

# Generation of tests for real-time systems with test purposes

Sébastien Salva  
LIMOS  
Campus des Cézeaux  
Université de Clermont Ferrand  
salva@iut.u-clermont1.fr

Patrice Laurençot  
LIMOS  
Campus des Cézeaux  
Université Blaise Pascal  
laurencot@isima.fr

## Abstract

Usually, the notion of time introduces space explosion problems during the generation of exhaustive tests, so test-purpose-based approaches have been developed to reduce the costs by testing (usually on the fly) the critical parts of the specification. In this paper, we introduce a test-purpose-based method which tests any behaviour and temporal properties of a real-time system. This method improves the fault detection in comparison with other similar approaches by using a state-characterization-based technique, which enables the detection of state faults on implementations. An example is given with the MAP-DSM protocol modelled with two clocks.

key words: Timed automata, conformance testing, test purpose

## 1. Introduction

Computer applications are being increasingly involved in critical, distributed and real-time systems. Their malfunctioning may have catastrophic consequences for the systems themselves, or for the ones who are using them. Testing techniques are used to check various aspects of such systems. Different categories of test can be found in literature: performance testing, robustness testing, interoperability testing and conformance testing which will be considered here.

Conformance testing consists in checking if the implementation is consistent with the specification by stimulating the implementation and observing its behaviour. *Test cases* which consist of interaction sequences are applied on the implementation via a test architecture [13, 23]. This one describes the configuration in which the implementation is experimented, which includes at least the implementation interfaces (called PCO, *point of control and observation*) and the tester which executes the test cases to establish the test verdict:

- *pass*: no error has been detected.
- *fail*: there is at least an error on the implementation.

- *inconclusive*: pass and fail cannot be given (the test cannot be performed).

Many testing methods have been proposed for generating automatically test cases from untimed systems [30, 9, 24, 3] or timed ones [5, 13, 10, 22, 29]. Most of the timed ones are exhaustive methods which generally transform specifications into larger automata (such as region graphs [13, 23], grid automata [29, 12], or SEA [19, 18]) to generate test cases on the complete specification. This kind of method is interesting and usable with small systems but can end in a space explosion problem (usually obtained from state explosion) with larger ones. So, others techniques called test-purpose-based approaches, have been proposed to test the most critical systems parts. These ones check local implementation parts from test requirements given by engineers, which are called *test purposes*. The conclusion of the test is given here by checking the satisfaction of the test purpose in the implementation.

Some test purpose based methods have been proposed [7, 8, 20, 28, 18] to test timed systems. These ones strongly reduce the test cost and can be generally used in practice to test specification properties on implementations. However, faults like extra(missing) states and transfer faults cannot be detected with the previous techniques. Such faults can modify the system internal state (this one becomes unknown and faulty), so detecting them is important.

In this paper, we introduce a test-purpose-based method which can test the conformance and the robustness of implementations, by testing any temporal or behaviour properties belonging to the specification (called Accept properties), but also any other ones given by designers (Refuse properties). The test case generation is performed by a *timed synchronous product* which combines the specification with the test purposes and prevents state explosion. With this product, we obtain a graph which includes the specification and the test purpose properties. Furthermore, to improve the fault detection, we use a state-characterization-based approach to identify each state visited in the implementation. So, missing and transfer faults can be detected.

This article is structured as follow: Section 2 describes the theoretical framework needed in this study. Section 3 provides an overview of testing methods, and a related works on timed testing with test purpose based approaches. Section 4 introduces the concept of Timed test purposes. The testing method is described in Section 5. We apply this one on a real system, which is a part of the MAP-DSM protocol. Then, we give the fault coverage of the method in Section 6. Finally, we give an overview of an academic test tool, which implements this testing method, in Section 7 and we conclude in Section 8.

## 2. Definitions

### 2.1. The Timed Input Output Automaton model

TIOA (Timed Input Output Automata) are graphs describing timed systems. This model, extended from the timed automaton one [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 sampling the time domain. Actions of the system are modelled by symbols labelled on transitions: input symbols, beginning with "?" are given to the system, and output ones, beginning with "!" are observed from it. A TIOA transition, labelled by an input symbol ?a, can be fired if the system receives ?a while its time constraint is satisfied. In the same way, a TIOA transition, labelled by an output symbol !a, is fired if !a is observed from the system while the time constraint is satisfied.

**Definition 1 (Clock zone)** A clock zone  $Z$  over a clock set  $C$  is a tuple  $\langle Z(x_1), \dots, Z(x_n) \rangle$  of intervals such that  $\text{card}(Z) = \text{card}(C)$  and  $Z(x_i) = [a_i, b_i]$  is a time interval for the clock  $x_i$ , with  $a_i \in \mathbb{R}^+$ ,  $b_i \in \{\mathbb{R}^+, \infty\}$ .

If  $X_i$  is the clock value of the clock  $x_i$ , we say that a clock valuation  $v = (X_1, \dots, X_n)$  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:

- $Z \cap Z' = \{v \mid v \models Z \text{ and } v \models Z'\}$
- $Z/Z' = \{v \mid v \models Z\} / \{v' \mid v' \models Z'\}$

### Definition 2 (Timed Input Output Automata (TIOA))

A TIOA  $\mathcal{A}$  is a tuple  $\langle \Sigma_{\mathcal{A}}, S_{\mathcal{A}}, s_{\mathcal{A}}^0, C_{\mathcal{A}}, E_{\mathcal{A}} \rangle$  where:

- $\Sigma_{\mathcal{A}}$  is a finite alphabet composed of input symbols and of output symbols,
- $S_{\mathcal{A}}$  is a finite set of states,  $s_{\mathcal{A}}^0$  is the initial one,
- $C_{\mathcal{A}}$  is a finite set of clocks,
- $E_{\mathcal{A}}$  is the finite transition set. A tuple  $(s, s', a, \lambda, Z)$  models a transition from the state  $s$  to  $s'$  labelled by the symbol  $a$ . The set  $\lambda \in C_{\mathcal{A}}$  gathers the clocks which are reset while firing the transition, and  $Z = \langle Z(1), \dots, Z(n) \rangle_{(n=\text{card}(C_{\mathcal{A}}))}$  is a clock zone.

A TIOA example, modelling a MAP-DSM part, is given in Figure 1. Among the protocols used with GSM (Global system for Mobile communication), nine protocols are grouped 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 [6]. The specification of Figure 1, describes the request of the MAP service by a user(?I3). This one can invoke several MAP requests(?I4) which aim to start some services(!O3). A dialog can be accepted then established or it can be abandoned(!O5 or !O9).

If we consider the transition  $(T_{mp2}, IDLE, !O9, \{X, Y\}, \langle X_{[4+\infty[}, Y_{[4+\infty[} \rangle)$ , the two clocks  $x$  and  $y$  must have an greater value than 4 so that the system produces the symbol !O9. After this execution,  $x$  and  $y$  are reset.

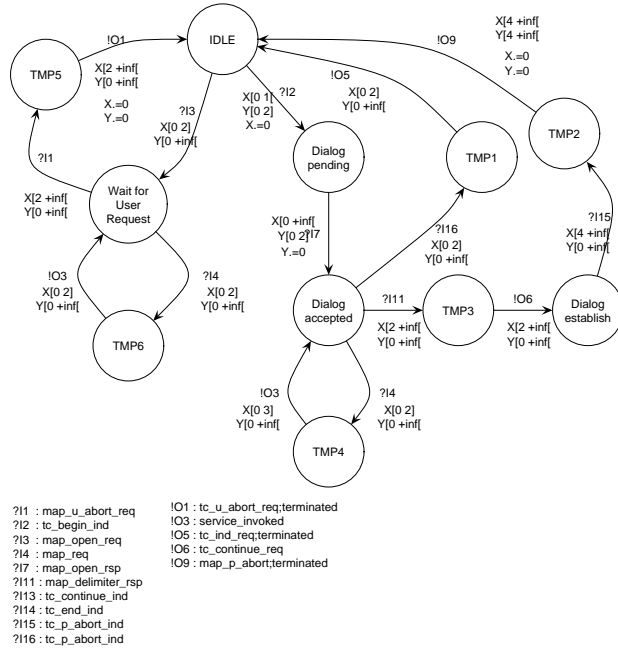


Figure 1. A TIOA

### 2.2. Fault model

The fault model is a set of potential faults (untimed and timed ones) that can be detected on implementations by the testing process. For the TIOA semantic, the fault model can be found in [13, 12]. This one is composed of:

- **Output faults:** An implementation produces an output fault, for a specification transition  $(s, s', !a, \lambda, Z)$ , if it does not respond with the expected output symbol !a.
- **Transfer faults:** An implementation produces a transfer fault if from a state, it goes into a state different from the expected one by accepting an input symbol or by giving an output one.

- **Extra state faults:** An implementation is said to have an extra (missing) state if its number of states must be reduced (increased) to be equal to the number of states of the specification.
- **Time constraint widening fault:** Such a fault occurs if the implementation does not respect the time delay granted by a specification clock constraint, that is if the upper (or lower) bound of a clock constraint is higher (smaller) in the implementation. This fault may occur on input or output symbols: for an output one, the implementation does not respond in the expected delay given by the specification, for an input symbol, the implementation accepts the input symbol in delays wider than the one given by the specification.
- **Time constraint restriction fault:** This fault occurs only with input symbols. An implementation produces this fault if it rejects an expected input symbol in delays satisfying the clock constraint given by the specification. In this case, the clock constraint of the implementation is more restrictive than the specification one. Since output symbol cannot be controlled by the system environment, an implementation that produces an output symbol in a more restrictive delay than the one specified is seen as a valid restriction of the specification.

### 3. Related Works

In the literature, testing methods can be grouped into two categories:

- **the exhaustive testing 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. To describe the set of correct implementations, a conformance relation is first defined, then test cases are given or generated from the specification to check if the relation is satisfied or not. Some works about timed systems testing can be found here [5, 13, 22, 29, 12].
- **the non exhaustive testing methods** [7, 8, 21, 20, 28, 11, 18, 2, 14], which aims to test local parts of implementations. This concept aims to check if a set of properties, called a test purpose, can be executed on an implementation during the testing process. Test purpose can be either manually given by designers, or can be automatically generated [7, 17]. Then, test cases are generally generated from the test purposes and from some specification parts, reducing the specification exploration in comparison with exhaustive methods (reducing in the same time the test costs). Finally, test cases are executed on the implementation to observe its reactions and to give a verdict [28].

In [8, 20], the authors use time automata to model the specification and the test purpose. Test cases are generated by synchronizing the specification with the test purpose and by extracting the paths which contain all the test purpose properties. During the synchronization, a reachability analysis is performed to keep only the reachable transitions. This method needs for each transition a resolution of linear inequalities and also a DFS algorithm to search some clocks constraints. The number of inequalities is proportional with the number of clocks and the transitions they constrained, consequently the resolution is generally costly.

In [28], the specification and the test purpose, modelled with timed automata are translated into region graphs to sample the time domain into polyhedrons. The test cases are generated by synchronizing the specification region graph and the test purpose one. Each region clock of the region graph is accessible from the initial one, so a final test case can be completely executed on implementations. However, the region graph generation is costly and can suffer from state explosion.

In [18], the test tool TGV [15] has been extended to test timed systems. This method can test non deterministic systems and takes into account the quiescence of states. Test purposes and specification are translated into non real time automata (SEA), then the TGV method is adapted and used to generate test cases.

In [2], the authors use specifications and test purposes modelled by TIOA. Then, they search for a feasible path which match the specification and the test purpose with a DFS algorithm.

In [14], test purposes are modelled by Message Sequence Charts (MSC). These ones are converted into TIOA. Then, the specification and test purposes are converted into grid automata. Finally, test cases are generated by using the synchronous product defined in [8].

In this paper, we propose a new test purpose definition to generate test cases which can test the conformance of timed system as well as their robustness by defining Refuse properties, that is test purpose properties which do not belong to the initial specification. So these ones can simulate the execution of different failures, like byzantine or scheduling ones, in order to check if the system can still respond correctly despite these errors. We do not translate timed automata into larger models to apply existing untimed test purpose methods on them [28, 18, 14]. We define a new timed synchronization product on timed automata which also takes into account Refuse properties. We also propose to improve the fault detection by enabling the detection of the missing state and transfer faults. We adapt a state characterization based approach, defined in

[26] for region graphs, to identify each system state with observable action sequences. With this state identification, missing and transfer faults can be detected.

Before describing the test case generation, we present our definition of timed test purposes.

#### 4. Timed test purpose

Test purposes are graphs describing the requirements that engineers wish to test on the system implementation. These requirements can be specification properties which should be satisfied in the implementation during tests. We call them Accept properties. But, test purposes could also be constructed with properties which do not belong to the specification, that we call Refuse properties. These ones can be used to test the system robustness by checking if the system responds correctly despite the execution of unspecified actions.

So, we define that a *Timed Test Purpose* is a TIOA whose the states are either labelled by "accept" or "refuse" to model that transitions are composed of accept or refuse properties. An accept transition of the test purpose must exist in the specification. Its clock zone may be however more restrictive than the specification one.

**Definition 3 (A Timed Test Purpose)** Let  $S = \langle \Sigma_S, S_S, s_S^0, C_S, E_S \rangle$  be a TIOA describing a specification. A timed test purpose  $\mathcal{TP}$  is a TIOA  $\langle \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s_{\mathcal{TP}}^0, C_{\mathcal{TP}}, I_{\mathcal{TP}}, E_{\mathcal{TP}} \rangle$  where:

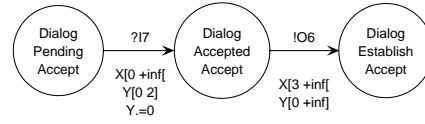
- $C_{\mathcal{TP}} \subseteq C_S$ ,
- $S_{\mathcal{TP}} \subseteq S_S \times \{accept, refuse\}$  is a set of states such that each state  $s' \in S_{\mathcal{TP}}$  is labelled either by:
  - **ACCEPT:** if  $s'$  is the initial state of  $\mathcal{TP}$ , or if  $\forall (s, s', a, \lambda, Z) \in E_{\mathcal{TP}}, \exists (s_1, s_2, a, \lambda_2, Z_2) \in E_S$  such as  $Z \subseteq Z_2, s \in \{(s_1, accept), (s_1, refuse)\}, s' = (s_2, accept)$
  - **REFUSE:** otherwise

**Definition 4 (Accept and Refuse transition)** We call a transition  $(s, s', a, \lambda, Z)$  an *accept transition* iff  $s'$  is labelled by **ACCEPT**. We call it a *REFUSE transition* otherwise.

A timed test purpose example is given in Figure 2. This one checks if after having a dialog accepted (?I7), a dialog can be established (!O6) during a more restrictive clock zone than the specification one.

#### Test purposes for testing the system robustness

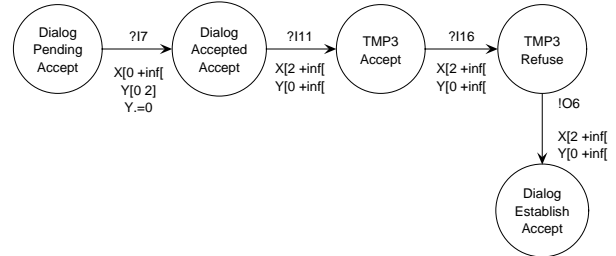
Robustness aims to check the system behaviour under the influence of external errors (byzantine failure, bus error, scheduling problem, ...). Mutations are generally injected into test cases to simulate these errors. Some well-known mutations can be found in [16]:



**Figure 2. A test purpose example for the MAP-DSM protocol**

1. Replacing an input action, to simulation that an unexpected action is received by the system from its external environment.
2. Changing the instant of an input action occurrence to simulate that the good input action is received later than expected
3. Exchanging two input actions to simulate an scheduling problem with external components to the system
4. Adding an unexpected action to simulate that an external component has send an additional action to the system.
5. Removing an action to simulate the lost of a information

Refuse properties are used here to model mutations, which are injected into test purposes and finally into test cases. Refuse properties can be added to test purposes by hands for specifying a precise error, or can be generated by some methods [16, 27]. The test purpose example of Figure 3 contains a refuse property which check that during the establishment of a connection between the MAP server and a service provider (?I11 !O6), the dialog cannot be aborted (?I16=tc\_p\_abort\_ind). The action ?I16 is an unexpected action for the system. The test purpose also checks that the system continue to establish the dialog despite ordering the abort.



**Figure 3. A test purpose with refuse properties**

## 5. Test case generation

### 5.1. Testing hypotheses

Some assumptions are required on the implementation under test and on the specification. The "Implementation Reset" and "Determinism" hypotheses are required to execute the test cases. Indeed, without reset function, the tester cannot execute several test cases on the implementation, and if the implementation is nondeterministic, it may be uncontrollable and thus not testable. The two last hypotheses are required for using a state characterization based approach. These ones ensure and allow to identify each specification state.

**Implementation Reset** After each test, implementations can be reset to the initial state.

**Determinism** The specification must be timed deterministic on the set of alphabet. 1. from any state, we cannot have two outgoing transitions labelled with the same symbol. 2. we cannot have an outgoing transition, labelled with an input symbol and an outgoing transition labelled with an output one, whose the timing constraints are satisfied simultaneously. These properties ensure that a determined implementation path can be covered during the tests.

**Minimality** The specification must be minimal on the state set.

**Completely specified system** The specification must be completely specified on the set of input symbols (each input symbol is enabled from each state).

**Remark 5** *To complete a specification on the set of input symbols, we propose to add a trap state  $s_{\square}$  and to complete each state  $s$  with outgoing transitions  $(s, s_{\square}, ?I, \lambda, G)$  from  $s$  to  $s_{\square}$ . These transitions model the external actions refused by  $\mathcal{A}$  and improve the observability and the controllability of the specification. So, the complete TIOA  $\mathcal{UP}_{\mathcal{A}} = \langle \Sigma_{\mathcal{UP}_{\mathcal{A}}}, S_{\mathcal{UP}_{\mathcal{A}}}, s_{\mathcal{UP}_{\mathcal{A}}}^0, C_{\mathcal{UP}_{\mathcal{A}}}, I_{\mathcal{UP}_{\mathcal{A}}}, E_{\mathcal{UP}_{\mathcal{A}}} \rangle$ , derived from  $\mathcal{A}$  can be obtained with these rules:*

- $\Sigma_{\mathcal{UP}_{\mathcal{A}}} = \Sigma_{\mathcal{A}}, S_{\mathcal{UP}_{\mathcal{A}}} = S_{\mathcal{A}} \cup \{s_{\square}\},$
- $s_{\mathcal{UP}_{\mathcal{A}}}^0 = s_{\mathcal{A}}^0, C_{\mathcal{UP}_{\mathcal{A}}} = C_{\mathcal{A}},$
- $I_{\mathcal{UP}_{\mathcal{A}}} = I_{\mathcal{A}} \cup \{Z' \mid \forall s' \in S_{\mathcal{A}}(s, s', ?I, \lambda, Z) \notin E_{\mathcal{A}}, Z' = \langle [0 + \infty[ \dots [0 + \infty[>\rangle\}$   
 $\cup \{Z' \mid \exists s' \text{ in } S_{\mathcal{A}}(s, s', ?I, \lambda, Z) \in E_{\mathcal{A}}, Z' = \langle [0 + \infty[ \dots [0 + \infty[> / Z \rangle\},$
- $E_{\mathcal{UP}_{\mathcal{A}}} = E_{\mathcal{A}} \cup \{(s, s_{\square}, ?I, \lambda', Z') \mid \forall s' \in S_{\mathcal{A}}(s, s', ?I, \lambda, Z) \notin E_{\mathcal{A}}, \lambda' = \emptyset, Z' = \langle [0 + \infty[ \dots [0 + \infty[>\rangle\}$   
 $\cup \{(s, s_{\square}, ?I, \lambda', Z') \mid \exists s' \text{ in } S_{\mathcal{A}}(s, s', ?I, \lambda, Z) \in E_{\mathcal{A}}, \lambda = \emptyset, Z' = \langle [0 + \infty[ \dots [0 + \infty[> / Z \rangle\}$   
 $\cup \{(s_{\square}, s_{\square}, ?I, \lambda, Z) \mid ?I \in \Sigma_{\mathcal{A}}\}$

Test purposes are often composed of some specification actions, but not of complete specification action sequences [8, 28, 11, 18, 2, 14]. Test purposes may also be inconsistent with the specification, especially when we use refuse properties. So, test purposes based methods generally synchronise the test purpose with the specification to obtain paths which can be completely executed from the initial system path. Moreover, our testing method needs a state characterization based step to detect missing and transfer faults. So, these two steps are first presented below:

### 5.2. Timed Synchronous Product

The timed synchronous product aims to combine a test purpose and a specification to obtain paths which can be executed on the implementation. In comparison with the timed synchronous product that we have defined in [28] for region graph models, this one takes into account Refuse properties and injects them into the final test cases.

Consider two transitions,  $s_1 \xrightarrow{A, Z_S} s_2$  of a specification  $\mathcal{S}$  and  $s'_1 \xrightarrow{A, Z_{\mathcal{TP}}} s'_2$  of a timed test purpose  $\mathcal{TP}$ , labelled with the same symbol "A". By synchronizing them, we generate different clock zones, depending on  $Z_S$  and  $Z_{\mathcal{TP}}$ . The different kinds of synchronized clock zones are:

- **PASS clock zone:** The clock zone  $Z_{pass}$  gathers the values which satisfy the execution of the two transitions, that is the ones which belong to  $Z_S \cap Z_{\mathcal{TP}}$ . If the transition is executed in this clock zone during the test, the test purpose transition is satisfied.
- **INCONCLUSIVE clock zone:** The clock zone  $Z_{inconclusive}$  represents the values which satisfy the execution of the specification transition, but not the execution of the test purpose one. INCONCLUSIVE clock zones ensure that test cases can be executed on implementations, even though the test purpose cannot be satisfied. INCONCLUSIVE clock zones allow to give an inconclusive result, that means some specification properties have been tested instead of the test purpose ones.  $Z_{inconclusive}$  contains values of  $Z_S / Z_{pass}$ .
- **FAIL clock zone:** The FAIL clock zones represent the values which do not satisfy the execution of the specification transition. In this case, if the transition is executed in a FAIL clock zone during the test, the implementation is faulty.

Figure 4 shows an example of synchronized clock zones.

Now, we give the definition of the timed synchronous product between a specification and a test purpose which may contain refuse properties.

**Definition 6 (Timed synchronous product)** *Let  $\mathcal{S} = \langle \Sigma_S, S_S, s_S^0, C_S, I_S, E_S \rangle$  and  $\mathcal{TP} = \langle \Sigma_{\mathcal{TP}}, S_{\mathcal{TP}}, s_{\mathcal{TP}}^0, C_{\mathcal{TP}}, I_{\mathcal{TP}}, E_{\mathcal{TP}} \rangle$  be two TIOA. The Timed Synchronous*

Product between  $\mathcal{S}$  and  $\mathcal{TP}$  is a graph  $\mathcal{SP} = \langle \Sigma_{\mathcal{SP}}, S_{\mathcal{SP}}, s_{\mathcal{SP}}^0, C_{\mathcal{SP}}, E_{\mathcal{SP}} \rangle$  defined by:

- $\Sigma_{\mathcal{SP}} \subseteq \Sigma_{\mathcal{S}} \cup \Sigma_{\mathcal{TP}}$ ,  $S_{\mathcal{SP}} \subseteq S_{\mathcal{S}} \cup S_{\mathcal{TP}}$ ,  $s_{\mathcal{SP}}^0 \subseteq s_{\mathcal{S}}^0$ ,  $C_{\mathcal{SP}} \subseteq C_{\mathcal{S}} \cup C_{\mathcal{TP}}$ ,
- $E_{\mathcal{SP}}$  is the set of transitions  $s_i \xrightarrow{a, \text{PASS}(Z), \text{INCONCLUSIVE}(Z')} s_{i+1}$ , with  $s_i \in S_{\mathcal{SP}}$ ,  $s_{i+1} \in S_{\mathcal{SP}}$ ,  $Z$  a PASS clock zone and  $Z'$  an INCONCLUSIVE one. This set is constructed with the following algorithm.

### Algorithm

**Input:**  $T$ (Test Purpose),  $S$ (Specification)

**Output:**  $SP$ (Synchronous Product)

**BEGIN:**

For each specification path  $PS$  of  $S$ , and For each test purpose path  $TP$  containing in the same order the accept transition symbols of  $TP$

We scan each transition  $tp \xrightarrow{A, Z_{TP}} tp'$  of  $TP$  and each

transition  $s \xrightarrow{B, Z_S} s'$  of  $PS$

**if** the symbol  $A == B$  **then**

//the specification and the test purpose transitions are synchronized

**if**  $\text{Label}(tp') == \text{REFUSE}$  **then** we add  $sp \xrightarrow{A, \text{PASS}(Z_{TP})} sp'$  to  $E_{SP}$   
**else** we add  $sp \xrightarrow{A, \text{PASS}(Z_S \cap Z_{TP}), \text{INCONCLUSIVE}(Z_S/Z_{TP})} sp'$  synchronizing the test purpose and the specification  
**endif**

**if** the symbol  $A \neq B$

//the specification and the test purpose transitions cannot be synchronized

**if**  $\text{Label}(tp') == \text{ACCEPT}$  **then** we add  $sp \xrightarrow{B, \text{PASS}(Z_S)} sp'$  to  $E_{SP}$  to reach a next synchronization  
**else** we scan  $PS$  to find if a synchronization on the symbol  $A$  with  $tp \xrightarrow{A, Z_{TP}} tp'$  is possible later  
**if** it is possible, **then** we add  $sp \xrightarrow{B, \text{PASS}(Z_S)} sp'$  to reach this synchronization.  
**else** we add  $sp \xrightarrow{A, \text{PASS}(Z_{TP})} sp'$   
**endif**  
**endif**

**if** some PTP transitions are not used **then** we add them to  $E_{SP}$   
**endif**

**END**

We illustrate the timed synchronous product with this simple example. Consider the path of Figure 5, derived from the specification of Figure 1. This one is synchronized with the test purpose of Figure 2. The timed synchronous product is expressed in Figure 6.

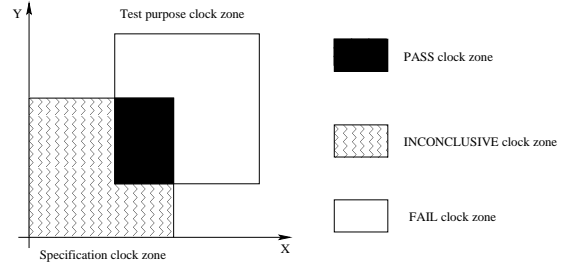


Figure 4. An example of synchronized clock zones with two clocks



Figure 5. A specification path

### 5.3. State characterization set of TIOA

We have defined the state characterization based approach for region graphs in [26]. We have shown that the identification of two states depends on output symbols, which are observed during the system execution, and on the moments of these observations, that is the clock zones, for TIOA. So, to distinguish two TIOA states, we look for a transition sequence which provides either different output symbols, or the same ones with different clock zones or both. A state  $s$  is characterized by a identification set  $W_s$  if this one is composed of transition sequences which distinguish  $s$  from the other states. Finally, the state characterization set  $W$  is the union of the subsets  $W_{s_i}$  which characterize each state  $s_i$ . This is formally described in the following definition.

#### Definition 7 (Timed State Characterization Set $W$ )

Let  $\mathcal{JA} = (\Sigma_{\mathcal{JA}}, S_{\mathcal{JA}}, s_{\mathcal{JA}}^0, I_{\mathcal{JA}}, E_{\mathcal{JA}})$  be a TIOA satisfying the hypotheses of Section 5.1. Two states  $S$  and  $S'$  of  $\mathcal{JA}$  are distinguished by a transition sequence  $\sigma = (t_1, t_2, A_1, \lambda, Z_1) \dots (t_n, t_{n+1}, A_n, \lambda_n, Z_n)$ , denoted  $S D_\sigma S'$  iff

1.  $\forall (t_k, t_{k+1}, A_k, \lambda_k, Z_k) (1 \leq k \leq n)$ , with  $A_k$  an output symbol, we have a path  $S \xrightarrow{A_1, \lambda_1 Z_1} S_2 \dots S_{k-1} \xrightarrow{A_{k-1}, \lambda_{k-1}, Z_{k-1}} S_k \in (E_{\mathcal{JA}})^k$  and  $(S_k, S_{k+1}, A_k, \lambda_k, Z_k) \in E_{\mathcal{JA}}$ ,
2.  $\exists (t_k, t_{k+1}, A_k, \lambda_k, Z_k) (1 \leq k \leq n)$ , with  $A_k$  an out-

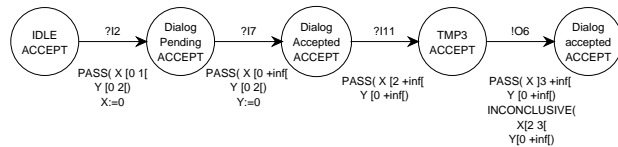


Figure 6. A synchronous product

put symbol, we have a path  $S' \xrightarrow{A_1, \lambda_1, Z_1} S_2 \dots$   
 $S_{k-1} \xrightarrow{A_{k-1}, \lambda_{k-1}, Z_{k-1}} S_k \in (E_{\mathcal{J}\mathcal{A}})^k$  and  
 $(S_k, S_{k+1}, A_k, \lambda_k, Z_k) \notin E_{\mathcal{J}\mathcal{A}}$ .

We denote  $W_S$ , the set of transition sequences allowing to distinguish  $S \in S_{\mathcal{J}\mathcal{A}}$  from the other states of  $S_{\mathcal{J}\mathcal{A}}$ .  $W_S = \{\sigma_i \mid \forall S' \neq S \in S_{\mathcal{J}\mathcal{A}}, S D_{\sigma_i} S'\}$ .

Finally, a Timed Characterization Set of  $\mathcal{J}\mathcal{A}$ , denoted  $W_{\mathcal{J}\mathcal{A}}$  equals to  $\{W_{S_1}, \dots, W_{S_n}\}$ , with  $\{S_1, \dots, S_n\} = S_{\mathcal{J}\mathcal{A}}$ .

A general algorithm of state characterization set generation can be found in [26].

If we take back our synchronous product example of Figure 6, the states can be distinguished with the following state-characterization sets. By applying this set to each pair of state, we always observe different output symbols at different time values, so we can distinguish them.

$$W_{TMP3} = \{(TMP3, Dialog\_establish, \{\}, !O6, < X_{[2+\infty]} Y_{[0+\infty]} >)\}$$

$$W_{Dialog\_accepted} = \{(Dialog\_accepted, TMP3, ?I11, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)(TMP3, Dialog\_establish, !O6, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)\}$$

$$W_{Dialog\_pending} = \{(Dialog\_pending, Dialog\_establish, ?I7, \{\}, < X_{[0+\infty]} Y_{[0+2]} >)(Dialog\_accepted, TMP3, ?I11, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)(TMP3, Dialog\_establish, !O6, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)\}$$

$$W_{IDLE} = \{(IDLE, Dialog\_pending, ?I2, \{\}, < X_{[0+1]} Y_{[0+2]} >)(Dialog\_pending, Dialog\_establish, ?I7, \{\}, < X_{[0+\infty]} Y_{[0+2]} >)(Dialog\_accepted, TMP3, ?I11, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)(TMP3, Dialog\_establish, !O6, \{\}, < X_{[2+\infty]} Y_{[0+\infty]} >)\}$$

$$W_{Dialog\_establish} = \{(Dialog\_establish, TMP2, ?I15, \{\}, < X_{[4+\infty]} Y_{[0+\infty]} >)(TMP2, IDLE, !O9, \{X Y\}, < X_{[4+\infty]} Y_{[4+\infty]} >)\}$$

#### 5.4. The testing method

The testing method is composed of four steps. Steps 1 and 2 synchronize the test purpose with the specification to generate paths, including the test purpose, which can be executed on the implementation. Step 3 applies a state-characterization-based approach on the synchronized paths. Finally, step 4 performs a reachability analysis on the paths obtained from the previous step and modify the clock zones to ensure that the test cases can be completely executed on the implementation.

These test case generation steps are detailed below:

Let  $\mathcal{S}$  be a TIOA, satisfying the previous hypotheses, and  $\mathcal{T}\mathcal{P}$  be a timed test purpose. The test case generation steps are:

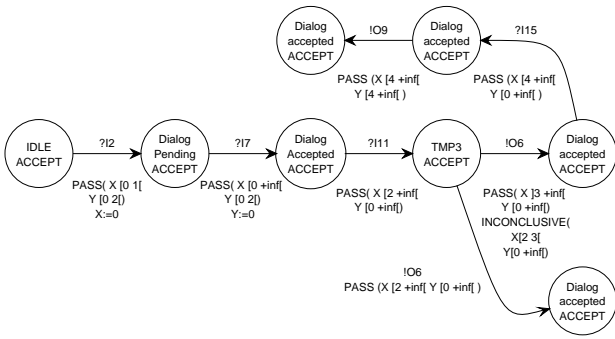
- **STEP1: Specification path search:** We extract the

specification paths which can be synchronized with the test purpose. Instead of synchronizing all the specification with the test purpose, we extract only the needed. So, the transition sequences of  $\mathcal{S}$ , containing in the same order all the Accept transition symbols of the test purpose, are first extracted and named  $TS_1(\mathcal{S}), \dots, TS_n(\mathcal{S})$ . If this set is empty, the process terminates and the following steps cannot be performed. We use a DFS (Depth First Path Search) algorithm to generate these paths. The path extraction is performed depth wise, so only one specification local path is memorized at a time.

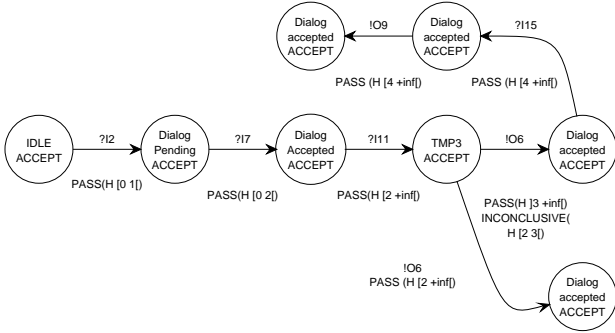
- **STEP2: Timed synchronous product:** Each transition sequence  $TS_i(\mathcal{S})$  is synchronized with  $\mathcal{T}\mathcal{P}$ . This operation generates a graph  $SP$ , including  $\mathcal{T}\mathcal{P}$  and respecting the temporal and behaviour properties of  $\mathcal{S}$ .
- **STEP3: State characterization set generation:** Each state  $S_i$  of  $SP$  is identified with  $W_{S_i}$  (cf Section 5.3). Then, we combine, with  $\Pi$ , the synchronous product and the state characterization set:  $\Pi = SP \otimes W = \{p.(s_i, s_j, a, \lambda, Z).\sigma_j \mid \forall (s_i, s_j, a, \lambda, Z) \in E_{SP}, p$  is a path of  $SP$  from  $s_0$  to  $s_i$ ,  $\sigma_j \in W_{s_j}$  if  $s_j$  is labelled by ACCEPT,  $\sigma_j = \emptyset$  otherwise $\}$ . It's the concatenation of a path  $p$  (finished by its state  $s_i$ ) with the state characterization set of  $s_i$ . If we combine the synchronous product example of Figure 6 and the state-characterization set of Section 5.3, we obtain the paths of Figure 7.
- **STEP4: Search of feasible paths:** Test cases are finally all the feasible paths of  $\Pi$  [2]. The feasibility problem, for a given path  $p = t_1..t_n$ , aims to determine if it exists a possible execution to reach the transition  $t_n$ , and to generate the clock zones over  $p$  for firing  $t_n$ . The approach, described in [2], adds a global clock  $h$  and then performs a reachability analysis from the first and the last transitions of the initial path. The obtained feasible path clock zones can be modelled with the global clock  $h$  or with the clocks used in the initial path. For example, the feasible paths of the  $\Pi$  set, illustrated in Figure 7, are given in Figure 8. These ones are the final test cases.

Test cases are then executed on the implementation from the initial state. Each input symbol is given to the implementation at a clock valuation of its PASS clock zone. If the system is not faulty, output symbols should be observed at clock valuations of PASS clock zones as well. So, by applying a test case transition  $t = (l, l', A, \lambda, PASS(Z), INCONCLUSIVE(Z'))$  on the implementation  $I$ , we can observe some reactions, denoted  $React(t)$ , and we can give a local verdict for the transition.  $React(t) =$

- $PASS_{action}$  iff  $A$  is an output symbol and  $A$  is received from the tester in the PASS clock zone, that is at a clock value  $v \models Z$ ,



**Figure 7. The synchronous product 6 combined with the state characterization sets**



**Figure 8. The test cases**

- $INCONCLUSIVE_{action}$  if  $A$  is an input symbol or if  $A$  is an output symbol and  $A$  is received from the tester in the  $INCONCLUSIVE$  clock zone, that is at a clock value  $v \models Z'$ ,
- $FAIL_{action}$  otherwise.

Finally, by executing the test cases and observing the implementation reactions, we can conclude on the success or on the failure of the test:

**Definition 8 (Verdict assignment)** Let  $I$  be a system under test and  $T = (l_1, l'_1, A_1, \lambda_1, PASS(Z_1), INCONCLUSIVE(Z'_1)) \dots (l_n, l'_n, A_n, \lambda_n, PASS(Z_n), INCONCLUSIVE(Z'_n))$  be a test case. The verdict assignment  $V(I, T)$ , obtained by applying  $T$  on  $I$ , is given by:

- Pass iff  $\forall t = (l_i, l'_i, A_i, \lambda_i, PASS(Z_i), INCONCLUSIVE(Z'_i)) \in T$ , with  $A_i$  an output symbol,  $React(t) = PASS_{action}$ ,
- Inconclusive iff  $\exists t = (l_i, l'_i, A_i, \lambda_i, PASS(Z_i), INCONCLUSIVE(Z'_i)) \in T$ , with  $A_i$  an output symbol,  $React(t) = INCONCLUSIVE_{action}$  and iff  $\forall t' = (l_j, l'_j, A_j, \lambda_j, PASS(Z_j), INCONCLUSIVE(Z'_j)) \in T$ , with  $A_j$  an output symbol,  $React(t) \neq FAIL_{action}$ ,

- Fail otherwise

**Method complexity:** If  $N$  is the number of state and  $K$  the number of transitions of the specification, the test case generation complexity of our method is proportional to  $C^2 * N + N * K + N + K$ . For the first step, we use a DFS algorithm whose the complexity is proportional to  $N + K$ . The timed synchronous product complexity depends on the length of the paths to combine. In the worst case, this length equals to  $N$  and there is at most  $K$  specification paths. So, the complexity of the synchronous product is proportional to  $N * K$ . The step 3 complexity is proportional to  $N^2 K$  [25]. In step 4, the search of feasible paths is proportional to  $C * C * N$  [2], with  $C$  the number of clocks.

## 6. Fault coverage of the proposed method

In this Section, we introduce the fault coverage of our testing method. As a test purpose doesn't test the whole implementation of a system, the fault coverage is analyzed on a implementation part, called  $I_{covered}$ . Furthermore, to generate test cases, we use some specification paths, needed for the timed synchronous product. Let  $S_{covered}$  be the set of these paths.  $I_{covered}$  corresponds to implementation part tested by the test cases obtained from  $S_{covered}$ .

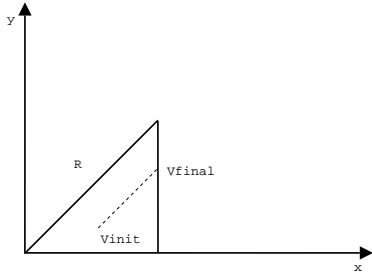
- **Output fault detection:** Output faults can be easily detected by firing all of the specification transitions. According to our definition of the timed synchronous product, each path of  $S_{covered}$  exists completely in at least one test case. Moreover, we suppose that the system is deterministic. So, each transition of  $I_{covered}$  is tested during the testing process.
- **Missing state fault detection:** Extra(missing) state faults are detected by checking if an extra or missing state exists in the implementation. As our method identifies each state, it can detect missing states on  $I_{covered}$ . Each state  $s_i$  of  $S_{covered}$  is identified in the implementation and tested by test cases of the form  $s_0 \xrightarrow{p} s_i.W_{s_i}$ , with  $p$  a path from the initial state  $s_0$  to  $s_i$  and  $W_{s_i}$  the subset allowing to identify  $s_i$ . Consequently, if a state is missing in  $I_{covered}$  at least one test case cannot be completely executed.
- **Transfer fault detection:** Transfer faults can be easily detected by identifying the states of the implementation. So, any state-identification based technique, and particularly our method, detects transfer faults on  $I_{covered}$ . Each transition  $t = (S_i, S_j, a, \lambda, G)$  of  $S_{covered}$  is tested by a test case of the form  $S_0 \xrightarrow{p} S_i \xrightarrow{a, \lambda, PASS(Z)} S_j.W_{S_j}$ , with  $p$  a path from the initial state  $S_0$  to  $S_i$  and  $W_{S_j}$  the subset allowing to characterize  $S_j$ . Consequently, the arrival state of the transition  $t$  in the implementation is tested and identified. So, transfer faults are detected.



- *Time constraint widening fault detection:* Time constraint widening faults are detected if at least an output symbol is not received by the tester in the time delay given by the specification. According to our definition of the timed synchronous product, each transition of  $S_{covered}$  is visited during the testing process by at least one test case. Consequently, a test case transition  $(s, s', a, \lambda, PASS(Z), INCONCLUSIVE(Z'))$  labelled by an output symbol, is tested by the tester which waits its receipt during the PASS clock zone  $Z$ . If no output symbol is received, a time constraint widening fault is detected on  $I$ .

For input ones, the method checks them only at clock valuations which belong to time delays given by the specification. So, time constraint widening faults can be detected with output symbols and not with input ones on  $I_{covered}$ .

- *Time constraint restriction fault detection:* In practice, it is unfeasible to detect all of the time constraint restriction faults. Consider a test case transition  $(s, s', ?a, PASS(Z), INCONCLUSIVE(Z'))$ , to detect the faults, the tester should send to the implementation the input symbol "?a" at all of the bounds of  $Z$  which are for each clock  $x_i$  the time values  $a_i$  and  $b_i$  such that  $Z(x_i) = [a_i, b_i]$ . Since the clocks are uncontrollable, these bounds are not necessary reached by the clocks.



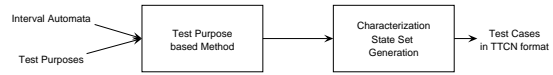
**Figure 9. Reaching all the clock region bounds: a difficult issue**

Consider the clock zone of the Figure 9.  $v_{init}$  represents the first clock valuation reached by the clocks in  $Z$  during an execution.  $v_{init}$  is not a bound of  $Z$ . So, if the implementation has a time constraint restriction fault between the bound of  $Z$  and  $v_{init}$ , the fault cannot be detected.

Consequently, we can detect such a fault if this one occurs during the execution. In this case, consider a test case  $p.(s_i, s_j, ?a, \lambda, PASS(Z), INCONCLUSIVE(Z')).p'.W_{s_k}$ . Let  $s_i$  be the implementation state reached by  $p$  and  $s_k$  the one reached by  $p.(s_i, s_j, a, \lambda, PASS(Z), INCONCLUSIVE(Z')).p'$ . If the implementation produces this fault,  $s_i$  rejects the input sym-

bol "a" in delays given by  $Z$ . Therefore, the implementation stays in its current state  $s_i$ . Here, either the implementation rejects  $p'.W_{s_k}$  too, or accepts it. If  $p'.W_{s_k}$  is rejected, the implementation enters in a deadlock. Output symbols of  $p'.W_{s_k}$  are not observed so the time constraint restriction fault is detected. If  $p'.W_{s_k}$  is accepted by the implementation, according to the hypotheses (Section 5.1),  $S$  is minimal and deterministic therefore there exists a unique path from  $s_i$  to  $s_k$ , covered with  $(s_i, s_j, a, \lambda, PASS(Z), INCONCLUSIVE(Z')) .p'$  from  $s_i$ . Thus, a state  $s_l$  different from  $s_k$  is reached with  $p'$  from  $s_i$ . Since the state reached by  $p'$  is identified with  $W_{s_k}$ , if this one is different from  $s_k$  an error is produced. So, in both cases, time constraint restriction faults are detected.

## 7. Prototype tool functionality



**Figure 10. The test tool TCG**

We have implemented the previous methodology in an academic prototype tool, called TCG (Timed Test Cases Generation). The description of its architecture is illustrated in Figure 10. The prototype tool takes specifications and test purpose modelled with TIOA. It is composed of two parts: the first one produces the timed synchronous product between the test purposes and some specifications paths. The second one produces the  $W$  set generation. The paths, obtained from the synchronous product and the state characterization set, are then concatenated to finally produce the test cases. These ones are given in TTCN or in Poscript format.

This tool has been written with the language C, excepted the second part which has been written in OpenMP to parallelize the  $W$  set generation. Clock zones modelling and operators on clock zone have been implemented with the Polylib library [31]. This one has a graphical interface which allows the user to load TIOA and test purposes. The amount of memory used depends on the specification. With the MAP-DSM specification (Figure 1), this one does not exceed 10 Mb.

## 8. Conclusion

In this paper, we have proposed a test purpose based approach which can test both the conformance and the robustness of implementations, by using test purposes composed of Accept and Refuse properties. This method uses a synchronous product between the specification and the test purpose to generate on the fly test cases and a state characterization based approach to improve the fault coverage by enabling the detection of transfer faults and miss-

ing state faults. The complexity is polynomial so we believe that this one can be used in practice.

Our approach could be extended for testing others aspects of timed systems like interoperability. The quiescence of critical states [4] could be tested with specific test purposes too, by checking if these states do not produce an output response without giving an input symbol. Moreover, this property could help to distinguish pair of states by considering the notion of quiescence as a special sort of output observation. As a consequence, the length of the state characterization set and so the test costs could be reduced.

## 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 *WRITES, satellite workshop of FME symposium*, pages 5–16, 2003.
- [3] C. Besse, A. Cavalli, M. Kim, and F. Zaidi. Two methods for interoperability tests generation. an application to the tcp/ip protocol. 2004.
- [4] 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.
- [5] R. Cardel-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proc. of the 5th. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of LNCS, pages 251–261. SpringerVerlag, 1998.
- [6] N. CARRERE. Dsm specification in lotos and test cases generation. *INT (French Telecommunication National Institute)*, 2001.
- [7] 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.
- [8] R. Castanet, P. Laurençot, and O. Kone. On the Fly Test Generation for Real Time Protocols. In *International Conference on Computer Communications and Networks, Louisiane U.S.A.*, 1998.
- [9] T. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, SE-4(3):178–187, 1978.
- [10] D. Clarke and I. Lee. Automatic Generation of Tests for Timing Constraints from Requirement. In *International Workshop on Object-Oriented Real-Time Dependable Systems, California*. IEEE Computer Society Press, 1997.
- [11] A. En-Nouaary and R. Dssouli. A guided method for testing timed input output automata. In *15th IFIP International Conference, TestCom 2003, Sophia Antipolis, France*, pages 211–225, May 2003.
- [12] A. En-Nouaary, R. Dssouli, and F. Khendek. Timed wp-method: Testing real-time systems. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, Nov. 2002.
- [13] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.
- [14] A. En-Nouaary and G. Liu. Timed test cases generation based on msc-2000 test purposes. In *Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL'04), part of the 15th IEEE International Symposium on Software Reliability Engineering (IS-SRE), Rennes, France*, Nov. 2004.
- [15] J. C. Fernandez, C. Jard, T. Jron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96. LNCS 1102 Springer Verlag*, 1996.
- [16] H. Fouchal, A. Rollet, and A. Tarhini. Robustness of composed timed systems. In *31st Annual Conference on Current Trends in Theory and Practice of Informatics, Liptovsky Jan, Slovak Republic, Europe, volume 3381 of LNCS*, pages 155–164, Jan. 2005.
- [17] 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.
- [18] 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.
- [19] A. Khoumsi and L. Ouedraogo. A new method for transforming timed automata. In *Brasilian Symposium on Formal Methods (SBMF), Recife, Brazil*, Nov. 2004.
- [20] O. Kone. A local approach to the testing of real time systems. *The computer journal*, 44:435–447, 2001.
- [21] O. Kone and R. castanet. Test generation for interworking sytems. *Computer communications, Elsevier Science*, 23:642–652, 1999.
- [22] B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In *TACAS01, vol. 2031 of LNCS, Genova, Italy*, pages 343–357, 2001.
- [23] E. Petitjean and H. Fouchal. From Timed Automata to Testable Untimeed Automata. In *24th IFAC/IFIP International Workshop on Real-Time Programming, Schloss Dagstuhl, Germany*, 1999.
- [24] A. Petrenko, N. Yevtushenko, and G. v. Bochmann. Testing Deterministic Implementations from Non-deterministic FSM Specifications. In *Proceedings of the 8th International Workshop on Test of Communicating Systems IWTCS'96 (Darmstadt, Germany)*, Amsterdam, september 1996. North-Holland.
- [25] S. Salva and P. Laurenot. Gnration de tests temporiss oriente caractrisation d'tats. In *Colloque Francophone de l'ingénierie des Protocoles, CFIP*, Oct. 2003.
- [26] S. Salva and P. Laurenot. A testing tool using the state characterization approach for timed systems. In *WRITES, satellite workshop of FME symposium*, 2003.
- [27] S. Salva and P. Laurenot. Gnration automatique dobjectifs de test pour systmes temporiss. In *Colloque Francophone de l'ingénierie des Protocoles, CFIP, Bordeaux*, 2005.
- [28] 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.
- [29] J. Springintveld, F. Vaandrager, and P. R. D'Argenio. Testing Timed Automata. *TCS*, 254(254):225–257, 2001.
- [30] J. Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.
- [31] D. K. Wilde. A library for doing polyhedral operations. Technical report, IRISA. <http://icps.u-strasbg.fr/PolyLib/>.