traces. If for a mutant we can find a trace that the other mutant cannot execute, then we can conclude that the two mutant are not duplicates. The heuristic suggests a pair of mutants as duplicates when no trace in the allocated simulation budget was able to distinguish one mutant in the pair from the other.
(3) We *detect duplicate mutants* using a novel approach that applies a timed bisimulation algorithm [40] to assess behavioural equivalence between two mutants. When duplicate mutants are detected, we keep only one of them.

Taken together, our three contributions enable the efficient identification of equivalent and duplicate mutants. Our overarching strategy is to apply these three methods successively, such that each method filters out a significant proportion of equivalent/duplicate mutants. For example, our behavioural testing method is faster than timed bisimulation, whose complexity class is EXPTIME-complete [10]). MUPPAAL generates timed traces using the

UPPAAL execution engine for TAs. Then it executes these timed traces on mutants using UPPAAL-TRON [34]. UPPAAL-TRON is an automated test generation tool which allows simulating the execution of the system by running traces in a traceable manner. Also, UPPAAL-TRON allows checking if the traces can be accepted by the mutants or models. We evaluate MUPPAAL on three case studies. We show the benefits of applying trace generation before bisimulation, reducing the use of the latter by up to 98% and execution time by up to 16 times. Yet, these benefits are utterly dependent on the heuristic parameters, whose optimal combinations are not stable across case studies, which is not the case of the bisimulation algorithm. MUPPAAL allows the exploration of such tradeoffs. We also show that our novel operator can eliminate up to 83% of the duplicate mutants. MUPPAAL implementation and full results of its evaluation are available on an anonymous website: https://anonymous.4open.science/r/Muppaal-91A7.

The remainder of this paper is organized as follows. Section 2 covers the background of our work. Section 3 presents our approach. Section 4 presents our MUPPAAL tool. Section 5 presents three case studies modelled in UPPAAL and used to illustrate our approach and tool. Section 6 presents our experimental results and validity threats. Section 7 presents related work, and Section 8 wraps up with concluding remarks and future work.

2 BACKGROUND

In this section, we present concepts that we use throughout the paper: timed traces, Timed Automata (TA), and Timed Input/Output Automata (TIOA). Additionally, we present the relevance of the refinement check and timed bisimulation for equivalent and redundant mutation, respectively.

2.1 Timed Traces and Timed Automata

The set of all *finite actions* over a finite alphabet Σ is denoted by Σ^*. Let $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ be the sets of natural, real and non-negative real numbers. A *timed trace* [3] over Σ is a finite sequence $\theta = ((\sigma_1, t_1), (\sigma_2, t_2) \dots (\sigma_n, t_n))$ of actions paired with non-negative real numbers (i.e., $(\sigma_i, t_i) \in \Sigma \times \mathbb{R}_{\geq 0}$) such that the timestamped sequence $t = t_1 \cdot t_2 \cdots t_n$ is non-decreasing (i.e., $t_i \leq t_{i+1}$). Timed Automata (TA) [3] are one of the most studied formalisms for modelling TS, which is successfully used in several model checkers such as UPPAAL [7],KRONOS [8], HYTECH [21]. TA are an extension of Finite State Automata (FSA) with a set of real-valued clocks increasing at the same rate. The real-valued clocks in a TA can be reset, which means updating the clock value to zero. This formalism allows enabling or disabling transitions using conditions on the values of the clocks and the actions can only be taken if the conditions are satisfied (i.e., *clock constraints*). Several extensions of TA have also been considered: TA with Inputs/Outputs (TAIO) [13] partition the actions into two disjoint sets for inputs and outputs and TA have been combined with a task model (TAT) [38]. In this paper, we use the TAIO.

2.2 Timed Input/Output Automata

A TIOA is a refined TA where the interaction between a system and its environment is modelled by using output and input actions [29]. The *clock constraints* are defined below.

Definition 1 (Clock constraints). *Let X be a finite set of clock names. A clock constraint $\phi \in \Phi(X)$ is a conjunction of comparisons of a clock with a natural constant c, given by the following grammar, where ϕ ranges over constraints $\Phi(X)$, $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, >, \leq, \geq, =\}$:*

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

Definition 2 (Clock valuations). *A clock valuation $v \in \mathbb{R}^X_{\geq 0}$ over X is a mapping $v : X \to \mathbb{R}_{\geq 0}$. For a time value $d \in \mathbb{R}_{\geq 0}$, we note $v + d$ the valuation defined by $(v + d)(x) = v(x) + d$. Given a clock subset $Y \subseteq X$, we note $v[Y \leftarrow 0]$ the valuation defined as follows: $v[Y \leftarrow 0](x) = 0$ if $x \in Y$ and $v[Y \leftarrow 0](x) = v(x)$ otherwise.*

Clocks and their constraints are used by TIOA to formulate behaviours by simulating TS. For this purpose, the transitions can be labelled with a *guard* that will allow the transitions to be taken or not, performing actions and resetting clocks. In TA, actions (or alphabet) are classified into two disjoint subsets of external observable and internal controllable input actions and external observable and internal controllable output actions according to the definition of [29]. The external actions of a TIOA can be shared with any other TIOA (behaviour observable from outside) but internal actions cannot be shared (behaviour not observable from outside). To represent the TIOA in UPPAAL, we call every internal action as *silent actions*, and are denoted by τ. We adapt the definition of [29], where the initial location is unique, variables are omitted, and the internal actions are the τ action. The formal definition of TIOA is:

Definition 3 (TIOA). *A TIOA is a tuple $(L, l_0, X, \Sigma_I, \Sigma_O, T, I)$, where:*

- *L is a finite set of locations,*
- *$l_0 \in L$ is a set of initial locations,*
- *X is a finite set of clocks,*
- *Σ_I is a finite set of input actions ($?$) and $\tau \notin \Sigma_I$,*
- *Σ_O is a finite set of output actions ($!$) and $\tau \notin \Sigma_O$, and $\Sigma = \Sigma_I \cup \Sigma_O$, is a finite set of internal actions, such that $\Sigma_I \cap \Sigma_O = \emptyset$. We denote by $\Sigma_\tau = \Sigma \cup \{\tau\}$ as the set of controllable not visible actions, and We denote by $\Sigma_C = \Sigma_O \cup \Sigma_\tau$ as the set of controlled actions,*
- *$T \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a finite set of edges called transitions, An edge is denoted by (l, a, ϕ, Y, l'),*
- *$I : L \to \phi(X)$ is a function that associates to each location a clock constraint called an invariant.*

For a transition $(l, a, \phi, Y, l') \in T$, we classically write $l \xrightarrow{a,\phi,Y} l'$ and call l and l' the source and target location, ϕ is the guard, a the action or alphabet, Y the set of clocks to be reset. The semantics of a TIOA is defined by a Timed Input/Output Transition System (TIOTS) where a *state* is a pair $(l, v) \in l \times \mathbb{R}^X_{\geq 0}$, where l denotes the current state with its accompanying clock valuation v, starting at (l_0, v_0) where v_0 maps each clock to 0. The transitions can be described in two ways. *Delay transitions* only let time pass without changing location. We only consider *legal* states, i.e. states that satisfy the current state of the invariant$v \vDash I(l)$. Discrete transitions instead, occur between a source location and a target location. The transition may occur only if the current clock values satisfy both the guard located in the transition and the invariant of the target location.

Definition 4 (Semantics of TIOA). *Let $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, T, I)$ be a TIOA. The semantics of TIOA $\mathcal{A}$ is given by a TIOTS$\mathcal{A}) = (S, s_0, \Sigma_I, \Sigma_O, \rightarrow)$ where:*

- $S \subseteq L \times \mathcal{V}(X)$ *is the set of states,*
- $s_0 = \{(l_0, v) | \ v(x) = 0$ *for all* $x \in X\}$,
- $\Sigma_\Delta = (\Sigma_I \cup \Sigma_O \cup \{\tau\}) \cup \mathbb{R}_{\geq 0}$,
- $\rightarrow \subseteq S \times \Sigma_\Delta \times S$ *is a transition relation defined by the following two rules:*
  - ***Discrete transition:*** $(l, v) \xrightarrow{a} (l', v')$, *for* $a \in \Sigma_\Delta$ *iff* $l \xrightarrow{a,\phi,Y} l'$, $v \vDash \phi$, $v' = v[Y \leftarrow 0]$ *and* $v' \vDash I(l')$ *and,*
  - ***Delay transition:*** $(l, v) \xrightarrow{d} (l, v + d)$, *for some* $d \in \mathbb{R}_{\geq 0}$ *iff* $v + d \vDash I(s)$.

A *path* in $\mathcal{A}$ is a finite sequence of consecutive delay and discrete transitions. A finite execution fragment of $\mathcal{A}$ is a path in TIOTS($\mathcal{A}$) starting from the initial state $s_0 = (l_0, v_0)$, with delay and discrete transitions alternating along the path: $\rho = (l_0, v_0) \xrightarrow{t_1} (l_0, v'_0) \xrightarrow{a_1} (l_1, v_1) \xrightarrow{t_2} (l_1, v'_1) \xrightarrow{a_2} (l_2, v_2) \ldots (l_{n-2}, v_{n-2}) \xrightarrow{t_{n-1}} (l_{n-2}, v'_{n-2}) \xrightarrow{a_{n-1}} (l_{n-1}, v_{n-1})$ where $v(x) = 0$ for every $x \in X$. An execution of $\mathcal{A}$ is *initial* if $s_0 = (l_0, v_0) \in$, S and *maximal* if it ends in a final state, or infinite. The set of all *timed traces* induced by $\mathcal{A}$ is denoted $\mathcal{L}(\mathcal{A}) = \mathcal{L}$(TIOTS($\mathcal{A}$))). We consider only deterministic TIOA.

Example 1. Let $\mathcal{A} = (L, l_0, X, \Sigma, T, I, F)$ be the TIOA depicted in Figure 1. $\mathcal{A}$ contains two locations: l_0 (being the initial one) and l_1. The actions in the transitions are subfixes with external input action (?) and external output actions (!) In particular, l_0 is the only location to define an invariant: $I(l_0) = (x < 7)$, forcing the TIOA to exit l_0 before x becomes 7. l_1 has a true invariant (thus not drawn), allowing to stay in l_1 forever. Suppose the current location is l_1. The transition $l_1 \xrightarrow{b?,(y=9),\{x:=0;y:=0\}} l_0$ specifies that when the external 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 until when it may be possible to stay in a given location) is different from using a guard (which specifies when the transition is enabled).

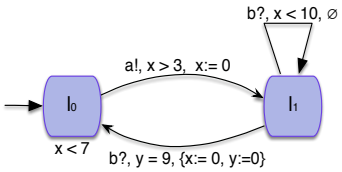


Figure 1: A TIOA with two clocks x and y.

Definition 5 (Deterministic TIOA.). *A TIOA $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, T, I)$ is deterministic iff $|l_0| = 1$ and for all locations l, for all actions $a \in \Sigma_I \cup \Sigma_O$, for every pair of edges of the form $(l, \phi_1, a, Y_1, l'_1)$ and $(l, \phi_2, a, Y_2, l'_2)$, the clock constraints ϕ_1 and ϕ_2 are mutually exclusive ($\phi_1 \wedge \phi_2$ is unsatisfiable).*

2.3 Timed Bisimulation and Trace Simulation

2.3.1 Trace-Based Simulation. *Simulation* is one of the most used and widely used techniques for performance analysis. However, *simulation* is not sufficient to prove the absence of errors in safety-critical systems, because they are not exhaustively analysed. Therefore, a better approach to verify the correctness of safety-critical systems is combining *simulation* and *formal verification* techniques [49]. One of the most used techniques in *simulation* is through the use of *traces*. A *simulation trace* is a sequence of information (i.e., external events) that have been collected during the simulation execution. A *simulation trace* in TIOA can be viewed as a sequence of states (system states), actions and time delay.

2.3.2 Timed Bisimulation. The State explosion problem [11] is one of the most serious problems treated by model checking [5], notably on TS where the number of states can be enormous. One of the most effective and successful techniques currently available to alleviate the state explosion problem is the generation of state space equivalents with respect to some desired properties of the system. Timed bisimulation relation [10] has been used to reason about the equivalent behaviour on the TS.

Definition 6 (**Timed Bisimulation** [10]). *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two TIOTS over the set of actions $\Sigma = \Sigma_I \cup \Sigma_O$. Let $S_{\mathcal{D}_1}$ (resp., $S_{\mathcal{D}_2}$) be the set of states of $\mathcal{D}_1$ (resp., $\mathcal{D}_2$). A timed bisimulation over TIOTS $\mathcal{D}_1$, $\mathcal{D}_2$ is a binary relation $\mathcal{R} \subseteq S_{\mathcal{D}_1} \times S_{\mathcal{D}_2}$ such that, for all $s_{\mathcal{D}_1} \mathcal{R} s_{\mathcal{D}_2}$, the following holds:*

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

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

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

2.4 Mutant Operators and Equivalent Problem

2.4.1 TA and Mutant Operators. Nilsson *et al.* [37] were among the first to extend the TA with a task model and mutant operators for testing the behaviour of TS. Nilsson *et al.* proposed six mutant 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, and *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.* [23] and Aichernig *et al.* [2] also proposed some mutant operators in UPPAAL for testing the behaviour of TS. Three of them are not time-related: *change action (CA)*, *change source/-target (CS)* 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 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*

Nilsson *et al.* [37]		Aichernig *et al.* [2]		Basile *et al.* [6]	
Operators	**Description**	**Operators**	**Description**	**Operators**	**Description**
ET	Execution time	**CA**	Change action	**TMI**	Transition missing
IAT	Inter-arrival time	**CS1**	Change target	**TAD**	Transition ADd
PO	Pattern offset	**CS2**	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.

invariant (CI) adds one time unit to the invariant constraint in an automaton location, *change invariant (CI)* and *invert reset (IR)* are two operators specific to TA.

Basile *et al.* [6] proposed some mutant operators for testing the behaviour of TS. In [6], the TS are modelled as TIOA, in which input actions are defined controllable and output actions are defined uncontrollable [6]. The mutant operators proposed in [6] avoid the generation of mutants that are subsumed by construction. Three of six mutant operators in [6] are not time-related and were also proposed by Fabbri *et al.* [16]: *transition missing (TMI)* removes a transition, *transition add operator (TAD)* adds a transition between two states, *state missing (SMI)* removes a state (other than the initial state) and all its incoming/outgoing transitions. The other three time-related operators are: *constant exchange L (CXL)* increases the constant of a clock constraint, *constant exchange S (CXS)* decreases the constant of a clock constraint and *clock constraint negation (CCN)* negates a clock constraint. The main idea in [6] is to perform a refinement check between the mutant and the system model, using ECDAR tool [32]. Table 1 shows the mutation operators retrieved from the considered contributions.

2.4.2 Equivalent/Duplicate mutation problem. MT is one of the most effective coverage criterion to evaluate the quality of a test suite [27, 42]. In addition, several recent empirical studies have evaluated the effectiveness and efficiency of MT [28, 41, 42]. However, despite being an effective and efficient technique to evaluate the quality of tests, MT is not widely used due to its high costs. Equivalent and duplicate mutants (i.e., useless mutants [41]) contribute to increasing costs [42]. Furthermore, recent empirical studies have described that the appearance of equivalent and duplicate mutants could reach between 60% and 70% (i.e., between 30% - 40% of equivalent mutants [36] and between 20% - 30% of duplicate mutants [42]) of total mutants generated. In this paper, we focus on minimizing the problem of the high cost of mutation testing by eliminating equivalent and duplicate mutants of the analysis.

2.5 UPPAAL and UPPAAL-TRON

UPPAAL is a tool environment for modelling, simulation and verification of TS depicted by a network of TA which is extended with structured data types, user functions, double and integer variables, local and global real-valued variables (clocks), and communication synchronous channels (channels are labelled by ! as emitting and ? as receiving). This tool has been used successfully in several case studies, as previously reported in [7]. UPPAAL-TRON [19] is a testing tool, based on UPPAAL, suited for online black-box conformance testing of TS. The modelling, simulated, and verification are performed in the UPPAAL model-checker, and UPPAAL-TRON is used for testing the Implementation Under Test (IUT). UPPAAL-TRON uses a randomized online testing algorithm, which is an extension of the UPPAAL model checker [7]. UPPAAL-TRON could generate and execute tests event by event in real-time through stimulating and monitoring the IUT. These two functions are performed by UPPAAL-TRON, computing the possible set of symbolic states based on the *timed trace* observed so far. A *timed trace* in UPPAAL-TRON consists of a sequence of input or output actions and time delay [34]. UPPAAL-TRON has been applied in different dominions such that rail-road intersection controllers, light controllers and electronic thermostat regulators [34].

3 OVERCOMING EQUIVALENT AND DUPLICATE MUTANT PROBLEM

At the moment, three methods have been proposed to tackle the equivalent (which can also be used to duplicate) mutant problem [36] : (1) to avoid equivalent mutants, (2) to suggest equivalent mutants, and (3) to detect equivalent mutants. In this section, we focus on addressing the equivalent/duplicate mutant problem using the three methods presented above. Then, we describe the timed mutation operators, which avoid the generation of certain equivalent and duplicate mutants (i.e., mutant generation and selection approach), and MBMT Analysis. An MBMT tool should be able to limit the generation of certain equivalent and duplicate mutants.

3.1 Avoiding Equivalent and Duplicate Mutants

The design of mutation operators is decisive to the effective functioning of a MBMT tool. The MBMT tool must be able to generate as few mutants as possible without losing efficiency (i.e., generate a set of mutants without equivalent and duplicate mutants). However, in [1, 23, 37], mutants are generated undistinguished, which means that the variation rules applied by the operators do not avoid the generation of equivalent or duplicate mutants. Here, we use the technique presented in [6] to avoid equivalent mutants. Using this technique, we have implemented a set of mutation operators for TIOA compatible with UPPAAL [1, 6]. We also propose to improve the model variation rules present in some mutation operators to avoid duplicate mutants. We introduce and implement new mutation operators for TIOA compatible with UPPAAL. Following, we formally define mutation operators.

3.1.1 Mutation Operators and Non-Equivalent Mutants. Here, we use guidelines and the six mutation operators introduced in [6] (see Table 1). Like [1], we redefine from [6] their mutation operators using the TIOA. Then, from our perspective, a mutation operator ($\mathcal{M}_\mu$) is a function that generates a set of mutants for a given specification (here a specification can be a TIOA). Each mutant in $\mathcal{M}_\mu$(TIOA) is generated by injecting only one fault in such a way that it is always well-formed. Following the guidelines from [6], a TIOA (mutant) is well-formed if the following conditions are met:

- All locations l are reachable,
- Every transition is executable in, at least, one timed trace,
- It is timelock-free,
- It is deterministic.

Definition 7. *Let $\mathbb{A}$ be a set of all possible TIOA and $\mathbb{B}$ be a set of all well-formed TIOA. A mutation operator is a function $\mathcal{M}_\mu: \mathbb{B} \rightarrow 2^{\mathbb{A}}$ where there is a possibility of generating non-well-formed TIOA.*

we use μ to refer to each specific operator presented in [6].

3.1.2 Our new Mutation Operator and Non-duplicate Mutants. Here, we introduce our new mutation operator (SMI-NR) to avoid duplicate mutants because with the six mutation operators introduced in [6] (see Table 1), equivalent mutants can be avoided before the generation, but duplicate mutants cannot be avoided. For example, the TMI, and SMI operators can potentially generate duplicate mutants. Figures 2, 3, and 4 illustrate a TIOA and duplicate mutants that the operators could generate. The duplicate mutant problem has been proposed by [28, 41]. As to be said before, a duplicate mutant has the same behaviour as another mutant, which means that one of the two mutants is redundant. Here, we introduce our operator for TIOA compatible with UPPAAL which automatically create a set of non-duplicate mutants. The following proposition considers the case when a TMI mutant and a SMI mutant are timed bisimilar.

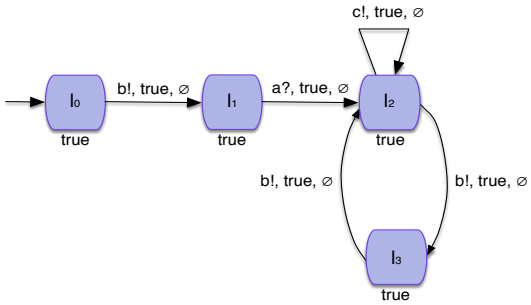


Figure 2: A TIOA.

Theorem 2 (TMI and SMI redundant mutants). *Let $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, E, I, F)$ be a well-formed TIOA and the mutants $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ where:*

$$\mathcal{A}_{tmi} \in \mathcal{M}_{tmi}(TIOA) \text{ where}$$
$$\mathcal{A}_{tmi} = (L, l_0, X, \Sigma_I, \Sigma_O, E \setminus \{e_{tmi}\}, I, F),$$
$$e_{tmi} = (l_1, a, \phi, Y, l_2) \in E \text{ and } a_{tmi} \in \Sigma_I, \text{ and}$$

$$\mathcal{A}_{smi} \in \mathcal{M}_{smi}(TIOA) \text{ where}$$

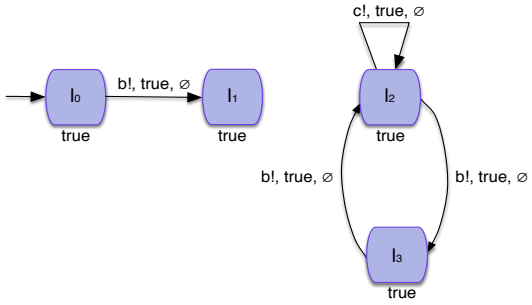


Figure 3: A mutant generated by TMI operator.

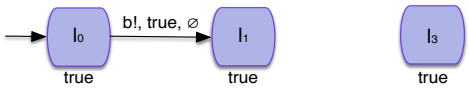


Figure 4: A mutant generated by SMI operator.

$$\mathcal{A}_{smi} = (L \setminus \{l_{smi}\}, l_0, X, \Sigma_I, \Sigma_O, E \setminus \{E_{smi}\}, I, F),\ l_{smi} \in L,$$
$$l_{smi} \neq l_0, E_{smi} = \{(l_1, a, \phi, Y, l_2) \in E \mid l_{smi} \neq l_1 \text{ and } l_{smi} \neq l_2\}$$

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

However, due to the high cost of verifying the condition present in Theorem 2, which would avoid duplicate mutants between the TMI and SMI operators, we describe another condition in Theorem 3 that avoids some duplicate mutants using the Breadth-first search algorithm in polynomial time.

Theorem 3 (Redundant mutants). *Let $\mathcal{A} = (L, l_0, X, \Sigma_I, \Sigma_O, E, I, F)$ be a well-formed TIOA and the mutants $\mathcal{A}_{tmi}$ and $\mathcal{A}_{smi}$ where:*

$$\mathcal{A}_{tmi} \in \mathcal{M}_{tmi}(TIOA) \text{ where}$$
$$\mathcal{A}_{tmi} = (L, l_0, X, \Sigma_I, \Sigma_O, E \setminus \{e_{tms}\}, I, F),$$
$$e_{tmi} = (l_1, a, \phi, Y, l_2) \in E \text{ and } a_{tmi} \in \Sigma_I, \text{ and}$$

$$\mathcal{A}_{smi} \in \mathcal{M}_{smi}(TIOA) \text{ where}$$
$$\mathcal{A}_{smi} = (L \setminus \{l_{smi}\}, l_0, X, \Sigma_I, \Sigma_O, E \setminus \{E_{smi}\}, I, F),\ l_{smi} \in L,$$
$$l_{smi} \neq l_0, E_{smi} = \{(l_1, a, \phi, Y, l_2) \in E \mid l_{smi} \neq l_1 \text{ and } l_{smi} \neq l_2\}$$

Then, if each path to the location l_{smi} has the same last transition, then $\mathcal{A}_{tmi} \sim \mathcal{A}_{smi}$.

Since Theorem 3 is not a biconditional proposition, some duplicate mutants can not be avoided before the generation with this condition. Some of these TIOA's are illustrated in figures 5, and 6.

3.2 Suggesting Duplicate Mutants

In the previous section, we presented an approach to overcome equivalent and duplicate mutants. Let us remember that two mutants are duplicates if they have the same behaviour (timed bisimilar). However, because our operators cannot discover all duplicate mutants before the generation (i.e., after running the test on the mutants, there are still surviving duplicate mutants). Here, we present an approach to suggest duplicate mutants by using automated behavioural simulation. We develop and implement our approaches in

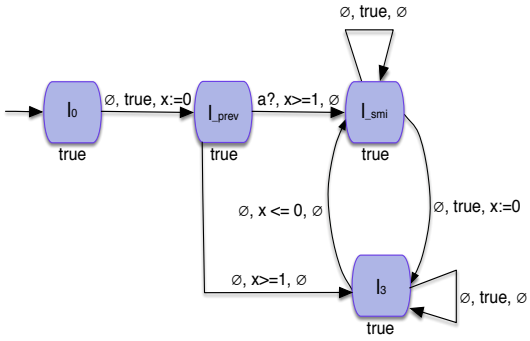


Figure 5: Removes transition.

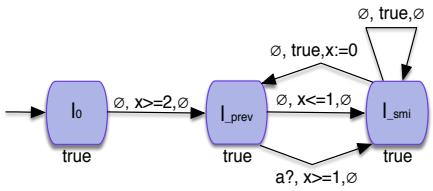


Figure 6: Removes location.

an MBMT tool called MUPPAAL. MUPPAAL uses the tools UPPAAL and UPPAAL-TRON to automatically suggest duplicate mutants. To better guide the suggestion of duplicate mutants, our tool first generates several (random) *timed traces* using UPPAAL, then executes these *timed traces* on mutants (using UPPAAL-TRON [34]).

3.2.1 Random Simulation Process. To suggest the duplicate mutants, 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 [14]. The aim of these traces is to check whether they are accepted by the mutants (i.e., whether the mutants are able to simulate the actions and delays). If a trace fails to run on one of the models, we will know that it is not possible for mutants to be bisimilar. However, if all traces are accepted, then the mutants are considered probably bisimilars (i.e., we cannot say anything about bisimilarity) and we need to use another formal method (timed bisimulation algorithm) to verify the bisimilarity between the mutants. In fact, our aim is to exclude a considerable number of non-equivalent mutants to have a reduced set of comparisons to which a formal method could be efficiently applied.

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 others [19]. In order to do the trace simulations on UPPAAL-TRON our translator tool uses ANTLR [44] to parse and translates the traces from UPPAAL into a *preamble* file and a *trace* file required by UPPAAL-TRON to check the actions and delays.

UPPAAL-TRON needs two files to monitor an execution: (1) the preamble that provides the required definitions to configure and prepare UPPAAL-TRON for test execution of the trace, and (2) the trace interpretation, which is a sequence of actions and delays to check if the model can execute.

For instance, the tool process between a pair of mutants (*Automaton* $\mathcal{A}$ and *Automaton* $\mathcal{B}$) to verify **N** random traces. First it

```
Input: A mutant M_1 ∈ M
A number of traces to generate N
A simulation time k
Output: An array with UPPAAL-TRON traces

T=[N]   // The set of N UPPAAL traces
T'=[N]  // The set of N UPPAAL-TRON traces
for(i=0;i<N; i++){
    //The seed for the pseudo-random generator
    r = random();
    //Get random trace from UPPAAL
    T[i] =VerifyTA(M_1, k, r);
    //Translate trace to UPPAAL-TRON format
    tree=parser(lexer(T[i]));
    T'[i] = tree.format();
}
return T'
```

Algorithm 1: Trace generation.

```
Input: A set of mutants MU
A number of traces to generate N
A simulation time k
Output: A list of probably bisimilar mutants
// The set of N UPPAAL-TRON traces per mutant
T'=[MU.size()][N]
for(i=0; i<MU.size(); i++){
    T'[i]=TraceGeneration(MU[i], N, k)
}
PBM = [()] // List of possible bisimilar pairs
for(i=0; i<MU.size(); i++){
    for(j=i+1; j<MU.size(); j++){
        for(k=0; k<N; k++){
            Pass1=Tron.check(MU[i], T'[j,k]);
            Pass2=Tron.check(MU[j], T'[i,k]);
            if(Pass1∧Pass2)
              PBM=PBM.add((MU[i], MU[j]));
        }
    }
}
return PBM
```

Algorithm 2: Trace Simulation Algorithm.

takes the *Automaton A* to generate traces using UPPAAL, the tool reads them parsering the trace t_i and with that data, it builds the preamble and per each t_i it makes a t_i' file with the UPPAAL-TRON format. Once traces are created, the tool uses UPPAAL-TRON to check if the *Automaton* $\mathcal{B}$ is able to execute the trace, returning as output "Passed" or "Filed". We use this information to avoid later the timed bisimulation check for every pair of mutants in the set. Hence, we know that all pairs that were not able to simulate their random traces are not bisimilar, afterwards we know it is required the bisimulation equivalence just among the pair of mutants that were able to simulate each other.

3.2.2 Random Simulation Algorithms. Below, we present the algorithms used to generate the random traces and suggest the probably bisimilar mutants. Algorithm 1 describes the random trace generation process of our tool MUPPAAL. The algorithm works as follows: at the first iteration, it computes the N random traces for the first mutant present in $\mathcal{M}$ (line 12) with a simulation time k and a random number generator r. In addition, the random traces generated from UPPAAL format are translated to UPPAAL-TRON format (line 15). The algorithm repeats until N random traces are generated for all mutants in $\mathcal{M}$ with a simulation time k and a random number generator r. Algorithm 2 describes the random trace simulation process of our tool MUPPAAL and uses Algorithm 1 for computing their UPPAA-TRON traces. The algorithm works as follows: at the first iteration, it checks the N random traces generated by the second mutant $\mathcal{MU}[j]$ on the first mutant $\mathcal{MU}[i]$ (line 16) and reciprocally (line 18). In addition, if a pair of mutants are probably bisimilar, $\mathcal{PBM}$ is actualized with the two mutants (line 20). At the second iteration, it is checked the N random traces generated by the other two following mutants in $\mathcal{MU}$ and so on up to the n_{th} iteration.

3.3 Detecting Duplicate Mutants

In the previous section, we presented an approach to suggest duplicate mutants. However, because our approach cannot detect all duplicate mutants after the generation, here, we present an approach to detect duplicate mutants (after the suggestion of probably bisimilar mutants) by using *timed bisimulation algorithm* [10, 40]. However, the computational cost of using *timed bisimulation* is usually high, which hinders its use in industry. Mainly, the computational cost is due to the process of comparing the behaviours of two models (i.e., the original model and mutants) and both functional and timing behaviour present in safety-critical TS. Furthermore, using this technique can be costly since empirical studies report that between 20% and 30% of all generated mutants are duplicates [36, 41]. However, this number can be increased to 30%, because the techniques proposed by [41] do not detect all duplicate mutants [31]. Timed bisimulation technique is decidable in EXPTIME [10]. Therefore, reducing the number of duplicate mutants before the detection could be interesting to reduce the computational cost of applying timed bismulation. Algorithm 3 describes the process to detect bisimilar mutants from a list of probably duplicate mutants with the relief of a timed bisimulation algorithm [40]. The algorithm works as follows: at the first iteration, it checkes if the pair of the two first mutants are bisimilar uses the BisimilarAlgo (line 7), $\mathcal{MU}^c$ is actualized with one of the two bisimilar mutants (line 9). For each pair of mutants in $\mathcal{PBM}$, it assesses whether the two mutants are bisimilar and update $\mathcal{MU}^c$" accordingly.

4 MUPPAAL TOOL

In this section, we present the functionalities and details related to the implementation of our MUPPAAL tool. MUPPAAL tool has the purpose of generating mutants and mutation score for UPPAAL tool. We used *Java* 8 version to develop MUPPAAL. Since UPPAAL GUI is a software with Java as a requirement, our tool run with the same JRE. Thus, we choose JAVA SE Development Kit 8 to build the tool. In the implementation of our tool, we use the command design pattern

```
1  Input: A list of probably bisimilar mutants PMB
2  A set of mutants MU
3  Output: A set of no duplicate mutants
4  MU^c ={} A set of complement mutants
5  for(i=0; i<PBM.size(); i++){
6    //Execute timed bisimulation algorithm
7    if(!BisimilarAlgo(PBM[i], PBM[i + 1])){
8      // Remove any of the two bisimilar mutants
9       MU^c.remove(PBM[i])
10     }
11 }
12 MU = MU – MU^c
13 return MU
```

Algorithm 3: Bisimuation Process.

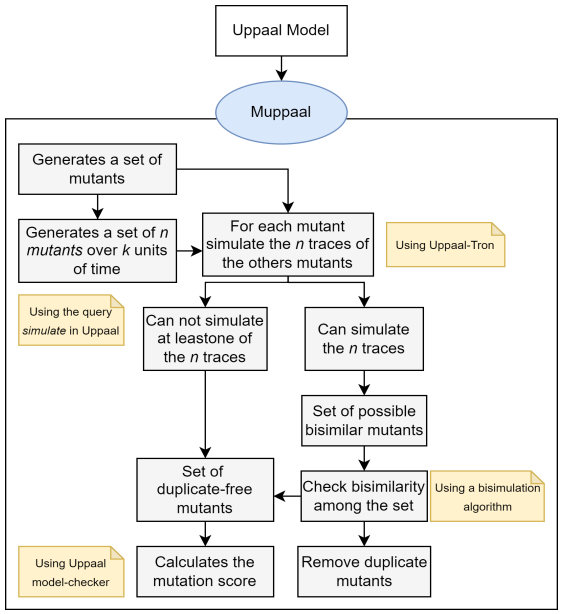


Figure 7: MUPPAAL Tool.

[51] to encapsulate each mutant operator. This architecture makes it possible to adapt the chosen operators in our tool. Furthermore, it is convenient due to its scalability to help us to implement new mutant operators and required data structures. We build our parser with the ANTLR tool to read the XML files generated by UPPAAL, which will be used by our tool to identify the system elements, and build mutants valid according to the syntax of UPPAAL. The flow of our MUPPAAL tool is depicted below in Figure 7.

The inputs of our tool are: (1) UPPAAL-Model file which has the XML format, (2) Properties file, which can be omitted if the properties are inside the UPPAAL model, and (3) the UPPAAL model checker (the program path to verifyta), which allows the verifier functionality accessible through command line. First, the tool takes the UPPAAL model and parses the file using the ANTLR library to generate n mutant files with the correct XML file. MUPPAAL tool is able to only generate the mutants. However, to calculate *the mutation score* is necessary to have the program path to *verifyta* as input to check the properties with every mutant and report

the mutation score as output. The mutation score is calculated, taking the number of killed mutants divided by the number of total mutants ($Mutation\ score = \frac{Killed\ mutants}{Total\ mutants}$). Any mutant that satisfies the properties is an alive mutant, whilst any mutant that does not satisfy the properties is a killed mutant. The code source and datasets of our MUPPAAL tool are available at https://anonymous.4open.science/r/Muppaal-91A7

5 CASE STUDIES

In this section, we describe the case studies that we used for the evaluation of MUPPAAL. For each case study, we consider one automaton from the network of automata. The UPPAAL specifications of these cases are available at https://github.com/farkasrebus/XtaBenchmarkSuite.

- *Case 1: Gear Control (GC):* The GC models a simple gear controller for vehicles [35]. 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 2 transitions have guards of the form $x < c$ and 2 transitions have guards of the form $x \geq c$, for some clock x and constant c.

- *Case 2: Collision Avoidance (CA):* The CA case models a protocol where different agents want to get access to Ethernet through a shared channel [25]. The CA model has 6 states and 12 transitions, of which 9 transitions have guards of the form $x == c$ and 4 transitions have guards of the form $x < c$, for some clock x and constant c.

- *Case 3: Train Gate Controller (TGC):* The TGC models a railway system which controls access to a bridge for several trains [7]. The bridge is a shared resource that may be accessed only by one train at a time. The TGC model has 14 states and all of them have invariants. All invariants are of the form $x < c$ for a clock x and constant c. There are 18 transitions, of which 4 transitions have guards of the form $x < c$ and 4 transitions have guards of the form $x > c$, for some clock x and constant c.

6 EVALUATION

We evaluate how MUPPAAL strategies help in avoiding equivalent mutants and identifying duplicate mutants.

6.1 Research Questions

We consider the following research questions:

- **RQ1** How many comparisons are effectively made and avoided by removing duplicate mutants?
- **RQ2** What is the impact on performance of timed traces vs timed bisimulation to detect duplicate mutants?
- **RQ3** How effective is our operator *SMI-NR* in avoiding duplicate mutants?

To answer these questions, we consider the three case studies presented in section 5. For each case, we apply the following procedure. First, we generate a set of mutants ($\mathcal{MU}$) from the model using the operators presented in Table 1 (Basile *et al.* [6]) and our novel operator SMI-NR. Second, we generate traces for six parameterisations of **N** and **k** using Algorithm 1. Third, we apply the trace simulation heuristic (Algorithm 2) followed by the exact timed bisimulation approach (Algorithm 3) and measure execution times as well as the number of comparisons we saved from using the more expensive timed bisimulation algorithm. We ran each parameterisation 10 times to mitigate randomness effects. Finally, by analysing mutants bisimilarity, we evaluate the effectiveness of our mutation operator (SMI-NR. We ran our experiments on a UBUNTU 21.10 × 86_64 GNU/Linux machine with 16 cores, 2.2 GHz, 32Gb RAM. The MUPPAAL tool and its evaluation are available: https://anonymous.4open.science/r/Muppaal-91A7.

6.2 Results and Discussion

Here, we present the results, and we answer the research questions. We generated 541 (GC), 41 (CA), and 222 (TGC) mutants (see Table 2). We have applied all mutants operators presented in [6] and our mutation operator (SMI-NR) to avoid duplicates. For each case study, we have used **N** number of traces and **k** units of time (see Table 3).

6.2.1 Answering RQ1. Table 2 presents: *(i)* the total number of mutants generated (#M), *(ii)* the number of comparisons performed (#CP) by the bisimulation algorithm and, *(iii)*, the number of comparisons avoided (#CA) for bisimulation via application of the trace simulation algorithm. Comparisons are 2-wise, meaning that there are $\frac{M * (M-1)}{2}$ evaluations for M mutants. For the sake of space, we only show the best, an intermediate and the worst parameterisation in Table2, but all details are available on the MUPPAAL companion website. The last row shows the total values, which are (1) the sums of the numbers of respective mutants in a given column and (2) the sums of the numbers of respective comparisons performed by removing duplicate mutants (3) the sums of the numbers of respective comparisons avoided by removing duplicate mutants. From Table 2 we can see that the total number of all comparisons made vs avoided is 59.7% vs 40.3% (GC), 58.5% vs 41.5% (CA) and 1.3% vs 98.7% (TGC). The gain is particularly notable for TGC where simulations are able to identify many non-duplicate pairs. We owe this result to the fact that TGC has invariants on all its states: slight changes have immediate consequences.

6.2.2 Answering RQ2. Table 3 presents the total number of mutants generated (#M), the total number of comparisons (#TC), the execution time (s) using timed trace algorithm (TR), using timed bisimulation algorithm (BI) and, using timed traces algorithm and timed bisimulation algorithm (TR+BI) on the 3 case studies. From Table 3 we can see the execution time per case study of the studied algorithms (columns TR and BI). The timed trace approach (TR) is faster than timed bisimulation approach (BI) in TGC (97%, 94% and 78% TGC), but in GC and CA TR is faster than BI for **N** = 2, **k** = 100 or 1000 and **N** = 2, **k** = 100 (89%, 31% GC and 92%, 80% CA). We can also see that it takes less time to use the combination of the two approaches (column TR + BI) than to use only BI in TGC (94%, 92% and 76% TGC), but in GC and CA TR+BI is faster than BI for **N** = 2, **k** = 100 (10% CA) and for **N** = 5, **k** = 100 (18% CA). In terms of execution times, there is always a winning parameterisation of TR+BI against bisimulation alone, though it varies across cases. Runtime gains are linked with the number of comparisons avoided thanks to trace simulation heuristics: as for RQ1, TGC allows the most notable differences between TR and TR+BI strategies. Figure 8 presents the execution time per case study of the algorithms (BI and TR+BI).

	GC			CA			TGC		
Operators	**#M**	**#CP**	**#CA**	**#M**	**#CP**	**#CA**	**#M**	**#CP**	**#CA**
TMI	13	4,046	2,792	9	234	90	14	208	2,795
TAD	501	84,451	56,831	26	402	313	178	295	23,290
SMI	12	3,008	3,310	2	59	20	12	24	2,562
SMI-NR	3	978	615	0	0	0	2	2	439
CXL	0	0	0	0	0	0	4	0	878
CXS	2	816	244	2	9	70	4	0	878
CCN	2	401	662	2	50	20	8	0	1,740
Totals	533	84,689	57,089	41	480	340	222	333	24,198

Table 2: The number of mutants (#M), number of comparisons performed (#CP), number of comparisons avoided (#CA).

Case study	#M	#TC	TR (s)	BI (s)	TR+BI (s)
GC	533	141,778	3,212 (**N**=2, **k**=100, **E**=10, st=3.1) 7,847 (**N**=5, **k**=100, **E**=10, st=3.4) 39,980 (**N**=10, **k**=1000, **E**=10, st=3.9)	39,493 (st = 6.0)	46,561 (**N**=2, **k**=100, **E** = 10, st=3.8) 32,568 (**N**=5, **k**=100, **E**=10, st=4.6) 90,612 (**N**=10, **k**=1000, **E**=10, st=4.3)
CA	41	820	38 (**N**=2, **k**=100, **E**=10, st=2.0) 248 (**N**=2, **k**=1000, **E**=10, st=2.3) 1,101 (**N**=10, **k**=1000, **E**=10, st=3.3)	359 (st=3.8)	324 (**N**=2, **k**=100, **E** = 10, st=3.0) 534 (**N**=2, **k**=1000, **E**=10, st=2.5) 1,353 (**N**=10, **k**=1000, **E**=10, st=3.9)
TGC	222	24,531	446 (**N**=2, **k**=100, **E**=10, st=4.8) 934 (**N**=10, **k**=100, **E**=10, st=2.5) 3,396 (**N**=10, **k**=1000, **E**=10, st=4.1)	15,020 (st=3.1)	932 (**N**=2, **k**=100, **E**=10, st=3.5) 1138 (**N**=10, **k**=100, **E**=10, st=3.9) 3,600 (**N**=10, **k**=1000, **E**=10, st=4.6)

Table 3: The total number of mutants (#M), and of comparisons (#TC), the average execution time (s) using traces (TR), using bisimulation (BI), using traces and bisimulation (TR+BI), the number of traces (N), and of runs (E), the units of time (k), and the standard deviation (st).

6.2.3 Answering RQ3. Table 4 presents the number of acutally bisimilar mutants found in the sets of mutants for each case study. Thus, we show the results of our work with particular cases, where it is evident that the mutants generated by SMI-NR are bisimilar with some SMI mutants, since the set of SMI-NR is a subset of SMI. We can also at the same time denote that all the mutants generated by the SMI operator are bisimilar with either SMI-NR or TMI, which shows us that adding SMI-NR effectively makes the use of SMI unnecessary when using the TMI operator, since SMI-NR generates the subset of SMI mutants that are bisimilar with TMI. From Table 2 (GC) we can see that 12 mutants were generated with the SMI operator, where all of them are bisimilar with SMI-NR or with TMI. Further, the set of SMI-NR presents the subset of SMI that are not bisimilar to TMI and its use would avoid the generation of 9 bisimilar mutants out of the 12 created (75%). From Table 2 (CA) we can see that 2 mutants were generated with the SMI operator, which are completely bisimilar with TMI, and in this case SMI-NR does not generate mutants since there is no subset of SMI that are not bisimilar with TMI, and its use would avoid the generation of 2 bisimilar mutants of the 2 created (100%). And finally, from Table 2 (TGC) we can see that 12 mutants were generated with the SMI operator, where all of them are bisimilars with SMI-NR or with TMI, while SMI-NR presents the subset of SMI that are not bisimilars to TMI, and its use would avoid the generation of 10 bisimilar mutants out of 12 created (83.3%). We can deduce that our novel SMI-NR operator, in addition to avoid duplicates, is also successfully able to capture the behaviour of the original SMI operator.

Bisimilar pairs	GC	CA	TGC
SMI-SMI-NR	3	0	2
SMI-TMI	9	2	10
TAD-TAD	20	1	9
Total bisimilar pairs	**32**	**3**	**21**

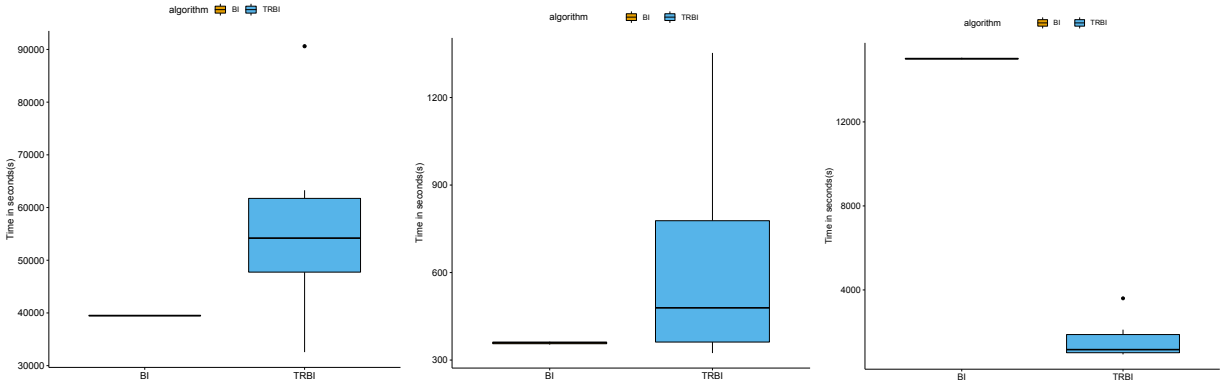
Table 4: Bisimilar mutants (Characterized by operator).

6.3 Threats to Validity

6.3.1 Internal validity. We selected three cases of different nature: a gear controller, a network communication model avoiding collisions, and a train gate controller. These models have different size and different numbers of clock constraints, which enabled to observe difference in detection and removal of redundant mutants. For better reproducibility, these cases are freely available online: https://github.com/farkasrebus/XtaBenchmarkSuite.

6.3.2 Construct validity. The **N** and **k** values for trace generation and simulation have been chosen to cover different situations. We cannot guarantee that our parameter values are relevant for any case study. They will rather depend on the model size, the number of clocks, the considered number of traces (**N**) unit of time (**k**), and the maximum time allowed for the bisimulation analysis. We ran each comparison 10 times to mitigate randomness effects.

6.3.3 External validity. We cannot of course guarantee that our results extends to all timed systems expressed in UPPAAL. Yet, we recall that the diversity of the selected cases were enough to show various tradeoffs between our proposed strategies, from having to use bisimulation in very rare occasions (1%) to a majority of the cases (60%).



(a) GC **(b) CA** **(c) TGC**

Figure 8: Execution time of the BI and TR+BI algorithms

7 RELATED WORK

The *equivalent mutant problem* is a long-standing one. Previous works have been focused on this problem and surveys covering this topic have been published [14, 26, 36, 39, 41]. The duplicate mutant problem is more recent [41], but it has only been addressed at the code level and not at the model level. MBMT gained traction more recently [1, 14–16] and some works and surveys in MBMT for TA and safety-critical TS exist [2, 6, 23, 32, 37, 47]. In [37], six mutation operators for TA with a task model were introduced, but the generation does not guarantee the absence of equivalent or duplicate mutants. In [2], eight mutation operators for TA were introduced, which are an extension of the standard mutation operators defined in [23]. However, these eight mutation operators also do not guarantee the non-generation of equivalent or duplicate mutants. In [2, 23, 37], the authors also do not propose solutions to avoid, suggest or detect equivalent and duplicates mutants. In [6], six mutation operators for TS were introduced. These mutation operators follow the same construction as those defined in [2, 23]. Furthermore, in [6] they use a timed refinement technique to avoid the generation of equivalent mutants, but they do not propose solutions to avoid, suggest or detect duplicate mutants. In [32] the problem of generating valid test cases for TS was tackled. Furthermore, mutants and the original model are modelled as TIOA in UPPAAL-ECDAR [12] and a refinement check technique is used between mutants and the original model. UPPAAL-ECDAR is a tool for compositional design and verification of TS. Again, their work focus on equivalent mutants while we focus on duplicates ones.

There are also some tools for MBMT using TA. Aichernig *et al.* designed a MBMT tool called MoMuT::TA [1]. MoMuT::TA maps TA to a formal semantics and performs a conformance check between mutants and the original model to generate test cases automatically. In [33] UPPAAL-ECDAR is used to unbounded conformance checks and obtain faster results than [1]. The tool UPPAL-TRON [19] has been added to UPPAAL environment. It can also be used to handle conformance tests on TS. UPPAL-TRON stimulates the IUT with input that is deemed relevant by the model, and it monitors the outputs and checks the conformance of these against the behaviour specified in the model. Hessel and Pettersson proposed a MBMT tool called Cover [22]. Cover generates test-cases based on TA and Timed Computation Tree Logic (TCTL). The properties written in TCTL can be used for verifying the test model. Similar approaches exist [20, 30]. μUTA introduces a test generation method which can be used to derive mutants from the specification and executes them via online testing. It was designed for robustness testing of web services [45].

8 CONCLUSION

In this paper, we proposed a set of algorithms to overcome the equivalent and duplicate mutant problem. This issue affects mutation testing in general, causing unnecessary and costly test executions and diminishing overall confidence in mutation scores. We exploited formal techniques to overcome this problem for safety-critical systems timed specifications. Our tool, MUPPAAAL, offer a duplicate-avoiding mutation operator in addition to supporting existing ones [6]. It also realises two algorithms to respectively approximate equivalent/duplicate mutant detection and to formally establish equivalence via a timed bisimulation algorithm. Our evaluation on three case studies demonstrate that the combination of algorithms is more efficient than applying timed bisimulation alone and that the new SMI-NR operator avoids duplicates while capturing the original SMI behaviour.

There is room for improvement. First, we would like to optimise our trace simulation algorithm to exploit syntactic differences between mutants, which could speed up non-duplicate detection and avoid additional comparisons required by the bisimulation algorithm. Another area of future work is to propose a full catalog of redudant-avoiding operators by design and extend mutations to networks of timed automata.

REFERENCES

[1] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. 2014. Model-Based Mutation Testing of an Industrial Measurement Device. In *Tests and Proofs (LNCS, Vol. 8570)*. Springer, 1–19.

[2] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. 2013. Time for Mutants - Model-Based Mutation Testing with Timed Automata. In *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7942)*, Margus Veanes and Luca Viganò (Eds.). Springer, 20–38. https://doi.org/10.1007/978-3-642-38916-0

[3] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theor. Comput. Sci.* 126, 2 (1994), 183–235. https://doi.org/10.1016/0304-3975(94)90010-8

[4] J H Andrews, L C Briand, Y Labiche, and A S Namin. 2006. Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. *Software Engineering, IEEE Transactions on* 32, 8 (2006), 608–624. https://doi.org/10.1109/TSE.2006.83

[5] C. Baier and J.P Katoen. 2008. *Principles of Model Checking*. MIT Press.

[6] Davide Basile, Maurice H. ter Beek, Maxime Cordy, and Axel Legay. 2020. Tackling the equivalent mutant problem in real-time systems: the 12 commandments of model-based mutation testing. In *SPLC '20: 24th ACM International Systems and Software Product Line Conference, Montreal, Quebec, Canada, October 19-23, 2020, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 30:1–30:11. https://doi.org/10.1145/3382025.3414966

[7] Gerd Behrmann, Alexandre David, and Kim G. Larsen. 2004. A Tutorial on Uppaal. In *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 (LNCS, 3185)*, Marco Bernardo and Flavio Corradini (Eds.). Springer–Verlag, 200–236.

[8] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. 1998. Kronos: a model-checking tool for real-time systems. In *Computer Aided Verification 10th International Conference, CAV'98 (Lecture Notes in Computer Science, Vol. 1427)*, Hu, Alan J.; Vardi, and Moshe Y. (Eds.). Springer, Vancouver, BC, Canada, 546–549. https://doi.org/10.1007/BFb0028779

[9] Timothy A. Budd and Ajei S. Gopal. 1985. Program testing by specification mutation. *Computer Languages* 10, 1 (Jan. 1985), 63–73. https://doi.org/10.1016/0096-0551(85)90011-6

[10] Kārlis Cerāns. 1993. Decidability of Bisimulation Equivalences for Parallel Timer Processes. In *Proceedings of the 4th International Workshop on Computer Aided Verification (CAV'92) (Lecture Notes in Computer Science, Vol. 663)*, Gregor von Bochmann and David K. Probst (Eds.). Springer-Verlag, 302–315.

[11] E. M. Clarke, W. Klieber, M. Novacek, and P. Zuliani. 2012. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification*, B. Meyer and M. Nordio (Eds.). Springer-Verlag.

[12] Alexandre David, Kim Guldstrand Larsen, Axel Legay, Ulrik Mathias Nyman, and Andrzej Wasowski. 2010. ECDAR: An Environment for Compositional Design and Analysis of Real Time Systems. In *Lecture Notes in Computer Science*, Vol. 6252/2010. Springer, Germany.

[13] Alexandre David, Kim G. Larsen, Axel Legay, Ulrik Nyman, and Andrzej Wasowski. 2010. 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* (Stockholm, Sweden) *(HSCC '10)*. ACM, New York, NY, USA, 91–100. https://doi.org/10.1145/1755952.1755967

[14] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2018. Model-based mutant equivalence detection using automata language equivalence and simulations. *Journal of Systems and Software* (2018), –. https://doi.org/10.1016/j.jss.2018.03.010

[15] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-based Mutation Analysis. In *International Conference on Software Engineering, ICSE*. ACM, Austin, TX, USA. https://doi.org/10.1145/2884781.2884821

[16] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Tatiana Sugeta, and Paulo Cesar Masiero. 1999. Mutation Testing Applied to Validate Specifications Based on Statecharts. In *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE '99)*. IEEE Computer Society, Washington, DC, USA, 210–. http://dl.acm.org/citation.cfm?id=851020.856195

[17] Gordon Fraser and Andrea Arcuri. 2014. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* (2014), 1–30. https://doi.org/10.1007/s10664-013-9299-z

[18] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2013. Comparing non-adequate test suites using coverage criteria. In *ISSTA*. ACM, 302–313.

[19] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. 2017. Uppaal Tron user Manual - docs.uppaal.org. https://docs.uppaal.org/extensions/tron/manual.pdf

[20] Tobias R. Gundersen, Florian Lorber, Ulrik Nyman, and Christian Ovesen. 2018. Effortless Fault Localisation: Conformance Testing of Real-Time Systems in Ecdar. *Electronic Proceedings in Theoretical Computer Science* 277 (Sep 2018), 147–160. https://doi.org/10.4204/eptcs.277.11

[21] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-toi. 1997. HyTech: A Model Checker for Hybrid Systems. *Software Tools for Technology Transfer* 1 (1997), 460–463.

[22] Anders Hessel and Paul Pettersson. 2007. Cover - A Test-Case Generation Tool for Timed Systems. 31–34.

[23] R. M. Hierons, S. Counsell, and M. AbouTrab. 2012. Specification Mutation Analysis for Validating Timed Testing Approaches Based on Timed Automata. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE Computer Society, Los Alamitos, CA, USA, 660–669. https://doi.org/10.1109/COMPSAC.2012.93

[24] William E. Howden. 1976. Reliability of the Path Analysis Testing Strategy. *IEEE Transactions on Software Engineering* 2, 3 (1976), 208–215.

[25] Henrik Jensen, Kim Larsen, and Arne Skou. 2002. Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL. *BRICS Report Series* 3 (01 2002). https://doi.org/10.7146/brics.v3i24.20005

[26] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Softw. Eng.* 37, 5 (Sept. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[27] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sept. 2011), 649–678.

[28] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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, 654–665.

[29] D. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. 2010. . https://doi.org/10.2200/S00310ED1V01Y201011DCT005

[30] Jin Hyun Kim, Kim G. Larsen, Brian Nielsen, Marius Mikucionis, and Petur Olsen. 2015. Formal Analysis and Testing of Real-Time Automotive Systems Using UPPAAL Tools. In *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings (Lecture Notes in Computer Science, Vol. 9128)*, Manuel Núñez and Matthias Güdemann (Eds.). Springer, 47–61. https://doi.org/10.1007/978-3-319-19458-5_4

[31] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. 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, 142–151. https://doi.org/10.1109/ICSTW.2016.41

[32] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. 2017. Mutation-Based Test-Case Generation with Ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 319–328. https://doi.org/10.1109/ICSTW.2017.60

[33] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman. 2017. Mutation-Based Test-Case Generation with Ecdar. (2017), 319–328. https://doi.org/10.1109/ICSTW.2017.60

[34] Kim G. Larsen, Marius Mikucionis, and Brian Nielsen. 2005. 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, 299 – 306. http://doi.acm.org/10.1145/1086228.1086283

[35] Magnus Lindahl, Paul Pettersson, and Wang Yi. 2001. Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer* 3, 3 (01 Aug 2001), 353–368. https://doi.org/10.1007/s100090100048

[36] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Trans. Software Eng.* 40, 1 (2014), 23–42.

[37] Robert Nilsson, Jeff Offutt, and Sten F. Andler. 2004. Mutation-Based Testing Criteria for Timeliness. In *Proceedings of the 28th Annual International Computer Software and Applications Conference - Volume 01 (COMPSAC '04)*. IEEE Computer Society, Washington, DC, USA, 306–311. http://dl.acm.org/citation.cfm?id=1025117.1025515

[38] Christer Norström and Anders Wall. 1999. Timed Automata as Task Models for Event-Driven Systems. In *In proceedings of RTCSA99*. IEEE Computer Society. http://www.es.mdh.se/publications/69-

[39] Jeff Offutt. 2011. A mutation carol: Past, present and future. *Information and Software Technology* 53, 10 (Oct. 2011), 1098–1107. https://doi.org/10.1016/j.infsof.2011.03.007

[40] James Jerson Ortiz, Moussa Amrani, and Pierre-Yves Schobbens. 2017. Multi-timed Bisimulation for Distributed Timed Automata. In *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10227)*, Clark Barrett, Misty Davies, and Temesghen Kahsai (Eds.). 52–67. https://doi.org/10.1007/978-3-319-57288-8

[41] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. 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, 936–946. https://doi.org/10.1109/ICSE.2015.103

[42] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* 112 (2018).

[43] Mike Papadakis and Nicos Malevris. 2010. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *ISSRE*. IEEE, 121–130. https://doi.org/10.1109/ISSRE.2010.38

[44] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2 ed.). Pragmatic Bookshelf, Raleigh, NC.

[45] Faezeh Siavashi, Junaid Iqbal, Dragos Truscan, and Juri Vain. 2017. *Testing Web Services with Model-Based Mutation*. Springer, 45–67. https://doi.org/10.1007/978-3-319-62569-0_3

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

[47] James Jerson Ortiz Vega, Gilles Perrouin, Moussa Amrani, and Pierre-Yves Schobbens. 2018. Model-Based Mutation Operators for Timed Systems: A Taxonomy and Research Agenda. *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)* (2018). https://doi.org/10.1109/qrs.2018.00045

[48] Jeffrey M. Voas and Gary McGraw. 1997. *Software Fault Injection: Inoculating Programs Against Errors.* John Wiley & Sons, Inc.

[49] Aznam Yacoub, Maamar El Amine Hamri, and Claudia Frydman. 2020. DEv-PROMELA: modeling, verification, and validation of a video game by combining model-checking and simulation. *SIMULATION* 96, 11 (2020), 881–910. https://doi.org/10.1177/0037549720946107 arXiv:https://doi.org/10.1177/0037549720946107

[50] Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman. 2017. *Model-Based Testing for Embedded Systems.* CRC Press.

[51] Apostolos V. Zarras. 2020. Common Mistakes When Using the Command Pattern and How to Avoid Them. *Proceedings of the European Conference on Pattern Languages of Programs 2020* (2020). https://doi.org/10.1145/3424771.3424773