A Model Based Approach to Combine Load and Functional Tests for Service Oriented Architectures

Afef Jmal Maâlej and Moez Krichen

Research Laboratory of Development and Control of Distributed Applications
National School of Engineers of Sfax, University of Sfax
BP 1173, 3038, Sfax-Tunisia
{afef.jmal,moez.krichen}@redcad.org

Abstract. We propose a new model-based framework to combine functional and load tests for service oriented architectures. The new framework is based on the model of extended timed automata with inputs/ouputs and shared integer variables. A test generation algorithm to produce analog-clock tests is proposed. To illustrate our approach, we report on a case study from the field of load balancing based architectures for BPEL compositions.

Keywords: Functional/Load Tests, Extended Timed Automata, Test Generation, Service Oriented Architecture.

1 Introduction

Nowadays testing has become an important phase in the development of any software system. During the last few decades very critical programming errors have been reported in different fields. Some of these errors were very dangerous and caused huge human/financial damages. For instance in 1996, the European rocket Ariane 5 exploded 37 seconds after launch[1]. The error was the result of a wrong reuse of code from Ariane 4. The financial loss caused by this accident was estimated to be about $400 million. A second example of critical software errors was encountered in the medical field. From 1985 to 1987, at least four patients died as a direct result of a radiation overdose received from the medical radiation therapy device Therac-25[2]. The four victims received up to 100 times the intended dose. The accident was the result of a bug in the software powering the Therac-25 device.

Clearly, many other critical errors happened in many other fields. However, we restrict ourselves to the two previous presented examples due to space limitation. The important point to emphasize here is that a good percentage of these

[1] http://www.ima.umn.edu/arnold/disasters/ariane.html

[2] http://therac25.net/about.php

errors could have been avoided by considering some more sophisticated testing efforts. However unfortunately, in practice such efforts are still minimal and the need for advanced testing solutions is still deep. Indeed software companies are still not making enough efforts at this level. A UK survey has noticed that 92% of test professionals perceive both the cost and complexity of software testing is increasing, but just 22% of test teams are willing to adopt this process[3].

The remainder of this paper is organized as follows. In Section 2, we give a brief recall about some types of testing. In Section 3, we define the extended timed input output conformance relation etioco, as an extension of our previous timed conformance relation tioco. Section 4 is dedicated to describe formally our testing approach. Then, we propose in Section 5 a test generation algorithm to produce analog-clock tests. In Section 6, we report on an example to illustrate our approach. Finally, Section 7 provides a conclusion that summarizes the paper and discusses items for future work.

2 A brief recall about some types of testing

In this paper we mainly concentrate on two particular types of software testing namely functional and load testings. First, we consider functional testing which allows to check whether the specified functionality in the system requirements works correctly or not. This is done by sending some particular sequences of inputs to the SUT and then checking whether the correct outputs are generated or not. This type of testing falls under the class of black box testing. Second, we consider load testing which is achieved to assess the SUT's behaviour under both normal and peak load conditions. More precisely, load testing aims at estimating the maximum amount of work the SUT can manage with no significant performance degradation. Obviously, load testing is mainly dedicated to mutliuser systems. In general, this kind of testing is accomplished by simulating a significant number of users accessing the SUT concurrently.

In this context and in addition to conventional functional testing procedures, load testing is a required procedure that reveals programming errors which would not appear if the SUT is executed with a small (limited) workload or for a short time. Such errors emerge when the system is executed under a heavy load or over a long period of time. On the other hand, a given process under test may be correctly implemented but fails under some particular load conditions because of external causes (e.g. misconfiguration, hardware failures, buggy load generator, etc.) [1]. Hence, it is important to identify and remedy these different problems. For that, we investigated in [2] the opportunities as well as challenges of load testing in general. A classification and evaluation of existing related works were also reported. In brief, recognizing problems under load is a challenging and time-consuming task due to the large amount of generated data and the long running time of load tests.

In this paper, we focus on conformance testing of a given SUT under various load conditions, which constitutes an important testing area that is often misun-

[3] http://www.uk.sogeti.com/News–Events/Press-Releases/

derstood or overlooked. Indeed, in some cases, a system may perform correctly and conformly to its specification under a certain load, but it disrespects this specification when the load increases and goes above the expected values. This may be due to some implementation errors which are discovered when stressing the SUT. For early bug detection, we make use of model-based testing [3], where the specification is described by a formal model from which a test suite is automatically generated. The obtained test cases are then applied to the SUT in order to check the conformance of both functional and non-functional constraints with respect to the specification in hand. Conceptually, testing consists of three phases: test case generation, test case execution and verdict assignment.

3 Extended Timed Automata

We extend the framework presented in a previous work [4], which treats only conformance testing without considering load conditions.

3.1 Timed Labelled Transition Systems

Let R be the set of non-negative reals, Q the set of non-negative rationals and N the set of non-negative integers. Given a finite set of *actions* Ac, the set $(\mathsf{Ac} \cup \mathsf{R})^*$ of all finite-length *real-time sequences* over Ac will be denoted $\mathsf{RT}(\mathsf{Ac})$. $\epsilon \in \mathsf{RT}(\mathsf{Ac})$ is the empty sequence. Given $\mathsf{Ac}' \subseteq \mathsf{Ac}$ and $\rho \in \mathsf{RT}(\mathsf{Ac})$, $\mathsf{P}_{\mathsf{Ac}'}(\rho)$ denotes the *projection* of ρ to $\mathsf{Ac}' \cup \mathsf{R}$, obtained by "erasing" from ρ all actions not in $\mathsf{Ac}' \cup \mathsf{R}$. Similarly, $\mathsf{DP}_{\mathsf{Ac}'}(\rho)$ denotes the (discrete) projection of ρ to Ac'. For example, if $\mathsf{Ac} = \{a, b\}$, $\mathsf{Ac}' = \{a\}$ and $\rho = a\,1\,b\,2\,a\,3$, then $\mathsf{P}_{\mathsf{Ac}'}(\rho) = a\,3\,a\,3$ and $\mathsf{DP}_{\mathsf{Ac}'}(\rho) = a\,a$. The time spent in a sequence ρ, denoted $\mathsf{duration}(\rho)$ is the sum of all delays in ρ, for example, $\mathsf{duration}(\epsilon) = 0$ and $\mathsf{duration}(a\,1\,b\,0.5) = 1.5$.

In the rest of the document, we assume given a set of actions Ac, partitioned in two disjoint sets: a set of *input actions* $\mathsf{Ac}_{\mathsf{in}}$ and a set of *output actions* $\mathsf{Ac}_{\mathsf{out}}$. Actions in $\mathsf{Ac}_{\mathsf{in}} \cup \mathsf{Ac}_{\mathsf{out}}$ are called *observable* actions. We also assume there is an *unobservable* action $\tau \notin \mathsf{Ac}$. Let $\mathsf{Ac}_\tau = \mathsf{Ac} \cup \{\tau\}$.

A *Timed Labelled Transition System* (TLTS) over Ac is a tuple $(S, s_0, \mathsf{Ac}, T_d, T_t)$, where:

- S is a set of *states*;
- s_0 is the initial state;
- T_d is a set of *discrete transitions* of the form (s, a, s') where $s, s' \in S$ and $a \in \mathsf{Ac}$;
- T_t is a set of *timed transitions* of the form (s, t, s') where $s, s' \in S$ and $t \in \mathsf{R}$.

Timed transitions must be deterministic, that is, $(s, t, s') \in T_t$ and $(s, t, s'') \in T_t$ implies $s' = s''$. T_t must also satisfy the following conditions: $(s, t, s') \in T_t$ and $(s', t', s'') \in T_t$ implies $(s, t + t', s'') \in T_t$; $(s, t, s') \in T_t$ implies that for all $t' < t$, there is some $(s, t', s'') \in T_t$.

We use standard notation concerning TLTS. For $s, s', s_i \in S$, $\mu, \mu_i \in \mathsf{Ac}_\tau \cup \mathsf{R}$, $a, a_i \in \mathsf{Ac} \cup \mathsf{R}$, $\rho \in \mathsf{RT}(\mathsf{Ac}_\tau)$ and $\sigma \in \mathsf{RT}(\mathsf{Ac})$, we have:

- General transitions:
  - $s \xrightarrow{\mu} s' \overset{Def}{=} (s, \mu, s') \in T_d \cup T_t$;
  - $s \xrightarrow{\mu} \overset{Def}{=} \exists s' : s \xrightarrow{\mu} s'$;
  - $s \not\xrightarrow{\mu} \overset{Def}{=} \nexists s' : s \xrightarrow{\mu} s'$;
  - $s \xrightarrow{\mu_1 \cdots \mu_n} s' \overset{Def}{=} \exists s_1, \cdots, s_{n+1} : s = s_1 \xrightarrow{\mu_1} s_2 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_{n+1} = s'$;
  - $s \xrightarrow{\rho} \overset{Def}{=} \exists s' : s \xrightarrow{\rho} s'$;
  - $s \not\xrightarrow{\rho} \overset{Def}{=} \nexists s' : s \xrightarrow{\rho} s'$.
- Observable transitions:
  - $s \xRightarrow{\epsilon} s' \overset{Def}{=} s = s'$ or $s \xrightarrow{\tau \cdots \tau} s'$;
  - $s \xRightarrow{a} s' \overset{Def}{=} \exists s_1, s_2 : s \xRightarrow{\epsilon} s_1 \xrightarrow{a} s_2 \xRightarrow{\epsilon} s'$;
  - $s \not\xRightarrow{a} \overset{Def}{=} \nexists s' : s \xRightarrow{a} s'$;
  - $s \xRightarrow{a_1 \cdots a_n} s' \overset{Def}{=} \exists s_1, \cdots, s_{n+1} : s = s_1 \xRightarrow{a_1} s_2 \xRightarrow{a_2} \cdots \xRightarrow{a_n} s_{n+1} = s'$;
  - $s \xRightarrow{\sigma} \overset{Def}{=} \exists s' : s \xRightarrow{\sigma} s'$;
  - $s \not\xRightarrow{\sigma} \overset{Def}{=} \nexists s' : s \xRightarrow{\sigma} s'$.

A sequence of the form $s_0 \xrightarrow{\mu_1} s \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s'$ is called a *run* and a sequence of the form $s_0 \xRightarrow{a_1} s \xRightarrow{a_2} \cdots \xRightarrow{a_n} s'$ an *observable run*.

3.2 Extended Timed Automata

We use timed automata [5] with deadlines to model urgency [4]. An *extended timed automaton over* Ac is a tuple $A = (Q, q_0, X, \mathsf{I}, \mathsf{Ac}, \mathsf{E})$, where:

- Q is a finite set of *locations*;
- $q_0 \in Q$ is the initial location;
- X is a finite set of *clocks*;
- I is a finite set of integer variables;
- E is a finite set of *edges*.

Each edge is a tuple $(q, q', \psi, r, inc, dec, d, a)$, where:

- $q, q' \in Q$ are the source and destination locations;
- ψ is the *guard*, a conjunction of constraints of the form $x \# c$, where $x \in X \cup \mathsf{I}$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$;
- $r \subseteq X \cup \mathsf{I}$ is the set of clocks and integer variables that are *reset* to zero;
- $inc \subseteq \mathsf{I}$ is the set of integer variables (disjoint from r) that are *incremented* by one;
- $dec \subseteq \mathsf{I}$ is the set of integer variables (disjoint from r and inc) that are *decremented* by one;
- $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*;
- $a \in \mathsf{Ac}$ is the action.

An example of an extended timed automaton $A = (Q, q_0, X, \mathsf{I}, \mathsf{Ac}, \mathsf{E})$ over the set of actions $\mathsf{Ac} = \{a, b, c, d\}$ is given in Figure 1 where :

- $Q = \{q_0, q_1, q_2, q_3\}$ is the set of locations;
- q_0 is the initial location;
- $X = \{x\}$ is the finite set of clocks;
- $\mathsf{I} = \{i\}$ is the finite set of integer variables;
- E is the set of edges drawn in the Figure.

The figure uses the following notation:

- "$x := 0$" means resetting the clock x to 0;
- "$i := 0$" means resetting the integer variable i to 0;
- "$i + +$" means incrementing i by 1; [4]
- "$i - -$" means decrementing i by 1. [5]

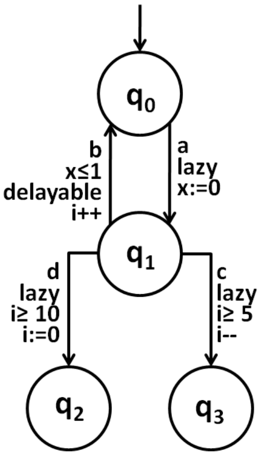

Fig. 1. An example of an extended timed automaton.

An extended timed automaton A defines an infinite TLTS which is denoted L_A. Its states are pairs $s = (q, v_X, v_\mathsf{I})$, where $q \in Q$, $v_X : X \to \mathsf{R}$ is a *clock valuation* and $v_\mathsf{I} : \mathsf{I} \to \mathsf{N}$ is an *integer variable valuation.* $\mathbf{0}_X$ is the valuation assigning 0 to every clock of A. $\mathbf{0}_\mathsf{I}$ is the valuation assigning 0 to every integer variable of A. S_A is the set of all states and $s_0^A = (q_0, \mathbf{0}_X, \mathbf{0}_\mathsf{I})$ is the initial state.

- *Discrete transitions* are of the form

$$(q, v_X, v_\mathsf{I}) \xrightarrow{a} (q', v'_X, v'_\mathsf{I})$$

where $a \in \mathsf{Ac}$ and there is an edge

$$(q, q', \psi, r, inc, dec, d, a)$$

[4] We can also use the usual notation "$i := i + 1$" instead.

[5] We can also use the usual notation "$i := i - 1$" instead.

such that (v_X, v_I) satisfies ψ and (v'_X, v'_I) is obtained by:

- resetting to zero all clocks and integer variables in r;
- incrementing integer variables in inc by one;
- decrementing variables in dec by one;
- leaving all other variables unchanged.

- *Timed transitions* are of the form

$$(q, v_X, v_\mathsf{I}) \xrightarrow{t} (q, v_X + t, v_\mathsf{I})$$

where $t \in \mathsf{R}, t > 0$ and there is no edge

$$(q, q'', \psi, r, inc, dec, d, a)$$

such that:

- either $d = \mathsf{delayable}$ and there exist $0 \leq t_1 < t_2 \leq t$ such that $(v_X + t_1, v_\mathsf{I}) \models \psi$ and $(v_X + t_2, v_\mathsf{I}) \not\models \psi$;
- or $d = \mathsf{eager}$ and $(v_X, v_\mathsf{I}) \models \psi$.

Lazy edges do not impact the semantics. They denote that an edge is neither delayable, nor eager. More precisely, lazy edges cannot block time progress, whereas delayable and eager edges can. We do not allow delayable edges with guards of the form $x < c$ since there is no *latest* time when the guard is still true. Similarly, we do not allow eager edges with guards of the form $x > c$ since there is no *earliest* time when the guard becomes true. A state $s \in S_A$ is *reachable* if there exists $\rho \in \mathsf{RT}(\mathsf{Ac})$ such that $s_0^A \xrightarrow{\rho} s$. The set of reachable states of A is denoted $\mathsf{Reach}(A)$.

3.3 Extended Timed Automata with Inputs and Outputs

An *extended timed automaton with inputs and outputs* (ETAIO) is an extended timed automaton over the partitioned set of actions

$$\mathsf{Ac}_\tau = \mathsf{Ac}_\mathsf{in} \cup \mathsf{Ac}_\mathsf{out} \cup \{\tau\}.$$

For clarity, we will explicitly include inputs and outputs in the definition of an ETAIO A and write

$$(Q, q_0, X, \mathsf{I}, \mathsf{Ac}_\mathsf{in}, \mathsf{Ac}_\mathsf{out}, \mathsf{E})$$

instead of

$$(Q, q_0, X, \mathsf{I}, \mathsf{Ac}_\tau, \mathsf{E}).$$

An ETAIO is called *observable* if none of its edges is labelled by τ.

Given a set of inputs $\mathsf{Ac}' \subseteq \mathsf{Ac}_\mathsf{in}$, an ETAIO A is called *input-enabled* with respect to Ac' if it can accept any input in Ac' at any state:

$$\forall s \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Ac}' \,:\, s \xrightarrow{a} .$$

It is simply said to be input-enabled when $\mathsf{Ac}' = \mathsf{Ac}_{\mathsf{in}}$. A is called lazy-*input* with respect to Ac' if the deadlines on all the transitions labelled with input actions in Ac' are lazy. It is called lazy-input if it is lazy-input with respect to $\mathsf{Ac}_{\mathsf{in}}$. Note that input-enabled does not imply lazy-input in general.

A is called *deterministic* if

$$\forall s, s', s'' \in \mathsf{Reach}(A)\,.\,\forall a \in \mathsf{Ac}_\tau \ :$$

$$s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Rightarrow s' = s''.$$

A is called *non-blocking* if

$$\forall s \in \mathsf{Reach}(A)\,.\,\forall t \in \mathsf{R}\,.\,\exists \rho \in \mathsf{RT}(\mathsf{Ac}_{\mathsf{out}} \cup \{\tau\}) \ :$$

$$\mathsf{duration}(\rho) = t \wedge s \xrightarrow{\rho} .$$

This condition guarantees that A will not block time in any environment.

The set of *timed traces* of an ETAIO A is defined to be

$$\mathsf{TTr}(A) = \{\rho \mid \rho \in \mathsf{RT}(\mathsf{Ac}_\tau) \wedge s_0^A \xrightarrow{\rho}\}.$$

The set of *observable timed traces* of A is defined to be

$$\mathsf{OTTr}(A) = \{\mathsf{P}_{\mathsf{Ac}}(\rho) \mid \rho \in \mathsf{RT}(\mathsf{Ac}_\tau) \wedge s_0^A \xrightarrow{\rho}\}.$$

The TLTS defined by an ETAIO is called a *timed input-output LTS* (TIOLTS). From now on, unless otherwise stated, all the considered ETAIO are defined with respect to the same sets $\mathsf{Ac}_{\mathsf{in}}$ and $\mathsf{Ac}_{\mathsf{out}}$ and unobservable action τ. As for ETAIO, a given TIOLTS L is denoted

$$(S, s_0, \mathsf{Ac}_{\mathsf{in}}, \mathsf{Ac}_{\mathsf{out}}, T_d, T_t)$$

instead of

$$(S, s_0, \mathsf{Ac}_\tau, T_d, T_t).$$

The two operators $\mathsf{TTr}(\cdot)$ and $\mathsf{OTTr}(\cdot)$ are extended in a natural way to the case of TIOLTS.

3.4 Parallel Composition of ETAIO with Shared Integer Variables

Let n be a non-negative integer such that $n \geq 2$. We consider n ETAIO $(A_i)_{1\leq i\leq n}$ where

$$A_i = (Q^i, q_0^i, X^i, \mathsf{I}, \mathsf{Ac}_{\mathsf{in}}^i, \mathsf{Ac}_{\mathsf{out}}^i, \mathsf{E}^i).$$

That is the set of integer variables I is shared between all the considered ETAIO $(A_i)_{1\leq i\leq n}$ while no other element from Q^i, X^i, $\mathsf{Ac}_{\mathsf{in}}^i$ and $\mathsf{Ac}_{\mathsf{out}}^i$ is shared whith the other ETAIO $(A_j)_{j\neq i}$. The TIOLTS

$$L^P = (S^P, s_0^P, \mathsf{Ac}_{\mathsf{in}}^P, \mathsf{Ac}_{\mathsf{out}}^P, T_d^P, T_t^P)$$

generated by the parallel product of the ETAIO

$$(A_i)_{1 \leq i \leq n}$$

is defined as follows

$$s_0^P = ((q_0^1, \cdots, q_0^n), (\mathbf{0}_{X^0}, \cdots, \mathbf{0}_{X^n}), \mathbf{0}_{\mathsf{I}})$$

$$\mathsf{Ac}_{\mathsf{in}}^P = \bigcup_{1 \leq i \leq n} \mathsf{Ac}_{\mathsf{in}}^i, \ \mathsf{Ac}_{\mathsf{out}}^P = \bigcup_{1 \leq i \leq n} \mathsf{Ac}_{\mathsf{out}}^i$$

and S^P, T_d^P and T_t^P are the smallest sets such that

- $s_0^P \in S^P$;
- For $s^P = ((q^1, \cdots, q^n), (v_{X^0}, \cdots, v_{X^n}), v_{\mathsf{I}}) \in S^P$ and $\delta \in \mathsf{R}$:

$$\forall 1 \leq i \leq n : (q_i, v_{X^i}, v_I) \xrightarrow{\delta} (q_i, v_{X^i} + \delta, v_I) \in T_t^i$$

$$\Rightarrow s'^P = ((q^1, \cdots, q^n), (v_{X^0} + \delta, \cdots, v_{X^n} + \delta), v_{\mathsf{I}}) \in S^P$$

and

$$s^P \xrightarrow{\delta} s'^P \in T_t \qquad (1)$$

- For $s^P = ((q^1, \cdots, q^n), (v_{X^0}, \cdots, v_{X^n}), v_{\mathsf{I}}) \in S^P$, $1 \leq i \leq n$ and $a_i \in \mathsf{Ac}_\tau{}^i$: [6]

$$(q_i, v_{X^i}, v_I) \xrightarrow{a_i} (q_i', v_{X^i}', v_I') \in T_d^i$$

$$\Rightarrow s'^P = (q'^p, v_X'^p, v_{\mathsf{I}}') \in S^P \quad \wedge \quad s^P \xrightarrow{a_i} s'^P \in T_d$$

where

$$q'^p = (q^1, \cdots, q^{i-1}, q'^i, q^{i+1}, \cdots, q^n)$$

and

$$v_X'^p = (v_{X^0}, \cdots, v_{X^{i-1}}, v_{X^i}', v_{X^{i+1}} \cdots, v_{X^n}) \qquad (2)$$

It is worth noticing here that it is possible to define the parallel composition of n copies $(A_i)_{1 \leq i \leq n}$ of the same ETAIO A. In this case we assume it is possible to distinguish the sets of inputs and outputs of the different instances, such as particular identifier corresponds to each instance. Obviously, the n instances share the set of integer variables of the ETAIO A. The obtained TIOLTS is denoted L_n^P.

[6] $\mathsf{Ac}_\tau{}^i = \mathsf{Ac}_{\mathsf{in}}^i \cup \mathsf{Ac}_{\mathsf{out}}^i \cup \{\tau\}$

3.5 Modelling Issues

In this section we illustrate some methodological aspects of our framework. First we explain how it is possible to combine both functional and load aspects within the same model. For instance in Figure 2 the response time to produce the output action b with respect to the input action a depends on the number of concurrent instances of the considered system under test as follows:

- output b is generated within at most 1 time unit if the number of concurrent instances is smaller or equal to 100;
- output b is generated within at most 2 time units if the number of concurrent instances is between 101 and 1000;
- output b is generated within at most 3 time units if the number of concurrent instances is greater or equal to 1001;

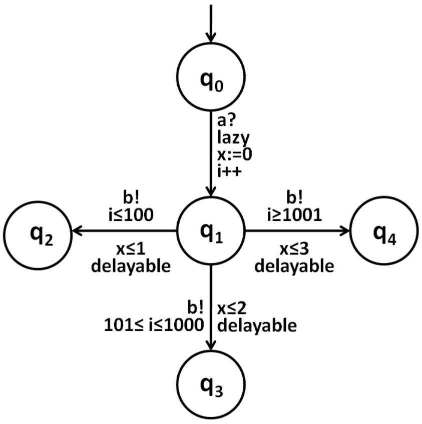

Fig. 2. An example showing how the time response of the SUT may depend on the number of concurrent instances.

In Figure 3 we show how to model the fact that the SUT may even produce different output actions with respect to the same input action depending on the current number of concurrent instances of the considered system. The SUT may produce either b, c or d. On the first hand, output a may be seen as the normal output generated by the SUT when the load is smaller or equal to 100. On the other hand, output b may correspond to the situation where the SUT still produces the same desired output action. However this time the output action is mixed with a warning message to inform the user that the system is starting entering a critical area (load between 101 and 1000). Finally output c may correspond to the production of an error message meaning that the SUT is no longer able to produce the desired output action since the load is too high (greater or equal to 1001).

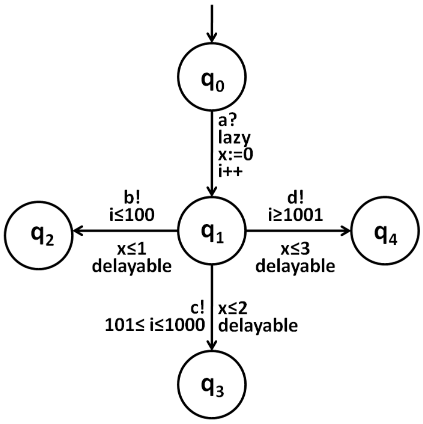

Fig. 3. An example where the SUT produces different output actions depending on the current load.

In Figure 4 we consider a more sophisticated situation where the SUT can produce complete different behaviours depending on the current load. Three distinct behaviours are possible according to the figure. Behaviour 1 can be considered as the nominal behaviour as in the previous example. The two other behaviours may correspond to the situation where the SUT is trying to find a suitable way to deal with the increase of the current number of concurrent instances and to improve the quality of the service. For instance a possible solution may consist in allocating additional resources to overcome the current critical situation.

4 Testing Framework

In this section, we are going to define a new extended timed input output conformance relation etioco. Then, we propose a new approach for deriving analog-clock tests from the SUT specification. Finally, we discuss both test execution and correctness requirements.

4.1 Conformance Relation

In order to formally define the conformance relation, we define a number of operators. Given a TIOLTS

$$L = (S^L, s_0^L, \mathsf{Ac}_{\mathsf{in}}^L, \mathsf{Ac}_{\mathsf{out}}^L, T_d^L, T_t^L)$$

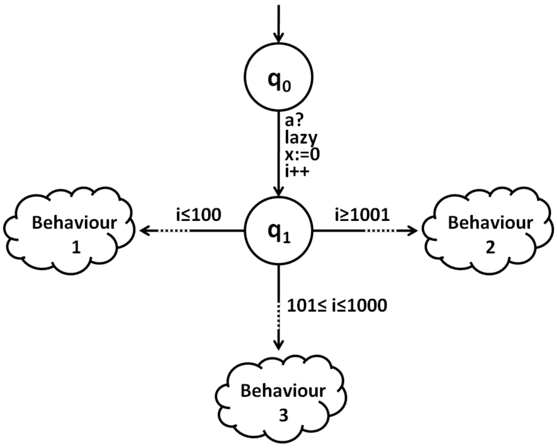

Fig. 4. An example where the SUT adopts different sophisticated behaviours depending on the current load.

and a timed trace

$$\sigma \in \mathsf{RT}(\mathsf{Ac}^L)$$

L after σ is the set of all states of L that can be reached by some timed sequence ρ whose projection to observable actions is σ. Formally:

$$L \ \mathsf{after}\ \sigma =$$

$$\{s \in S^L \mid \exists \rho \in \mathsf{RT}({\mathsf{Ac}_\tau}^L)\ :\ s_0^L \xrightarrow{\rho} s \wedge \mathsf{P}_{\mathsf{Ac}}(\rho) = \sigma\}.$$

Given state $s \in S^L$, $\mathsf{elapse}(s)$ is the set of all delays which can elapse from s without L making any observable action. Formally:

$$\mathsf{elapse}(s) =$$

$$\{t > 0 \mid \exists \rho \in \mathsf{RT}(\{\tau\})\ :\ \mathsf{duration}(\rho) = t \wedge s \xrightarrow{\rho}\}.$$

Given state $s \in S^L$, $\mathsf{out}(s)$ is the set of all observable "events" (outputs or the passage of time) that can occur when the system is at state s. The definition naturally extends to a set of states S. Formally:

$$\mathsf{out}(s) = \{a \in \mathsf{Ac}_{\mathsf{out}}^L \mid s \xrightarrow{a}\} \cup \mathsf{elapse}(s)$$

and

$$\mathsf{out}(S) = \bigcup_{s \in S} \mathsf{out}(s).$$

The specification of the system to be tested is given as a non-blocking ETAIO A_S while the implementation can be modelled as a non-blocking, input-enabled

ETAIO A_I. [7] For $n \geq 1$, let $L^P_{S,n}$ (resp., $L^P_{I,n}$) be the parallel composition of n copies of A_S (resp., A_I).

Input-enabledness is required so that the implementation can accept inputs from the tester at any state.

The *extended timed input-output conformance relation*, denoted etioco, is an extension of our previous conformance relation tioco [4, 6]. The new relation etioco is defined as

$$A_I \ \mathsf{etioco}\ A_S$$

$$\textit{iff} \ \ \forall n \geq 1 \wedge \sigma \in \mathsf{OTTr}(L^P_{S,n}) \ :$$

$$\mathsf{out}(L^P_{I,n} \ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(L^P_{S,n} \ \mathsf{after}\ \sigma).$$

The relation states that an implementation A_I conforms to a specification A_S iff for any number of copies n of A_S and any observable behaviour σ of $L^P_{S,n}$, the set of observable outputs of $L^P_{I,n}$ after any behaviour "matching" σ must be a subset of the set of possible observable outputs of $L^P_{S,n}$.

Notice that observable outputs are not only observable output actions but also time delays. Also notice that in case we consider only $n = 1$, the definitions of etioco and tioco become the same.

4.2 Analog-Clock Tests

A test (or *test case*) is an experiment performed on the implementation by an agent (the *tester*). There are different types of tests, depending on the capabilities of the tester to observe and react to events. In general, one may consider either *Analog-clock* or *Digital-clock* tests [7]. In this work, we consider only analog-clock tests. The latters can measure precisely the delay between two observed actions and can emit an input at any point in time.

It should be noted that we consider *adaptive* tests (following the terminology of [8]), where the action the tester takes depends on the observation history.

For $n \geq 1$, let Ac^n (resp., $\mathsf{Ac}^n_{\mathsf{in}}$) denote the union of all observable actions (resp., all input actions) of n copies of the specification A_S. An analog-clock test for n parallel executions of A_S is a total function

$$T_n : \mathsf{RT}(\mathsf{Ac}^n) \rightarrow \mathsf{Ac}^n_{\mathsf{in}} \cup \{\mathsf{Wait}, \mathsf{Pass}, \mathsf{Fail}\}.$$

$T_n(\rho)$ specifies the action the tester must take once it observes ρ:

- If $T_n(\rho) = a \in \mathsf{Ac}^n_{\mathsf{in}}$ then the tester emits input a.
- If $T_n(\rho) = \mathsf{Wait}$ then the tester waits (lets time elapse).
- If $T_n(\rho) \in \{\mathsf{Pass}, \mathsf{Fail}\}$ then the tester produces a verdict (and stops).

[7] A_I may be unknown. We assume it simply exists.

4.3 Test Execution and Correctness Requirements

The execution of the test T_n on the implementation A_I can be defined as the *parallel composition* of the TIOLTS defined by T_n and $L^P_{I,n}$ the TIOLTS corresponding to n copies of A_I, with the usual *synchronization* rules for transitions carrying the same label. We will denote the product TIOLTS by $L^P_{I,n} \| T_n$. The execution of the test reaches a pass/fail verdict after bounded time.

Formally, we say that A_I *passes* the test, denoted A_I passes T_n, if state Fail is not reachable in the product $L^P_{I,n} \| T_n$. We say that an implementation passes (resp. fails) a set of tests (or *test suite*) $\mathcal{T}$ if it passes all tests (resp. fails at least one test) in $\mathcal{T}$.

We say that an analog-clock test suite $\mathcal{T}$ is *sound with respect to* A_S if

$$\forall A_I \ : \ A_I \ \mathsf{etioco} \ A_S \Rightarrow A_I \ \mathsf{passes} \ \mathcal{T}.$$

We say that $\mathcal{T}$ is *complete with respect to* A_S if

$$\forall A_I \ : \ A_I \ \mathsf{passes} \ \mathcal{T} \Rightarrow A_I \ \mathsf{etioco} \ A_S.$$

5 Test Generation

We adapt the untimed test generation algorithm of [3]. Roughly speaking, the algorithm builds a test in the form of a tree. A node in the tree is a set of states S of the specification and represents the "knowledge" of the tester at the current test state. The algorithm extends the test by adding successors to a leaf node, as illustrated in Figure 5.

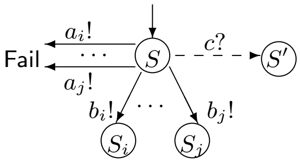

Fig. 5. Generic test-generation scheme.

For all *illegal* outputs a_i (outputs which cannot occur from any state in S) the test leads to Fail. For each legal output b_i, the test proceeds to node S_i, which is the set of states the specification can be in after emitting b_i (and possibly performing unobservable actions). If there exists an input c which can be accepted by the specification at some state in S, then the test may decide to

```
$S \leftarrow \mathsf{tsucc}(\{s^P_{n,0}\}, 0)$;
while(true)
  $x \leftarrow 0$; /* $x$ is a clock measuring elapsing time */
  await(output $b$ is received at $x < T$ or $x = T$)
  if($b$ received at $x$)
    $S \leftarrow \mathsf{dsucc}(\mathsf{tsucc}(S, x), b)$;
  else
    $S \leftarrow \mathsf{tsucc}(S, T)$;
  endif;
  if($S = \emptyset$)
    announce Fail;
    exit;
  endif;
  if($\mathsf{valid\_inputs}(S) \neq \emptyset$)
    $i \leftarrow \mathsf{pick}(\{0, 1\})$; /* 0 to send an input */
                    /* and 1 to continue observation */
  endif;
  if($i = 0$)
    $a \leftarrow \mathsf{pick}(\mathsf{valid\_inputs}(S))$;
    $S \leftarrow \mathsf{dsucc}(S, a)$;
  endif;
endwhile;
```

Algorithm 1: On-the-fly analog-clock test generation.

emit this input (dashed arrow from S to S'). At any node, the algorithm may decide to stop the test and label this node as Pass.

Analog-clock tests cannot be directly represented as a finite tree, because there is an a-priori infinite set of possible observable delays at a given node. To remedy this, we use the idea of [9]. We represent an analog-clock test as an *algorithm*. The latter essentially performs subset construction on the specification automaton, during the execution of the test. Thus, our analog-clock testing method can be classified as on-the-fly or *on-line*, meaning that the test is generated at the same time it is executed. More precisely, the tester will maintain a set of states S of the TIOLTS $L^P_{S,n}$. S will be updated every time an action is observed or some time delay elapses. Since the time delay is not known a-priori, it must be an input to the update function. We define the following operators:

$$\mathsf{dsucc}(S, a) = \{s' \mid \exists s \in S \;:\; s \xrightarrow{a} s'\}$$

$$\mathsf{tsucc}(S, t) =$$
$$\{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \;:\; \mathsf{duration}(\rho) = t \wedge s \xrightarrow{\rho} s'\}$$

where $a \in \mathsf{Ac}^n$ and $t \in \mathsf{R}$. $\mathsf{dsucc}(S, a)$ contains all states which can be reached by some state in S performing action a. $\mathsf{tsucc}(S, t)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions and takes exactly t time units.

The test operates as follows. It starts at state

$$S_0 = \mathsf{tsucc}(\{s^P_{n,0}\}, 0)$$

where $s^P_{n,0}$ is the initial state of $L^P_{S,n}$. Given current state S:

- if output a is received t time units after entering S, then S is updated to $\mathsf{dsucc}(\mathsf{tsucc}(S,t),a)$.
- If ever the set S becomes empty, the test announces Fail.
- At any point, for an input b, if $\mathsf{dsucc}(S,b) \neq \emptyset$, the test may decide to emit b and update its state accordingly.

On-line analog-clock test generation is performed by Algorithm 1. The algorithm keeps running as long as no non-conformance is detected. At any time the tester can stop testing and declare Pass. The algorithm uses the following notation. Given a nonempty set X, $\mathsf{pick}(X)$ chooses randomly an element in X. Given a set of states S, $\mathsf{valid\_inputs}(S)$ is defined as the set of valid inputs at S, that is:

$$\mathsf{valid\_inputs}(S) =$$

$$\{a \in \mathsf{Ac}^n_{\mathsf{in}} | \mathsf{dsucc}(\mathsf{tsucc}(S,0),a) \neq \emptyset\}.$$

Following the same methodology as in our previous work [4] we can prove that the proposed test generation algorithm is both sound and complete.

6 Illustrative Example: the Round-Robin Algorithm

Considering the scalability of load balancing based architectures, it is increasingly necessary to develop appropriate quality assurance methodologies and techniques, of which Testing is widely adopted and used one. In this context, we proposed in [10] a distributed platform for on-line checking of the conformance between the real functioning of a given load balancer and its specified requirements. Our solution is based on Timed Automata as model for testing supported load balancing algorithms. We also developed a prototype tool support, LBACT, which is implemented for quality assurance of load balancing based architectures.

For simplicity and due to restrictions on page number, we present in this section the modelling of the round-robin load balancing algorithm, where properties are supposed to be available only for the case in which two servers are clustered. Adding more servers would require a different model for the corresponding algorithm. However, we highlight that the basic ideas are the same as for considering two clustered servers.

In fact, when a load balancer is configured to use the round-robin method, it rotates incoming requests around the servers that it manages. Figure 6 describes this principle. Actually, a node in the cluster is modelled by a state. LB, host2 and host3 represent the load balancer and two servers. To simplify, we designed host2 and host3 by the following respective values 1 and 2. Indeed, the load balancer should assign a request to a server conforming to the following conditions:

- The identifier of the current instance (i) is less than the number of total requests (nb_req), which corresponds to test instances number.
- The execution of the instance of number (i) is completed by a server which is different from the server that treated the previous instance (i-1). This condition is checked by the function verif(currentserver) which return type is Boolean.

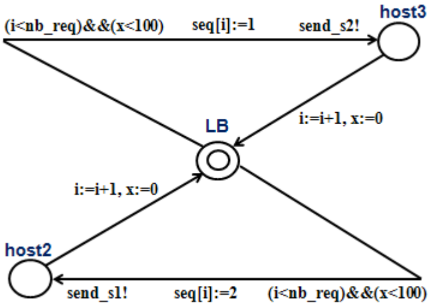

Fig. 6. Specification of the round-robin algorithm.

We underline that, according to Figure 6, the top and bottom transitions are enabled when a new request has arrived (by virtue of (nb_req) being externally incremented), so the internal counter (i) is less than (nb_req), enabling the transition firing. Furthermore, in order to ensure that the tester does not remain in an infinite wait of a sent request from the load balancer to a server, we considered a temporal constraint on network timeout. To model this constraint, we define a clock x which value does not exceed, for example, 100 milliseconds (tmax) as shown in Figure 6. Thus, the proposed model is characterized by:

- send_si!, where i belongs to {1, 2}, represents a synchronization message.
- seq[] is an array which size is equal to the total number of handled requests by the load balancer. The value of an element seq[i] determines the server that treats the instance of number (i).
- A transition between a server and the load balancer is of type *update*. In fact, it allows incrementing the value of the current instance and the initialization of the clock value to 0.

7 Conclusion and Perspectives

In this work, we proposed a new formal model-based framework to combine functional and load tests. Our solution is based on the model of extended timed automata with inputs/ouputs and shared integer variables. The latter allows high expressiveness for concurrent systems since it guarantees partial-observability and parallel composition. In addition, we defined the new extended timed input output conformance relation etioco which allows to compare a given implementation with respect to its specification in our new framework. We also provided a new technique for deriving analog-clock tests from the specification of the SUT. An important contribution in this work is to use a rich formalism to model mutli-user systems and to combine functional and load tests, which constitutes an important testing area that is usually misunderstood or omitted.

Many extensions are possible for this work. First, we need selection techniques based on coverage criteria in order to guide test generation and to reduce the number of generated tests. Second, we can adapt our approach to consider digital-clocks since they allow to take into account time imprecision. We may also combine off-line and on-line testing within the same testing architecture aiming to better balance the space/time trade-off. As a future work direction, we are intending to implement our testing methodology in the context of distributed concurrent software architectures.

References

1. Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of ICSM'08*, pages 307–316, Beijing, China, September 28 - October 4 2008. IEEE.
2. A. J. Maâlej, M. Krichen, and M. Jmaïel. A comparative evaluation of state-of-the-art load and stress testing approaches. *International Journal of Computer Applications in Technology (IJCAT)*, 51(4):283–293, 2015.
3. J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, volume 1664 of *LNCS*. Springer, 1999.
4. M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.
5. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
6. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Proceedings of SPIN'04*, volume 2989 of *LNCS*. Springer, 2004.
7. T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *Proceedings of ICALP'92*, volume 623 of *LNCS*. Springer, 1992.
8. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.
9. S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*. Springer, 2002.
10. A. J. Maâlej, Z. B. Makhlouf, M. Krichen, and M. Jmaiel. Conformance testing for quality assurance of clustering architectures. In *Proceedings of the 2nd International QASBA'13 Workshop, in conjunction with ISSTA'13*, pages 9–16, Lugano, Switzerland, July 15 2013. ACM.