2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)

Fault Detection in Timed FSM with Timeouts by SAT-Solving

Omer Nguena Timo
Computer Research Institute of Montréal
Montréal, Canada
omer.nguena-timo@crim.ca

Dimitri Prestat
University of Québec in Montréal
Montréal, Canada
prestat.dimitri@courrier.uqam.ca

Florent Avellaneda
Computer Research Institute of Montréal
Montréal, Canada
florent.avellaneda@crim.ca

***Abstract*—Faults in safety critical real-time systems are not only logical, but they can correspond to violations of timing constraints. They must be detected to avoid system failures with adverse consequences. Developing efficient fault detection techniques for varieties of system models is still challenging. In this paper, we deal with fault detection for timed finite state machines with timeouts (TFSMs-T). TFSM-T is an extension of FSM to model timing constraints in safety-critical real-time systems. We lift a fault detection approach developed for FSM to generate tests detecting both logical faults and violations of time constraints in TFSMs-T. The approach is based on constraint solving and uses mutation machines to represent domains of faulty implementations (mutants) of a specification TFSMs-T. It also avoids enumerating the implementations one by one. We develop a prototype tool and we conduct experiments to evaluate the scalability of the proposed methods.**

***Index Terms*—Fault detection; Mutation testing; Test generation; Timed finite state machines; Timeouts; SAT-solving;**

I. Introduction

The fault domain coverage criterion can be adopted to generate tests revealing faults in safety/security critical systems under test (SUT). The domain can be built from referenced databases or expert knowledge. Efficient test generation methods are needed especially for the fault domains of important sizes, which has motivated the development of an approach [1], [2] leveraging recent advances in the field of (Boolean) constraint solving. The approach has been elaborated to detect logical faults in reactive systems specified with finite state machines (FSMs). We propose to lift the approach to detect both logical faults and violations of time constraints in reactive systems; in particular, we focus on reactive systems specified with timed FSMs with timeouts (TFSMs-T).

TFSM-T [3], [4] is an extension of the classical FSM with timeout transitions for expressing time constraints. Although they express limited types of time constraints as compared to other timed FSMs [4], TFSMs-T have been used to specify reactive systems such as web applications [5] and protocols [6]–[8]. Logical faults in TFSMs-T correspond to unexpected outputs or unexpected state changes. Reducing and increasing waiting time are violations of time constraints. An implementation for a given specification TFSM-T can be represented with a mutated version of the specification TFSM-T also called a mutant. A mutant can be obtained by seeding the specification with an arbitrary number of faults. A fault domain for a specification is then a finite set of possible mutants; it can be built from a list of identified faults to be detected in systems under test. A mutant is nonconforming if its timed output sequence differs from that of the specification for some timed input sequences (tests). Tests covering a fault domain detect all nonconforming mutants in it.

Approaches have been investigated for FSM-based test generation with guaranteed fault coverage [9], [10]. FSMs have been extended to express time constraints, which has resulted in a variety of timed FSMs [4], [5], [11]–[13]. Timed FSMs are not compared to the well-known timed automata for which testing approaches exist [14]–[16]. The testing approaches [5], [12], [13], [17] for timed FSMs integrate the reasoning on time constraints in well-known FSM-based testing approaches [10]. The methods in [1], [2] to verify and generate tests for (extended) FSM specifications are based solving constraints or Boolean expressions, which allows one to take advantage of the efficiency of constraint/SAT solvers; this is a novelty as compared to the work in [18] and the well-known approaches such as the W-method [19]. The high efficiency of using constraint solving in testing software code was demonstrated in [20]. The constraints specifying the mutants surviving given tests; they are defined over the transitions in executions of the mutants. The executions are selected with a so-called distinguishing automaton of the specification and the fault domain that is compactly modeled with a nondeterministic FSM called a mutation machine. A solution of the constraints is a mutant which, if it is nonconforming, allows to generate a test detecting the mutant and many others; then the constraints are upgraded to generate new tests.

Our contribution is to lift the methods in [1], [2] for verifying and generating complete tests to cover fault domains for TFSM-T specifications. The work in [17] addressed similar problems by lifting the W-method. Our approach based on SAT-solving and elimination of FSMs in a fault domain could take advantage of the efficiency of existing solvers. In our work, specifications and mutants are deterministic and input-complete. We define a new distinguishing automaton with timeouts for a TFSM-T specification and a fault domain. The automaton serves to extract transitions in detected mutants and build constraints for specifying test-surviving mutants. Extracting the transitions, we pair input/output transitions with timeout-unexpired transitions allowing to perform the input/output transitions; this is formalized with a new notion of "comb". We have implemented the methods in a prototype tool which we use to evaluate the efficiency of the methods and compare our results with those of the related work.

Organization of the paper: the next section introduces a fault model for TFSMs-T and the coverage of fault models with tests. In Section III we build constraints for the analysis of timed input sequences and the generation of complete test suites. The analysis and generation methods are presented in Section IV. Section V presents an empirical evaluation of the efficiency of the methods with the prototype tool. We conclude the paper in Section VI.

II. Preliminaries

Let $\mathbb{R}_{\geq 0}$ and $\mathbb{N}_{\geq 1}$ denote the sets of non-negative real numbers and non-null natural numbers, respectively.

A. TFSM with Timeouts

Definition 1. A *timed finite state machine with Timeouts* (shortly, TFSM-T) is a 6-tuple $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ where S, I and O are finite non-empty set of *states*, *inputs* and *outputs*, respectively, s_0 is the *initial* state, $\lambda_{\mathcal{S}} \subseteq S \times I \times O \times S$ is an input/output transition relation and $\Delta_{\mathcal{S}} \subseteq S \times (\mathbb{N}_{\geq 1} \cup \{\infty\}) \times S$ is a timeout transition relation defining at least one timeout transition in every state.

TFSMs-T can be described with state transition graphs; the nodes of the graph correspond to the states and the transitions are described with labelled and directed arrows between the states. Figure 1 presents state transition graphs for TFSMs-T.

Our definition of TFSM-T extends the definition in [4] by allowing multiple timeout transitions in the same state, which we use later to compactly represent sets of TFSMs-T. An input/output transition $(s, i, o, s') \in \lambda_{\mathcal{S}}$ defines the output o produced in its source state s when input i is applied. A timeout transition $(s, \delta, s') \in \Delta_{\mathcal{S}}$ defines the timeout δ in state s. A timeout transition can be taken if no input is applied at the current state before the timeout of the transition expires. It is not possible to expect an input beyond the maximal timeout defined in the current state.

A TFSM-T uses a single clock for recording the time elapsing in the states and determining when timeouts expire. The clock is reset when the transitions are performed. A *timed state* of TFSM-T $\mathcal{S}$ is a pair $(s, x) \in S \times \mathbb{R}_{\geq 0}$ where $s \in S$ is a state of $\mathcal{S}$ and $x \in \mathbb{R}_{\geq 0}$ is the current value of the clock and $x < \delta$ for some $\delta \in \mathbb{N}_{\geq 1} \cup \{\infty\}$ such that $(s, \delta, s') \in \Delta_{\mathcal{S}}$. An *execution step* of $\mathcal{S}$ in timed state (s, x) corresponds either to the time elapsing or *performing* an input/output or timeout transition; it is *permitted* by a transition of $\mathcal{S}$. We say that $stp = (s, x)a(s', x') \in (S \times \mathbb{R}_{\geq 0}) \times ((I \times O) \cup \mathbb{R}_{\geq 0}) \times (S \times \mathbb{R}_{\geq 0})$ is an execution step if it satisfies one of the following conditions:

- (timeout step) $a \in \mathbb{R}_{\geq 0}$, $x' = 0$ and $x + a = \delta$ for some δ such that $(s, \delta, s') \in \Delta_{\mathcal{S}}$; then (s, δ, s') is said to permit the step.
- (time-elapsing step) $a \in \mathbb{R}_{\geq 0}$, $x' = x + a$, $s' = s$ and $x + a < \delta$ for some δ and $s'' \in S$ such that there exists $(s, \delta, s'') \in \Delta_{\mathcal{S}}$; then (s, δ, s'') is said to permit the step.
- (input/output step) $a = (i, o)$ with $(i, o) \in I \times O$, $x' = 0$ and there exists $(s, i, o, s') \in \lambda_{\mathcal{S}}$; then (s, i, o, s') is said to permit the step.

Time-elapsing steps satisfy the following time-continuity property w.r.t the same timeout transition: if $(s_1, x_1)d_1(s_2, x_2)d_2(s_3, x_3) \dots d_{k-1}$ (s_k, x_k) is a sequence of time-elapsing steps permitted by the same timeout transition t, then $(s_1, x_1)d_1 + d_2 + \dots + d_{k-1}(s_k, x_k)$ is a time-elapsing step permitted by t. In the sequel, any time-elapsing step permitted by a timeout transition can be represented with a sequence of time-elapsing steps permitted by the same timeout transitions.

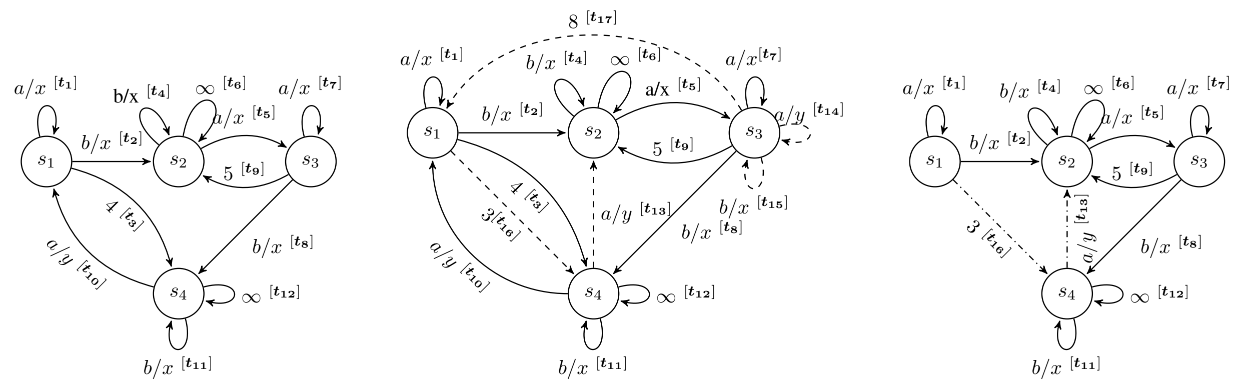
An *execution* of $\mathcal{S}$ in timed state (s, x) is a sequence of steps $e = stp_1 stp_2 \dots stp_n$ with $stp_k = (s_{k-1}, x_{k-1})a_k(s_k, x_k)$, $k \in [1, n]$ such that the following conditions hold:

- $(s_0, x_0) = (s, x)$,
- stp_1 is not an input/output step,
- stp_k is an input/output step implies that stp_{k-1} is a time-elapsing step for every $k \in [1..n]$.

If needed, the elapsing of zero time units can be inserted between a timeout step and an input/output step. Let $d_1 d_2 \dots d_l \in \mathbb{R}^l_{\geq 0}$ be a non-decreasing sequence of real numbers, i.e., $d_k \leq d_{k+1}$ for every $k = 1..l-1$. The sequence $\sigma_e = ((i_1, o_1), d_1)((i_2, o_2), d_2) \dots ((i_l, o_l), d_l)$ in $((I \times O) \times \mathbb{R}_{\geq 0})^*$ with $l < n$ is a *timed input/output sequence* of execution e if $(i_1, o_1)(i_2, o_2) \dots (i_l, o_l)$ is the maximal sequence of input/output pairs occurring in e. The delay d_k for each input/output pair (i_k, o_k), with $k = 1..l$, is the amount of the time elapsed from the beginning of e to the occurrence of (i_k, o_k). The *timed input sequence* and the *timed output sequence* of e are $(i_1, d_1)(i_2, d_2)...(i_l, d_l)$ and $(o_1, d_1)(o_2, d_2)...(o_l, d_l)$, respectively. We let $inp(e)$ and $out(e)$ denote the timed input and output sequences of execution e. Given a timed input sequence α, let $out_{\mathcal{S}}((s, x), \alpha)$ denote the set of all timed output sequences which can be produced by $\mathcal{S}$ when α is applied in s, i.e., $out_{\mathcal{S}}((s, x), \alpha) = \{out(e) \mid e \text{ is an execution of } \mathcal{S} \text{ in } (s, x) \text{ and } inp(e) = \alpha\}$.

A TFSM-T $\mathcal{S}$ is *deterministic* (DTFSM-T) if it defines at most one input-output transition for each tuple $(s, i) \in S \times I$ and exactly one timeout transition in each state; otherwise, it is *nondeterministic*. $\mathcal{S}$ is *initially connected* if it has an execution from its initial state to each of its states. $\mathcal{S}$ is *complete* if for each tuple $(s, i) \in S \times I$ it defines at least one input-output transition. Note that the set of timed input sequences defined in each state of a complete machine $\mathcal{S}$ is $(I \times \mathbb{R}_{\geq 0})^*$.

We consider distinguishability and equivalence relations between states of complete TFSMs-T [5]. Let p and s be the states of two complete TFSMs-T over the same inputs and outputs. Given a timed input sequence α, p and s are *distinguishable* (with distinguishing input sequence α), denoted $p \not\simeq_\alpha s$, if the sets of timed output sequences in $out_{\mathcal{S}}((p, 0), \alpha)$



(a) A specification TFSM $\mathcal{S}_1$ (b) A mutation TFSM $\mathcal{M}_1$ (c) A mutant TFSM $\mathcal{P}_1$

Fig. 1: Examples of TFSMs, state s_1 is initial. Dashed arrows represent mutated transitions and solid arrows represent transitions of the specification. Names of transitions appear in brackets.

and $out_{\mathcal{S}}((s,0),\alpha)$ differ; otherwise they are *equivalent* and we write $s \simeq p$.

TFSM-T $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ is a *submachine* of TFSM-T $\mathcal{P} = (P, p_0, I, O, \lambda_{\mathcal{P}}, \Delta_{\mathcal{P}})$ if $S \subseteq P$, $s_0 = p_0$, $\lambda_{\mathcal{S}} \subseteq \lambda_{\mathcal{P}}$ and $\Delta_{\mathcal{S}} \subseteq \Delta_{\mathcal{P}}$.

Example 1. Figure 1 presents two initially connected TFSMs-T $\mathcal{S}_1$ and $\mathcal{M}_1$. $\mathcal{M}_1$ is nondeterministic; it defines two timeout transitions in states s_1 and s_3. $\mathcal{S}_1$ is a complete deterministic submachine of $\mathcal{M}_1$. Here are two executions of $\mathcal{M}_1$: $(s_1,0) \xrightarrow[t_3|t_{16}]{2} (s_1,2) \xrightarrow[t_2]{b/x} (s_2,0) \xrightarrow[t_6]{1} (s_2,1) \xrightarrow[t_5]{a/x} (s_3,0) \xrightarrow[t_{17}]{8} (s_1,0) \xrightarrow[t_3]{4} (s_4,0) \xrightarrow[t_{12}]{0.5} (s_4,0.5) \xrightarrow[t_{10}]{a/y} (s_1,0)$ and $(s_1,0) \xrightarrow[t_{16}]{3} (s_4,0) \xrightarrow[t_{12}]{0.5} (s_4,0.5) \xrightarrow[t_{11}]{b,x} (s_4,0) \xrightarrow[t_{12}]{1} (s_4,1) \xrightarrow[t_{13}]{a,y} (s_2,0) \xrightarrow[t_6]{12.5} (s_2,12.5) \xrightarrow[t_5]{a,x} (s_3,0)$. Let us explain the first execution. It consists of 8 steps represented with arrows between timed states. The label above an arrow is either a delay or an input-output pair. The label below an arrow indicates the transitions permitting the step. The first, third and seventh steps are time-elapsing. The second, fourth and last steps are input-output. The fifth and the sixth steps are timeout. The first step is permitted by either t_3 or t_{16} because their timeouts are not expired 2 units after the machine has entered state $(s_1,0)$. The timeout of t_{12} permitting the seventh step has not expired before t_{10} is performed at the last step. The timed input/output sequence for the first execution is $((b,x),2)((a,x),3)((a,y),15.5)$.

B. Complete Test Suite for Fault Models

Henceforth the TFSMs-T are complete and initially connected. Let $\mathcal{S} = (S, s_0, I, O, \lambda_{\mathcal{S}}, \Delta_{\mathcal{S}})$ be a DTFSM-T, called the *specification* machine.

Definition 2. A nondeterministic TFSM-T $\mathcal{M} = (M, m_0, I, O, \lambda_{\mathcal{M}}, \Delta_{\mathcal{M}})$ is a *mutation machine* for $\mathcal{S}$ if $\mathcal{S}$ is a submachine of $\mathcal{M}$. Transitions in $\lambda_{\mathcal{M}}$ but not in $\lambda_{\mathcal{S}}$ or in $\Delta_{\mathcal{M}}$ but not in $\Delta_{\mathcal{S}}$ are called *mutated.*

We use mutation machines to compactly represent possible implementations of the specification machines. A *mutant* is a deterministic submachine of $\mathcal{M}$ different from the specification. We let *Mut*($\mathcal{M}$) denote the set of mutants in $\mathcal{M}$. Every mutant represents an implementation seeded with faults. Faults are represented with mutated transitions which can be viewed as can be alternatives for transitions of the specification machines. Mutated transitions can represent transfer faults, output faults, changes of timeouts and adding of extra-states. Every mutant defines a subset of mutated transitions.

A transition t is *suspicious* in $\mathcal{M}$ if $\mathcal{M}$ defines another transition t' from the source state of t such that both transitions either have the same input or are timeout transitions. In other words, we can substitute t for t' in a mutant or the specification to obtain another mutant. A transition of the specification is called *untrusted* if it is suspicious in the mutation machine; otherwise, it is *trusted*. Every trusted transition is defined in each mutant.

Let $\mathcal{P}$ be a mutant with an initial state p_0 of the mutation machine $\mathcal{M}$ of $\mathcal{S}$. We use the state equivalence relation $\simeq$ to define conforming mutants.

Definition 3 (Conforming mutants and detected mutants)**.** Mutant $\mathcal{P}$ is *conforming* to $\mathcal{S}$, if $p_0 \simeq s_0$; otherwise, it is *nonconforming* and a timed input sequence α such that $out_{\mathcal{P}}((p_0,0),\alpha) \neq out_{\mathcal{S}}((s_0,0),\alpha)$ is said to *detect* $\mathcal{P}$. $\mathcal{P}$ *survives* α if α does not detect $\mathcal{P}$.

The set *Mut*($\mathcal{M}$) of all mutants in mutation machine $\mathcal{M}$ is called a *fault domain* for $\mathcal{S}$. If $\mathcal{M}$ is deterministic and complete then *Mut*($\mathcal{M}$) is empty. A *fault model* is the tuple $\langle \mathcal{S}, \simeq, \mathit{Mut}(\mathcal{M}) \rangle$ following [1], [2], [21]. Let $\lambda_{\mathcal{M}}(s,i)$ denote the set of input/output transitions defined in state s with input i and $\Delta_{\mathcal{M}}(s)$ denote the set of timeout transitions defined in state s. The number of mutants in *Mut*($\mathcal{M}$) is given by the formula $|\mathit{Mut}(\mathcal{M})| = \Pi_{(s,i) \in S \times I} |\lambda_{\mathcal{M}}(s,i)| \times \Pi_{s \in S} |\Delta_{\mathcal{M}}(s)| - 1$.

Definition 4. A test for $\langle \mathcal{S}, \simeq, \mathit{Mut}(\mathcal{M}) \rangle$ is a timed input sequence. A *complete test suite* for $\langle \mathcal{S}, \simeq, \mathit{Mut}(\mathcal{M}) \rangle$ is a set

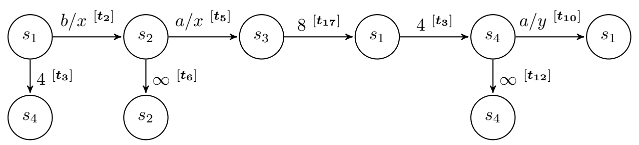


Fig. 2: A comb for the execution in Example 1.

of tests detecting all nonconforming mutants in $\mathit{Mut}(\mathcal{M})$.

Mutated transitions representing faults can be introduced with different types of mutation operations. These operations include changing target states, outputs, or timeouts in transitions. Changing timeouts is irrelevant for classical FSM and the corresponding faults could be detected with adequate tests.

To generate a complete test suite, a test can be computed for each nonconforming mutant by enumerating the mutants one-by-one, which would be inefficient for huge fault domains. We avoid the one-by-one enumeration of the mutants with constraints specifying only test-surviving mutants.

III. Specifying test-surviving mutants

The mutants surviving a test cannot produce any execution with an unexpected timed output sequence for the test. We encode them with Boolean formulas over Boolean transition variables of which the values indicate the presence or absence of transitions of the mutation machine in mutants.

A. Revealing Combs and Involved Mutants

The mutants detected by a test α exhibit a revealing execution which produces an unexpected timed output sequence and has α as the test. Such an execution is permitted by transitions forming a comb-subgraph in the state transition diagram of the mutation machine. Intuitively, a comb for an execution is nothing else but a path augmented with timeout-unexpired transitions, i.e., transitions of which the timeouts have not expired prior to an input-output step. These additional timeout transitions are also needed to specify detected mutants and eliminate them from the fault domain. We simply represent combs with sequences of transitions.

A *comb* of an execution $e = stp_1 stp_2 \ldots stp_n$ is the sequence of transitions $t_1 t_2 \cdots t_n$ such that t_i permits stp_i for every $i = 1..n$. As exemplified in Figure 2, combs are kinds of non linear graph-structures. We say that comb $t_1 t_2 \cdots t_n$ is *enabled* by the input sequence of e. Each timeout or input/output step in e is permitted with a unique transition. However, each time-elapsing step is permitted by a timeout transition with an unexpired timeout, i.e., the timeout is not greater than the clock value in the source timed state of the time-elapsing step. We will follow timeout transitions which permit time-elapsing steps with the symbol "$\curvearrowright$". So, several combs can permit the same execution since several timeout transitions permit the same time-elapsing step.

Example 2. The first execution in Example 1 corresponds to two combs; this is because its first step is permitted by either t_3 or t_{16}. The first comb, $t_3 \curvearrowright t_2 t_6 \curvearrowright t_5 t_{17} t_3 t_{12} \curvearrowright t_{10}$ is represented in Figure 2. Transition t_3 is not performed, but it permits a time-elapsing step before t_2 is performed; then it is followed by $\curvearrowright$. The timeouts of the (performed) transitions represented with horizontal arrows have expired. The first comb is deterministic; the second comb $t_{16} \curvearrowright t_2 t_6 \curvearrowright t_5 t_{17} t_3 t_{12} \curvearrowright t_{10}$ is nondeterministic because t_{16} and t_3 are two suspicious timeout transitions defined in s_1.

Combs for executions with unexpected timed output sequences reveal nonconforming mutants unless they belong only to nondeterministic submachines. The combs belonging only to nondeterministic submachines have two transitions which are not defined in the same mutant; they are called nondeterministic. A comb is *nondeterministic* if it has two suspicious input-output transitions with the same input or two timeout transitions defined in an identical state of the mutation machine; otherwise, it is a *deterministic comb*. Clearly, combs in a mutant or the specification are deterministic. A nondeterministic submachine of a mutation machine can contain both deterministic and nondeterministic combs.

Definition 5. Let π be a comb of an execution e_1 from $(s_0, 0)$ and α be a timed input sequence. We say that π is *α-revealing* if there exists an execution e_2 of $\mathcal{S}$ such that $\alpha = inp(e_1) = inp(e_2)$, $out(e_1) \neq out(e_2)$. Comb π is revealing if it is β-revealing for some timed input sequence β.

The specification contains no revealing comb because its executions always produce expected timed output sequences. Only mutants, nondeterministic or incomplete submachines of a mutation machine can contain revealing combs; but mutants contain only deterministic revealing combs.

Let $Rev_\alpha(\mathcal{P})$ denote the set of deterministic α-revealing combs of machine $\mathcal{P}$.

Lemma 1. $Rev_\alpha(\mathcal{M}) = \bigcup_{\mathcal{P} \in \mathit{Mut}(\mathcal{M})} Rev_\alpha(\mathcal{P})$, for any test α.

Lemma 2. Let $\pi \in Rev_\alpha(\mathcal{M})$ for a test α and $\mathcal{P} \in \mathit{Mut}(\mathcal{M})$. $\mathcal{P}$ is not detected by α if and only if $\pi \notin Rev_\alpha(\mathcal{P})$.

The comb for the second execution in Example 1, namely $t_{16} t_{12} \curvearrowright t_{11} t_{12} \curvearrowright t_{13} t_6 \curvearrowright t_4$ is revealing and contained in the mutant and mutation machine in Figure 1. To prevent the fourth execution, we must prevent one of the transitions in the comb. For example, if we prevent t_{16}, the other timeout transition t_3 defined in s_1 will be performed, yielding to another execution which cannot be performed in the mutant $\mathcal{P}_1$, but rather in $\mathcal{S}_1$. Clearly $\mathcal{S}_1$ is not detected by $(b, 3.5)(a, 4.5)(a, 17)$, in the contrary of $\mathcal{P}_1$.

We introduce a distinguishing automaton which will serve to compute deterministic combs.

Definition 6. Given a specification machine $\mathcal{S} = (S, s_0, I, O, \lambda_\mathcal{S}, \Delta_\mathcal{S})$ and a mutation machine $\mathcal{M} = (M, m_0, I, O, \lambda_\mathcal{M}, \Delta_\mathcal{M})$, a finite automaton $\mathcal{D} = (C \cup \{\nabla\}, c_0, I, \lambda_\mathcal{D}, \Delta_\mathcal{D}, \nabla)$, where $C \subseteq S \times M \times (\mathbb{N}_{\geq 0} \cup \{\infty\}) \times (\mathbb{N}_{\geq 0} \cup \{\infty\})$, $\lambda_\mathcal{D} \subseteq C \times I \times C$ is the input transition relation, $\Delta_\mathcal{D} \subseteq C \times (\mathbb{N}_{\geq 1} \cup \{\infty\}) \times C$ is the timeout transition relation

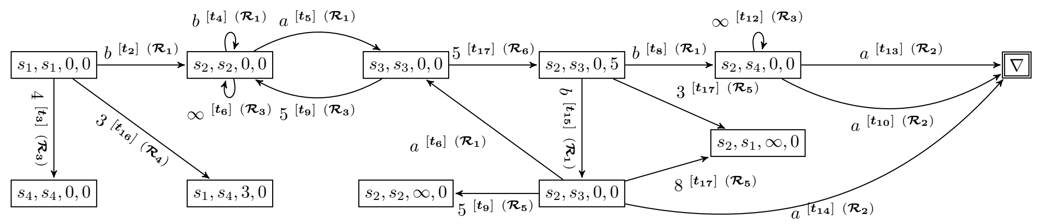

$s_1, s_1, 0, 0$
b [t_2] ($\mathcal{R}_1$)
b [t_4] ($\mathcal{R}_1$)
a [t_5] ($\mathcal{R}_1$)
$s_2, s_2, 0, 0$
$s_3, s_3, 0, 0$
∞ [t_6] ($\mathcal{R}_3$)
5 [t_9] ($\mathcal{R}_3$)
5 [t_{17}] ($\mathcal{R}_6$)
$s_2, s_3, 0, 5$
b [t_8] ($\mathcal{R}_1$)
∞ [t_{12}] ($\mathcal{R}_3$)
$s_2, s_4, 0, 0$
a [t_{13}] ($\mathcal{R}_2$)
∇
3 [t_{17}] ($\mathcal{R}_5$)
a [t_{10}] ($\mathcal{R}_2$)
$s_2, s_1, \infty, 0$
4 [t_3] ($\mathcal{R}_3$)
3 [t_{16}] ($\mathcal{R}_4$)
a [t_6] ($\mathcal{R}_1$)
b [t_{15}] ($\mathcal{R}_1$)
$s_4, s_4, 0, 0$
$s_1, s_4, 3, 0$
$s_2, s_2, \infty, 0$
$s_2, s_3, 0, 0$
5 [t_9] ($\mathcal{R}_5$)
8 [t_{17}] ($\mathcal{R}_5$)
a [t_{14}] ($\mathcal{R}_2$)

Fig. 3: A fragment of the distinguishing automaton of $\mathcal{S}_1$ and $\mathcal{M}_1$ with timeouts; $(s_1, s_1, 0, 0)$ is initial.

and ∇ is the accepting (sink) state, is the *distinguishing automaton* with timeouts for $\mathcal{S}$ and $\mathcal{M}$, if it holds that:

- $c_0 = (s_0, m_0, 0, 0)$
- For each $(s, m, x_s, x_m) \in C$ and $i \in I$

($\mathcal{R}_1$) : $((s, m, x_s, x_m), i, (s', m', 0, 0)) \in \lambda_{\mathcal{D}}$ if there exists $(s, i, o, s') \in \lambda_{\mathcal{S}}, (m, i, o', m') \in \lambda_{\mathcal{M}}$ s.t. $o = o'$

($\mathcal{R}_2$) : $((s, m, x_s, x_m), i, \nabla) \in \lambda_{\mathcal{D}}$ if there exists $(s, i, o, s') \in \lambda_{\mathcal{S}}$, $(m, i, o', m') \in \lambda_{\mathcal{M}}$ s.t. $o \neq o'$

- For each $(s, m, x_s, x_m) \in C$ and the only timeout transition $(s, \delta_s, s') \in \Delta_{\mathcal{S}}$ defined in the state of the deterministic specification

($\mathcal{R}_3$) : $((s, m, x_s, x_m), \delta_m - x_m, (s', m', 0, 0)) \in \Delta_{\mathcal{D}}$ if there exists $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ s.t. $\delta_s - x_s = \delta_m - x_m$ and $\delta_m - x_m > 0$

($\mathcal{R}_4$) : $((s, m, x_s, x_m), \delta_m - x_m, (s, m', x_s + \delta_m - x_m, 0)) \in \Delta_{\mathcal{D}}$ if there exists $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ s.t. $\delta_m - x_m < \delta_s - x_s$ and $\delta_s \neq \infty$ and $\delta_m - x_m > 0$

($\mathcal{R}_5$) : $((s, m, x_s, x_m), \delta_m - x_m, (s, m', \infty, 0)) \in \Delta_{\mathcal{D}}$ if there exists $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ s.t. $\delta_m - x_m < \delta_s - x_s$ and $\delta_s = \infty$ and $\delta_m - x_m > 0$

($\mathcal{R}_6$) : $((s, m, x_s, x_m), \delta_s - x_s, (s', m, 0, x_m + \delta_s - x_s)) \in \Delta_{\mathcal{D}}$ if there exists $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ s.t. $\delta_s - x_s < \delta_m - x_m$ and $\delta_m \neq \infty$ and $\delta_s - x_s > 0$

($\mathcal{R}_7$) : $((s, m, x_s, x_m), \delta_s - x_s, (s', m, 0, \infty)) \in \Delta_{\mathcal{D}}$ if there exists $(m, \delta_m, m') \in \Delta_{\mathcal{M}}$ s.t. $\delta_s - x_s < \delta_m - x_m$ and $\delta_m = \infty$ and $\delta_s - x_s > 0$,
where $\infty - x = \infty$ if x is finite or infinite and $\infty + \infty = \infty$

- $(\nabla, x, \nabla) \in \lambda_{\mathcal{D}}$ for all $x \in I$ and $(\nabla, \infty, \nabla) \in \Delta_{\mathcal{D}}$

The seven rules $\{\mathcal{R}_i\}_{i=1..7}$ introduce input transitions and timeout transitions in $\mathcal{D}$. Each state (s, m, x_s, x_m) of $\mathcal{D}$ is composed of a state s of $\mathcal{S}$, a state m of $\mathcal{M}$, the value x_s of the clock of $\mathcal{S}$ and the value x_m of the clock of $\mathcal{S}$. Input transitions are introduced with $\mathcal{R}_1$ and $\mathcal{R}_2$. Rule $\mathcal{R}_2$ adds a transition to accepting state ∇ if different outputs are produced in s and m for the same input; otherwise $\mathcal{S}$ and $\mathcal{M}$ move to next states, as described with rule $\mathcal{R}_1$.
The rules $\{\mathcal{R}_i\}_{i=3..7}$ introduce timeout transitions of the form $((s, m, x_s, x_m), \delta, (s', m', x'_s, x'_m))$. δ is the delay for reaching the only timeout δ_s defined in s from (s, x_s) or a timeout defined in m from (m, x_m), since multiple timeouts can be defined in states of $\mathcal{M}$. δ can be greater than the delays for reaching some timeouts defined in m; however, δ is never greater than $\delta_s - x_s$, the delay for reaching the only timeout δ_s in s. So, $x'_s = 0$ if $\delta = \delta_s - x_s$; a similar statement holds for the clock and a selected timeout transition of $\mathcal{M}$. In $\mathcal{R}_3$, both the timeout in s and a timeout in m expire after δ time units. In $\mathcal{R}_4$ only a timeout defined in m expires after δ time units and the only finite timeout defined in s does not expire after δ time units. A similar phenomenon is described with $\mathcal{R}_5$; but contrarily to $\mathcal{R}_4$, the only timeout in s is ∞ and we set the clock of the specification to ∞. Setting the clock to ∞ expresses the fact that we do not care any more about finite values of x'_s because only the infinite timeout in s must be reached. Without this latter abstraction on the values of x'_s, the size of C could be infinite because we could have to apply $\mathcal{R}_4$ infinitely. The rules $\mathcal{R}_6$ and $\mathcal{R}_7$ are similar to $\mathcal{R}_4$ and $\mathcal{R}_5$, but the only timeout in s expires before a timeout in m.

An execution of $\mathcal{D}$ from a timed state (c, x) is a sequence of steps between timed states of $\mathcal{D}$; it can be defined similarly to that for a TFSM-T. An execution starting from $(c_0, 0)$ and ending at ∇ is called *accepted.* As for TFSMs-T, we can associate every execution of $\mathcal{D}$ with a comb and a timed input sequence. A comb of $\mathcal{D}$ is accepted if it corresponds to an accepted execution. Each transition in $\mathcal{D}$ is defined by a transition of $\mathcal{S}$ and a transition of $\mathcal{M}$. So, every comb of $\mathcal{D}$ is defined by a comb of $\mathcal{S}$ and a comb of $\mathcal{M}$, i.e., it has been obtained by composing the transitions in two combs.

Lemma 3. A comb π of $\mathcal{M}$ is revealing if it defines an accepted comb of $\mathcal{D}$.

A revealing comb can be common to many mutants, in which case those mutants are said to be *involved* in the comb. Mutants involved in a revealing comb are detected by the tests enabling the comb. We let $Susp_X$ denote the set of suspicious transitions in X.

Lemma 4. A mutant $\mathcal{P}$ is involved in a revealing comb π of $\mathcal{M}$ if and only if $Susp_\pi \subseteq Susp_{\mathcal{P}}$.

Lemma 4 assumes that mutants are known and indicates how to check if a mutant is involved in a given (deterministic or nondeterministic) revealing comb. However, we want to avoid the enumeration of the mutants in eliminating the nonconforming mutants detected by a test; we also want to generate tests corresponding to deterministic revealing combs because they detect nonconforming mutants as stated in

Lemma 2. So we will focus on computing only deterministic α-revealing combs in $Rev_\alpha(\mathcal{M})$ from $\mathcal{D}$. This is done by a Breadth-first search of sink state ∇ in $\mathcal{D}$ while performing transitions of $\mathcal{D}$ defined with transitions of $\mathcal{M}$ which cannot be defined in an identical mutant. The search is step-wise and guided by timed inputs in α; it consists to perform a timeout transition in $\mathcal{D}$ whenever the delay between the current and the previous input in α is greater than the timeout of the transition or to perform an input transition in $\mathcal{D}$ when the current state in $\mathcal{D}$ defines a timeout smaller than delay between the current and the previous input.

Example 3. Figure 3 presents a fragment of the distinguishing automaton for the $\mathcal{S}_1$ and $\mathcal{M}_1$ in Figure 1. It is relevant for the revealing combs for $\alpha = (b, 0.5)(a, 1)(b, 6.7)(a, 7.2)$. $[t](\mathcal{R})$ indicates an input/output transition or a timeout transition t of the mutation machine defining the transition of the automaton introduced with rule $\mathcal{R}$; e.g., the timeout transition $((s_3, s_3, 0, 0), 5, (s_2, s_3, 0, 5))$ is defined by t_{17} and the timeout of t_{17} has not expired when $\mathcal{R}_6$ is applied. One of the six deterministic α-revealing combs is: $\boldsymbol{t}_3 \curvearrowright t_2 t_6 \curvearrowright t_5 \boldsymbol{t}_{17} \curvearrowright \boldsymbol{t}_8 t_{12} \curvearrowright \boldsymbol{t}_{10}$; the transitions in bold are suspicious.

B. Encoding Submachines Involved in Revealing Combs

We introduce a Boolean variable for each suspicious transition in mutation machine $\mathcal{M}$; Based on Lemma 4, we build Boolean formulas over these variables to encode the mutants involved in revealing combs. A solution of such a formula assigns a truth value to every transition variable. We say that a solution of a formula *determine* a submachine $\mathcal{P}$ of $\mathcal{M}$ if $\mathcal{P}$ is composed of all trusted transitions and every suspicious transition for which the value of the corresponding variable is $True$. In general, the submachine for the solution of a formula can be non-initially connected, nondeterministic or incomplete. Later we encode mutants (deterministic and complete submachines) with additional formulas. For now, let us encode the submachines involved in revealing combs of $\mathcal{M}$ with Boolean formulas.

Let α be a test and $Rev_\alpha(\mathcal{M}) = \{\pi_1, \pi_2, \ldots, \pi_n\}$ be the set of deterministic revealing combs of $\mathcal{M}$ enabled by α. We encode a comb $\pi = t_1 t_2 \ldots t_m$ of $\mathcal{M}$ with the Boolean formula $\varphi_\pi = \bigwedge_{t_i \in Susp_\pi} t_i$, the conjunction of all the suspicious transitions in π. Clearly, any solution of φ_π determines a submachine (of the mutation machine) containing the comb π; The executions associated with π are defined in such a submachine which is detected by α. Conversely, each submachine determined by a solution of the negation of φ_π does not contain π; it cannot define any execution associated with π and is not detected by α. Such a submachine is not necessarily a mutant because it can be nondeterministic or incomplete. For the set of deterministic revealing combs in $Rev_\alpha(\mathcal{M})$, let us define the formula $\varphi_\alpha = \bigvee_{\pi \in Rev_\alpha(\mathcal{M})} \varphi_\pi$. The set of (possibly nondeterministic or incomplete) submachines of $\mathcal{M}$ detected by α is determined by a solution of φ_α. A submachine of $\mathcal{M}$ surviving α cannot contain any comb in $Rev_\alpha(\mathcal{M})$ and it cannot be determined by a solution of the negation of φ_α, as stated in Lemma 5.

Lemma 5. A submachine of $\mathcal{M}$ survives a test α if and only if it can be determined by a solution of $\neg\varphi_\alpha$.

To obtain the mutants surviving α, we remove from the solutions of $\neg\varphi_\alpha$ those determining nondeterministic or incomplete submachines. This is possible with a Boolean formula encoding only the mutants in $\mathcal{M}$.

C. Encoding the Mutants in a Mutation Machine

Let $T = t_1, t_2, \ldots t_n$ be a set of Boolean variables for all the transitions t_i of $\mathcal{M}$, $i = 1..n$. Let us define the Boolean formula ξ_T as follows:

$$\xi_T = \bigwedge_{k=1\ldots n} \bigwedge_{l=k+1..n} (\neg t_k \vee \neg t_l) \wedge \bigvee_{k=1..n} t_i$$

A solution of ξ_T assigns $True$ to exactly one selected variable and assigns $False$ to all other variables. Note that ξ_T is a CNF-SAT formula and it can be solved [22].

Let $\mathcal{M}$ be a mutation machine for the specification machine $\mathcal{S}$. Clearly, $\lambda_\mathcal{S} \subseteq \lambda_\mathcal{M}$ and $\Delta_\mathcal{S} \subseteq \Delta_\mathcal{M}$. A deterministic and complete submachine of $\mathcal{M}$ selects one transition in $\lambda_\mathcal{M}(s, i)$ and one transition in $\Delta_\mathcal{M}(s)$ for every state s and input i; it is therefore determined by a solution of $\varphi_\mathcal{M}$ defined as follows:

$$\varphi_\mathcal{M} = \bigwedge_{(m,i) \in M \times I} \xi_{\lambda_\mathcal{M}(m,i)} \wedge \bigwedge_{m \in M} \xi_{\Delta_\mathcal{M}(m)} \wedge \bigvee_{t \in \lambda_\mathcal{S} \cup \Delta_\mathcal{S}} \neg t$$

The specification cannot be determined by a solution of $\varphi_\mathcal{M}$ because its subformula $\bigvee_{t \in \lambda_\mathcal{S} \cup \Delta_\mathcal{S}} \neg t$ encodes the rejection of transitions of the specification, since $\lambda_\mathcal{S} \cup \Delta_\mathcal{S} \subseteq \lambda_\mathcal{M} \cup \Delta_\mathcal{M}$. The graph composed of the transitions selected by a solution can be disconnected, in which case it does not represent any mutant; a mutant can be obtained by extracting the transitions connected to the initial state.

Lemma 6. A submachine of $\mathcal{M}$ is complete and deterministic if and only if it is determined by a solution of $\varphi_\mathcal{M}$.

Theorem 1. A mutant survives the test α if it is determined by a solution of $\neg\varphi_\alpha \wedge \varphi_\mathcal{M}$.

Considering the revealing combs for $\alpha = (b, 0)(a, 0)$ $(b, 5)(a, 5)$, we use the suspicious transitions in the six revealing combs in Example 3 to compute $\neg\varphi_\alpha = (\neg t_3 \vee \neg t_{17} \vee \neg t_8 \vee \neg t_{10}) \wedge (\neg t_3 \vee \neg t_{17} \vee \neg t_8 \vee \neg t_{13}) \wedge (\neg t_3 \vee \neg t_{17} \vee \neg t_{15} \vee \neg t_{14}) \wedge (\neg t_{16} \vee \neg t_{17} \vee \neg t_8 \vee \neg t_{10}) \wedge (\neg t_{16} \vee \neg t_{17} \vee \neg t_8 \vee \neg t_{13}) \wedge (\neg t_{16} \vee \neg t_{17} \vee \neg t_{15} \vee \neg t_{14})$. The mutant composed with the transitions $t_1, t_2, t_4, t_6, t_5, t_7, t_9, t_{15}, t_{13}, t_{12}, t_{11}$ and t_{16} is determined by a solution of $\neg\varphi_\alpha$ and it survives α. The submachine with the transitions t_1, t_2, t_3, t_4 and t_6 is determined by another solution of $\neg\varphi_\alpha$; it is neither a mutant nor a solution of $\varphi_{\mathcal{M}_1}$.

The mutants surviving a test α can be partitioned into conforming mutants and nonconforming mutants which can only be detected with a test different from α. Nonconforming mutants can be used to generate additional tests and upgrade the constraints. The generated test suite is complete if the solutions of the constraints determine only conforming mutants. This is the intuition of the test verification and generation methods below. The methods avoid a one-by-one enumeration of the mutants because a single test eliminates many of them.

```
Procedure Verify_completeness ($\varphi_{fd}, E, \mathcal{D}$);
Input  : $\varphi_{fd}$, a Boolean expression specifying a fault domain
Input  : $E$, a (possibly empty) set of tests
Input  : $\mathcal{D}$, the distinguishing automaton of $\mathcal{M}$ and $\mathcal{S}$
Output : $\alpha \neq \varepsilon$, a test detecting a nonconforming mutant surviving $E$; $\alpha = \varepsilon$, if $E$ is complete
initialization : $\varphi_E := \bigwedge_{\alpha \in E} \neg\varphi_\alpha$    $\varphi_{fd} := \varphi_{fd} \wedge \varphi_E$
$\varphi_\mathcal{P}$ := False    $\alpha := \varepsilon$;
repeat
    $\varphi_{fd} := \varphi_{fd} \wedge \neg\varphi_\mathcal{P}$;
    $\mathcal{P}$ := Determine_a_submachine($\varphi_{fd}$);
    if $\mathcal{P} \neq null$ then
        Build $\mathcal{D}_\mathcal{P}$, the distinguishing automaton of $\mathcal{S}$ and $\mathcal{P}$;
        if $\mathcal{D}_\mathcal{P}$ has no sink state then
            $\varphi_\mathcal{P} := \bigwedge_{t \in \lambda_\mathcal{P} \cup \Delta_\mathcal{P}} t$;
        else
            Set $\alpha$ to the timed input sequence of an accepted comb of the distinguishing automaton $\mathcal{D}_\mathcal{P}$;
        end
    end
until $\alpha \neq \varepsilon$ or $\mathcal{P} = null$;
return ($\varphi_{fd}, \alpha$);
```
Algorithm 1: Verifying the completeness of given tests.

IV. Verifying and Generating a Complete Test Suite

Let $E = \{\alpha_1, \alpha_2, \dots, \alpha_n\}$ be a test suite and $\langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$ be a fault model. Our method for verifying whether E is complete works in three steps. First we build the Boolean expression $\bigwedge_{\alpha \in E} \neg\varphi_\alpha \wedge \varphi_\mathcal{M}$ encoding the mutants surviving E; this is based on Theorem 1. Secondly, we use a solver to determine a mutant surviving the Boolean expression. Thirdly, we decide that E is a complete test suite if there is no mutant surviving E or all the mutants surviving the tests in E are conforming. Procedure *Verify_completeness* in Algorithm 1 implements the method. It makes a call to *Determine_a_submachine* for obtaining a mutant in a fault domain specified with φ_{fd}. *Determine_a_submachine* can use an efficient SAT-solver to solve φ_{fd} and build mutants from solutions. *Determine_a_submachine* returns *null* when φ_{fd} is unsatisfiable, i.e., the fault domain is empty. *Verify_completeness* always terminates; this is because the size of the fault domain and the number of revealing combs for a test are finite, and the SAT problem is decidable.

Procedure *Generate_complete_test_suite* in Algorithm 2 implements the iterative generation of a complete test suite. In each iteration step, a new test is generated to detect a surviving mutant returned by *Verify_completeness* if the mutant is nonconforming; otherwise the mutant is discarded from the set of surviving mutants. *Generate_complete_test_suite* always terminates because there are finitely many mutants in the fault domain, *Verify_completeness* always terminates and the

```
Procedure Generate_complete_test_suite
 ($E_{init}, \langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$);
Input  : $E_{init}$, an initial (possibly empty) set of timed input sequences
Input  : $\langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$, a fault model
Output : $E$, a complete test suite for $\langle \mathcal{S}, \simeq, Mut(\mathcal{M})\rangle$
Compute $\varphi_\mathcal{M}$, the boolean formula encoding all the mutants in $Mut(\mathcal{M})$;
Build $\mathcal{D}$, the distinguishing automaton of $\mathcal{S}$ and $\mathcal{M}$;
$\varphi_{fd} := \varphi_\mathcal{M}$;
$E := \emptyset$;
$E_{curr} := E_{init}$;
repeat
    $E := E \cup E_{curr}$;
    $(\varphi_{fd}, \alpha)$ := Verify_completeness($\varphi_{fd}, E_{curr}, \mathcal{D}$);
    $E_{curr} := \{\alpha\}$;
until $\alpha = \epsilon$;
return $E$;
```
Algorithm 2: Generating a complete test suite E from E_{init}.

number of surviving mutants is reduced at every iteration step.

The result of an execution of *Verify_completeness* with input $E_{init} = \{(b, 0.5)(a, 1)(b, 6.7)(a, 7.2)\}$ is the nonempty test $(a, 3)$, which indicates that E_{init} is not complete. We can generate additional tests to be added to E and obtain a complete test suite. An execution of *Generate_complete_test_suite* with E_{init} produces five tests detecting all the 31 mutants in the fault domain. The tests are the following: $(b, 0.5)(a, 1)(b, 6.7)(a, 7.2)$, $(a, 3)$, $(a, 4)(a, 8)$, $(b, 0)(a, 0)(b, 0)(a, 0)$ and $(b, 0)(a, 0)(a, 0)$. The generated test suite includes identical untimed sequences applied after different delays, i.e., the delays are needed for the fault detection.

V. Experimental results

We implemented in the C++ language a prototype tool for an empirical evaluation of the efficiency of the proposed methods. The experiment was accomplished with a computer equipped with the processor Intel(R) Core(TM) i5-7500 CPU @ 3.40 GHz and 32 GB RAM. The tool uses the solver cryptoSAT [22]. We use the tool to evaluate the scalability of the proposed methods with examples of TFSMs-T.

We consider a TFSM-T specification of the Trivial File Transfer Protocol (TFTP) [7]. TFTP is timeouts-dependent and it has already been tested in [6]. Our model focuses on the behavior of reading files; it is inspired by [6], [7]. No more than a given number of packets are transferred and the timeout for expecting a packet equals three seconds. Moreover, we assume the file exists. The number of packets determines the number of states in the specification machine.

For two packets, the specification has 3 states and 23 transitions. We manually built a mutation machine consisting of 3 states and 198 mutated transitions; it specifies 1404928 mutants. The tool generated within 0.31s a complete test suite

	#mutants in the fault domain			
#states	$\simeq 10^4$	$\simeq 10^8$	$\simeq 10^{12}$	$\simeq 10^{18}$
4 states	(9, 0.04)	(26, 9.17)	(30, 319.19)	N/A
8 states	(9, 0.5)	(19, 0.65)	(32, 5.31)	(68, 864.06)
10 states	(9, 4.73)	(20, 21.44)	(30, 682.97)	(58, 250.72)
12 states	(8, 56.36)	(20, 1.32)	(25, 66.1)	(47, 593.07)
15 states	(5, 168.24)	(17, 227.56)	(33, 418.72)	(58, 64.55)

TABLE I: Size of the generated complete test suites and generating time ; for an entry (x, y), x is the size of the test suite and y is the generating time in seconds.

of size 23. The maximal length of the tests is 5. The size of the generated complete test suite could be reduced to 16 by removing the seven tests which are prefixes of the others. For 15 packets, the specification has 16 states and we built a mutation machine with 9438 mutated transitions specifying 1.9×10^{46} mutants; then we generated a complete test suite of size 98 within 555.14s. The test suite can be amputated from 23 tests' prefixes. The maximal length of the tests is 17.

Table I presents the evaluation results for randomly generated specification and mutation machines equipped with 2 inputs and 2 outputs. The maximal timeout in the specification machines is 3 and the one in the mutation machines is 5.

We have generated complete test suites for fault domains of important sizes and TFSM-T of reasonable size. In the automotive domain, controllers can be represented with fewer than 13 states, which let us believe that the proposed testing approach is suitable for industrial-sized TFSMs-T.

VI. Conclusion

We have proposed an approach to detecting logical and timing faults in systems represented with TFSMs-T. We have identified the types of faults to be detected and we have defined mutation machines to represent a fault domain for a specification TFSMs-T. The proposed approach includes a method of checking whether a test suite is complete for a fault domain and a method of generating a complete test suite. The methods are inspired from constraint solving-based test generation methods developed for FSM. We defined the distinguishing automaton with timeouts which is used to build SAT constraints, verify the completeness of test suites and generate complete test suites. We evaluated the scalability the methods with a prototype tool we developed.

Further work includes enhancing the developed prototype tool, reducing the size of the generated test suites, comparing the efficiency of the approach w.r.t. the approach in [4], [17] and lifting the proposed methods to TFSMs expressing time constraints beyond the timeouts.

Acknowledgment

This work was partially supported by MEI (Ministère de l'Économie et Innovation) of Gouvernement du Québec.

References

[1] A. Petrenko, O. Nguena Timo, and S. Ramesh, "Test generation by constraint solving and FSM mutant killing," in *Proceedings of the 28th IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. LNCS, vol. 9976. Springer, 2016, pp. 36–51.

[2] O. Nguena-Timo, A. Petrenko, and S. Ramesh, "Checking sequence generation for symbolic input/output fsms by constraint solving," in *Proceedings of 15th International Colloquium on Theoretical Aspects of Computing*, ser. LNCS, vol. 11187. Springer, 2018, pp. 354–375.

[3] M. G. Merayo, M. Núñez, and I. Rodríguez, "Extending efsms to specify and test timed systems with action durations and time-outs," *IEEE Transactions on Computers*, vol. 57, no. 6, pp. 835–844, June 2008.

[4] D. Bresolin, K. El-Fakih, T. Villa, and N. Yevtushenko, "Deterministic timed finite state machines: Equivalence checking and expressive power," in *Proceedings of 5th International Symposium on Games, Automata, Logics and Formal Verification*, ser. EPTCS, vol. 161, 2014, pp. 203–216.

[5] M. Zhigulin, N. Yevtushenko, S. Maag, and A. R. Cavalli, "FSM-Based Test Derivation Strategies for Systems with Time-Outs," in *Proceedings of the 11th International Conference on Quality Software*. IEEE Computer Society, 2011, pp. 141–149.

[6] M. Zhigulin, S. Prokopenko, and M. Forostyanova, "Detecting Faults in TFTP Implementations using Finite State Machines with Timeouts," in *Proceedings of the Young Researchers Colloquium on Software Engineering*, Russia, 2012.

[7] K. Sollins, "The tftp protocol (revision 2)," 1992. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1350.txt

[8] Y. M. Huang and C. V. Ravishankar, "Constructive protocol specification using Cicero," *IEEE Transactions on Software Engineering*, vol. 24, no. 4, pp. 252–267, 1998.

[9] G. V. Bochmann and A. Petrenko, "Protocol Testing: Review of Methods and Relevance for Software Testing," in *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 1994, pp. 109–124.

[10] M. Broy, B. Jonsson, J. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-Based Testing of Reactive Systems, Advanced Lectures*, ser. LNCS, vol. 3472. Springer, 2005.

[11] M. G. Merayo, M. Núñez, and I. Rodríguez, "Formal Testing from Timed Finite State Machines," *Comput. Netw.*, vol. 52, no. 2, pp. 432–460, 2008.

[12] K. El-Fakih, N. Yevtushenko, and H. Fouchal, "Testing Timed Finite State Machines with Guaranteed Fault Coverage," in *Proceedings of the 21st IFIP WG 6.1 International Conference TESTCOM and 9th International Workshop FATES*, ser. LNCS, vol. 5826. Springer, 2009, pp. 66–80.

[13] K. El-Fakih, N. Yevtushenko, and A. Simao, "A Practical Approach for Testing Timed Deterministic Finite State Machines with Single Clock," *Science of Computer Programming*, vol. 80, pp. 343 – 355, 2014.

[14] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Formal Methods in System Design*, vol. 34, pp. 238–304, 2009.

[15] R. Dssouli, A. Khoumsi, M. Elqortobi, and J. Bentahar, "Chapter Three - Testing the Control-Flow, Data-Flow, and Time Aspects of Communication Systems: A Survey," ser. Advances in Computers, A. M. Memon, Ed. Elsevier, 2017, vol. 107, pp. 95 – 155.

[16] A. David, K. G. Larsen, M. Mikucionis, O. Nguena Timo, and A. Rollet, "Remote testing of timed specifications," in *Proceedings of the 25th IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. LNCS, vol. 8254. Springer, 2013, pp. 65–81.

[17] A. Tvardovskii, K. El-Fakih, and N. Yevtushenko, "Deriving tests with guaranteed fault coverage for finite state machines with timeouts," in *Proceedings of the 30th IFIP WG 6.1 International Conference on Testing Software and Systems*, ser. LNCS, vol. 11146. Springer, 2018, pp. 149–154.

[18] A. Simão and A. Petrenko, "Fault Coverage-Driven Incremental Test Generation," *Comput. J.*, vol. 53, no. 9, pp. 1508–1522, Nov. 2010.

[19] M. P. Vasilevskii, "Failure diagnosis of automata," *Cybernetics*, vol. 9, no. 4, pp. 653–665, 1973.

[20] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation Using Constraint Solving Techniques," *SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 53–62, 1998.

[21] A. Petrenko and N. Yevtushenko, "Test Suite Generation from a FSM with a Given Type of Implementation Errors," in *Proceedings of the 12th IFIP TC6/WG6.1 International Symposium on Protocol Specification, Testing and Verification*, ser. IFIP Transactions, vol. C-8. North-Holland, 1992, pp. 229–243.

[22] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, ser. LNCS, vol. 5584. Springer, 2009, pp. 244–257.