Time for Networks

Cortés, David; Ortiz Vega, James Jerson; Basile, Davide; Aranda Bueno, Jesús Alexander; Perrouin, Gilles; Schobbens, Pierre-Yves

Publication date:
2024

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
Cortés, D, Ortiz Vega, JJ, Basile, D, Aranda Bueno, JA, Perrouin, G & Schobbens, P-Y 2024, 'Time for Networks: Mutation Testing for Timed Automata Networks', Paper presented at 12 International Conference On Formal Methods In Software Engineering, Lisbon, Portugal, 14/04/24 - 15/04/24.

Time for Networks: Mutation Testing for Timed Automata Networks

David Cortés
david.cortes@correounivalle.edu.co
Universidad del Valle
Cali, Colombia

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

Davide Basile
davide.basile@isti.cnr.it
ISTI CNR
Pisa, Italy

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

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

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

ABSTRACT

Mutation Testing (MT) is a technique employed to assess the efficacy of tests by introducing artificial faults, known as mutations, into the system. The goal is to evaluate how well the tests can detect these mutations. These artificial faults are generated using mutation operators, which produce a set of mutations derived from the original system. Mutation operators and frameworks exist for a variety of programming languages, and model-based mutation testing is gaining traction, particularly for timed safety-critical systems. This paper focuses on extending MT to Networks of Timed Automata (NTAs), an area that has not been extensively explored. We introduce mutation operators designed for NTAs specified in UPPAAL, aiming to create temporal interaction faults. We assess the effectiveness of these operators on five UPPAAL NTAs sourced from the literature, specifically examining the generation of equivalent and duplicate mutants. Our results demonstrate a varied prevalence of equivalent mutants (from 12% to 71%) while the number of duplicates is less. In all cases, timed bisimulation was able to process each mutant pair in less than one second.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; **• Theory of computation → Timed and hybrid models**.

KEYWORDS

Model-Based Mutation Testing, UPPAAL, Bisimulation

ACM Reference Format:
David Cortés, James Ortiz, Davide Basile, Jesús Aranda, Gilles Perrouin, and Pierre-Yves Schobbens. 2024. Time for Networks: Mutation Testing for Timed Automata Networks. In *Formal Methods in Software Engineering (FormaliSE) (FormaliSE '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3644033.3644378

1 INTRODUCTION

Digital systems have become an integral part of modern life. They are used in almost every area, including business, scientific research, and our daily routines. They range from smartphones, computers, and common household appliances, to critical systems such as industrial control systems, medical devices, braking mechanisms, and avionic systems. As a result, it is imperative to understand and ensure the accuracy and reliability of these systems, using, among others, software testing and formal verification.

Here, we focus on safety-critical Timed Systems (TS), where strict time constraints must be met to ensure correctness. Such systems are currently validated using testing, as mandated by *e.g.*, DO-178, ISO26262, etc. But how to ensure the quality of tests? For this purpose, Mutation Testing (MT) can be used, where artificial faults, called mutations, are planted in the system to check whether its test suite can detect them [33]. Any mutant that is not distinguished from the original by the test suite is called *alive*, otherwise, it is called *killed* [25]. The mutation score (MS) that measures the quality of the test suite: $MS = M_K/M_T$, where M_K is the number of killed mutants and M_T is the total number of generated mutants [25]. Indeed, the type of mutations introduced and their impact on the system are crucial for an effective and comprehensive Quality Assurance (QA) process in a safety-critical system. When dealing with formal models and other formalisms like Timed Systems (TS), mutations are applied not to the actual system but to a formal model faithfully mirroring the implementation. This approach is known as Model-Based Mutation Testing (MBMT). Timed Automata (TA), Timed Petri Nets (TPN), and Networks of Timed Automata (NTA) are some of the formalisms commonly used to model these systems [4, 42].

In this context, we focus on NTA because they provide a more expressive, declarative, and, depending on the application, a more natural modeling experience (e.g., for communication protocols, I/O architectures, etc.) compared to TA, TPN, and UML, among others. However, most of the existing mutation operators involving NTA were initially defined for TA or a closely related language [3, 7, 8, 15, 32, 39, 42]. These operators indirectly mutate the NTA by targeting its TA components. However, most case studies involve real applications exhibiting synchronous and asynchronous characteristics. For example, 14 of the 20 (70%) case studies listed on [41] fall into this category. To our knowledge, specific mutation operators for NTA have not been proposed or implemented. This represents an opportunity to exploit by designing operators that encode faults mainly present in safety-critical systems with parallel characteristics. The main contributions of this paper are as follows:

(1) We propose a set of novel mutation operators for NTA inspired from other works, emphasizing sync/async aspects;
(2) We implement our operators on top of the MUPPAAL tool;
(3) We evaluate their propensity to yield equivalent and duplicate mutants for five case studies in UPPAAL.

Open Science Policy. The implementation of the operators, UPPAAL models we used, and the tool's outputs are available at https://gitlab.com/formalise/ntaiomutants.

Outline. In Section 2 we present mutation operators for TA and other TS. Theoretical background on TA, NTA, and modeling tools are presented in Section 3. In Section 4, we introduce our eight mutation operators with their respective formal definitions. We evaluate such operators in Section 5 on five case studies. Finally, Section 8 concludes and presents possible future directions.

2 RELATED WORK

Mutation testing has a rich history spanning various domains, from mainstream programming to formal languages [37]. In particular, in the area of Timed Systems (TS), the work of Aichernig et al. [3] stands as a pivotal contribution, introducing a mutation testing framework for Timed Automata (TA). This seminal work laid the foundation for subsequent research, inspiring the creation of fundamental mutation operators as well as their implementation by other researchers [4, 32, 39]. Aboutrab *et al.* and Aichernig *et al.* [1, 3] proposed seven mutation operators for UPPAAL to test TS. Nilsson *et al.* [35] extended TA with tasks: Every i-th task of a set of n (real-time) tasks has a period T_i, a worst-case execution time C_i, and a relative deadline D_i. Furthermore, Basile *et al.* [7] proposed six mutation operators on TAIO, with the intent of avoiding the generation of subsumed mutants a priori. The main idea in [7, 8] was to perform a refinement check between a given mutant and the system model, using ECDAR [29]. Siavash *et al.* [39] explored mutation-based testing capabilities to assess vulnerabilities in Web services where multiple users interact. They used UPPAAL and some classic mutation operators, as well as three of their own: *Remove Actions*, which randomly deletes one action at a time, *Duplicate Actions*, which randomly copies an action to different parts of a model, *Remove Guards*, which randomly selects an action and removes its guard. The newly defined operators provided erroneous behaviors that were not revealed by other mutants.

The need for further exploration and development of mutation operators for systems explicitly expressing time, such as TA was highlighted in [42]. The paper provided a taxonomy and survey of existing operators for TS across different formalisms. It emphasized the importance of investigating new operators that could expose subtle errors inherent in systems with explicit time representation.

Cuartas *et al.* [15] proposed an approach for the static detection of duplicate mutants in TS, introducing a new operator tailored for remove transition/remove state. This work contributes to ongoing efforts to improve mutation testing techniques for TS. Table 1 was retrieved from [15] and summarizes the operators from the considered contributions.

Cuartas *et al.* [14] highlighted the need to explore model-based mutation testing (MBMT) in Networks of Timed Automata (NTA). This need became apparent when certain mutants were no longer detected as erroneous when considered as part of a network, emphasizing the importance of evolving mutation testing methodologies to address the complexity of networked systems.

Alberto *et al.* [4] instantiates the notions of mutation testing and the associated formal theory of testing for the state-rich concurrent language Circus [4]. In particular, Alberto *et al.* are concerned with linking mutations that are discovered within a Circus specification to traces of the denotational semantics of Circus that characterize tests that include the mutation. The theory and languages on which Circus is based, such as Communicating Sequential Processes (CSP) [40], Unifying Theories of Programming (UTP) [2], have employed mutation testing. However, it is novel to investigate a state-rich process algebra for refinement with an UTP semantics. Furthermore, Alberto *et al.* examine and modify existing mutation operators [40] for the underlying languages and develop new ones specifically for Circus to propose new mutation operators.

In terms of implementations, there are several tools dedicated to mutation-based testing and analysis of TA, the most well-known and used being MoMuT: TA [31] and more recently, **μ**UTA [39], and the one on which this work builds [14], all work with Networks of Timed Automata (NTA) in UPPAAL, but neither performs mutations on the entire network as a whole, but on each automaton, and all derived work uses only the *classical* operators defined in [3].

Particularly for the UPPAAL tool, there is a dearth of research and implementations using mutation operators in NTA. Researching mutation operators on entire NTA could contribute to a deeper comprehension of the value and suitability of mutation-based testing in distributed and concurrent environments.

3 BACKGROUND

3.1 Clocks and Timed Automata

We use non-negative real-valued variables known as *clocks* to represent the continuous time domain. Clocks are variables that advance synchronously at a uniform rate; they are the basis of TA [5]. Model checkers such as UPPAAL [9], KRONOS [11], and HYTECH [22] support and extend TA. Clock constraints within TA control transitions. Here, we work with an extension of TA known as Timed Automata with Inputs and Outputs (TAIO) [3]. In TAIO, actions are divided into inputs and outputs [3, 17].

3.2 Timed Automata with Inputs and Outputs

A TAIO is an extended TA where the interaction between a system and its environment is modeled using output and input actions [3].

Definition 1 (Clock constraints). *Let X be a finite set of clock variables ranging over $\mathbb{R}_{\geq 0}$ (non-negative real numbers). $\Phi(X)$ are the* clock constraints *over X. A* clock constraint *$\phi \in \Phi(X)$ is 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 $\sim \in \{<, >, \leq, \geq, =\}$.

Table 1: Mutation operators for TA, from [15].

Nilsson *et al.* [35]		Aichernig *et al.* [3]		Basile *et al.* [7]	
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	-	-

Definition 2 (Clock valuations). *Given a finite set of clocks X, a* clock valuation $v : X \longrightarrow \mathbb{R}_{\geq 0}$ *is a function assigning to each clock $x \in X$ a non-negative value $v(x)$. We denote $\mathbb{R}^X_{\geq 0}$ the set of all valuations. For a clock valuation $v \in \mathbb{R}^X_{\geq 0}$ and a delay $d \in \mathbb{R}_{\geq 0}$, $v + d$ is the valuation given by $(v + d)(x) = v(x) + d$ for each $x \in X$. Given a clock subset $Y \subseteq X$, we denote $v[Y \longleftarrow 0]$ the valuation defined as follows: $v[Y \longleftarrow 0](x) = 0$ if $x \in Y$ and $v[Y \longleftarrow 0](x) = v(x)$ otherwise.*

In TAIO, the transitions are guarded by a clock constraint. A transition also can carry out an action and can reset clocks. TAIO divide actions into two disjoint sets: input actions (marked with ?) and output actions (marked with !) [3]. The output actions of a TAIO $\mathcal{A}$ can be input actions of another TAIO $\mathcal{B}$. We adapt the definition of [26] as:

Definition 3 (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,*
- *$I : L \longrightarrow \Phi(X)$ is a function that associates to each location a clock invariant.*

We write a transition $(l, a, \phi, Y, l') \in T$ as $l \xrightarrow{a,\phi,Y} l'$ where l and l' are the source and target locations, respectively, ϕ a guard, a the action, 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 $(l, v) \in L \times \mathbb{R}^X_{\geq 0}$, where l denotes the current location with its accompanying clock valuation v, starting at (l_0, v_0) where v_0 maps each clock to 0.

Definition 4 (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$(\mathcal{A})$ = $(S, s_0, \Sigma_\Delta, \longrightarrow)$ where:*

- *$S = L \times \mathbb{R}^X_{\geq 0}$ is the set of states,*
- *$s_0 = (l_0, v_0)$ with $v_0(x) = 0$ for all $x \in X$ and $v_0 \models I(l_0)$,*
- *$\Sigma_\Delta = \Sigma \uplus \mathbb{R}_{\geq 0}$,*
- *$\longrightarrow \subseteq S \times \Sigma_\Delta \times S$ is a transition relation defined by the following two rules:*
  - ***Discrete transition:** $(l, v) \xrightarrow{a} (l', v')$, for $a \in \Sigma$ iff $l \xrightarrow{a,\phi,Y} l'$, $v \models \phi$, $v' = v[Y \longleftarrow 0]$ and $v' \models I(l')$ and,*
  - ***Delay transition:** $(l, v) \xrightarrow{d} (l, v + d)$, for some $d \in \mathbb{R}_{\geq 0}$ iff $v + d \models I(l)$.*

3.3 Network of Timed Automata I/O

A definition of NTAIO and of their the semantics is presented below. Basically, an NTAIO is a parallel product of TAIO, where as usual input is blocking, and the synchronizations are broadcast (i.e., one sender and zero or more receivers).

Definition 5 (NTAIO). *A Network of TAIO (NTAIO) is the parallel product of TAIO $\mathcal{A}_i = (L_i, l^0_i, X_i, \Sigma^I_i, \Sigma^O_i, \Sigma_i, T_i, I_i)$, $1 \leq i \leq n$, noted $\mathcal{N} = \mathcal{A}_1 || \dots || \mathcal{A}_n$.*

Below, given a vector $\bar{l}$, we denote as $\bar{l}_i$ the i-th element of $\bar{l}$.

Definition 6 (Semantics of NTAIO). *The semantics of the NTAIO $\mathcal{N}$ is a TIOTS $(S, s_0, \Sigma_\Delta, \longrightarrow)$ where:*

- *$S = \{(\bar{l}, v) \in (L_1 \times \ldots \times L_n) \times \mathbb{R}^X_{\geq 0} \mid v \models \bigwedge_{i=1}^n I_i(l_i)\}$,*
- *$s_0 = (\bar{l}_0, v_0)$ such that $\forall x \in X$, $v_0(x) = 0$,*
- *$\Sigma_\Delta = \Sigma \cup \mathbb{R}_{\geq 0}$,*
- *$\longrightarrow \subseteq S \times \Sigma_\Delta \times S$ is the transition relation defined by:*
- ***Discrete transition:** $(\bar{l}, v) \xrightarrow{a} (\bar{l}', v')$ for some $a \in \Sigma_\Delta$, iff*
  - *let $Synch \subseteq \{1, \ldots, n\}$ be such that $\bar{l}_i = \bar{l}'_i$ if $i \notin Synch$,*
  - *for all $i \in Synch$ there exists some $t_i = (l_i, a, \phi_i, Y_i, l'_i) \in T_i$ such that $\bar{l}_i = l_i$, $\bar{l}'_i = l'_i$,*
  - *there exists $i \in Synch$ such that $a \in \Sigma^O_i$, and for all $j \in Synch$, $j \neq i$ it holds that $a \in \Sigma^I_j$.*
  - *$v \models \bigwedge_{i \in Synch} \phi_i$, $v' = v[\bigcup_{i \in Synch} Y_i \longleftarrow 0]$, and $v' \models \bigwedge_{i=1}^n I_i(\bar{l}'_i)$*
- ***Delay transition:** $(\bar{l}, v) \xrightarrow{d} (\bar{l}, v + d)$ iff $v \models \bigwedge_{i=1}^n I_i(\bar{l}_i)$ and $v + d \models \bigwedge_{i=1}^n I_i(\bar{l}_i)$.*

Example 1. *Let $\mathcal{N}$ be the NTAIO depicted in Fig 1. $\mathcal{N}$ contains two TAIO ($\mathcal{N} = \mathcal{A}_1 \mid\mid \mathcal{A}_2$) with two clocks x and y, and two locations: S_0 (initial) and S_1 (for $\mathcal{A}_1$), S_2 (initial) and S_3 (for $\mathcal{A}_2$). We denote input actions with (?) and output actions with (!) (a, b, c). Thus, a, b and c require synchronization of the two automata (i.e., two automata communicate using synchronization actions). In particular, S_1 and S_3 are the locations in which the invariants are not trivially true:*

$I(S_1) = (x \leq 4)$, forcing the TAIO ($\mathcal{A}_1$) to exit S_0 before x becomes 4 and $I(S_2) = (y \leq 3)$, forcing the TAIO ($\mathcal{A}_2$) to exit S_3 before y becomes 3. Locations S_0 and S_3 have true invariants (thus not drawn), allowing it to stay in S_0 and S_3. The transition $S_1 \xrightarrow{c?,(x==4)} S_0$ ($\mathcal{A}_1$) specifies that when the input action $c?$ occurs and that action is synchronized with the output action $c!$ of $\mathcal{A}_2$), and the guard $x == 4$ holds, this enables the transition, leading to a new current location S_0 for the automaton $\mathcal{A}_1$ and the new current location S_2 for the automaton $\mathcal{A}_2$ (with transition $S_3 \xrightarrow{c!,y:=0} S_2$). Similarly, the input and output actions $a?$ ($a!$) and $b?$ ($b!$) are synchronized between the two automata.

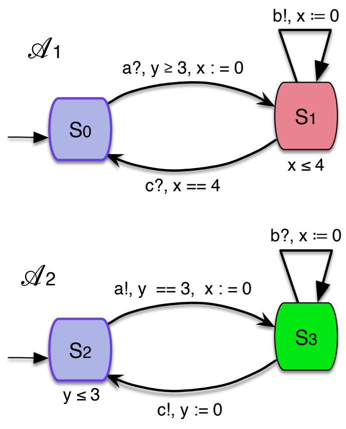


Figure 1: Network of TAIO with two clocks x and y.

3.4 Timed Bisimulation

Timed bisimulation [12] is a relation between system states that captures behavioral equivalence concerning both functional behavior and timing constraints. Timed bisimulation is useful to verify systems modeled as TA, or any other formalism that has a TIOTS semantics.

Definition 7 (**Timed Bisimulation** [12]). *Let $\mathcal{D}_1$ and $\mathcal{D}_2$ be two TIOTS over the set of actions Σ. 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.

3.5 UPPAAL

UPPAAL is a tool for the modelling, simulation, and verification of NTA extended with, among the others, data types, variables, functions and clocks that can be declared either as global or local. Furthermore, in UPPAAL there are two synchronization mechanisms, binary and broadcast synchronization, which translate to unicast and broadcast communication channels [9] (actions of TAIO are also called channels in UPPAAL).

In Definition 6 all actions are formalized as broadcast. Unicast synchronizations can be formalized as follows. The set of actions Σ is further partitioned into broadcast and unicast actions. In Definition 6, whenever $(\bar{l}, v) \xrightarrow{a} (\bar{l}', v')$ and a is unicast, the further constraint $|Synch|= 2$ is required.

UPPAAL also allows to define *committed* and *urgent* locations. Transitions outgoing committed locations have higher priority than any other discrete/delay transition of any other automaton in the network. Whenever a location is urgent, no delay transition is allowed. Basically, whenever the network is in a state $(\bar{l}, v)$ where for some i it holds that $\bar{l}_i$ is committed, then from $(\bar{l}, v)$ it is only possible to execute discrete transitions such that $i \in Synch$. Furthermore, whenever the network is in a state $(\bar{l}, v)$ where for some i it holds that $\bar{l}_i$ is urgent, then no delay transition is allowed. The concept of urgency is extend to action/channels: whenever a source location has an enabled outgoing transition labelled by an urgent action, no delay is possible from that location. In UPPAAL it is also possible to have transitions with no action label. In the following, we write $t = (l, \emptyset, \phi, Y, l')$ whenever transition t has no action label.

3.6 Equivalent/Duplicate mutation problem.

The equivalent/duplicate mutant problem [38] poses a significant challenge in mutation analysis, where two program variants exhibit identical behavior, making them indistinguishable by test cases. This problem has a significant impact on both test suite generation and evaluation. In test suite generation, resources are wasted trying to eliminate mutants that cannot be killed, while in evaluation, results are biased by evaluating the same (erroneous) behavior multiple times. The problem includes equivalence between mutants and the original system and equivalence between two mutants that do not affect the original system.

In the context of model-based mutation testing (MBMT), mutation testing (MT) [19] is a highly effective coverage criterion for evaluating test suite quality. However, MT comes at a high cost, and the presence of equivalent and duplicate mutations further increases this cost [37] [33] [38]. Studies at the code level have shown that between 30% and 40% of mutants are equivalent, and 20% to 30% are duplicate mutants [37].

- **Equivalent mutants**: Describe the same behavior as the original model. It is impossible to kill them, and they do not contribute to an adequate mutation score [25].
- **Redundant mutants**: Killed whenever other mutants are killed. Two types of redundant mutants: (1) *duplicated* mutants when two mutants are equivalent; and (2) *subsumed* mutants when a mutant is killed whenever the other one is also killed (but they are not necessarily equivalent) [25]. In this paper, we focus our analyses on equivalent and duplicate mutants.

4 PROPOSED OPERATORS

Here, we introduce new mutation operators defined specifically for NTAIO based on the UPPAAL syntax. The new operators are presented in Table 2. The first column indicates whether the operator affects the network directly (requires a network) or indirectly (requires only an automaton). The second column shows the name of the operator, followed by a description of the operator. We will now give a formal definition of each operator.

Proposed operators

	Name	Acronym	Type
NTAIO	*syncSeq*	SS	Sequential
	delSync	DS	Interleave
	maskVarh	MVCh	Shadow global channel
	maskVarc	MVCc	Shadow global clock
	urgChan	UC	Urgent Channel
	BroadChan	BC	Broadcast Channel
TAIO	*urgLoc*	UL	Urgent Location
	commLoc	CL	Committed Location

Table 2: Set of proposed operators.

A mutation operator is a function $\mathcal{M}_\mu$ that generates a set of mutants from a NTAIO $\mathcal{N}$. Here, we will use μ to refer to each specific operator presented in Table 2.

4.1 SyncSeq mutation (SS)

The objective is to make two synchronizing transitions execute sequentially. This is achieved by a mutation that removes the synchronization labels, adds a clock reset and a guard, and commits one of the source locations to force a specific sequential timing behavior.

Definition 8 (SyncSeq operator (SS)). *Let $\mathcal{N}$ be a NTAIO and $\mathcal{M}_{syncSeq}(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ using the μ = syncSeq mutation operator. The mutated NTAIO $m \in \mathcal{M}_{syncSeq}(\mathcal{N})$ is of the form $(\mathcal{A}_1||\dots||\mathcal{A}'_i||\dots||\mathcal{A}'_j||\dots||\mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0i}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, \Sigma_i, (T\backslash\{t_i\}) \cup \{t'_i\}, I_i)$ and $\mathcal{A}'_j = (L_j, l_{0j}, X_j, \Sigma_{Ij}, \Sigma_{Oj}, (T\backslash\{t_j\}) \cup \{t'_j\}), I_j)$, such that:*

- *$t_i = (l_i, a, \phi_i, Y_i, l'_i)$ and $t_j = (l_j, a, \phi_j, Y_j, l'_j)$ where either $a \in \Sigma^I_i \cap \Sigma^O_j$ or $a \in \Sigma^O_i \cap \Sigma^I_j$,*
- *$t'_i = (l_i, \emptyset, \phi_i, Y_i \cup \{x\}, l'_i)$, $t'_j = (l_j, \emptyset, \phi_j \cup \{x > k\}, Y_j, l'_j)$, and*
- *l_i is a committed location and $k > 0$.*

Example 2. *Let $\mathcal{N}$ be the NTAIO depicted in Fig 2. $\mathcal{N}$ contains two TAIO ($\mathcal{N} = \mathcal{A}_1 \,||\, \mathcal{A}_2$) where the locations: S_1 (initial), S_1 (for $\mathcal{A}_1$), Q_1 (initial), Q_2 (for $\mathcal{A}_2$). The two automata communicate using synchronization actions (a, b, and c). The mutation operator SyncSeq deletes the synchronized actions (a!, a?), makes the location Q_1 committed, add the clock reset $y := 0$ and add the guard $y > 3$.*

4.2 DelSync mutation (DS)

Two synchronous actions are interleaved by deleting the input/output action labels. This emulates a timing error in which two processes skip a step in their synchronization.

Definition 9 (DelSync operator (DS)). *Let $\mathcal{N}$ be a NTAIO and $\mathcal{M}_{delSync}(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ used the μ = delSync mutation operator. The mutated NTAIO $m \in \mathcal{M}_{delSync}(\mathcal{N})$ is of the form $(\mathcal{A}_1||\dots||\mathcal{A}'_j||\dots||\mathcal{A}'_j||\dots||\mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0i}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, \Sigma_i, (T_i\backslash\{t_i\}) \cup \{t_{m_i}\}, I_i)$ and $\mathcal{A}'_j = (L_j, l_{0j}, X_j, \Sigma_{Ij}, \Sigma_{Oj}, (T_j\backslash\{t_j\}) \cup \{t_{m_j}\}, I_j)$, such that:*

- *$t_i = (l_i, a, \phi_i, Y_i, l'_i)$, $t_j = (l_j, a, \phi_j, Y_j, l'_j)$,*
- *$a \in \Sigma_{Ii}, a \in \Sigma_{Oj}$ or $a \in \Sigma_{Ij}, a \in \Sigma_{Oi}$, and*
- *$t_{m_i} = (l_i, \emptyset, \phi_i, Y_i, l'_i)$, $t_{m_j} = (l_j, \emptyset, \phi_j, Y_j, l'_j)$,*

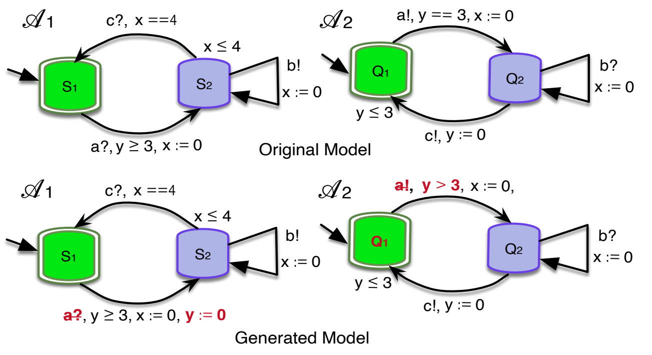


Figure 2: SyncSeq Operator Example

To avoid deadlocks in binary synchronization, the labels of the input and output channels must be deleted. However, if the synchronization is done via broadcast channels, this is not necessary because the broadcast transmission is non-blocking according to UPPAAL semantics.

Example 3. *Let $\mathcal{N}$ be the NTAIO depicted in Fig 3. $\mathcal{N}$ contains three TAIO ($\mathcal{N} = \mathcal{A}_1 \,||\, \mathcal{A}_2 \,||\, \mathcal{A}_3$). The three automata communicate using binary synchronization actions (a, b, c, d). The mutation operator DelSync deletes the synchronized actions (a!, a?).*

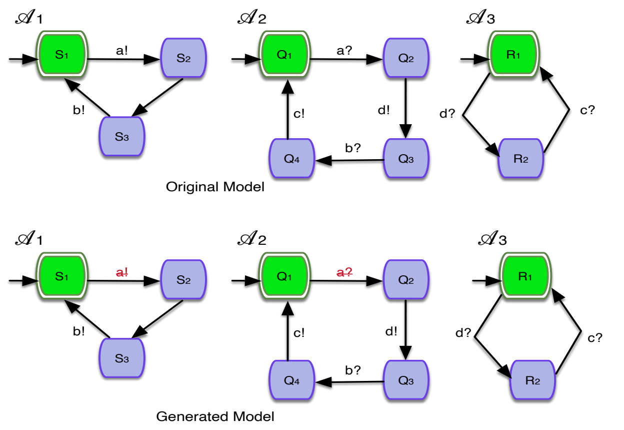


Figure 3: DelSync Operator Example

4.3 MaskVar Mutation (MVC(c/h))

This operator simulates a memory inconsistency error by substituting a global declaration of a variable, channel, clock, etc., into a local declaration within a NTAIO, effectively masking the global declaration. In a real-world scenario, such inconsistency could arise from the variable's value not updating fast enough, updating too quickly, or the programmer inadvertently masking the global variable with a local one. Since this operator re-declares a variable already declared, but within a local scope, the introduced differences will only be visible from the perspective of UPPAAL (and the XML file).

First, we define the mutation on clocks, where x is a clock declared globally and $\hat{x}$ is the mutated clock that is declared locally.

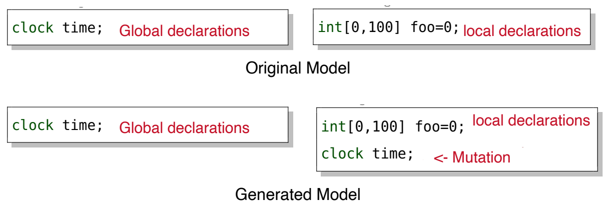
Definition 10 (MaskVar with clocks (MVCc)). *Let $\mathcal{N}$ be a NTAIO. Let x and $\hat{x}$ be two clocks, where $x \in X$ and $\hat{x} \notin X$ and $\mathcal{M}_{maskVarc}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ used the $\mu = maskVarc$ mutation operator, where $m \in \mathcal{M}_{maskVarc}(\mathcal{N})$ is of the form $(\mathcal{A}_1 || \dots || \mathcal{A}'_i || \dots || \mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0i'}, ((X_i - \{x\}) \cup \{\hat{x}\}), \Sigma_{Ii}, \Sigma_{Oi}, \Sigma_i, T_i, I_i)$.*

We also define the mutation on channels/actions, where c is a channel/action declared globally and $\hat{c}$ is the mutated channel/action that is declared locally.

Definition 11 (MaskVar with channels (MVCh)). *Let $\mathcal{N}$ be a NTAIO. Let c and $\hat{c}$ be two actions (or channels), where $c \in \Sigma$ and $\hat{c} \notin \Sigma$ and $\mathcal{M}_{maskVarh}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ used the $\mu = maskVarh$ mutation operator, where $m \in \mathcal{M}_{maskVarh}(\mathcal{N})$ is of the form $(\mathcal{A}_1 || \dots || \mathcal{A}'_i || \dots || \mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0i}, X_i, \Sigma'_{Ii}, \Sigma'_{Oi}, ((\Sigma_i - \{c\}) \cup \{\hat{c}\}), T_i, I_i)$.*

In the above definition, $\Sigma'_{Ii}, \Sigma'_{Oi}$ are mutated as side effect of the mutation on Σ_i.

Example 4. *Figure 4 depicts the use of the MaskVar operator MVCc, in which we declare a clock variable in a global environment and subsequently redeclare it in a local environment (mutation).*



```
clock time;
```

```
int[0,100] foo=0;
```

```
clock time;
```

```
int[0,100] foo=0;
clock time;
```

Figure 4: MaskVar (MVCc) Operator Example

4.4 BroadChan and UrgChan mutation

4.4.1 BroadChan Mutation. The broadChan mutation allows for one-to-many synchronization. When a broadcast synchronization of this type is enabled, it mandates that all receivers must synchronize. In other words, when one of these broadcast synchronizations is enabled, all receivers involved in the synchronization must also synchronize. This mutation is useful for modeling scenarios where information or signals must be broadcast to multiple receivers, ensuring that all intended receivers receive and process the broadcast data. The following mutation mutates a broadcast channel into a unicast channel.

Definition 12 (BroadChan operator (BC)). *Let $\mathcal{N}$ be a NTAIO. Let c and $\hat{c}$ be two actions/channels, where $c \in \Sigma$ and $\hat{c} \notin \Sigma$ and $\mathcal{M}_{BroadChan}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ using the $\mu = BroadChan$ mutation operator, where $m \in \mathcal{M}_{BroadChan}(\mathcal{N})$ is of the form $(\mathcal{A}_1 || \dots || \mathcal{A}'_i || \dots || \mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0'}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, ((\Sigma_i - \{c\}) \cup \{\hat{c}\}), (T_i \backslash U) \cup V, I_i)$ where:*

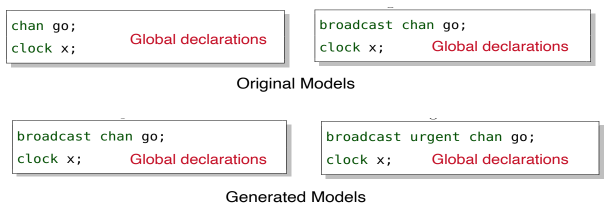
- *$c \in \Sigma$, is a binary action/channel,*
- *$\hat{c} \notin \Sigma$ is a broadcast action/channel,*
- *$U \subseteq T_i$ is the set of all edges whose action is c,*
- *$V = \{(l_i, \hat{c}, \phi_i, Y_i, l'_i) \mid (l_i, c, \phi_i, Y_i, l'_i) \in U\}$.*

4.4.2 UrgChan Mutation. The urgChan mutation introduces urgency into a synchronization/shared action, preventing delays in the source state if an edge with the specified urgent synchronization is enabled. An urgent channel implies that there should be no delay when a transition with an urgent action is enabled, meaning that the next state must involve an action step, which may or may not be synchronized. In essence, urgent channels are used to prevent delays/waiting and encourage the next transition through that channel to be synchronized. In addition, channels can be both broadcast and urgent. This combination can significantly change the behavior of the system. However, to generate valid mutants, this operator can only be applied to edges without clock guards.

Definition 13 (UrgChan operator (UC)). *Let $\mathcal{N}$ be a NTAIO. Let c and $\hat{c}$ be two actions/channels, where $c \in \Sigma$ and $\hat{c} \notin \Sigma$ and $\mathcal{M}_{UrgChan}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ used the $\mu = UrgChan$ mutation operator, where $m \in \mathcal{M}_{UrgChan}(\mathcal{N})$ is of the form $(\mathcal{A}_1 || \dots || \mathcal{A}'_i || \dots || \mathcal{A}_n)$ where $\mathcal{A}'_i = (L_i, l_{0i}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, ((\Sigma_i - \{c\}) \cup \{\hat{c}\}), (T_i \backslash U) \cup V, I_i)$ where:*

- *$c \in \Sigma$ is a binary action/channel,*
- *$\hat{c} \notin \Sigma$ is a urgent action/channel,*
- *$U \subseteq T_i$ is the set of all edges whose action is c,*
- *$V = \{(l_i, \hat{c}, \phi_i, Y_i, l'_i) \mid (l_i, c, \phi_i, Y_i, l'_i) \in U\}$.*

Example 5. *Figure 5 depicts the use of the BroadChan and UrgChan operators, in which we declare a channel variable and broadcast channel in a global environment and subsequently redeclare the channel variable as broadcast channel variable and broadcast channel variable as broadcast urgent channel variable in the same environment (mutation).*



```
chan go;
clock x;
```

```
broadcast chan go;
clock x;
```

```
broadcast chan go;
clock x;
```

```
broadcast urgent chan go;
clock x;
```

Figure 5: BroadChan and UrgChan Operators Example

4.5 UrgLoc and CommLoc mutation

4.5.1 UrgLoc Mutation. As stated in Section 3.5, we recall that in UPPAAL locations can be labeled as urgent. In an urgent location, time cannot be waited for. In other words, it is not possible to introduce delays while occupying that location. However, leaving the urgent location can still occur normally. This type of location significantly alters the sequence of events and their timing. Nonetheless, defining a location as urgent can be a common modeling mistake, especially for individuals new to the field. The introduction of this novel operator is justified based on the inherent complexities and potential pitfalls associated with the use of urgent locations [23].

Definition 14 (UrgLoc operators (UL)). *Let $\mathcal{N}$ be a NTAIO. Let $\mathcal{M}_{UrgLoc}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ using the μ = UrgLoc mutation operator, where $m \in \mathcal{M}_{UrgLoc}(\mathcal{N})$ is of the form $(\mathcal{A}_1|| \ldots ||\mathcal{A}'_i|| \ldots ||\mathcal{A}_n)$ where $\mathcal{A}'_i = ((L_i - \{l_i\}) \cup \{\hat{l}_i\}, l_{0i}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, \Sigma_i, (T_i \backslash T_{l_i}) \cup T_{\hat{l}_i}, I_i)$ where:*

- *$\hat{l}$ is a urgent location, substituing the non-urgent location l_i,*
- *$T_{l_i} = \{t \mid t \in T_i$ has source or target location $l_i\}$ and*
- *$T_{\hat{l}_i}$ is obtained from T_{l_i} by swapping l_i with $\hat{l}_i$ in all elements of T_{l_i}.*

4.5.2 CommLoc Mutation. As stated in Section 3.5, we recall that in UPPAAL, locations can also be declared committed. Committed locations freeze time, but must guarantee atomicity. The idea of this operator is to change any urgent or normal location into a committed location and analyze the results. Due to the massive changes a committed location introduces into a system, this operator will likely generate a large number of non-equivalent mutants or even error-revealing ones.

Definition 15 (CommLoc operators (CL)). *Let $\mathcal{N}$ be a NTAIO. Let $\mathcal{M}_{CommLoc}$ $(\mathcal{N})$ be the set of mutants generated from $\mathcal{N}$ used the μ = CommLoc mutation operator, where $m \in \mathcal{M}_{CommLoc}(\mathcal{N})$ is of the form $(\mathcal{A}_1|| \ldots ||\mathcal{A}'_i|| \ldots ||\mathcal{A}_n)$ where $\mathcal{A}'_i = ((L_i - \{l_i\}) \cup \{\hat{l}_i\}, l_{0i}, X_i, \Sigma_{Ii}, \Sigma_{Oi}, \Sigma_i, (T_i \backslash T_{l_i}) \cup T_{\hat{l}_i}, I_i)$ where:*

- *$\hat{l}_i$ is a committed location, l_i is not committed,*
- *$T_{l_i} = \{t \mid t \in T_i$ has source or target location $l_i\}$ and*
- *$T_{\hat{l}_i}$ is obtained from T_{l_i} by swapping l_i with $\hat{l}_i$ in all elements of T_{l_i}.*

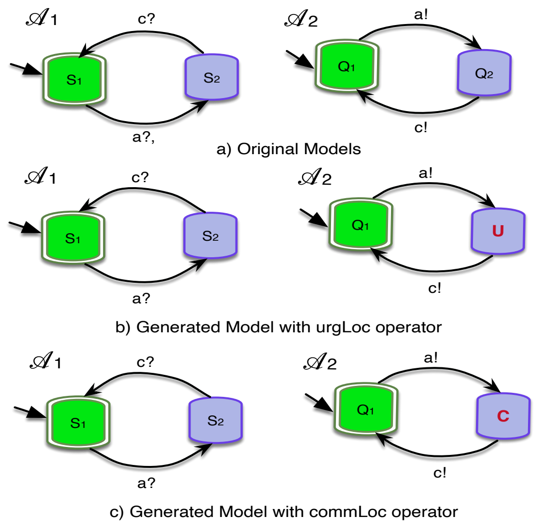
Example 6. *Let $\mathcal{N}$ be the NTAIO depicted in Fig 6. $\mathcal{N}$ contains two TAIO ($\mathcal{N} = \mathcal{A}_1 \mid\mid \mathcal{A}_2$) where the locations: S_1 (initial), S_1 (for $\mathcal{A}_1$), Q_1 (initial), Q_2 (for $\mathcal{A}_2$) in Fig 6 (a). The two automata communicate using synchronization actions (a and c). The mutation operator urgLoc change the normal location Q_2 into an urgent location (Fig 6 (b)). The mutation operator commLoc change the normal location Q_2 into a committed location (Fig 6 (c)).*

5 EVALUATION

5.1 Case Studies

Our studies come from UPPAAL specifications of these cases and are available at [18]. In addition, we considered a mechanical ventilator [13]. Our case studies consist of 5 models in total. The models Collision Avoidance, Train Gate Controller, Tram Door, Gear Control, Mechanical Ventilator are all cyclic and deadlock-free models. For each case study, we focused on the Network of TA (UPPAAL). Table 3 provides structural metrics for each case study.

5.1.1 Collision Avoidance (CA). The CA case models a protocol where different agents want to get access to Ethernet through a shared channel [24]. The model consists of two instances (TAIO), which together form a NTAIO. The CA model has 12 locations in total and 26 transitions.



𝒜1
c?
S1
S2
a?,
𝒜2
a!
Q1
Q2
c!
a) Original Models
𝒜1
c?
S1
S2
a?
𝒜2
a!
Q1
U
c!
b) Generated Model with urgLoc operator
𝒜1
c?
S1
S2
a?
𝒜2
a!
Q1
C
c!
c) Generated Model with commLoc operator

Figure 6: urgLoc and commLoc Operators Example

5.1.2 Train Gate Controller (TGC). The TGC models a railway system that controls access to a bridge for several trains [6]. The model consists of 3 instances (TAIO), which together form a NTAIO. The bridge is a shared resource accessible by only one train at a time. The TGC model has 17 locations in total. 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.

5.1.3 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 consists of 5 instances (TAIO), which together form a NTAIO. The model has 32 locations in total, 36 transitions, and 2 clocks.

5.1.4 Gear Control (GC). The GC models a simple gear controller for vehicles [30]. The model consists of 2 TAIO, forming a NTAIO. Overall, the GC model contains 48 locations, of which 20 have invariants and 60 transitions.

5.1.5 Mechanical Ventilator (MV). The MV model describes the basic functionality and behavior of a mechanical ventilator [16]. The model consists of 5 TAIO, forming a NTAIO. The model has 24 locations in total and 29 transitions.

5.2 Research Questions

To evaluate our newly introduced operators, we measure the number of mutants they can produce and their proneness to generate equivalent and duplicate mutants. Therefore, we form the following research questions:

RQ1 How many mutants each operator generates?
RQ2 How do NTAIO and TAIO operators compare regarding equivalent mutants?
RQ3 How do NTAIO and TAIO operators compare regarding duplicate mutants?
RQ4 What is the cost of generating mutants and removing equivalent and duplicate ones?

Table 3: Case Studies Details.

Case Studies	Instances	Locations	Transitions	Clocks	Channels	Broad. Channels	Urg. Channels
CA	2	12	26	1	0	11	0
GC	2	48	60	1	0	20	0
TGC	3	17	18	1	0	9	0
MV	5	24	29	6	7	12	0
TD	5	32	36	2	0	11	0

5.3 Evaluation Procedure

We ran our experiments on a UBUNTU 21.10 × 86_64 GNU/Linux machine with 4 cores, 4.7 GHz, 32GB RAM. To mitigate randomness effects, we ran the mutant generation 5 times. The computation of equivalent and duplicate mutants relies on the timed bisimulation algorithm presented in Section 3. Figure 7 shows the general workflow for our five systems (see Section 5.1). For each case, we first generate a set of mutants ($\mathcal{MU}$) using our novel operators presented in Table 4. This results in 779 total mutants for all case studies, as shown in Table 4. For each case study, the generated mutants are grouped in all possible pairs, and the timed bisimulation algorithm presented in [36] is applied to each pair to check how many mutants are equivalent.

6 RESULTS AND DISCUSSION

6.1 RQ1: Number of Generated Mutants

Table 4: Average number of generated mutants per operator

	CA	GC	TGC	MV	TD
DS	25	56	36	17	36
MVCh	22	40	16	60	55
MVCl	2	2	7	9	8
UC	11	20	8	12	11
UL	12	46	28	17	36
CL	12	46	28	20	31
SS	9	18	7	7	10
BC	0	0	0	7	0
Total	93	228	130	149	179

Table 4 presents the average number of generated mutants for each operator. Indeed, the implementation of the operators involves some randomness and may create invalid ones that we discard. We can see that the delete synchronization operator (DS) generates the maximum number of mutants for three out of the five systems. The masking channel operator (MVCh) is also able to generate many mutants, in particular for the mechanical ventilator and tram door, because these models have many synchronizations. Logically, MVCc operator yields more mutants when there are more clocks. In contrast, the BC operator was only applied to the mechanical ventilator model. This is due to the nature of channels. Indeed, in all but the ventilator model, channels were already broadcast ones, preventing the application of the operator.

RQ1 Summary: Our operators exhibit diversity in the number of mutants they can generate, from widely applicable channel masking and interleaving to operators focusing on TA-specific elements such as clocks and non-broadcast channels.

6.2 RQ2: NTAIO vs TAIO Equivalent Mutants

Table 5 shows the number of equivalent mutants per operator for each UPPAAL network. The last column shows varied averages of equivalent mutants, from 12% to 71%. In particular, the masking channel operator (MVCh) is particularly likely to produce equivalent mutants. Interestingly, the clock masking operator (MVCc) generates fewer equivalent mutants (42% compared to 71% for (MVCh), denoting a very distinct impact on model behavior. While operators producing high numbers of equivalent mutants may seem irrelevant, we mitigate this aspect by having an exact procedure (timed bisimulation) to remove them. Generally, network-aware operators yield either the maximum or the minimum number of mutants.

RQ2 Summary: Our operators also have diverse behavior regarding equivalent mutants. The operator MVCh is by far the largest contributor to equivalence, followed by UC. Thus, network operators are more prone to yield equivalence.

6.3 RQ3: NTAIO vs TAIO Duplicate Mutants

Tables 6 to 10 detail occurrences of duplicates per study and the most involved combinations of operators yielding such duplicates. For example, one can interpret the first row of Table 7 as "The operator that generated the most mutants duplicated with DS was UL, its mutants comprising 28% of the total mutants that are bisimilar with other DS mutants." In three out of the five models, the MVCh operator is the most involved in duplicate pairs, followed by the UL and CL that are tied in the two remaining models. The MVCh operator is also often appearing in duplicate pairs when compared to other operators (*e.g.*, for the mechanical ventilator, see Table 10). The CL and UL operators seem to contribute equally in duplicate pairs for three of our systems.

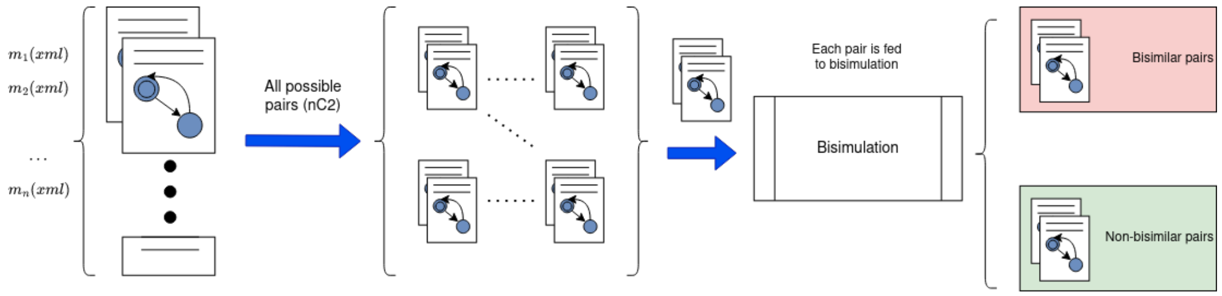


Figure 7: Experimentation workflow

Table 5: Average ratio of Equivalent Mutants Computed via Timed Bisimulation.

Case	Collision Avoidance	Gear Control	Train Gate Controller	Mechanical Ventilator	Tram Door	Average
DS	4/25 (16%)	9/56 (16%)	3/36 (8%)	15/17 (88%)	1/36 (3%)	26%
MVCh	3/22 (14%)	38/40 (95%)	14/16 (87%)	56/60 (93%)	36/55 (65%)	71%
MVCc	0/2 (0%)	1/2 (50%)	4/7 (57%)	6/9 (66%)	3/8 (37%)	42%
UC	5/11 (45 %)	19/20 (95%)	5/8 (62%)	8/12 (66%)	6/11 (54%)	64%
UL	4/12 (33%)	43/46 (93%)	21/28 (75%)	6/17 (35%)	24/36 (66%)	60%
CL	3/12 (25%)	43/46 (93%)	21/28 (75%)	9/20 (45%)	20/31 (64%)	60%
SS	0/9 (0%)	1/18 (5%)	0/7 (0%)	4/7 (57%)	0/10 (0%)	12%
BC	0 (0%)	0 (0%)	0 (0%)	2/7 (28%)	0 (0%)	28%
Total	19/93 (20%)	154/228 (67%)	75/130 (58%)	106/149 (71%)	90/187 (48%)	52%

Table 6: Average redundant mutants for the CA case study.

Operator	No. duplicated pairs	Most duplicated operator
DS	56/8556 (0.65%)	MVCH 10 (27%)
MVCh	1055 /8556 (12.3%)	CL & UL 528 (50%)
MVCl	116 /8556 (1.3%)	MVCH 44 (38%)
UC	591 /8556 (7%)	MVCH 242 (40%)
UL	639 /8556 (7.5%)	MVCH 264 (41%)
CL	636 /8556 (7.5%)	MVCH 264 (42%)
SS	0/8556(0%)	N/A

Table 7: Average duplicate mutants for the TGC case study.

Operator	No. duplicated pairs	Most duplicated operator
DS	643 / 16770 (4%)	UL 153 (23%)
MVCh	1292 / 16770 (8%)	CL & UL 896 (69%)
MVCl	354 / 16770 (2%)	CL & UL 224 (63%)
UC	680 / 16770 (4%)	CL & UL 448 (65%)
UL	2107 / 16770 (12.5%)	CL 784 (37%)
CL	2108 / 16770 (12.5%)	UL 784 (37%)
SS	28 / 16770 (0.1%)	CL 12 (42%)

RQ.3 Summary: The MVCh operator is the most important contributor to pairs of duplicate mutants. This network operator is more involved in duplicates than non-network ones. The "locations" operators (UL and CL) contribute equally to duplicates.

Table 8: average duplicate pairs of mutants for each operator, with the TD case study.

Operator	No. duplicated pairs	Most duplicated operator
DS	44 / 31,862 (0.1%)	MVCH 14 (31%)
MVCh	5569 / 31,862 (17.5%)	UL 1705 (30%)
UC	1356 / 31,862 (4.2%)	MVCH 605 (44%)
UL	3509 / 31,862 (11%)	MVCH 1705 (48%)
CL	3511 / 31,862 (11%)	MVCH 1704 (48%)
SS	3 / 31,862 (0.009%)	CL 1 (33%)

Table 9: Average duplicate pairs of mutants for each operator, with the GC case study.

Operator	No. duplicated pairs	Most duplicated operator
DS	939/ 51,756 (1.8%)	CL 167 (17%)
MVCh	5515 / 51,756 (10.6%)	CL & UL 3680 (66%)
MVCl	316 / 51,756 (0.6%)	CL & UL 184 (58%)
UC	2971 / 51,756 (5.7%)	CL & UL 1840 (62%)
UL	6221 / 51,756 (12%)	CL 2116 (34%)
CL	6221 / 51,756 (12%)	UL 2116 (34%)
SS	193 / 51,756 (0.3%)	UL 58 (30%)

6.4 Answering RQ4: Mutation Costs

Table 11 presents the costs of mutation, that is the time required to generate mutants and perform equivalent and duplicate mutant analysis. As we can expect, mutant generation is quite fast, and most of the cost is formed by the comparison amongst mutants

Table 10: Average duplicate pairs of mutants for each operator, with the MV case study.

Operator	No. duplicated pairs	Most duplicated operator
DS	2448 / 22,052 (11%)	CL 167 (17%)
MVCh	4837 / 22,052 2 (22%)	MVCH 1546 (31%)
MVCl	972 / 22,052 (4.4%)	MVCH 504 (51%)
UC	1278 /22,052 (5.7%)	MVCH 672 (52%)
UL	909 / 22,052 (4.1%)	MVCH 379 (41%)
CL	1225 / 22,052 (5.5%)	MVCH 548 (44%)
BC	128 / 22,052 (0.5%)	MVCH 55 (42%)
SS	685 / 22,052 (3.1%)	MVCH 340 (49%)

Table 11: Average times for mutant generation, bisimulation for all pairs, and mean bisimulation per pair (left to right) across case studies.

Case study	Mutant generation	Bisimulation of all pairs	Bisimulation per pair
CA	1 s	24 min	660 ± 41 ms
GC	3 s	172 min	1 s ± 126 ms
TGC	2 s	153 min	748 ± 52 ms
MV	2.5 s	165 min	1 s ± 177 ms
TD	3 s	195 min	726 ± 67 ms

using timed bisimulation, which can take more than three hours. A comparison takes less than one second, and it is not necessarily the largest model (in terms of locations and transitions) that takes more time to analyze. For example, the mechanical ventilator is smaller but we see the influence of the number of clocks on the analysis times.

RQ4 Summary: Our timed bisimulation algorithm can analyze an equivalent/duplicate pair in less than one second. The analysis times are only partially dependent on the model's structural characteristics suggesting a good scalability of timed bisimulation.

6.5 Discussion

6.5.1 Number of equivalent mutants. Compared to code-based mutation, where the number of equivalent mutants is around 30% [37], the number of equivalent mutants may seem important (up to 71%). This was one of the motivations when designing network operators to assess this point. This is in line with recent studies showing that, even for operators not operating at the network level, there could be a large ration of subsumed mutants refining the original system (that are thus equivalent). In particular, the MVCh operator creates a masking problem that cannot be detected easily without proceedings to resets. The analysis of non-equivalent mutants in terms of introduced faults is left to future work.

6.5.2 Dynamic vs Static Equivalent/Duplicate Detection. Our operators exhibit a variety of behaviors, and some may be more easily detected by static analysis rather than a more costly bisimulation. For example, given the number of equivalent/duplicates generated by MVCh, it can be of further interest to avoid masking in generation guidelines [7, 8]. Further, studying the tradeoffs between precision and analysis speed for each operator could pave the way for hybrid equivalence detection approaches.

6.5.3 On Timed Bisimulation. This paper focuses on timed bisimulation equivalence as the primary equivalence relationship between NTAIO. The following reasons motivate this choice. First, while for deterministic systems timed trace equivalence is also possible, timed bisimulation is more efficient [20, 28]. Second, timed bisimulation being the strongest equivalence relation, it will finely distinguish behaviour due to subtle faults, allowing to keep stubborn mutants (*i.e.*, hard-to-kill mutants), improving bug detection ability of test suites. Yet, it would be of interest to compare the influence of simulation relations (such as TIOCO [27]) on the effectiveness of timed mutation testing.

7 THREATS TO VALIDITY

Internal validity. To mitigate effects due to randomness we ran each operator on each system five times. The timed bisimulation algorithm is exact and is therefore not subject to parameterization issues.

External validity. We cannot guarantee that our results extend to all timed systems expressed in UPPAAL. To mitigate this threat, we selected five cases of different natures: a gear controller, a network communication model avoiding collisions, a train gate controller, a tram door, and a mechanical ventilator. These models have different sizes and numbers of clock constraints. They enabled us to observe differences in detecting and removing duplicate mutants.

8 CONCLUSION

In this paper, we designed novel operators for Networks of Timed Automata with Input and Output (NTAIO). Our experiments showed various behaviors regarding equivalent and duplicate mutant occurrences. Timed bisimulation is a cost-effective way to remove them.

There is room for future work. First, we would like to evaluate the influence of resets on masking operators, which may lead to many equivalent and duplicate mutants. The invert reset operator [3] is a candidate for this. Second, concerning the quality of the remaining mutants, we would like to assess their *stubbornness* and determine a minimally adequate subset of mutation operators via subsumption analyses. We will evaluate our operators for test generation and assessment. Finally, we would like to explore the transferability of our operators to other formalisms or languages, such as Timed Petri Nets, Timed Process Algebras, and Real-time Logics [10, 21, 34, 43].

ACKNOWLEDGMENTS

James Ortiz and Pierre Yves Schobbens were partially supported by the Fonds de la Recherche Scientifique FNRS under PDR Grant n° T.0199.21 (Scaling Up Variability). Special thanks to the agreement between the University of Namur and Universidad del Valle. Davide Basile acknowledges the MUR PRIN 2022 PNRR P2022A492B project ADVENTURE 2023-2025, funded by European Union Next-Generation EU. Gilles Perrouin is a FNRS Research Associate. This manuscript reflects only the authors' views and opinions.

REFERENCES

[1] M Saeed AbouTrab, Steve Counsell, and Robert M Hierons. 2012. Specification mutation analysis for validating timed testing approaches based on timed automata. In *2012 IEEE 36th Annual Computer Software and Applications Conference*. IEEE, 660–669.

[2] Bernhard K. Aichernig and He Jifeng. 2008. Mutation testing in UTP. *Formal Aspects of Computing* 21, 1–2 (Feb. 2008), 33–64. https://doi.org/10.1007/s00165-008-0083-6

[3] Bernhard K Aichernig, Florian Lorber, and Dejan Ničković. 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 7*. Springer, 20–38.

[4] Alex Donizeti Betez Alberto. [n. d.]. *Formal mutation testing in Circus process algebra*. Doutorado em Ciências de Computação e Matemática Computacional. https://doi.org/10.11606/T.55.2019.tde-04012019-112931

[5] Rajeev Alur and David L Dill. 1994. A theory of timed automata. *Theoretical computer science* 126, 2 (1994), 183–235.

[6] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 1993. Parametric real-time reasoning. In *STOC*. ACM, 592–601.

[7] Davide Basile, Maurice H ter Beek, Sami Lazreg, Maxime Cordy, and Axel Legay. 2022. Static detection of equivalent mutants in real-time model-based mutation testing: An Empirical Evaluation. *Empirical Software Engineering* 27, 7 (2022), 160. https://doi.org/10.1007/s10664-022-10149-y

[8] 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, Volume A*, Roberto Erick Lopez-Herrejon (Ed.). ACM, 30:1–30:11. https://doi.org/10.1145/3382025.3414966

[9] Gerd Behrmann, Alexandre David, and Kim G Larsen. 2006. A tutorial on Uppaal 4.0. *Department of computer science, Aalborg university* (2006).

[10] B. Berthomieu and M. Diaz. 1991. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering* 17, 3 (1991), 259–273. https://doi.org/10.1109/32.75415

[11] 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 Vancouver, BC, Canada, June 28–July 2, 1998 Proceedings 10*. Springer, 546–550.

[12] 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.

[13] David Cortes. 2023. Mechanical Ventilator Case Study. https://github.com/ventynet/ventynet.

[14] Jaime Cuartas. 2022. *Model-Based Mutation Testing Prototype for Timed Automata*. Bachelor's Thesis.

[15] Jaime Cuartas, Jesús Aranda, Maxime Cordy, James Ortiz, Gilles Perrouin, and Pierre-Yves Schobbens. 2023. MUPPAAL: Reducing and Removing Equivalent and Duplicate Mutants in UPPAAL. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 52–61.

[16] Jaime Cuartas, David Cortés, Joan S. Betancourt, Jesús Aranda, José I. García, Andrés M. Valencia, and James Ortiz. 2023. Formal Verification of a Mechanical Ventilator Using UPPAAL. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Formal Techniques for Safety-Critical Systems* (Cascais, Portugal) *(FTSCS 2023)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3623503.3623536

[17] 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*. 91–100.

[18] Rebeka Farkas. 2023. Case Studies. https://github.com/farkasrebus/XtaBenchmarkSuite.

[19] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. 1997. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software* 38, 3 (1997), 235–253.

[20] Pierre Ganty, Nicolas Manini, and Francesco Ranzato. 2023. Computing Reachable Simulations. arXiv:2204.11804 [cs.LO]

[21] M. R. Hansen. 1998. *Duration Calculus: A Logical Approach to Real-Time Systems*. DFKI Saarbrücken. http://www2.compute.dtu.dk/pubdb/pubs/1902-full.html

[22] Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. 1997. HyTech: A model checker for hybrid systems. In *Computer Aided Verification: 9th International Conference, CAV'97 Haifa, Israel, June 22–25, 1997 Proceedings 9*. Springer, 460–463.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[24] 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

[25] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[26] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. 2010. *The theory of timed I/O automata*. Morgan & Claypool Publishers.

[27] Moez Krichen and Stavros Tripakis. 2009. Conformance testing for real-time systems. *Formal Methods Syst. Des.* 34, 3 (2009), 238–304. https://doi.org/10.1007/S10703-009-0065-1

[28] Antonín Kučera and Richard Mayr. 2002. Why is Simulation Harder than Bisimulation?. In *Proceedings of the 13th International Conference on Concurrency Theory (CONCUR'02) (Lecture Notes in Computer Science, Vol. 2421)*, Luboš Brim, Petr Jančar, Mojmír Křetínský, and Antonín Kučera (Eds.). Springer-Verlag, 594–609.

[29] Kim G Larsen, Florian Lorber, Brian Nielsen, and Ulrik M Nyman. 2017. Mutation-based test-case generation with ecdar. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 319–328.

[30] 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

[31] Florian Lorber. 2015. Model-Based Mutation Testing of Synchronous and Asynchronous Real-Time Systems. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015,*. IEEE Computer Society, 1–2. https://doi.org/10.1109/ICST.2015.7102615

[32] Florian Lorber, Kim G Larsen, and Brian Nielsen. 2018. Model-based mutation testing of real-time systems via model checking. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 59–68.

[33] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2013. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *IEEE Transactions on Software Engineering* 40, 1 (2013), 23–42.

[34] Xavier Nicollin and Joseph Sifakis. 1994. The Algebra of Timed Processes, ATP: Theory and Application. *Inf. Comput.* 114, 1 (1994), 131–178. https://doi.org/10.1006/INCO.1994.1083

[35] 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, 2004. COMPSAC 2004*. IEEE, 306–311.

[36] 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*. 52–67.

[37] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*. Vol. 112. Elsevier, 275–378.

[38] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Márcio Ribeiro. 2019. A systematic literature review of techniques and metrics to reduce the cost of mutation testing. *Journal of Systems and Software* 157 (2019), 110388.

[39] Faezeh Siavashi, Dragos Truscan, and Jüri Vain. 2018. Vulnerability Assessment of Web Services with Model-Based Mutation Testing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. https://doi.org/10.1109/QRS.2018.00043

[40] Thitima Srivatanakul, John A. Clark, Susan Stepney, and Fiona Polack. 2003. Challenging Formal Specifications by Mutation: a CSP Security Example. In *Proceedings of the 10th Asia-Pacific Software Engineering Conference (APSEC'03)*. Chiang Mai, Thailand, 340–350.

[41] UppalTeam. [n. d.]. Uppaal Case Studies. https://uppaal.org/casestudies/.

[42] 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. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 325–332.

[43] Wang Yi. 1991. CCS + time = an interleaving model for real time systems. In *Automata, Languages and Programming*, Javier Leach Albert, Burkhard Monien, and Mario Rodríguez Artalejo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 217–228.