Timed Transition Tour for Race Detection in Distributed Systems*

Evgenii Vinarskii[1,†], Natalia Kushik[1], Nina Yevtushenko[2,3], Jorge López[4] and Djamal Zeghlache[1]
[1]*SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France*
[2]*Ivanikov Institute for System Programming, Russian Academy of Sciences, Moscow, Russia*
[3]*Higher School of Economics, Moscow, Russia*
[4]*Airbus, Issy-Les-Moulineaux, France*

Keywords: Races, Model Based Testing, Timed Finite State Machines, Timed Transition Tour.

Abstract: The paper is devoted to detecting *output* races in distributed systems. We perform such detection through testing their implementations. As an underlying model for our test generation strategy we consider a Timed Finite State Machine or a TFSM (for short), where each input/output transition is augmented with a timed guard and an output delay. A potential output race can thus be simulated as an *output delay* mutant; this formalism is introduced in the paper. In order to build a test suite, we adapt a well-known test generation strategy, a transition tour method. The novelty of the proposed method relies on choosing appropriate timestamps for inputs, yielding a *timed* transition tour. We discuss its fault coverage for output race detection. As an application case study, we consider a Software Defined Networking (SDN) framework where the system under test is represented by the composition of a controller and a switch. Experimental results show that the timed transition tour can detect races in the behavior of the widely used ONOS controller.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Software and hardware (distributed) systems become more complex and thorough testing and verification of them is crucial. If a system implementation has races its behavior can be different from the specification (expected behavior) and the system can produce wrong responses to submitted requests, can have deadlocks and livelocks (Baier and Katoen, 2008), etc. Correspondingly, detecting races in a system under test is important. In order to guarantee the absence of races, model based verification and testing can be utilized (see Section 2 for the related work). In Model Based Testing (MBT), the system specification and its implementation are described by the same model; finite transition systems are widely used in MBT (Benharrat et al., 2017). When talking about tests for detecting races, the model of a classical Finite State Machine (FSM) becomes useless as there are no output races in this model. Instead, Input/Output automata (Lynch and Tuttle, 1989) are more appropriate for this purpose. However, in an Input/Output automaton there are no restrictions regarding how long a tester can wait for an output, and thus, the testing process can become really hard or even chaotic. One possible solution is to use proper timed FSMs where inputs can be applied in a row without waiting for a produced output. However, in order to escape potential chaos, an output should be produced only after the appropriate number of time units.

In this paper, we rely on the notion of a Timed Finite State Machine as defined in (Vinarskii and Zakharov, 2020). In the aforementioned model, the behavior depends on its current state, the time instance when an input is applied, and the time required to process the input. Each input/output transition in the TFSM is augmented with a timed guard and an output delay; a transition is only executed if the corresponding input is applied at a time instance which belongs to the interval *guarding* it. The output delay reflects the number of time units needed for the output to be produced after the input has been applied. Note that, in this case, a TFSM can accept an input while the output to the previous one has not been produced yet. In other words, a TFSM considered in the paper implicitly contains concurrent procedures for handling inputs (see Section 3 for more details), and thus races between outputs are relevant. We assume that a

*The work was partially supported by Erasmus program
†On leave from Higher School of Economics, Russia

Copyright © 2023 by SCITEPRESS – Science and Technology Publications, Lda. Under CC license (CC BY-NC-ND 4.0)

TFSM is (output) race-free if for each input sequence there are no two different outputs that can be produced at the same time instance. When deriving test suites, we assume that the specification of the system is output race-free, however, its implementations can have output races. Further, we showcase that the behavior of each implementation can be adequately modeled by *output delay mutants* of the specification that have the same transition graph, but output delays differ. As the classical transition tour for FSMs is known to detect many implementation faults, in this paper, we focus on studying the properties of a *timed* transition tour, and its capabilities for detecting output races in TFSM implementations (see Section 4).

As an application scenario, we consider Software Defined Networking systems and related components. As a system under test, we study the composition of a controller and a switch, in order to assure that the implemented composition is output race-free. Experimental results show that output races in the SDN components can be detected by the timed transition tour. However, the choice of the timestamps significantly affects the fault coverage. This position paper therefore, raises a number of challenges that concern an appropriate assignment of the timestamps to the test inputs as well as the timed transition tour fault coverage against other types of faults, e.g., transition/output faults. We discuss some of these challenges and provide some findings and observations (see Section 5).

The main contributions of this work are: i) transition tour adaption for timed FSMs and its effectiveness for detecting output races in distributed systems, ii) experimental evaluation of the timed transition tour for detecting output races in a real distributed system (i.e., a programmable network framework).

2 RELATED WORK

The problem of race detection in distributed systems, as well as preventive design strategies for race-free systems have been discussed in a number of publications. One of the known approaches in this case relies on the definition of a specific (partial) order between possible events, i.e., the idea is to establish a *happened-before* $\prec_{hb}$ relationship between events relevant to races (Pereira et al., 2020). This order is presented by so-called "logical clocks" C which indicate the timestamps of the events' execution, i.e., $e_1 \prec_{hb} e_2$ if and only if $C(e_1) < C(e_2)$. Such Happened-Before model (Φ_{hb}) has been studied, for example, in (Wen et al., 2022) and (Pereira et al., 2020). Given a pair (e_1, e_2) representing the competing actions, the system is *race-sensitive* if there exist such logical clocks that the following holds: $\varphi_{race} = \Phi_{hb} \wedge (C(e_1) = C(e_2))$. We note that there are race detection tools, based on the Happened-Before (HB) model. For example, in (Liu et al., 2017), the authors present the tool DCatch, which uses the z3 SMT solver (de Moura and Bjørner, 2008) to check whether φ_{race} holds for each pair (e_1, e_2). Moreover, in (Pereira et al., 2020), this approach is improved by eliminating redundant events from the distributed system without affecting the accuracy of race detection (corresponding statements are proven in (Pereira et al., 2020)). A similar approach can be applied for programmable networks (Lu et al., 2019), (El-Hassany et al., 2016) and (Li et al., 2022). In particular, in (El-Hassany et al., 2016) and (Li et al., 2022) the authors have used the HB model for SDN, and related races can be detected by the tools SDNRacer and SPIDER. In (Vinarskii et al., 2019), the authors have studied the advantages of model checking techniques for proactive testing in SDN race detection.

Another possibility is to take a *preventive path*, i.e., to derive the components of a distributed system that are carefully synchronized, so that races cannot show up (McClurg et al., 2017), (McClurg, 2021). In (McClurg et al., 2017), an approach which inserts a number of synchronization processes to the SDN controller is proposed for avoiding races in SDN frameworks. This approach has been improved in (McClurg, 2021). Another prevention strategy is proposed in (Rouzaud-Cornabas et al., 2010) and (Raducu et al., 2022). Namely, the authors focus on preventing a so-called Time Of Check To Time Of Use (TOCTTOU) bug. The latter means that during the time between checking the possibility of execution of a function f and its actual execution, no requests changing the internal state of a program can be received. In (Rouzaud-Cornabas et al., 2010), a race-free formula for preventing the TOCTTOU bug is presented together with the tool that implements the proposed solution.

To the best of our knowledge, there are no existing works in the area of test derivation strategies for detecting races in Timed FSMs, and in this paper, we study the effectiveness of the timed transition tour to provoke races between observable actions.

3 BACKGROUND

As mentioned in the introduction, classical finite state machines are widely used when deriving test suites with guaranteed fault coverage for discrete event and hybrid systems. In this section, we introduce the notion of a TFSM model whose behavior depends on its current state, the time instance when an input is applied, and the time required to process the input. A *timestamp* is represented by a real number $t \in \mathcal{R}_0^+$, which indicates a time instance, when the system receives an input or generates an output. A *timed guard* is an interval (u, v) where $u, v \in \mathcal{N}^+$ and $u < v$ which indicates the period of time when a transition is enabled for processing an input. An *output delay* (or simply a *delay*) $d \in \mathcal{N}^+$ indicates the time needed for producing an output after receiving an input.

Let I (O) be a finite input (output) alphabet, $i \in I$ ($o \in O$) be an input (output) and t be a timestamp. As usual, a *timed input* is a pair (i,t) and a *timed output* is a pair (o,t). A *timed input sequence* is a finite sequence $\alpha = (i_1,t_1)(i_2,t_2)\dots(i_n,t_n)$ where the sequence $t_1t_2\dots t_n$ is monotonically increasing. At the same time, a *timed output sequence* is defined as a finite sequence $\beta = (o_1,t_1)(o_2,t_2)\dots(o_n,t_n)$ where the sequence $t_1t_2\dots t_n$ is monotonically non-decreasing. An untimed projection of β (α), i.e., $\beta \downarrow_O = o_1o_2\dots o_n$ ($\alpha \downarrow_I = i_1i_2\dots i_n$), denotes the sequence obtained after deleting the timestamps (Bresolin et al., 2021).

A *TFSM* $\mathcal{A}$ is a tuple $(S, I, O, G, h_S, D, s_0)$, where I and O are finite input and output alphabets, S is a finite non-empty set of states, G is a finite non-empty set of timed guards, $h_S \subseteq (S \times I \times G \times O \times D \times S)$ is a set of transitions, D is a finite non-empty set of non-zero integer delays, s_0 is the *initial state*. A *transition* $(s,i,g,o,d,s') \in h_S$ is denoted as $s \xrightarrow{i,g/(o,d)} s'$. If the machine receives an input i after t time units being at state s, where $t \in g$, then the machine moves to state s' and produces output o after d time units. Given $t_0 = 0$, a *run* of the TFSM $\mathcal{A}$ for a timed input sequence $\alpha = (i_1,t_1)(i_2,t_2)\dots(i_n,t_n)$ is a finite sequence $run(\mathcal{A},\alpha) = s_0 \xrightarrow[(o_1,\tau_1)]{(i_1,t_1)} s_1 \xrightarrow[(o_2,\tau_2)]{(i_2,t_2)} \dots \xrightarrow[(o_n,\tau_n)]{(i_n,t_n)} s_n$ such that for each $j \in \{1,\dots,n\}$ there exists a transition $s_{j-1} \xrightarrow{i_j,g_j/(o_j,d_j)} s_j$ of $\mathcal{A}$ for which $t_j - t_{j-1} \in g_j$, and $\tau_j = t_j + d_j$. If $t_j - t_{j-1} \in g_j$ for each j, we say that the timed input sequence α is *enabled* for $\mathcal{A}$. Notation $s_{j-1} \xrightarrow[(o_j,\tau_j)]{(i_j,t_j)} s_j$ means that if the machine being at state s_{j-1} receives input i_j at time instance t_j, then the machine immediately moves to state s_j and produces o_j at time instance $\tau_j = t_j + d_j$. TFSM $\mathcal{A}$ is *deterministic* if for every two transitions $s \xrightarrow{i,g/(o,d)} s'$ and $s \xrightarrow{i,g'/(o',d')} s''$ it holds that $g \cap g' = \emptyset$.

In order to get an output reaction for $\alpha = (i_1,t_1)(i_2,t_2)\dots(i_n,t_n)$, the timed outputs of the sequence $(o_1,\tau_1)(o_2,\tau_2)\dots(o_n,\tau_n)$ where $\tau_1 = t_1 + d_1$, $\tau_2 = t_2 + d_2$, $\dots$, $\tau_n = t_n + d_n$ are ordered in such a way that the timed instances are not decreasing. Note that even a deterministic TFSM can produce several output reactions for a timed input sequence. Consider TFSM $\mathcal{M}_{e_1}^4$ shown in Fig. 2a[1] and timed input sequence $\alpha_1 = (i_1,1.5)(i_1,3.0)(i_1,4.5)$ which is enabled for $\mathcal{M}_{e_1}^4$. TFSM $\mathcal{M}_{e_1}^4$ produces both timed output sequences $\beta_1 = (o_2,4.0)(o_1,5.5)(o_3,5.5)$ and $\beta_2 = (o_2,4.0)(o_3,5.5)(o_1,5.5)$, i.e., $\beta_1 \downarrow_O = o_2o_1o_3$ and $\beta_2 \downarrow_O = o_2o_3o_1$, due to the competition of two outputs that can be produced at the same time instance. In this case, we say that a TFSM is *output/output* (or simply, *output*) race-sensitive[2]. Note that there is a known criterion (Vinarskii and Zakharov, 2018) for checking if a TFSM is output race-free.

Thus, given a deterministic TFSM $\mathcal{A}$, $\alpha = (i_1,t_1)(i_2,t_2)\dots(i_n,t_n)$ with a run $run(\mathcal{A},\alpha) = s_0 \xrightarrow[(o_1,\tau_1)]{(i_1,t_1)} s_1 \xrightarrow[(o_2,\tau_2)]{(i_2,t_2)} \dots \xrightarrow[(o_n,\tau_n)]{(i_n,t_n)} s_n$, an output reaction of $\mathcal{A}$ for α is a set of all such permutations of $(o_1,\tau_1)(o_2,\tau_2)\dots(o_n,\tau_n)$ that the time instances are not decreasing, written $out(\mathcal{A},\alpha)$. Let $out(\mathcal{A},\alpha)\downarrow_O$ be a set of all output projections for timed output sequences of $out(\mathcal{A},\alpha)$. Coming back to the TFSM $\mathcal{M}_{e_1}^4$ and $\alpha_1 = (i_1,1.5)(i_1,3.0)(i_1,4.5)$, the output reaction $out(\mathcal{M}_{e_1}^4,\alpha_1) = \{(o_2,4.0)(o_1,5.5)(o_3,5.5),(o_2,4.0)(o_3,5.5)(o_1,5.5)\}$.

Differently from a classical TFSM (Bresolin et al., 2021), for the TFSM considered in this paper, the next timed input can be applied before the machine has produced an output to the previous one. Consider a TFSM $\mathcal{S}$ in Fig. 1. If the machine is at state s_0 then input i_1 can be processed if and only if it is applied at time instance $t_1 \in (1,2)$. Therefore, the transition $s_0 \xrightarrow{i_1,(1,2)/(o_1,5)} s_1$ is enabled for timed input $(i_1,1.3)$ and $\mathcal{S}$ moves to state s_1. Output o_1 will be produced 5 time units later, i.e., there will be a timed output $(o_1,6.3)$. Transition $s_1 \xrightarrow{i_2,(1,2)/(o_2,2)} s_0$ is enabled for the timed input $(i_2,2.6)$ due to the fact that $2.6 - 1.3 \in (1,2)$, and $\mathcal{S}$ moves to state s_0. Output o_2 is produced 2 time units after applying the input i_2, i.e., there is a timed output $(o_2,4.6)$. The run of the TFSM $\mathcal{S}$ on $\alpha = (i_1,1.3)(i_2,2.6)$ is $r = run(\mathcal{S},\alpha) = s_0 \xrightarrow[(o_1,6.3)]{(i_1,1.3)} s_1 \xrightarrow[(o_2,4.6)]{(i_2,2.6)} s_0$, and the timed output sequence is $\beta = (o_2,4.6)(o_1,6.3)$, i.e., $\beta \downarrow_O = o_2o_1$.

With this example, we illustrate the key difference between the TFSM of interest and timed machines considered in (Bresolin et al., 2021). Being at state s and receiving a timed input (i,t), the TFSM immediately moves to state s' simultaneously running the procedure f in order to handle the input i and compute the output o; the execution of the procedure f takes d

[1]We will further explain the notation $\mathcal{M}_{e_1}^4$.

[2]A formal definition is introduced in Section 4.1.

time units (an output delay). The next input can be applied when the TFSM has not produced the output yet. In this case, the TFSM moves immediately to the next state s'' and simultaneously runs the procedure f' in order to compute the output o' in a parallel way with the procedure f which is not finished yet. Therefore, the TFSM considered in the paper implicitly contains concurrent procedures and thus, represents potential output races in distributed systems. We say that the TFSM is *feasible* if the number of procedures running in parallel is always finite. We refer to (Vinarskii and Zakharov, 2020), for checking if a TFSM is feasible. We hereafter consider only feasible TFSMs.

4 TRANSITION TOUR FOR TFSMs AND OUTPUT RACES

In MBT, the behavior of the specification and an Implementation Under Test (IUT) is described by the same model; in our case a TFSM. We would like to check whether there exists an IUT for which two outputs could be produced at the same time instance, i.e., two outputs could compete. In this section, we formally introduce the notion of an *output/output* (or simply *output*) race for a TFSM and consider so-called output delay mutants of the specification TFSM which can have such races. As a transition tour of the specification that is a classical FSM is known to detect many transition and output faults, we define a timed transition tour for a TFSM and analyze its effectiveness with respect to output delay mutants. As an application scenario, we utilize a distributed networking system represented by an SDN framework.

4.1 Output Races in TFSMs

Given a TFSM $\mathcal{A}$, a timed input sequence $\alpha = (i_1,t_1)(i_2,t_2)\dots(i_n,t_n)$ such that α is enabled for $\mathcal{A}$ with a run $run(\mathcal{A},\alpha) = s_0 \xrightarrow[(o_1,\tau_1)]{(i_1,t_1)} s_1 \xrightarrow[(o_2,\tau_2)]{(i_2,t_2)} \dots \xrightarrow[(o_n,\tau_n)]{(i_n,t_n)} s_n$, we say that α *provokes* an *output race* if there exist $k,m \in \{1,\dots,n\}$ such that $o_k \neq o_m$ and $\tau_k = \tau_m$. This means that o_k and o_m compete to be produced, and can be produced in any order: $\dots o_k o_m \dots$ or $\dots o_m o_k \dots$. TFSM $\mathcal{A}$ is *output race-sensitive* if there exists an input sequence α which provokes an output race in $\mathcal{A}$. Otherwise, $\mathcal{A}$ is *output race-free*.

As an example of an output race-sensitive TFSM, consider a machine $\mathcal{M}^4_{e_1}$ shown in Fig. 2a. $\alpha_1 = (i_1,1.5)(i_1,3.0)(i_1,4.5)$ is enabled for $\mathcal{M}^4_{e_1}$ with the corresponding $r = q_0 \xrightarrow[(o_1,5.5)]{(i_1,1.5)} q_1 \xrightarrow[(o_2,4.0)]{(i_1,3.0)} q_2 \xrightarrow[(o_3,5.5)]{(i_1,4.5)} q_1$. Therefore, $\tau_1 = \tau_3 = 5.5$, and o_1 and o_3 compete to be produced at time instance 5.5, i.e., α_1 provokes an output race in TFSM $\mathcal{M}^4_{e_1}$. Similarly, $\alpha_2 = (i_1,1.3)(i_1,2.6)(i_1,3.9)(i_2,5.3)$ provokes an output race between o_1 and o_2 at time instance $\tau_1 = \tau_4 = 6.3$ in TFSM $\mathcal{M}^1_{e_4}$ shown in Fig. 2b as $r' = q_0 \xrightarrow[(o_1,6.3)]{(i_1,1.3)} q_1 \xrightarrow[(o_2,3.6)]{(i_1,2.6)} q_2 \xrightarrow[(o_3,4.9)]{(i_1,3.9)} q_1 \xrightarrow[(o_2,6.3)]{(i_1,5.3)} q_0$.

Note that races and in particular, output races can appear in both, software specifications and implementations. In this section, we assume that the specification is always race-free and an output race can happen only in an implementation. In other words, we do not focus on the specification validation, but rather study test generation strategies where the test purpose is to detect as many output races as possible. Following classical model based testing approaches, we assume that the specification and its implementation can be modeled by the same formalism, which in our case, are TFSMs, and we further discuss output race-sensitive implementations which can be obtained as output delay mutants of the specification.

4.2 Output Delay Mutants of TFSMs

Let $Spec = (S,I,O,G,h_S,D,s_0)$ be an initially connected race-free specification TFSM and $e = s \xrightarrow{i,(u,v)/(o,d)} s' \in h_S$ be one of its transitions. Consider also such a delay $d' \in \mathcal{N}^+$ that $d \neq d'$ and a TFSM $\mathcal{M}^{d'}_e(Spec) = (S,I,O,G,h'_S,D',s_0)$ where h'_S differs from h_S in one transition only, i.e., $h'_S \backslash h_S = \{e'\}$ where $e' = s \xrightarrow{i,(u,v)/(o,d')} s' \in h'_S$. In other words, $\mathcal{M}^{d'}_e(Spec)$ has a transition e' instead of e. We refer to $\mathcal{M}^{d'}_e(Spec)$ as to a *first order output delay mutant* of the TFSM *Spec*.

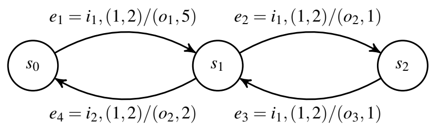

$e_1 = i_1,(1,2)/(o_1,5)$
$e_2 = i_1,(1,2)/(o_2,1)$
s_0
s_1
s_2
$e_4 = i_2,(1,2)/(o_2,2)$
$e_3 = i_1,(1,2)/(o_3,1)$

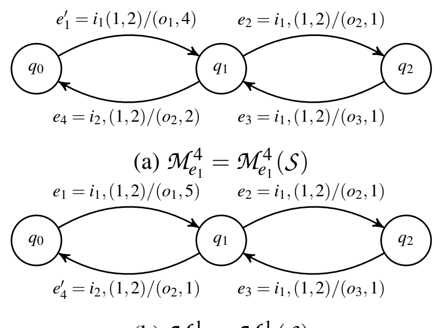
Figure 1: Running example, TFSM $\mathcal{S}$.

By definition, the following statement holds.

Proposition 1. *Given Spec, $\mathcal{M}^{d'}_e(Spec)$ and a timed input sequence α, it holds that 1) α is enabled for Spec if and only if α is enabled for $\mathcal{M}^{d'}_e(Spec)$ and 2) Spec is feasible if and only if $\mathcal{M}^{d'}_e(Spec)$ is feasible.*

Proposition 1 assures that any sequence which can be applied to *Spec*, can be also applied to its mutant $\mathcal{M}^{d'}_e(Spec)$. Moreover, the number of parallel procedures for computing the output reaction of $\mathcal{M}^{d'}_e(Spec)$

is finite and guaranteed by the same property of *Spec*.


$e_1' = i_1(1,2)/(o_1,4)$
$e_2 = i_1,(1,2)/(o_2,1)$
q_0
q_1
q_2
$e_4 = i_2,(1,2)/(o_2,2)$
$e_3 = i_1,(1,2)/(o_3,1)$
$e_1 = i_1,(1,2)/(o_1,5)$
$e_2 = i_1,(1,2)/(o_2,1)$
q_0
q_1
q_2
$e_4' = i_2,(1,2)/(o_2,1)$
$e_3 = i_1,(1,2)/(o_3,1)$

(a) $\mathcal{M}_{e_1}^4 = \mathcal{M}_{e_1}^4(\mathcal{S})$

(b) $\mathcal{M}_{e_4}^1 = \mathcal{M}_{e_4}^1(\mathcal{S})$

Figure 2: Two mutants of TFSM $Spec = \mathcal{S}$ in Fig. 1.

Fig. 2 contains two first order output delay mutants for the specification TFSM $\mathcal{S}$ in Fig. 1, for transitions e_1 and e_4 respectively. The output delay in a mutated transition of $\mathcal{M}_{e_4}^1$ is 1, while the output delay in $\mathcal{S}$ is 2. TFSM $\mathcal{M}_{e_1}^4$ is another mutant of $\mathcal{S}$ where the delay d in e_1 is assigned to 4 instead of 5.

Given $\alpha = (i_1,t_1)(i_2,t_2)\ldots(i_n,t_n)$ which is enabled for *Spec* and $\mathcal{M}_e^{d'} = \mathcal{M}_e^{d'}(Spec)$ which is an output race-sensitive mutant, according to Proposition 1, α is enabled for $\mathcal{M}_e^{d'}$. We say that α *detects* a race-sensitive mutant $\mathcal{M}_e^{d'}$ if α provokes a race in $\mathcal{M}_e^{d'}$, i.e., $out(\mathcal{M}_e^{d'},\alpha)\downarrow_O$ is not a singleton. To check that α detects an output race in $\mathcal{M}_e^{d'}$, during testing, we need to observe the set $out(\mathcal{M}_e^{d'},\alpha)\downarrow_O$ which can contain several output sequences. Therefore, when applying α, to observe all possible outputs of $out(\mathcal{M}_e^{d'},\alpha)\downarrow_O$, we need to rely on the "*all weather conditions*" assumption (Milner, 1980). This means that a tester possesses enough resources to apply as many times as needed the sequence α, for the implementation to *show* all possible output reactions[3].

In our example, both race-sensitive mutants $\mathcal{M}_{e_1}^4$ and $\mathcal{M}_{e_4}^1$ of $\mathcal{S}$ (Fig. 2) are detected by $\alpha_1 = (i_1,1.5)(i_1,3.0)(i_1,4.5)$ and $\alpha_2 = (i_1,1.3)(i_1,2.6)(i_1,3.9)(i_2,5.3)$, accordingly. A *timed test suite* (*TTS*) for TFSM *Spec* is a set of timed input sequences which are enabled for *Spec*. TTS *detects* a race-sensitive implementation whose behavior is described by a mutant $\mathcal{M}_e^{d'}$, if there exists such α in TTS that detects $\mathcal{M}_e^{d'}$.

4.3 TTT for Detecting Output Races

Given the specification TFSM *Spec* and $\alpha = (i_1,t_1)\ldots(i_n,t_n)$ which is enabled for *Spec* with a run $s_0 \xrightarrow[(o_1,\tau_1)]{(i_1,t_1)} s_1 \xrightarrow[(o_2,\tau_2)]{(i_2,t_2)} \ldots \xrightarrow[(o_n,\tau_n)]{(i_n,t_n)} s_n$, we say that α *covers* a transition $e = s \xrightarrow{i,(u,v)/(o,d)} s' \in h_S$ if there exists such $j \in \{1,\ldots,n\}$ that $s_{j-1} = s$, $s_j = s'$, $i_j = i$ and $t_j - t_{j-1} \in (u,v)$. A *timed transition tour* (*TTT*) for *Spec* is a finite set of timed input sequences enabled for *Spec* such that for each transition e of *Spec*, the TTT contains a timed input sequence which covers e. When deriving input sequences of the TTT it is not only crucial to carefully choose each input, but also each time instance when the corresponding input should be applied. TTT does not introduce any optimality constraints on the derivation of the test sequences. Consider $\mathcal{S}$ in Fig. 1 and two TTTs $ts_1 = \{(i_1,1.3)(i_1,2.6)(i_1,3.9)(i_2,5.3)\}$ and $ts_2 = \{(i_1,1.3)(i_1,2.6)(i_1,3.9),(i_1,1.3)(i_2,2.6)\}$. The second test suite is longer and requires a tester to submit one RESET between two test sequences. Unlike a classical transition tour, in TTT we have the freedom in choosing timestamps. For example, $ts_1' = \{(i_1,1.4)(i_1,2.7)(i_1,4.0)(i_2,5.4)\}$ is a TTT for $\mathcal{S}$ which is different from ts_1 only in timestamps.

4.4 Experimenting with SDN and Related Output Races

In this section, we study the timed transition tour effectiveness for a real use-case. We consider an SDN framework as a system under test and check which output races can be detected and also discuss some *hardly detectable* races and related TFSM mutants.

4.4.1 SDN Application Scenario

We performed a preliminary experimental study where the IUT was a composition of two network entities. Namely, we studied SDN that allows implementing a user request on a data plane, when the SDN controller pushes the rules of interest to the switches (forwarding devices) (McKeown et al., 2008). In our case, the IUT is the composition of the ONOS controller (McKeown et al., 2008) and an Open vSwitch. The main test objective was to assure the absence of output races in such composition, that can provoke potential misconfigurations on the data plane.

In order to derive the TFSM modeling the system behavior, we chose a very simple external application which can only request to push two flow rules. Those are: the flow rule 1 (2) with the priority 40001 (40002) and the hard[4] timeout of 5 (2) seconds. The SDN controller receives these requests and pushes the

[3]In our particular case, it is enough that two output reactions are shown to conclude there exists a race.

[4]The number of time units (after installation) for the flow rule to expire in the switch.

rules to the switch. Therefore, our IUT is assumed to have only two inputs: PostFlow1 ($pf1$) and PostFlow2 ($pf2$) and two outputs: FlowExpire1 ($fe1$) and FlowExpire2 ($fe2$). The input $pf1$ ($pf2$) indicates that the composition receives a request for adding the flow rule with the priority 40001 (40002) and the hard timeout 5 (2). The output $fe1$ ($fe2$) indicates that the composition produces a message of expiring a flow rule with the priority 40001 (40002). We derived the output race-free specification TFSM with four states (Fig. 3). This specification is output race-free and contains the following states: $empty$ – the initial state, $F1$ ($F2$) – the state that indicates that the flow rule 1 (2) has been received (to be pushed to the flow table), and $F1\_F2$ – the state that indicates that the switch has received the request to push both flow rules to its flow table.

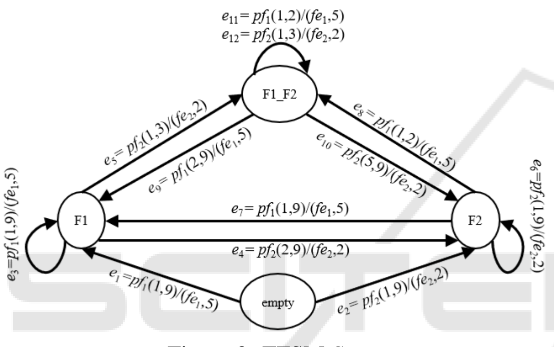


Figure 3: TFSM *Spec*.

SCIENCE AND TECHNOLOGY PUBLICATIONS

We focus on output race detection related to the flow rule expiration which is why we are interested in expressing the states related to changes in the switch flow table. The output delays reflect the hard timeouts of the rules, accordingly. As an example, consider a transition $F1 \xrightarrow{pf2(1,2)/(fe2,2)} F1\_F2$, which means that if the SDN controller and switch composition resides at state $F1$ while receiving $pf2$ after $t \in (1,2)$ seconds, then it moves to state $F1\_F2$, i.e., the second rule is added and this rule will expire in 2 seconds.

We derived a timed transition tour $ts = \{\alpha_1, \alpha_2\}$, where $\alpha_1 = (pf_1,2)(pf_2,4)(pf_1,5.5)(pf_2,7.5)(pf_1,10)(pf_1,12)$ and $\alpha_2 = (pf_2,2)(pf_2,8)(pf_2,11)(pf_1,13)(pf_1,14.5)(pf_2,20)$. The set of expected output responses is $\{\beta_1 \downarrow_O, \beta_2 \downarrow_O\}$ where $\beta_1 \downarrow_O = fe_2 fe_1 fe_2 fe_1 fe_1 fe_1$ and $\beta_2 \downarrow_O = fe_2 fe_2 fe_2 fe_1 fe_1 fe_2$. The test cases were executed against the ONOS controller composed with an Open vSwitch by pushing the flow rules via the REST interface. For that matter, we ran the Mininet (de Oliveira et al., 2014) simulator. Experiments were performed on a virtual machine running on Ubuntu 20.04 LTS with 16GB of RAM. All scripts utilized in the experimental setup are accessible via (Vinarskii, 2023).

The timed transition tour ts was executed two times. The output reaction of the IUT to the test suite ts was different. On the first execution we observed that the flow rule with the priority 40001 expired before the flow rule with the priority 40002. On the second execution we noticed that the flow rule with the priority 40002 expired before the flow rule with the priority 40001[5]. Therefore, we can conclude that the SDN composition is output race-sensitive.

4.4.2 Hardly Detectable Output Delay Mutants and TTT Fault Coverage

As shown above, TTT seems to provide good practical results when it comes to output delay faults that provoke races. However, it is interesting to check how often these faults are detected by the TTT[6]. In order to evaluate such fault coverage, for the specification of the SDN framework, considered above (Fig. 3), we generated a set of first order output delay mutants and verified if they can be detected by various TTTs. Note that output delays in the specification TFSM represent the expiration time for flow rules in the SDN framework, therefore it is interesting to see if this hard timeout for a rule is implemented wrongly but "not far away" from the original value. In other words, we propose a *slight change* of these values, i.e., $d' = d \pm 1$ for a transition $e = s \xrightarrow{i,g/(o,d)} s'$ in $\mathcal{M}_e^{d'}(Spec)$.

Table 1: TTT fault coverage.

Transition in *Spec*	Mutation	Does ts detect races ?	Does ts' detect races ?
$empty \xrightarrow{pf_1(1,9)/(fe_1,5)} F_1$	$d' = 4$	Yes	No
$F_1 \xrightarrow{pf_1(1,9)/(fe_1,5)} F_1$	$d' = 4$	No	No
$F_1 \xrightarrow{pf_2(1,3)/(fe_2,2)} F_2$	$d' = 3$	Yes	No
$F_2 \xrightarrow{pf_1(1,9)/(fe_1,5)} F_1$	$d' = 4$	No	No
$F_2 \xrightarrow{pf_1(1,2)/(fe_1,5)} F_1\_F_2$	$d' = 4$	No	No
$F_1\_F_2 \xrightarrow{pf_1(2,9)/(fe_1,5)} F_1$	$d' = 4$	No	No
$F_1\_F_2 \xrightarrow{pf_1(1,2)/(fe_1,5)} F_1\_F_2$	$d' = 4$	Yes	Yes
$F_1\_F_2 \xrightarrow{pf_2(1,3)/(fe_2,2)} F_1\_F_2$	$d' = 3$	Yes	Yes

In this way, we mutated several transitions of *Spec* and obtained output race-sensitive mutants. In total, 8 mutated output race-sensitive TFSMs were obtained, that are shown in Table 1. We considered various TTTs and observed some interesting results. For the test suite $ts = \{\alpha_1, \alpha_2\}$ that detected output races in the ONOS

[5]See (Vinarskii, 2023) for a detailed description.

[6]It is possible to show that the TTT does not deliver a complete test suite against output delay faults.

framework, with $\alpha_1 = (pf_1,2)(pf_2,4)(pf_1,5.5)(pf_2,7.5)(pf_1,10)(pf_1,12)$ and $\alpha_2 = (pf_2,2)(pf_2,8)(pf_2,11)(pf_1,13)(pf_1,14.5)(pf_2,20)$, only 4 mutants were detected. Another test suite is $ts' = \{\alpha'_1,\alpha'_2\}$ where $\alpha'_1 = (pf_1,2)(pf_2,4.5)(pf_1,6)(pf_2,8)(pf_1,10)(pf_1,12)$ and $\alpha'_2 = (pf_2,2)(pf_2,7)(pf_2,11)(pf_1,13)(pf_1,14.5)(pf_2,20)$ which differs from ts only in timestamps. However, ts' detected only 2 mutants. These experiments showcase the importance of properly choosing timestamps to detect output races (Section 5).

The latter confirms our assumption that a slight change in the output delay value can significantly affect the fault coverage of the timed transition tour. As not many mutants of this kind were detected, we refer to them as *hardly detectable output delay mutants*.

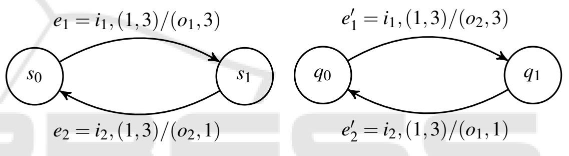
5 DISCUSSIONS

On Choosing Timestamps for a TTT. According to the definition of the timed transition tour, there can be different sets of finite timed input sequences that contain exactly the same untimed input sequences. Indeed, different timestamps for each input i to be applied next, provide different test cases which have different abilities for detecting output races. The question therefore arises: to traverse a transition $e = s \xrightarrow{i,(u,v)/(o,d)} s' \in h_S$, how should we choose the timestamp t for the input i? Shall we get closer to the lower or the upper bound of the interval (u,v) or would it be better to consider its mean value? This choice can have impact on the fault coverage.

Consider again the specification TFSM $\mathcal{S}$ in Fig. 1 and two output race-sensitive implementations in Fig. 2. Let $ts = \{\alpha_1,\alpha_2\}$ ($ts' = \{\alpha'_1,\alpha'_2\}$) be the timed transition tour where all timestamps are close to the lower bound (mean value) of the interval. That is, $\alpha_1 = (i_1,1.3)(i_1,2.6)(i_1,3.9)$, $\alpha_2 = (i_1,1.3)(i_1,2.6)(i_1,3.9)(i_2,5.3)$, $\alpha'_1 = (i_1,1.5)(i_1,3.0)(i_1,4.5)$ and $\alpha'_2 = (i_1,1.5)(i_1,3.0)(i_1,4.5)(i_2,6.0)$, i.e., ts and ts' contain the same set of untimed input sequences. Due to the fact that $out(\mathcal{M}^1_{e_4},\alpha_2)\downarrow_O = \{o_2o_3o_1o_2, o_2o_3o_2o_1\}$, ts detects an output race in $\mathcal{M}^1_{e_4}$ (o_1 and o_2 compete to be produced), however, ts cannot detect an output race in $\mathcal{M}^4_{e_1}$. Similarly, as $out(\mathcal{M}^4_{e_1},\alpha'_1)\downarrow_O = \{o_2o_3o_1, o_2o_1o_3\}$, ts' detects an output race in $\mathcal{M}^4_{e_1}$ (o_1 and o_3 compete to be produced), however, ts' cannot detect an output race in $\mathcal{M}^1_{e_4}$. Finally, note that $ts'' = \{\alpha'_1,\alpha_2\}$ detects output races in both mutants. Thus, even a slight change in one or several timestamps can affect the overall race detection capability of the TTT.

It seems that the timestamps should be chosen based on the types of mutants which we want to detect. If we want to provoke a race and push some outputs to permutate in the implementation, we need to "play" over the output delays d and d' for two of them to be produced at the same time. This knowledge can probably allow backtracking of some feasible timestamps but this is out of the scope of this position paper. Instead, this is a challenge for future work.

On Detecting Output Faults by a TTT. Transition tour of a classical untimed FSM is capable to detect all output faults, i.e., whenever a transition (s,i,o,s') is changed in an implementation to the transition (s,i,o',s'), $o \neq o'$ such a fault will always be detected by a transition tour (*guaranteed* fault coverage). Naturally, the question arises: does a TTT detect all output faults as well? Interestingly, this is not necessarily the case which can simply be illustrated by using a counterexample shown in Fig. 4.


$e_1 = i_1,(1,3)/(o_1,3)$
s_0
s_1
$e_2 = i_2,(1,3)/(o_2,1)$
$e'_1 = i_1,(1,3)/(o_2,3)$
q_0
q_1
$e'_2 = i_2,(1,3)/(o_1,1)$

(a) TFSM $\mathcal{S}_1$ (b) $\mathcal{M}^{o_2,o_1}_{e_1,e_2}(\mathcal{S}_1)$

Figure 4: Specification TFSM and its output fault mutant.

A timed input sequence $\alpha = (i_1,2.0)(i_2,4.0)$ covers all transitions in $\mathcal{S}_1$, i.e., $\{\alpha\}$ is a timed transition tour for $\mathcal{S}_1$. Nevertheless, α does not detect a wrong implementation described by a second order output fault mutant shown in Fig. 4, because $out(\mathcal{S}_1,\alpha)\downarrow_O = out(\mathcal{M}^{o_2,o_1}_{e_1,e_2},\alpha)\downarrow_O = \{o_1o_2, o_2o_1\}$. It is crucial that in the counterexample above, the mutant contains not a single, but multiple faults, i.e., outputs for two transitions are mutated (second order mutant). Therefore, differently from classical FSMs, output fault detection does not hold for a timed transition tour for multiple faults. On the contrary, it can be shown that single output faults can always be detected by any timed transition tour. A corresponding proposition can be proven by contradiction. It is thus interesting to study other capabilities of the timed transition tour. Our paper focuses on specific output delay mutants that represent output races, but does not consider, for example, transition faults, or faults in time intervals. This is another open challenge left for future work.

6 CONCLUSION

This paper adapted the transition tour test generation strategy for timed FSMs, with the test purpose of detecting output races in implementations. Such faulty implementations can be represented by first order output delay mutants of the specification TFSM. To estimate the fault coverage of the timed transition tour against output race detection in distributed systems, we performed a preliminary experimental study. In particular, an SDN framework was considered as a case study. As a result, we observed races between the flow rules in the ONOS controller. The order of the flow rule expiration in ONOS implementation can differ from the specified order and the timed transition tour detects this difference. This work-in-progress raises a number of research challenges. For future work, we plan to further study how to properly choose the timestamps in the timed transition tour. The TTT test suite completeness should also be thoroughly studied, against races and other types of faults, for various types of distributed systems. Finally, TTT should be compared with other test generation strategies with respect to performance and fault coverage, and we plan to make such comparison as well.

REFERENCES

Baier, C. and Katoen, J. (2008). *Principles of model checking*. MIT Press.

Benharrat, N., Gaston, C., Hierons, R. M., Lapitre, A., and Gall, P. L. (2017). Constraint-based oracles for timed distributed systems. In *Testing Software and Systems - 29th IFIP WG 6.1 International Conference*, pages 276–292.

Bresolin, D., El-Fakih, K., Villa, T., and Yevtushenko, N. (2021). Equivalence checking and intersection of deterministic timed finite state machines. *Formal Methods Syst. Des.*, 59(1):77–102.

de Moura, L. M. and Bjørner, N. S. (2008). Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference*, pages 337–340.

de Oliveira, R. L. S., Schweitzer, C. M., Shinoda, A. A., and Prete, L. R. (2014). Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing*, pages 1–6.

El-Hassany, A., Miserez, J., Bielik, P., Vanbever, L., and Vechev, M. T. (2016). Sdnracer: concurrency analysis for software-defined networks. In *Proceedings of the 37th ACM SIGPLAN*, pages 402–415.

Li, A., Padhye, R., and Sekar, V. (2022). Spider: A practical fuzzing framework to uncover stateful performance issues in sdn controllers. https://doi.org/10.48550/arXiv.2209.04026.

Liu, H., Li, G., Lukman, J. F., Li, J., Lu, S., Gunawi, H. S., and Tian, C. (2017). Dcatch: Automatically detecting distributed concurrency bugs in cloud systems. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 677–691.

Lu, G., Xu, L., Yang, Y., and Xu, B. (2019). Predictive analysis for race detection in software-defined networks. *Sci. China Inf. Sci.*, 62(6):62101:1–62101:20.

Lynch, N. A. and Tuttle, M. R. (1989). An introduction to input/output automata. *CWI quarterly*, 2:219–246.

McClurg, J. (2021). Correct-by-construction network programming for stateful data-planes. In *SOSR'21: The ACM SIGCOMM Symposium on SDN Research, Virtual Event*, pages 66–79.

McClurg, J., Hojjat, H., and Cerný, P. (2017). Synchronization synthesis for network programs. In *Computer Aided Verification - 29th International Conference*, pages 301–321.

McKeown, N., Anderson, T. E., Balakrishnan, H., Parulkar, G. M., Peterson, L. L., Rexford, J., Shenker, S., and Turner, J. S. (2008). Openflow: enabling innovation in campus networks. *Comput. Commun. Rev.*, 38(2):69–74.

Milner, R. (1980). *A Calculus of Communicating Systems*. Springer.

Pereira, J. C., Machado, N., and Pinto, J. S. (2020). Testing for race conditions in distributed systems via SMT solving. In *Tests and Proofs - 14th International Conference, TAP@STAF 2020*, pages 122–140.

Raducu, R., Rodríguez, R. J., and Álvarez, P. (2022). Defense and attack techniques against file-based TOCTOU vulnerabilities: A systematic review. *IEEE Access*, 10:21742–21758.

Rouzaud-Cornabas, J., Clemente, P., and Toinard, C. (2010). An information flow approach for preventing race conditions: Dynamic protection of the linux OS. In *Fourth International Conference on Emerging Security Information Systems and Technologies*, pages 11–16.

Vinarskii, E. (2023). Timed transition tour for race detection in distributed systems. https://github.com/vinevg1996/Timed-Transition-Tour.

Vinarskii, E. and Zakharov, V. (2020). On some properties of timed finite state machines. *System Informatics*, pages 11–20.

Vinarskii, E. M., López, J., Kushik, N., Yevtushenko, N., and Zeghlache, D. (2019). A model checking based approach for detecting SDN races. In *Testing Software and Systems - 31st IFIP WG 6.1 International Conference*, pages 194–211.

Vinarskii, E. M. and Zakharov, V. A. (2018). On the verification of strictly deterministic behavior of timed finite state machines. In *Proceedings of ISP RAS*, pages 325–340.

Wen, C., He, M., Wu, B., Xu, Z., and Qin, S. (2022). Controlled concurrency testing via periodical scheduling. In *44th IEEE/ACM 44th International Conference on Software Engineering*, pages 474–486.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS