Test purpose generation for timed protocol testing

Sébastien Salva
LIMOS - UMR CNRS 6158
Université d'Auvergne, Campus des Cézeaux,
Aubière, France
Email: salva@iut.u-clermont1.fr

Antoine Rollet
LABRI - UMR CNRS 5800
Université de Bordeaux, Talence cedex, France
Email: rollet@labri.fr

***Abstract*—Test purposes are requirements, usually constructed by hand, which aim at testing critical properties on implementations. These ones are then used by testing methods to generate test cases. Writing them manually is a heavy task, this is why we propose several techniques to generate test purposes automatically or semi-automatically from a specification. These methods use different strategies for test purpose generation in order to focus on specific communication protocol properties like the test of critical states and actions, or the test of service invocations. These methods are basically based on a test purpose design language which takes into consideration testability. This one mesures the test quality of a system with the evaluation of several criteria (observability, controllability,...). Measuring the testability while generating test purposes helps to reduce the test costs and helps to improve the fault detection during the testing process.**

Key-words : Timed protocols, conformance testing, test purpose, testability.

I. Introduction

Testing techniques are now well established methods in development processes and are used to check various aspects of an implementation such as its interoperability, its performance, its robustness, or its conformance which is the topic of this paper. This last one aims at checking the consistency between an implementation, seen as a black box, and its specification. Testing methods are necessary to produce reliable software, systems or communication protocols. Nevertheless they tend to be costly, especially when specifications are large, timed or distributed. Indeed, testing the whole system with exhaustive testing methods often leads to a space explosion problem (and as a consequence to a test case explosion).

To reduce these costs a solution, well-known in the protocol development area since twenty years, is the construction of test purposes, used later for testing. Test purposes are test requirements (scenarios or suite of properties) which are usually given by designers, and sometimes added to the specification. They can be used for testing the current system but also the upgraded ones later. Basically, test purposes aim at checking the critical system parts such as the critical communications, the internal states or the invocations of external partners or services. Test cases are then derived and executed with test purpose based methods or by hand.

Although using this approach greatly reduces the test costs, the first issue encountered is that test purposes are essentially constructed by hand. And this is particularly difficult when the system is large, has real-time constraints or is distributed. Some works [5], [10] tackle to the test purpose generation for untimed systems, but to our knowledge none of them gives generation methods for timed systems and timed protocols.

This is why we propose in this paper several techniques to generate test purposes automatically or semi-automatically from timed systems modeled with timed automata. During the generation, we take into account some specific properties of protocols, such as critical states or service invocations of another layer. We propose two categories :
(1) the first one is composed of two methods which produce semi-automatically test purposes from critical properties. Designers just have to give either the critical states (for the first method) or transitions (for the second one) to compute the test purposes that will be able to test them.
(2) The second category is composed of two methods which generate test purposes automatically. Each one uses a specific strategy, to guide the test purpose generation. The first method tries to recognize the invocation of services in the specification and generates test purposes which can call them. The second method detects the critical states of the system and produces test purposes to test their conformance.

We also focus on the effectiveness of the generated test purposes. It is indeed more interesting to generate test purposes which can be always executed, which can detect the greatest number of faults and which require the lowest time execution. This "effectiveness of test" is defined by the system testability [9], [6]. Testability is evaluated with the analysis of different criteria and is measured precisely with testability degrees. So, to take into consideration testability, we define a new test purpose design language, composed of operators which evaluate a list of testability degrees while generating test purposes.

This paper is structured as follow: Section II provides an overview of test purpose based methods, and our motivations. Section III describes the theoretical framework needed in this study. We give an overview of the testability evaluation in Section IV. Section V presents our test purpose model, based on a rational language which takes into account testability and which may represent test purpose sets with short expressions. The test purpose generation methods are described in Section VI and a brief overview on test case generation is given in Section VII. We conclude in Section VIII.

II. Related work and motivations

Testing consists in checking if the implementation conforms to the specification by stimulating the implementation and observing its behaviour. Testing methods can be grouped into two categories. On the one hand, we have exhaustive methods which involve generation of test cases on the complete specification, execution of the test cases on the implementation and analysis of the test results. On the other hand, we have non exhaustive methods, like test purpose based ones, [5], [11], [2], [8], [7], which aim to test only local parts of the implementation. These methods are oriented: designers or experts who have a good knowledge of the system, describe the requirements to be tested (test purposes), which are generally the important or critical parts of the system. Then, either test cases are constructed manually or are generated from these requirements and from specification parts, thereby reducing the specification exploration in comparison with exhaustive methods.

Some test purpose based methods [13], [11], [2], [8] are introduced below: In [13], specifications and test purposes are modelled with TIOA (Timed Input Output Automata). Specification paths which can be synchronized with test purposes are extracted. Then, these paths and test purposes are transformed into region graph to sample the time domain. Specification paths are synchronized with test purposes to generate test cases. In [11], test cases are constructed by adapting the TGV untimed method (Test Generation with Verification methodology). Specification and test purposes, modelled with timed automata, are synchronized. The result is translated into a non real-time model (SEA). Then, test cases are obtained by extracting the visible behaviour and by separating input and output actions. In [2], the authors use specifications and test purposes modelled by TIOA. Specification paths are first extracted to synchronize them with the test purposes, then time intervals of the test purpose and of the specification are synchronized. The main interest of this method lies in the computation of feasible paths which are the paths that can be executed from their initial state to their final one. The final test case set is composed of these feasible paths.

Test purpose based methods are suitable for testing: test costs are reduced and large systems or protocols can be tested without having a memory space explosion. Nevertheless, such methods lead to a significant issue : test purposes are essentially constructed by hand. This is why some works about test purpose generation have been proposed, for untimed systems [5] and distributed ones [10]. We make our contribution in this field by proposing test purpose generation methods for timed protocols. Another originality of this paper is to define a test purpose description language which takes into account testability. Thanks to this language, our methods target the tests of protocol properties (critical states, service invocation,...) by giving some specific expressions.

III. Definitions and notations

A. The Timed Input Output Automaton model

TIOA (Timed Input Output Automata) are graphs describing timed systems. This model, extended from the timed automaton [1], expresses time with a set of clocks which can take real values (dense time representation) and by time constraints, called clock zones, composed of time intervals which sample the time domain. Actions of the system are modelled by symbols labelled on transitions. Clocks are all evolving with the same speed, and it is possible to reset some of them while firing a transition. Two kinds of symbols are used, the input ones given to the system, and the output ones observed from it. Actions may be executed if the clock zone of the transition is satisfied by the clocks.

Definition III.1 (Clock zone) *A clock zone Z over a clock set C is a tuple $< Z(x_1), ...Z(x_n) >$ of intervals such as $card(Z) = card(C)$. $Z(x_i)$ is a time interval for the clock x_i, $Z(x_i) \in \{[a_i\ b_i],\ [a_i\ +\infty[\mid a_i \in \mathbb{R}^+, b_i \in \mathbb{R}^+\}$.*

We say that a clock valuation $v = (X_1, ..., X_n)$ satisfies Z, denoted $v \models Z$ iff $X_i \in Z(x_i)$, with $1 \leq i \leq n$.

For two clock zones Z and Z', we denote some operators: $Z \cap Z' = \{v \mid v \models Z$ and $v \models Z'\}$ and $Z + d = \{[a_k\ b_k + d] \mid [a_k\ b_k] \in Z\}$

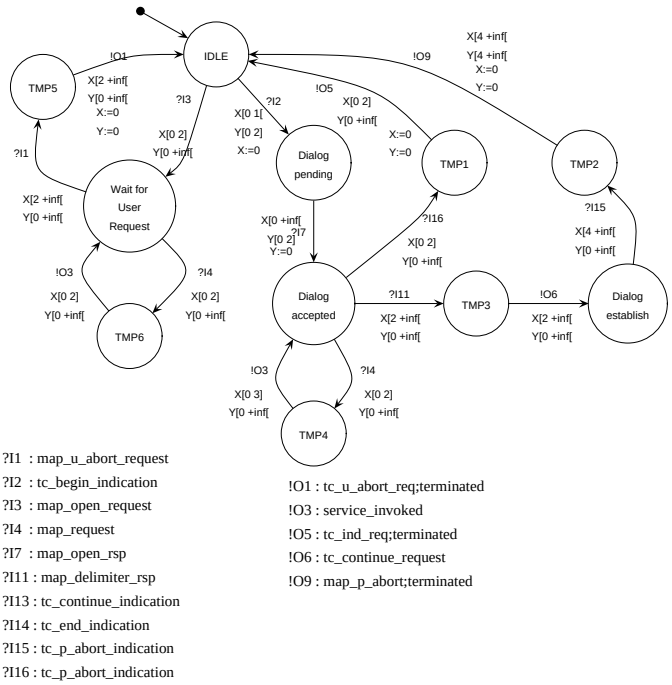

Fig. 1. A TIOA modelling a part of the MAP-DSM protocol

Definition III.2 (Timed Input Output Automata (TIOA)) *A TIOA $\mathcal{A}$ is a tuple $< \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}} >$ such as:*

- *$\Sigma_{\mathcal{A}}$ is a finite alphabet composed of input symbols (beginning by "?") and of output symbols (beginning by "!"),*

- *$S_{\mathcal{A}}$ is a finite set of states, $s^0_{\mathcal{A}}$ is the initial one,*
- *$C_{\mathcal{A}}$ is a finite set of clocks,*
- *$E_{\mathcal{A}}$ is the finite transition set. A tuple $< l, l', a, \lambda, Z >$ models a transition from the state l to l' labelled by the symbol a. The set $\lambda \subseteq C_{\mathcal{A}}$ gathers the clocks which are reset while firing the transition, and $Z =< Z(1), ..., Z(n) >_{(n=card(C_A))}$ is a clock zone.*

A TIOA, modelling a part of the MAP-DSM protocol, is given in figure 1. Among the protocols used with GSM (Global system for Mobile communication), nine protocols are grouped together into the MAP (Mobile application part). Each one corresponds to a specific service component. The Dialog State Machine (DSM) manages dialogs between MAP services and their instantiations (opening, closing...). A DSM description can be found in [4]. The specification of figure 1 describes the request of the MAP service by a user(?I3). This one can invoke several requests (?I4) which aim to start some services (!O3). A dialog can be accepted then established or it can be stopped (!O5 or !O9).

IV. Testability

Testability [9] gathers quality criteria which evaluate the capacity of the system to reveal its faults, the accessibility of its components and the test costs (test execution, state space) depending on the system modelling. Testability is measured with the specification analysis and on the evaluation of factors, called *testability degrees*. These degrees help designers to improve the specification so that the final system should be completely tested. Figure 2 summarizes the software life cycle which takes into account testability measurements.

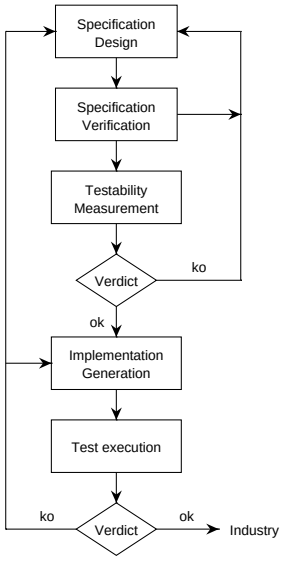

Fig. 2. Testability in life cycle

In [12], we have defined the testability of timed systems modelled with TIOA. We have proposed several testability degrees based on the time properties analysis. We give here an overview of these degrees. Their complete definitions as well as examples and algorithms, can be found in [12]. Each degree is based on a polynomial algorithm. These ones are:

- the **Timed shape Degree**, denoted S_t, $0 \leq S_t \leq 1$, measures, for each specification path, the amount of infinite clock zones, that is the clock zones composed of unbounded intervals, and measures the number of intervals which are unbounded per clock zone. Infinite clock zones are always bounded arbitrarily by testing methods, consequently, infinite clock zone are not covered by the tests. So, the system is only partially tested. Thereby, for a path p, the closer to 1 $S_t(p)$ is, the fewer p has infinite clock zones and the more covered by the tests, the system is,
- the **Time interval reachability** Degree, denoted R_t, $0 < R_t \leq 1$ evaluates the reachability of clock zones and of transitions from the initial one. This factor performs a reachability analysis on the clock zones to determine if specification paths can be completely executed. If some clock zones are not reachable by the clocks, then several transitions cannot be reached and tested. So the system is not testable and can only be partially tested. For a clock zone Z and a path p, the closer to 1 $R_t(Z, p)$ is, the more the clock zone Z is reachable from the initial one, and the more testable p is. To compute this degree, we use a forward symbolic analysis using a $post$ operator, able to construct the next clock zone of a state after firing a transition,
- The **Observability Degree**, denoted Obs, $0 \leq Obs \leq 1$, measures the observability of system paths. For a path p, this degree analyzes if for a transition labelled by an input symbol, there exists a consecutive transition labelled by an observable output one. The closer to 1 $Obs(p)$ is, the more observable and testable p is.
- The **Determinism Degree**, denoted Det, $0 < Det \leq 1$, measures the system determinism. It is well known that if a system is not deterministic, it cannot be controlled and tested. For a path p, this factor determines whether each state of p is deterministic. For a state s, we consider the different cases of indeterminism: 1. outgoing transitions labelled by the same symbol, 2. outgoing transitions labelled by different symbols but labelled by clock zones with a non-empty intersection, 3. at least one outgoing transition labelled by an input symbol and transitions labelled by output symbols, all of these ones labelled with clock zones with a non-empty intersection. The closer to 1 $Det(p)$ is, the more deterministic and testable the states of p are.
- The **Time Execution Degree**, denoted TE_t, $0 \leq TE_t \leq 1$, measures the execution cost of system paths. For timed systems, this cost depends basically on the minimal and the maximal execution time of each path (E_{min} and E_{max}). For a path p, the closer to 1 $TE_t(p)$ is, the shorter the average time execution of p is and the more reduced the test costs of p are.

During the test purpose generation, a vector $(x_1, ...x_n)$

composed of the previous degrees is used to generate the most testable test purposes by using the most testable paths of the specification. The choice of the vector belongs to designers but we suggest to use all the previous degrees since each expresses a specific criteria. The testability degrees Obs, Det, S_t and R_t enable to use the most testable paths to construct test purposes, that is the most observable (Obs), whose states are the most deterministic (Det) and whose time intervals are the most reachable and bounded (R_t, S_t). The Time Execution degree TE_t chooses the paths with the lowest average time execution. Therefore, the overall time of the test execution is reduced as well.

Notice that it can be useful to refine the testability evaluation by adding a "weight" which gives more or less significance to each degree. If we suppose that we give a vector $W = (w_1, ..., w_n)$ of weights, then the testability evaluation in the algorithm would become $D_{p_i} = \sum_{1 \leq j \leq n} w_j x_j(p_i)$. Since the principle is similar, we do not consider this in the following.

V. TIMED TEST PURPOSE MODELLING

To express the test purpose generation in the methods of section VI, we need of a model which can take into consideration testability degrees and which can easily model test purpose sets with few expressions. Since the TIOA model is not sufficient, we propose to define a rational language, called TPDL (Test Purpose Description Language). Comparing to the TIOA model, this one is still composed of clock zones, states and transitions. So, it is still possible to add and test temporal restrictions associated to transitions. But it is also composed of operators which handle testability degrees and return path sets. From one TPDL expression, and after having executed the expression operators, we obtain a test purpose set composed of TIOA states and transitions. So, these test purposes could be later used with most of the test purpose based methods [2], [8], [11], [7] to produce the final test cases (see section VII). Note that this language may be easily extended, according the user needs, with new operators.

Definition V.1 (TPDL Language) *Let* $\mathcal{A} =< \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s^0_{\mathcal{A}}, C_{\mathcal{A}}, E_{\mathcal{A}} >$ *be a TIOA and* $(x_1, ..., x_n)$ *be a testability degree vector chosen by designers. The rational language* $TPDL$*, modelling test purposes over* $\mathcal{A}$*, is composed of the keywords described in figure 3.*

The interesting feature of this language is the use of operators which automatically produce the most testable paths between two states. Designers don't need to search them in the specification. The max and $noloop$ operators produce specification paths by using a classical DFS (Depth First path Search) algorithm. This algorithm performs a search of the shortest path between two states deep-wise in polynomial time. It stores only one path at time and so uses few memory space. The $noloop$ operator algorithm corresponds to DFS one between two states s_i and s_j. The max operator, which takes an important place in the language (used by Pre and Post operators), computes the most testable path whose the testability degrees $(x_1, ...x_n)$ are the closest to 1. For instance, if the degree list is (Obs, TE_t) this is the path, the most observable and which requires the less time to be executed. The algorithm of the max operator is given below.

$Label$	$\langle a, Z, \lambda \rangle$ with $a \in \Sigma_{\mathcal{A}}, Z$ a clock zone and λ a clock reset set
$State$	$State_k$ with $s_k \in S_{\mathcal{A}}$
$Trans$	$State_i.\langle a, Z, \lambda \rangle.State_k$ with $(s_i, s_k, a, \lambda, Z) \in E_{\mathcal{A}}$
$noloop$	$State_i.noloop.State_j$ models a set of paths P from s_i to s_j without loop: $\forall p \in P, \forall$ path r, s, $t \in E^*_{\mathcal{A}}$ such as $path = r.s.t, s_i \xrightarrow{r} s_k \xrightarrow{s} s_l \xrightarrow{t} sj \Longrightarrow s_k \neq s_l$. $noloop = \emptyset$ iff $P = \emptyset$.
max	$State_i.max_{(x_1,...x_n)}.State_j$ models a path p of $State_i.noloop.State_j$ such as p is the most testable path of $State_i.noloop.State_j$. (algorithm given below).
Pre	$Pre_{(x_1,...x_n)}.State_j = s^0_{\mathcal{A}}.max_{(x_1,...x_n)}$. $State_j$ models the most testable path from the initial state to s_j.
$Post$	$State_j.Post_{(x_1,...x_n)}$ models the system reset. If it exists a reset function $reset()$ then $State_j.Post_{(x_1,...x_n)} = reset()$, otherwise $State_j.Post_{(x_1,...x_n)} = State_j.max_{(x_1,...x_n)}.State_0$
$Path =$	$\emptyset \mid Trans \mid State_i.max_{(x_1,...x_n)}.$ $State_j \mid Pre_{(x_1,...x_n)}.State_j \mid State_j.$ $Post_{(x_1,...x_n)} \mid Path^* \mid Path \cup Path$
$TP =$	$Pre.Path.Post = \{tp_1, ..., tp_n\} (n \geq 1)$ is a test purpose set where tp_i is composed of a preamble, a path and a postamble of the specification.

Fig. 3. TPDL keywords

Algorithm

$state_i.max(x_1, ...x_n).state_j$
Input: $\mathcal{A}$(TIOA), $(x_1, ...x_n)$ (a vector of testability degrees)
Output: p (a path)
P a path set
$P = DFS(s_i, s_j)$
For each $p_i \in P$
  Compute the testability degrees $x_1(p_i), ..., x_n(p_i)$,
  $D_{p_i} = \sum_{1 \leq j \leq n} x_j(p_i)$
EndFor
p is the path of P such as $D_p = max(D_{p_i} \mid p_i \in P)$

With this language, we can construct test purposes by giving mere expressions. For instance, $TP = Pre_{(x_1,...,x_n)}$.

$State_{dialog\_accepted}. <?I4, (X[01]Y[01]), \emptyset >$.
$Post_{(x_1,...,x_n)}$, models one test purpose which uses the most testable path to reach the state *Dialog_accepted*, fires the transition labeled by "?I4" with a restrictive clock zone, and uses the most testable postamble to reset the system. After having executed each operator, we automatically obtain the specification path composed of the labels "?I2?I7?I16!O5" (see figure 1).

VI. Test purpose generation

We propose two method categories : (1) the first one is composed of two semi-automatic approaches which test the accessibility of critical states or of transitions. (2) the second category gathers two automatic approaches which use different strategies to generate test purposes: the first one test service invocations and the second one the conformance of critical states.

A. Semi-automatic test purpose generation

1) Semi-automatic test purpose generation for critical state testing: This method generates test purposes to test whether each critical state of a set S_F can be reached from an initial one s_{init}. This method is especially suitable when the specification states have a precise meaning, which is often the case in protocols.

So, let $\mathcal{A} =< \Sigma_\mathcal{A}, S_\mathcal{A}, s^0_\mathcal{A}, C_\mathcal{A}, E_\mathcal{A} >$ be a TIOA. And let $s_{init} \in S_\mathcal{A}$ be a state and $S_F = \{q_1, ..., q_m\} \subset S_\mathcal{A}$ be a critical state set. The expression, given below, produces a set of test purposes which test if each state $q_j \in S_F$ can be visited from s_{init} with all the acyclic paths from s_{init} to q_j. The most testable preamble is used to reach s_{init} from the initial state s_0, and the most testable postamble is used to reset it.

$$TP = \bigcup_{1 \le j \le m} \{Pre_{(x_1,...,x_n)}.State_{init}.noloop.State_j. Post_{(x_1,...,x_n)} \mid State_{init}.noloop.State_j \neq \emptyset\}$$

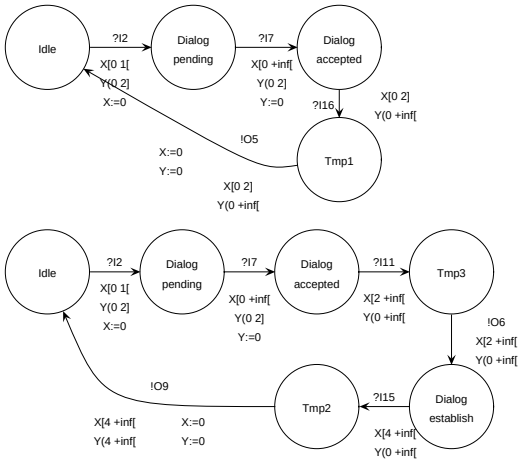

Fig. 4. Test purposes for testing critical states

For instance, on the specification of figure 1, to test if a pending dialog can be established, we can use the initial state $s_{init} = Dialog\_pending$ and the states $S_F = \{Dialog\_accepted, Dialog\_establish\}$. With the previous expression, we obtain the two test purposes illustrated in figure 4.

2) Semi-automatic test purpose generation from a transition list T: This second method generates test purposes to test if a successive list of critical transitions can be executed. So, let $\mathcal{A} =< \Sigma_\mathcal{A}, S_\mathcal{A}, s^0_\mathcal{A}, C_\mathcal{A}, E_\mathcal{A} >$ be a TIOA and let $T = \{(q_1, q'_1, a_1, \lambda_1, Z_1)... (q_n, q'_n, a_n, \lambda_n, Z_n)\}$ be a transition list. The following expression constructs one test purpose which tries to fire each transition of T successively. The first transition is reached, from the initial state s_0, by the most testable preamble, and the other ones with the most testable paths obtained by the operator "max".

$$TP = Pre_{(x_1,...,x_n)}.State_1. < a_1, Z_1, \lambda_1 > .State'_1. F_2.F_3...F_n.Post_{(x_1,...,x_n)}\}$$

where $F_i = max_{(x_1,...,x_n)}.State_i. < a_i, Z_i, \lambda_i > .State'_i$

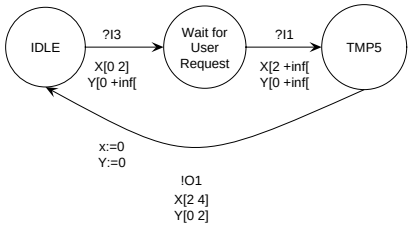

Fig. 5. Test purpose for critical transition testing

For instance, on the specification described in figure 1, to test if after having requested the end of a map invocation, this one is stopped only 2 seconds later (with the clock zone $X[2\ 4]Y[0\ 2]$), we can use the transition list $\{Wait\_for\_User\_Request \xrightarrow{?I1\ X[2\ +\infty[Y[0\ +\infty[} TMP5, TMP5 \xrightarrow{!O1\ X[2\ 4]Y[0\ 2]} IDLE\}$. By using the previous expression, the resulting test purpose is described in figure 5.

B. Automatic test purpose generation

The following methods automatically generate test purposes from a specification. The main issue is to construct a limited number of test purposes, without accomplishing an exhaustive test. To do this, we propose two methods which use specific strategies. The first one produces test purposes for testing protocol service invocations by supposing that invocations are named classically ("service_name" followed by "begin","open","request",...). The second method tests the critical state conformance by supposing that the critical states are the most visited ones. These assumptions can be modified, since they do not interact on the test purpose expressions.

1) Test purpose generation for testing service invocation: Protocols are often composed of service invocations that call functionalities of other layers or systems. Services are usually named and are called explicitly by their names with transition labels. So, this method tries to recognize all the complete service invocations and generates test purposes to test them. Usually, a service invocation corresponds to a list of steps (labels) which are: (1) the service can be initialized (though not necessarily) which is often modelled by a label of the form "service_name" followed by "begin", or "open", or "request". (2) the service can be called (used) which is often modelled by a label of the form "service_name" followed by "request" or "call". It may receive a response which is modelled by a label of the form "service_name" followed by "indication" or "response". (3) the service can be terminated (not necessary) which is often modelled by a label of the form "service_name" followed by "end" or "terminated" or "abort" or "confirmation".

First, this method searches for the service invocation lists and gathers them into a set, denoted IS. Then, for each list $(l_1, ..., l_n) \in IS$, the method extracts, from the specification, the tuples of transitions $T = \{(t_1, ..., t_n) \mid t_i \text{ is labelled by } l_i\}$. Finally, the method generates one test purpose for each transition tuple $(t_1, ..., t_n) \in T$ which aims to test if each transition can be fired. The final test purpose set is:

$$TP = \bigcup_{(t_1,...,t_n)\in T} \{tp_{(t_1,...,t_n)}\}$$

with $tp_{(t_1,...,t_n)}$ the test purpose expression used for testing the invocation $(t_1, ..., t_n)$:

$$tp_{(t_1,...,t_n)} =$$

$$\begin{cases} \emptyset \text{ if } \exists(1 \leq i < n) \mid State'_i.max_{(x_1,...,x_n)}.State_{i+1} = \emptyset \\ Pre_{(x_1,...,x_n)}.State_1. < a_1, Z_1, \lambda_1 > .State'_1.F_2.F_3... \\ F_n.Post_{(x_1,...,x_n)} \text{ otherwise} \end{cases}$$

where $F_i = max_{(x_1,...,x_n)}.State_i. < a_i, Z_i, \lambda_i > .State'_i$

This expression produces one path which aims to invoke the service completely from the initial state of the specification, with the condition that each transition t_{i+1} could be reached from the previous one t_i. Otherwise, the service cannot be completely invoked, consequently the test purpose is not generated (the max operator produces an empty path $\emptyset$).

The specification, described in figure 1, is composed of several service invocations. For example, the labels $?map\_open\_request$ $?map\_request$ $?map\_u\_abort\_request$ (?I3?I4?I1) model one of them. From these labels, we obtain one tuple of transitions : $(IDLE \xrightarrow{?I3} Wait\_for\_user\_request, Wait\_for\_user\_request \xrightarrow{?I4} TMP6, Wait\_for\_user\_request \xrightarrow{?I1} TMP5)$. The previous expression gives one test purpose, expressed in figure 6.

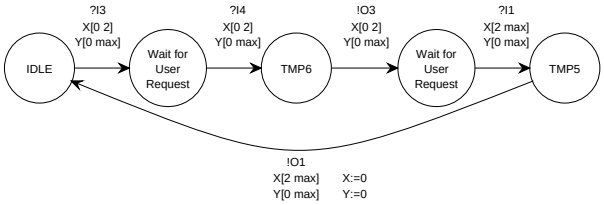

Fig. 6. Testing services with test purposes

2) Test purpose generation for testing critical states: Test purposes are often used to test the critical parts or properties of implementations. This method tries to determine the critical states of TIOA and generates test purposes to test their conformance. It is not obvious to set which state is critical since no general and formal definition is given in literature. So, in this paper, we suggest that the critical states are the most potentially visited states in the specification from its initial one. Nevertheless, other criteria could be chosen, such as the less visited states, or the quiescent states ([3]).

To detect these critical states, we use the following algorithm on the specification $\mathcal{A}$ to extract its acyclic paths. While constructing them, the algorithm counts the number of times each state is visited. The most visited ones are the critical states. The algorithm, derived from the DFS one, has a polynomial complexity.

Algorithm

$Critical - state(\mathcal{A}, s)$
Input: $\mathcal{A} =< \Sigma_\mathcal{A}, S_\mathcal{A}, s^0_\mathcal{A}, C_\mathcal{A}, E_\mathcal{A} >$(TIOA), $s = s_0$
Output: CS (Critical state set)
For each $t = (s, s', a, \lambda, Z) \in OutgoingTransition(s)$
  If Label(t) == UNEXPLORED
    Label(t)=VISITED, Count(s)=Count(s)+1
    $Critical - state(\mathcal{A}, s')$
  Else Count(s')=Count(s')+1
End For
$CS = \{$ state $s \mid Count(s) > \frac{\sum_{s_i \in S_\mathcal{A}} Count(s_i)}{card(S_\mathcal{A})}\}$

Then, we propose to test all the outgoing transitions of each critical state with test purposes. So, for a state s_i and for each outgoing transition $t = (s_i, s'_i, a_i, \lambda_i, Z_i)$, the expression, given below, constructs one test purpose which aims to reach s_i, to fire t and to reset the system.

$$TP = \bigcup_{s_i \in CS} \{tp(s_i)\}$$

where $tp(s_i)$ is one test purpose expression which aims to test all the outgoing transitions of the state s_i.

$$tp(s_i) = \bigcup_{(s_i,s'_i,a_i,\lambda_i,Z_i)\in E_\mathcal{A}} \{Pre_{(x_1,...,x_n)}.State_i. \langle a_i, Z_i, \lambda_i\rangle.State'_i.Post_{(x_1,...,x_n)}\}$$

With the specification of figure 1, we have 3 critical states $IDLE$, $Dialog\_accepted$ and $Wait\_for\_user\_request$. For the state $IDLE$, we obtain the test purposes of figure 7. These ones test the outgoing transitions of $IDLE$ and reset the system.

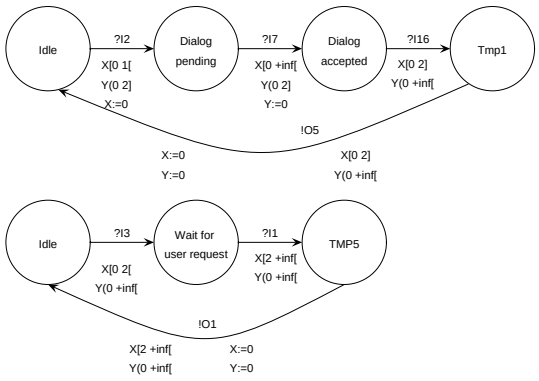

Fig. 7. Test purposes testing critical states

VII. Test case generation and execution

Once test purposes have been constructed, test methods are used to generate test cases [11], [8]. We give in this section an overview of this generation. These methods basically synchronize test purposes and the specification to produce: complete paths (specification paths from the initial state which include the test purpose labels in the same order), pass clock zones (time constraints which satisfy both the specification and the test purpose) and inconclusive ones (time constraints which satisfy the specification but not the test purpose). These inconclusive clock zones are used to give an inconclusive verdict if the test purpose is not satisfied and if the implementation does not contradict the specification.

By using our proposed methods, the generated test purposes are already complete specification paths (and the most testable ones). So, the test case generation corresponds to the construction of the inconclusive clock zones only. If the inconclusive verdicts are not required then our generated test purposes are the final test cases where the clock zones are the PASS ones.

Test case execution methods can be found in [13], [8]: each test case is executed by a tester; if all observable output actions, of each test case, can be successively observed in the Pass clock zones, then the PASS verdict is given which means the test purpose is satisfied during the test execution. If one output action is observed in an inconclusive clock zone then the test purpose is not satisfied but the implementation is not faulty (inconclusive verdict). Otherwise, the verdict is fail.

VIII. Conclusion

We have proposed, in this paper, four semi-automatic and automatic test purpose generation methods for timed protocols, modelled by TIOA. The originality of these methods is the use of testability degrees by mean of the TPDL language. With the testability degree evaluation during the generation, the test purposes are the most efficient for detecting faults and are executed with the lowest time execution.

We have experimented with success these methods on the complete timed MAP-DSM communication protocol. We have firstly applied the automatic methods, and have observed that many critical properties may be automatically tested, but of course not all of them. So, to refine the tests, we have used the semi-automatic methods to target the test of other states and especially the test of critical transitions with reduced clock zones. For instance, we have generated test purposes to test whether some communication delays between a client and a MAP-DSM server could be reduced without modifying the MAP-DSM server behaviour (any following action should be yet executable).

The TPDL language can be easily extended to propose other test strategies for timed systems or can be improved to test other systems. For example, we are implementing a method which tries to detect all the autonomous sub-systems of a complete TIOA for generating test purposes whose the goal will be to test completely each sub-system. Obviously, other properties like the quiescence or some specific clock constraints can also be easily tested by constructing new expressions.

References

[1] R. Alur and D. Dill. A theory of timed automata. *TCS*, 126:183–235, 1994.

[2] I. Berrada, R. Castanet, and P. Felix. A formal approach for real-time test generation. In *WRTES, satellite workshop of FME symposium*, pages 5–16, 2003.

[3] L. B. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *FATES04 (Formal Approached to Testing of Software),Kepler University Linz, Austria*, pages 71–85, 2004.

[4] N. Carrere. Dsm specification in lotos and test cases generation. *INT (French Telecommunication National Institute)*, 2001.

[5] R. Castanet, C. Chevrier, O. Kone, and B. L. Saec. An Adaptive Test Sequence Generation Method for the User Needs. In *IWPTS'95, Evry, France*, 1995.

[6] K. Drira, P. Azéma, and P. de Saqui Sannes. Testability analysis in communicating systems. *Computer Networks*, 36:671–693, 2001.

[7] A. En-Nouaary. A test purpose based method for testing timed input output automata. In *International Journal of Software Testing, Verification, and Reliability (JSTVR)*, 2007.

[8] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *15th IFIP International Conference, TestCom 2003, France*, pages 211–225, May 2003.

[9] R. S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6), june 1991.

[10] O. Henniger, M. Lu, and H. Ural. Automatic generation of test purposes for testing distributed systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, pages 185–198, Oct. 2003.

[11] A. Khoumsi, T. Jeron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES03 (Formal Approaches for Testing Software), Canada*, Oct. 2003.

[12] S. Salva and H. Fouchal. Some Parameters for Timed System Testability. In *ACS/IEEE International Conference on Computer System and Applications, AICCSA'01 (Beirut, Lebanon)*, June 2001.

[13] S. Salva, E. Petitjean, and H. Fouchal. A simple approach to testing timed systems. In *FATES01 (Formal Approaches for Testing Software), a satellite workshop of CONCUR, Aalborg, Denmark*, Aug. 2001.