Symbolic and on the fly testing with real-time observers

Rachid Bouaziz and Ousmane Koné

University of Toulouse - CNRS IRIT
31062 Toulouse Cedex - France
{bouaziz,kone}@irit.fr

Abstract. Analyzing real-time specifications involves new difficulties in the test generation process. In addition to usual combinatory explosion, issues like tests executability and controllability become more problematic. To deal with such issues, the new method proposed in this paper combines both on the fly computation (not on line) and optimized symbolic analysis with the underlying concept of real-time observers. A symbolic forward analysis is used for test executability and a backward analysis is performed to refine the tests controllability in view of avoiding inconclusive verdicts. The featured observers and the backward computation are the basis for a more targeted test selection. To illustrate the method, the work example is a process control communication system. Finally, we introduce Real-time Ethernet and the related tests produced with our method.

1 Introduction

Real time applications are those are applications for which real-time (i.e. *physical time*) is the main execution constraint. They are concerned with business lines such as aeronautics, aerospace, automotive or telecommunications. These applications often manage the systems and even the people security and therefore must be designed with very rigorous techniques. For testing real-time systems, one must carefully define when to submit an input to the Implementation Under Test (IUT) and when to observe an output. A major issue of automatic test selection from a formal specification is the combinatory explosion of the analyzed behaviour. In the presence of real time constraints, the test selection problem is worsened as a huge number of time instances are relevant to test. Moreover controllability and test executability becomes non trivial [10]. To deal with such problems, the method proposed in this paper combines different strategies to improve the efficiency of automatic test selection.

Symbolic techniques were firstly introduced in the field of formal methods, as they produced reachability graph of reduced size [13, 4, 17, 14]. These techniques have recently been used for the purpose of "real-time testing", with some idea of on line testing [9]. But in these testing approaches, the meaning of "*on line*" is that the test sequences are not computed in advance, before test execution. There is no test suite available before, but the tests are calculated "on line", event after event, during test execution. Since these approaches are based on the reuse of model-checking tools, the resulting tests can not be directly reproduced as they often correspond to diagnostic traces or run time executions. This also disables a reasonable *prediction* of test suite coverage. Meanwhile, an explicit test suite selection strategy can be performed when using test purposes to characterize the expected tests [8, 6, 7] and, above all, "to guide" the test selection process, "*off line*", before test execution. This is a different and now classical basis for test selection against combinatory explosion. Indeed, the "*on the fly traversal*" consists of searching the test pattern from the system model, while avoiding the whole exploration of the model. For timed systems, the principles of on the fly test selection from test purposes were presented a few years ago [6, 15]. The new features and improvements here are the combination of on-the-fly traversal and modern symbolic abstractions together with enhanced real-time observers, for

improving the efficiency of test case selection. Here, real-time observers are used to target the expected tests. Then a symbolic forward analysis is used for test executability and a backward analysis is performed to refine the tests controllability in view of avoiding inconclusive verdicts. For illustration, an example of process control system is presented and finally the case study started with Real-time Ethernet protocol is introduced. The current work is being implemented in a prototype tool named OOTEST, and which architecture will be introduced. The papers is organized as follows. Section 2 presents the formalism and notations related timed automata. The kernel of the test methodology is presented in section 3. Section 4 and 6 are complementary comments and concluding remarks.

2 Timed Input Output Automata

Let $\mathbf{R}^+$ be a set of non-negative real numbers and let X be a set of non negative real-time valued variables called *clocks*. The set of *Guards* $G(X)$ is defined by the grammar $g := x \sim c \mid x - y \sim c \mid g \wedge g \mid \mathit{true}$, where x and $y \in X$, $c \in \mathbf{R}^+$ and $\sim \in \{<, \leq, >, \geq\}$. We denote by $\mathbf{T}$ the finite sequences of elements in $\mathbf{R}^+$ called *time domain*, and by Σ the finite set of *actions*. A *time sequence* over $\mathbf{T}$ is a finite no decreasing sequence $\rho = t_1, t_2, \ldots, t_n$ and a *timed word* $w = (a_1, t_1,), (a_2, t_2) \ldots (a_n, t_n)$ is an element of $(\Sigma \times \mathbf{R}^+)^*$. A *clock valuation* is a function $\nu : X \rightarrow \mathbf{R}^+$, if $\delta \in \mathbf{T}$ the $\nu + \delta$ denotes the valuation such that for each clock $x \in X, (\nu + \delta)(x) = \nu(x) + \delta$. If $r \subseteq X$ then $\nu[r := 0]$ denotes the valuation such that for each clock $x \in X \setminus r, \nu[r := 0](x) = \nu(x)$ and for each clock $x \in r$, $\nu[r := 0](x) = 0$. $[r := \infty]\nu$ denotes the valuation such that for each clock $x \in X \setminus r, [r := \infty]\nu(x) = \nu(x)$ and for each clock $x \in r$, $[r := \infty]\nu(x) = \infty$.

Definition. A *Timed Automaton* (TA) is a tuple $A = (L, L_0, L_f, X, \Sigma, E, I)$, where L is a finite set of locations, $L_0(L_f) \subset L$ is a subset of initial (final) locations, X is a finite set of clocks. Σ is a finite set of events. If the set of events (actions) is partitioned in two disjoint subsets $\Sigma^?$ and $\Sigma^!$, where $\Sigma^?$ is the set of *input* actions and $\Sigma^!$ is the set of *output* actions, the TA A is called *Timed Input Output Automaton* (TIOA). $E \subseteq L \times G(X) \times \Sigma \times R(X) \times L$ is a set of edges. We write $l \xrightarrow{a,g,r} l'$ iff $(l, a, g, r, l') \in E$, where $l, l' \in L$ are the source and destination locations, $g \in G(X)$ is a conjunction of constraints in $G(X)$, $a \in \Sigma$ is the action (or event), $r \in R(X)$ is the set of clocks to be *reset*. $I : l \rightarrow G(X)$ assigns invariants to locations.

We use the notation such as $l \xrightarrow{a}$ (resp. $l \not\xrightarrow{a}$) to denote that there exists l' such that $l \xrightarrow{a} l'$ (resp. there is no such l'). This notation naturally extends to time sequences. We write $l \xrightarrow{(a,t)}$ if from location l, a can be executed at time t. A TIOA A is said to be *complete*, if it accepts every action in Σ at every time. It is said to be input-complete if it accepts every input action in $\Sigma_!$ at every time. A TIOA is called deterministic if $\forall l, l', l'' \in L \cdot \forall a \in \Sigma \cdot \forall t \in \mathbf{R}^+ \cdot l \xrightarrow{(a,t)} l' \wedge l \xrightarrow{(a,t)} l'' \Rightarrow l' = l''$. It is called non-blocking if $\forall l \in L, \forall a \in \Sigma^! \cup \mathbf{R}^+ \cdot l \xrightarrow{a}$.

A *Path* P in TA A is a finite sequence of consecutive transitions $l_0 \xrightarrow{g_1,a_1,r_1} l_1 \xrightarrow{g_2,a_2,r_2} l_2 \ldots$. It is said to be *Accepting* if it starts in an initial location ($l_0 \in L_0$) and ends in a final location ($l_f \in L_f$). A *Run* of the automaton along the path P is a sequence of the form $(l_0, \nu_0) \xrightarrow[t_1]{g_1,a_1,r_1} (l_1, \nu_1) \xrightarrow[t_2]{g_2,a_2,r_2} (l_2, \nu_2) \ldots$, where $\sigma = t_1, t_2 \ldots$ is a time sequence in $\mathbf{T}$, and $\nu_i (i = 1, 2 \ldots)$ is a clock valuation such that: $\nu_0(x) = 0, \forall x \in X$; $\nu_{i-1} + (t_i - t_{i-1}) \models g_i$; $\nu_i = [r_i := 0](\nu_{i-1} + (t_i - t_{i-1}))$. The label of the run is the timed word $\omega = (a_1, t_1), (a_2, t_2), \ldots (a_n, tn)$. The set of all timed words in A is denoted $\mathsf{Traces}(A)$. If the path P is accepting the timed word ω then it is said to be accepted by the TA A.

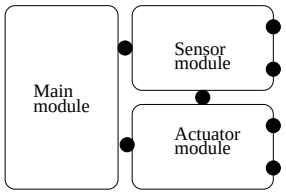

Fig. 1. The process control system - Communicating sensor and actuator

Example. The following presents a process control communication system in an energy production center. The system operates under real-time constraints with a fault recovery mechanism. The overall system (figure 1) is composed of a `Sensor` module, an `Actuator` module and a `Main` module. The different modules communicate through synchronization ports. In the sequel, we focus on the `Sensor` module only, which model in represented with figure 2. The sensor tries to detect the temperature signal from the environment (`?t`). If no signal was received within 4 time units, the famine signal (`!f`) is sent to the main module. Otherwise the system proceeds with checking the pressure (`?p`) and then sends the (`!v`) signal to the actuator, for subsequent operation. In case of failure during this walk, an error procedure (`!e`) is started for an new tentative. Finally the sensor resets (`!r`) and returns to initial state, etc.

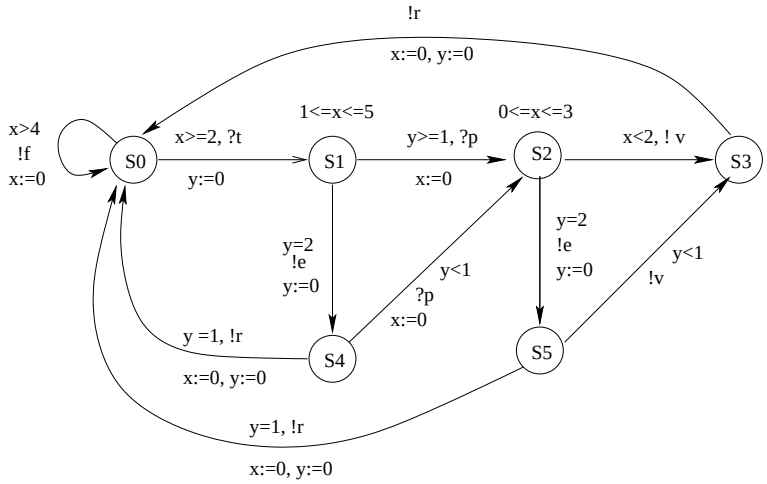

Fig. 2. A process control communication system

The automaton of figure 1 contains six locations $L = \{S_0, S_1, S_2, S_3, S4, S_5\}$, where the set of initial locations is $L_0 = \{S_0\}$, the set of final locations is $L_f = \{S_0\}$. Two clocks x and y and ten transitions are used. An *accepted path* of the automaton can be represented by :

$$S_0 \xrightarrow{(x\geq 2),?t,(y)} S_1 \xrightarrow{(y\geq 1),?p,(x)} S_2 \xrightarrow{(y=2),!e,(y)} S_5 \xrightarrow{(y=1),!r,(x,y)} S_0.$$

An *accepting run* of this path is: $(S_0,(3,\infty)) \xrightarrow[3]{(x\geq 2),?t,(y)} (S_1,(4.5,1.5)) \xrightarrow[4.5]{(y\geq 1),?p,(x)} \ldots$
$\ldots\ (S_2,(0.5,2)) \xrightarrow[5]{(y=2),!e,(y)} (S_5,(1.5,1)) \xrightarrow[6]{(y=1),!r,(x,y)} S_0(0,0).$

where $(4.5, 1.5)$ is the valuation ν associated to the clock x and y such that $\nu(x) = 4.5$ and $\nu(y) = 1.5$.
An *accepted timed word* of this run is : $(?t, 3), (?p, 4.5), (!e, 5), (!r, 6)$.

3 Test design

3.1 From symbolic abstraction to executability and controllability

Since actual IUT runs correspond to test execution, the executability problem turns to standard reachability analysis that handles all the possible executions of the system. For timed automata, symbolic reachability is now a well established technique addressing the explosion problem of the reachability graph. Rather than enumerating all the states (e.g. like regions) the states are characterized and gathered by a Boolean formula. Different approaches exist for representing the nodes of the reachability graph (Zones represented by Difference Bound Matrix, Clock Difference Diagram, State classes etc). Currently, we have started to implement them as options in our `OOTEST` tool, in view of a further performance comparison and analysis. In the sequel, we illustrate the method with DBM (Difference Bound Matrix) which have an intuitive representation. For instance, the constraint $0 \leq x \leq 4 \wedge y = 0$ defines a zone that can be represented using a DBM. The DBM is $(n+1)\times(n+1)$ matrix where n is the number of clocks. Each element $D_{i,j}$ of a matrix D is an upper bound of the difference of two clocks x_i and x_j ($x_i - x_j \prec D_{i,j}$ where $\prec \in \{\leq, <\}$). Given a TA A, the reachability graph (or *Reachable Automaton*) RA associated with A is a graph where the nodes (the *symbolic states*) have the form (l, z), where l is a location of A and z is a DBM. The construction of RA is performed with a standard reachability algorithm, augmented with the computation of *successors* (future after time progress) of zones. This construction is called *Forward Reachability* as it computes symbolic states (l, z) which are reachable from initial locations of A. The following procedure computes the successor of a symbolic state (l, z).

Input: $(l, z) \xrightarrow{g,a,r} (l', I(l'))$ such that $z \subseteq I(l)$.
Output: (l', z'): Reachable target state.

1. Compute $z_1 = \{\nu + \delta \cdot \nu \in z \text{ and } \delta \in \mathbf{R}^+\}$; The clock valuations that can be reached from z by delaying.
2. Compute $z_2 = z_1 \cap I(l)$; The clock valuations where the constraints of both z_1 and the invariant of l are satisfied.
3. Compute $z_3 = z_2 \cap g$; The clock valuations where the constraints of both z_2 and the guard condition are satisfied.
4. Compute $z_4 = z_3[r := 0] = \{\nu[r := 0] \cdot \nu \in z_3\}$; The clock valuations obtained by resetting clocks in r in the zone z_3.
5. Compute $z' = z_4 \cap I(l')$; The initial reachable clock valuations in l'.

The *Forward Reachability* will guarantee the tests *executability*. But as explained in the following section, we are also interested in improving the *controllability* of the tests execution towards some specific (Pass) states. So we also need to compute the sufficient and necessary constraints that can lead from an initial configuration to such particular states. For that, the *backward* propagation of constraints from the target states to the initial states should be computed. Such computation is called *Backward Reachability* as it calculates symbolic states (l, z) with the *actual* necessary constraints for reaching a given state (l', z'). The following procedure computes the predecessor of a symbolic state (l', z').

Input: $(l', z') \xleftarrow{g,a,r} (l, z)$.
Output (l, z''): Actual predecessor state.

1. Compute $z'_1 = \{\nu \cdot \exists \delta \in \mathbf{R}^+ \cdot \nu + \delta \in z'\}$; The set of clock valuations that can reach z' by delaying.
2. Compute $z'_2 = z'_1[r := 0]$; The clock valuations just after performing the transition between l and l' and thus after resetting clocks in r.
3. Compute $z'_3 = [r := \infty]z'_2$; The clock valuations just before executing the transition; All clocks in r are allowed to have any values.
4. Compute $z'_4 = z'_3 \cap g$; The set of clock valuations where the constraints of both z'_2 and the condition that allowed to fire the transition are satisfied.
5. Compute $z'' = z'_4 \cap z$; Concrete predecessor constraints in l obtained by the intersection between the clock valuations in z'_4 and the invariant of l (included in z).

A full example will be presented later with figure 8.

3.2 Real-time observers

The current developments are aimed to be used with the observers currently designed in the framework of the Open source TOPCASED project[11, 12]. These observers have been initially defined for capturing several extra features like fault tolerance, diagnosis, etc, with an underlying structure of extended timed automata. In this paper, for the time being, we consider the aspects related to functional conformance requirements. In addition to standard timing requirements, the observers define not only the expected behaviour expressed in terms of *test purpose*, but also some aspects of the behaviour that we are not currently interested in. The latter are excluded during the search of the target test sequence and therefore enable restricting the behaviour to be computed. It is up to the test engineer to define a given test observer. We assume that he has the required knowledge for deciding which features are to be included or excluded. In the following, we present an approach to model and construct a real-time observer.

Modelling the observer. The observer is used for checking the interactions within the test environment. Obviously, its design will be related to the one of the reference specification considered.

Let $A_S = (L_S, L_{0S}, L_{fS}, X_S, \Sigma^?_S, \Sigma^!_S, E_S, I_S)$ be a TIOA model of the Specification. An observer of specification is a TIOA $A_O = (L_O, L_{0O}, L_{fO}, X_O, \Sigma^?_O, \Sigma^!_O, E_O, I_O)$ with the following characteristics:

- The set of locations L_O is equipped with two new disjoint locations $Accept \in L_{fO}$ and $Reject \in L_{fO}$. If the location *Accept* is reached, we conclude that the functionality modeled by the observer has been satisfied. An efficient set of test cases can be extracted from all paths reaching and *no traversing* a location *Accept*. All path reaching or traversing a *Reject* location should be ignored (cf below).
- $\Sigma_O = \Sigma_S$ such that $\Sigma^!_O = \Sigma^!_S$ *and* $\Sigma^?_O = \Sigma^?_S$.
- A_O is *non blocking, deterministic*, and *complete* TIOA.

In this paper, we use passive observers. They basically model a test purpose which characterizes some expected functionality involving a *Pass verdict*, but they also capture undesired (even if correct) behaviour that lead to inconclusive verdicts. An example is presented in figure 3. The construction of an observer can be done by completing the test purpose basic functionality. From every location l in the test purpose, outgoing transitions are added according to the following cases :

1. For every $a \in \Sigma_S$ such that $l \xrightarrow{g,\ a}$ is an outgoing transition in the test purpose
   - If we'd like to test a under (g) only, then a transition $l \xrightarrow{\neg g,\ a} Reject$ is added to (E_O). ($\neg g$ is the negation of g)

- If we are interested in the occurrence of a under another constraint (g'), then
  (a) A loop $l \xrightarrow{a,g'} l$ is added;
  (b) A transition $l \xrightarrow{a,\neg g \wedge \neg g'} Reject$ is added.

2. If the action $a \in \Sigma_S$ is not specified in the outgoing transitions of l
   - If we are interested in the occurrence of a under the constraints (g) then a loop in $l \xrightarrow{a,g} l$ and a transition $l \xrightarrow{\neg g,\ a} Reject$ are added to (E_O).
   - If we do not take care of the occurrence of a, a loop $l \xrightarrow{a,true} l$ is added to the (E_O).

Example. Let us consider the process control example of figure 2. A main characteristic of that real-time system is that only "fresh signals" must be transmitted. We may be interested in the following properties (test purposes).

1. The system sends signal ($!v$) to the actuator in less than 1.3 time units after the reception of the temperature signal ($?t$).
2. If an error occurs, it can be corrected within 1 time unit.
   etc ...

The figure 3 (right part) represents the observer modelled from the first functionality described above. In this observer all traces starting with $(!f, n \geq 4)$ are rejected. Such traces increase the behaviour to be analyzed while they are useless for the expected tests. On the other hand, $(!f, n < 4)$ should be preserved as it can detect conformance violation.

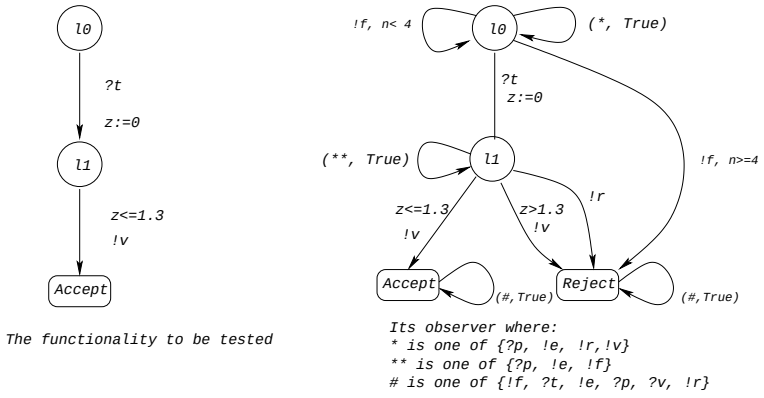

Fig. 3. An example of real-time observer

3.3 On the fly traversal

The main characteristic of our test selection approach is *to perform an on-the-fly traversal of the Reachable (symbolic) Automaton of specification* A_S until the current observer A_O is exhausted (Accept state reached). As usual, this can be formulated on the basis of a synchronous product of A_S and A_O. This is done in the following, in terms of full synchronization as the observer is assumed to be complete, by its construction.

Let $A_S = (L_S, L_{0S}, L_{fS}, X_S, \Sigma_S^?, \Sigma_S^!, E_S, I_S)$ and $A_O = (L_O, L_{0O}, L_{fO}, X_O, \Sigma_O^?, \Sigma_O^!, E_O, I_O)$. The synchronous product of A_S and A_O is a TIOA $A_{SP} = (L_{SP}, L_{0SP}, L_{fSP}, X_{SP}, \Sigma_{SP}, E_{SP}, I_{SP})$ where:

- $L_{SP} = L_S \times L_O$ is the set of states equipped by two distinguished sets of Accepting and Rejecting states which are defined as follows :
  - $Accept_{SP} = L_S \times \{Accept\} = L_{fSP}$;
  - $Reject_{SP} = L_S \times \{Reject\}$.
    An accepting state is an element from $Accept_{SP}$, and it has the form $(l_S, Accept)$, and a rejecting state is an element of $Reject_{SP}$ and it has the form $(l_S, Reject)$.
- $L_{0SP} = L_{0S} \times L_{0O}$ is the set of initial locations;
- $\Sigma_{SP} = \Sigma_{SP}^? \cup \Sigma_{SP}^!$ such that : $\Sigma_{SP}^? = \Sigma_S^? = \Sigma_O^?$, and $\Sigma_{SP}^! = \Sigma_S^! = \Sigma_O^!$ is the set of actions;
- $X_{SP} = X_S \cup X_O$ is the set of clocks ;
- E_{SP} is the set of transitions defined by the following minimal rule :

$$\frac{(l_S \xrightarrow{a, g_S, r_S} l'_S) \wedge (l_O \xrightarrow{a, g_O, r_O} l'_O)}{(l_S, l_O) \xrightarrow{a, g_S \wedge g_O, r_S \cup r_O} (l'_S, l'_O)}$$

- I_{SP} is such that $I((l_S, l_O)) = I(l_S) \wedge I(l_O)$ is the invariants to locations in A_{SP}.

The synchronous product is illustrated with the specification of figure 2 and the observer of figure 3. For space and readability reasons, the picture has been decomposed into figure 4 and figure 5, and is partially drawn.

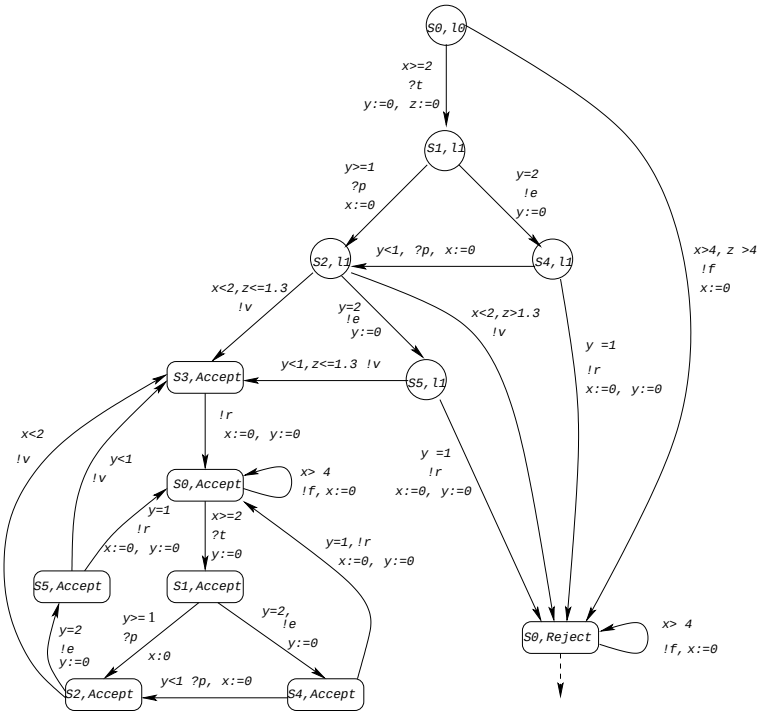

Fig. 4. *Synchronous product of Specification and Observer*

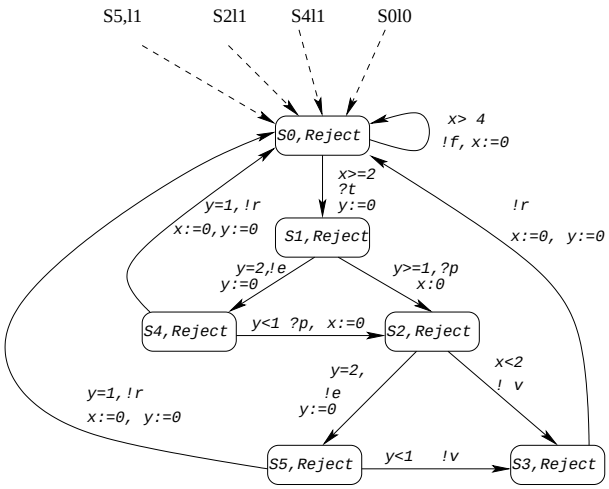

Fig. 5. *Synchronous product of Specification and Observer (cont.)*

Optimizing the on the fly traversal. The intuition behind the use of the *Accept* and *Reject* states is to eliminate some behaviours and therefore improve (or optimize) the computation during the search of target test case. The search must be stopped if *Accept* state is reached while transitions towards *Reject* state are not considered. This improves the reachability of the resulting test graph since much of the behaviour is eliminated. The test graph (the tester structure) can be defined by the following operations on the synchronous product.

1. Each element of Accepting states in synchronous product TIOA must be transformed in *PASS* state.
2. Each element of Rejecting states in synchronous product TIOA must be transformed in *INCONCLUSIVE* state.
3. Each transition which initial location is a *PASS* must be removed.
4. Each transition which initial location is an *INCONCLUSIVE* state must be removed.

Starting from specification of figure 2 and observer of figure 3, and their product in figure 4 and figure 5, the operations below produce the test graph of figure 6 which is transformed into the final tester structure in figure 7. In the latter, the inputs of specification are transformed into outputs and vice versa, and a *Fail* state is added to capture all the unexpected outputs.

The test graph defines the **subset** of the product automaton that is considered for the reachability. Moreover, during the symbolic reachability analysis, the target is the (Accept/PASS) state and all the transitions that lead to (Reject/INCONCLUSIVE) are not to be traversed. This obviously improves the performance of the forward analysis. An on the fly traversal of the synchronous product (restricted to the test graph) produces the test path depicted in figure 8-(A). This accepting path is submitted to symbolic analysis for generating test cases, which can be performed during the on the fly traversal (next subsection).

3.4 Test paths executability and controllability improved

Forward symbolic for executability. Let us consider the accepting test path depicted in figure 8-(A). A forward symbolic reachability detects that a state such as $(S_1; (x = 3, y = 1.7))$ is not actually reachable in this path because whenever the automaton occupies location S1, the

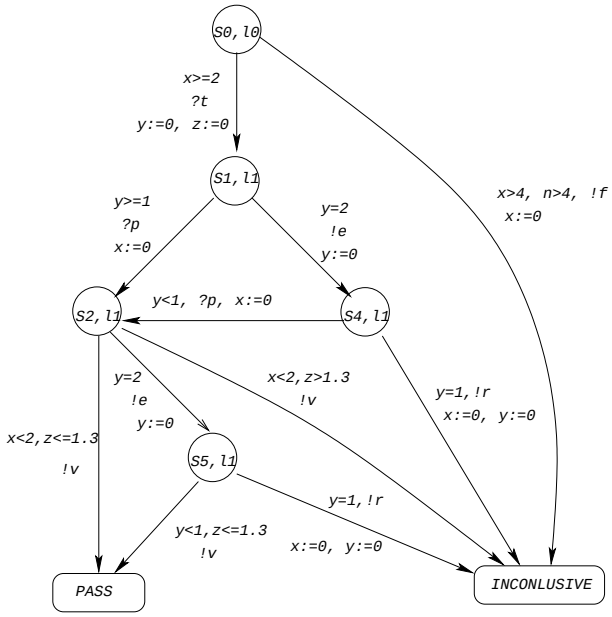

Fig. 6. *Reduced/optimised graph of synchronous product*

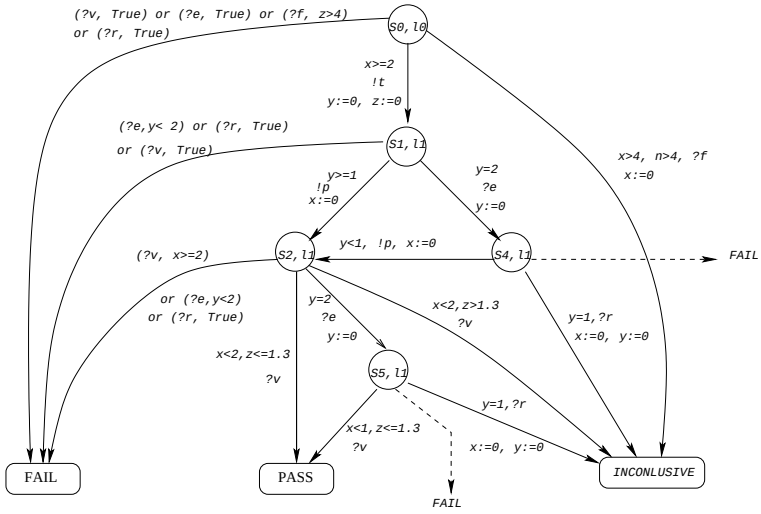

Fig. 7. *Complete test graph*

difference between x and y is at last 2 time units ($x - y >= 2$). Therefore, a test run containing such state is unsound and may fail with a correct implementation. The following presents such unsound test case.

$$(S_0,(2,2)) \xrightarrow[t_1]{x>=2,!t,(y)} (S_1,(3,1.7)) \xrightarrow[t_2]{y>=1,!p,(x)} (S_2,(0.1,1.8)) \xrightarrow[t_3]{x<2,?v,()} (S_3,(0.2,1.8)).$$

To guarantee the executability and the test soundness, the "forward" procedure previously presented must be performed, throughout the sequence of transitions in the test path, so that to insure the correctness of the propagated constraints (in the related zones). Figure 8-(B) presents the computed (correct) symbolic states that enable test executability and soundness.

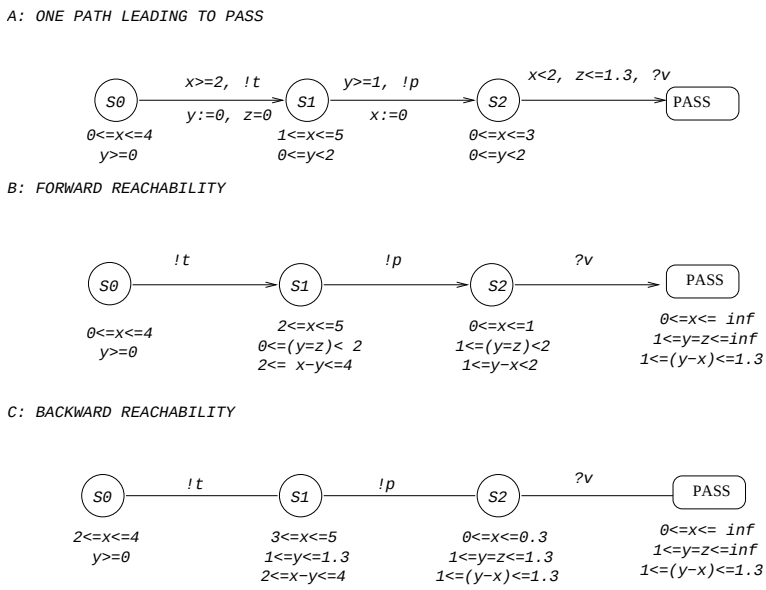

Fig. 8. *Forward and backward reachability analysis for one accepting path*

Backward symbolic for controllability. The future of an IUT run is potentially a tree structure and one can not always control its evolution towards a specific expected state. This often turns to inconclusive verdict (There was no error but the expected test could not be completed). To try to avoid such situation, one must compute the minimal constraints necessary to lead the IUT towards the expected state. Such constraint is computed in a backward manner with the target symbolic state as input. The "backward" procedure previously presented must be performed, throughout the sequence of transitions in the **executable** test path so that to refine the propagated constraints (in the related zones) until the beginning of the path.

Figure 8-(C) indicates that the PASS state is reachable only if the input $!p$ is sent when the clock x takes a value in the interval $[3,5]$ rather than in $[2,5]$, and the clock y takes a value in $[1,1.3]$ rather than $[1,2]$. In state the S1 of figure 2, such interval $[1.3,2]$ where $y > 1.3$ can not allow the interaction $!p$ to be executed in a timely manner, which implies an inconclusive outcome. Figure 9 shows the *controllable* zone in S1 in which we must submit the action $!p$, where Z is the reachable state space and Z_f is the controllable zone. If the action $!p$ is sent

in the Zone $Z1 \cup I(S1) \setminus Z$, an unsound test is generated, and if it is sent in the zone $Z2$, an INCONCLUSIVE test is generated. To avoid such situations, $!p$ should be submitted only in the controllable zone Z_f.

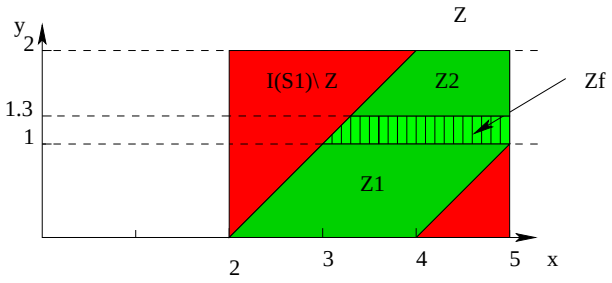

Fig. 9. *Controllable zone of S1*

Finally, to instantiate concrete tests, particular valuations (ν) of clocks can be chosen and propagated in the backward reachability. One could for instance consider extreme values like "minimal/maximal clock valuation" etc.

Example of successful test run. Let us consider the path shown in figure 8-(A). After computing forward and backward reachability graphs (figures 8-(B) and 8-(C)), we obtain the actual symbolic test path below.

$$R = S_0 \xrightarrow{(2\leq x\leq 4),!t} S_1 \xrightarrow{(3\leq x\leq 5,1\leq y\leq 1.3,2\leq (y-x)\leq 4),!p} S_2 \xrightarrow{(0\leq x\leq 0.3,1\leq y\leq 1.3,1\leq (y-x)\leq 1.3),?v} S_3$$

The symbolic test path above can be instantiated with the following test case:

$$S0 \xrightarrow[2]{x=2,!t} S1 \xrightarrow[3]{y=1,!p} S2 \xrightarrow[3.3]{x\leq 0.3,?v} \text{PASS}$$

This test case means that the tester sends signal $!t$ at 2 time units, waits at last 1 time unit and submits signal $!p$. Then it should receive $?v$ no later than 0.3 time unit after $!p$ was performed.

4 Further comments on the proposed method

Algorithms and complexity - Efficiency, savings of the method. In this paper, the test selection algorithm proceeds in two steps: First, it uses an on the fly traversal of the product (specification, observer) with a standard DFS (Depth First Search) performed in conjunction with the symbolic forward computation. Second, the intermediate test pattern obtained at this step is analysed in a backward manner for controllability refinement.

The on the fly DFS algorithm is linear with respect to the transitions relation and state space of the product. Moreover, our approach performs the on the fly selection with an optimized graph, with a reduced transition relation and state space: Check the savings by comparing the reduced/optimized graph (figure 6) against the synchronous product (figure 4 continued in figure 5). Finally, symbolic computation is known to produce fewer reachable vertices, which leads to better performances.

The other aspect is related to controllability during test experiment. As the implementation is "free", it can happen that one does not manage to carry out the desired scenario. The test

experiment thus is to be replayed several times in the hope of exhibit the desired behavior, which is very expensive in times of development. The controllability analysis is necessary for the selection of more targeted scenarios and for saving the coast of tests (cf previous comments on the refinements and gain in figure 9).

Fault detection and conformance relation. For timed systems, the principal models of errors identified in the literature are : *output error* (when an unexpected output action arrived), *transfer error* (when an unexpected state is reached), and *time constraints errors* (when an output action arrived too early or too late). The method presented in this paper detects *output* and *time constraints errors*. Moreover, it can be related to the implementation relation, referred to as tioco below, since the detection of such errors imply a violation of this implementation relation. The tioco relation is a time extension of the ioco implementation relation used for input-output systems: Let A_I and A_S be two TIOAs modelling the IUT and the specification. The IUT *conforms* to its specification if for each behavior of specification, the possible outputs of the IUT after this behavior is a subset of possible outputs of the specification behaviors. To formally define this, we use the following notations: Given a run σ, $A\mathsf{after}\sigma$ is the set of all states of A that can be reached after the execution of σ. Formally, A after $\sigma = \{l \in L_A \cdot l_0 \xrightarrow{\sigma} l\}$. The set out refers to the set of all output actions or *delay* that can occur when the automaton reach l : $\mathsf{out}(l) = \{a \in \Sigma^! \cup \mathbf{R}^+ \cdot l \xrightarrow{a}\}$. The relation tioco is defined as follows:

$$A_I \text{ tioco } A_S \text{ iff } \forall \sigma \in \mathsf{Traces}(A_S) \cdot \mathsf{out}(A_I \text{ after } \sigma) \subseteq \mathsf{out}(A_S \text{ after } \sigma).$$

Coverage. The test purpose is a natural basis to coverage analysis. It defines/specifies the tests to be computed from the system model. The method proposed in the paper computes at least one test case for a given test purpose, if it exists. One could try to compute all the test paths corresponding to a given test purpose. This option is easily implemented, but we incur combinatory explosion, and incur loosing the benefits of the on the fly search. Moreover, without additional hypotheses on the time domains, it is impossible to compute all the possible instances of timed tests because of dense time.

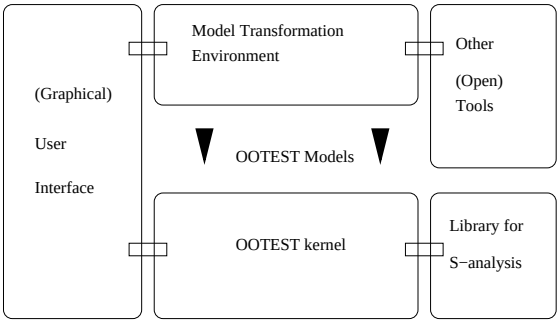

Fig. 10. Architecture of OOTEST environment

Architecture and status of OOTEST. The prototype tool OOTEST is under a very "Beta version". The test paths generated before backward analysis are not always the shortest one (there is no search optimization implemented yet). The architecture is presented in figure 10. The tool is designed to be flexible, evolutive and must be connected to other platforms and tools. Currently the tool

inputs are timed automata generated from a Graphical User Interface, developed in the french `Averroes` project [2]. For symbolic analysis, the tool reuses existing libraries for the manipulation of polyhedra. Many such libraries are available as open source and we have currently used some extensions of the `Polylib` library [5]. We can manipulate structures equivalent to DBM, and few modifications enable the manipulation of Clock Difference Diagrams, or state classes. The `Averroes` platform partially implements some model transformation and generates automata in `XML` format that can be parsed towards over model-checking tools.

5 Real-time Ethernet protocol

In this section we briefly comment the case study that we are currently carrying in our laboratory, for testing RT-EP (*Real Time Ethernet Protocol*). More details can be found in the report [3].

RT-EP [16] has been designed to avoid collision in the Ethernet media, and to achieve a relatively high speed mechanism for real time communication at a low cost, while keeping the predictable timing behaviors required in the distributed hard real time communication. Each station (processing or CPU) in RT-EP has a transmission queue and a set of reception queues. The number of reception queues can be configured depending on the number of applications threads running in the system and requiring reception of messages.

The network is logically organized as a ring. Each station knows which other station is its predecessor and its successor. The protocol works by rotating a token in this ring. The token holds information about the station having a highest priority packet to be transmitted and its priority value. The network operates in two phases. The first phase corresponds to the priority arbitration, and the second phase to the transmission of an application message.

The following operations show the functionality of RT-EP.

- Firstly, each station in RT-EP reads a configuration file describing the token ring and gets configured as one of its station. The station configured as initial *token-master* sends the *Initial Token (In-Token)* to the successor station.
- Each station listens for the arrival of any packet. When a packet is received, a check is done to determine its type:
  - If it is an information packet (*infos*) the information is written into the appropriate reception queue and the station becomes the *token-master*.
  - If it is a token packet, the station checks its type. (1) If it is a *Regular Token (Rg-Token)* the station compares the priority carried by the token with the highest priority element on its transmission queue, changes the regular token if its own priority is higher, and sends the *Update Token (Up-Token)* to the next station. (2) If the token is the *Initial Token* the station sends information (infos) if it has the highest priority, or sends the permission (action *Tr-Token*) to the highest priority station. (3) If the token is the *Transmit Token* the state has the highest priority on the ring and it is allowed to transmit it.

To recover faults due to the loss of packets, each station, after sending a packet (information or token) listens to the media for an acknowledge (action *ack*), which is the transmission of the next frame by the receiving station. If no acknowledge is received after some specified *timeout*, the station assumes that the packet is lost and retransmits it. The station repeats this process until an acknowledge is received or a specified number of retrials is produced. In the latter case the receiving station is considered as a failing station and will be excluded from the ring (action *dk*).

Each station in RT-EP can be modeled by the two concurrent automata: a sender module and a receiver module. The sender is shown in Figure 12, it uses 3 clocks (x, y, ω).

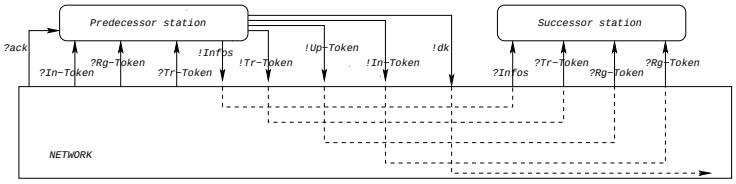

Fig. 11. *Operations in RT-EP*

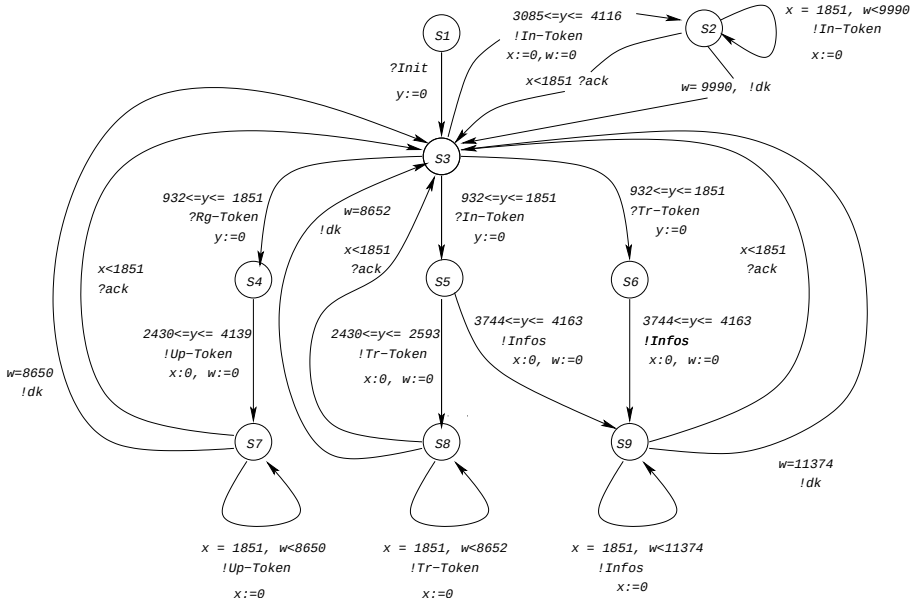

Fig. 12. *transmission module*

More details on RT-EP can be found in [16, 3].

For testing RT-EP, if we want to test the ability of the protocol to handle faults due to the loss of packets, we study the following examples:

1. A station should not be excluded from the ring only if it cannot response after 4 retransmission from the predecessor station.
2. When an Information packet is received, the station can submit an acknowledge no later than 1851 *ns*.

The observer related to the first property is shown in figure 13-(B).

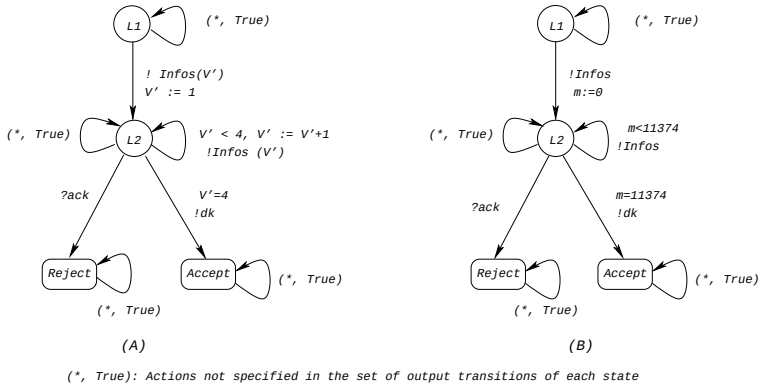

Fig. 13. *Observer of the first property; (A) with variable and (B) with temporal interpretation*

As examples of test sequences leading to the PASS we have:

$$Tc_1 = (S1, L1) \xrightarrow[y:=0]{!Init} (S3, L1) \xrightarrow[y:=0]{y=932,!Tr-Token} (S6, L1) \xrightarrow[x:=0,w:=0,m:=0]{y=4163,?Infos} (S9, L2) \ ...$$

$$... \ (S9, L2) \xrightarrow{w:=11374,m:=11374,?dk} PASS.$$

$$Tc_2 = (N1, F1) \xrightarrow[m:=0]{!Init} (N2, F1) \xrightarrow[m:=0]{y=2119,!Infos} (N2, F2) \xrightarrow{k<1851,m<1851,?ack} PASS.$$

The first test case is generated from the complete test graph of the transmission module specification of RT-EP (Figure 12) and the observer of Figure 13. Here we propagated the minimal value of the time in the case of emission and the maximal value of the time in the case of reception. This test means that after initialization the tester emits to the IUT a transmit token packet at 930 *ns* and should observe the information packet no later than 4163 after sending the transmit token, waits 11374 *ns* and *should* observe the output (dk). For receiving the output (dk) the tester should not submit to the IUT the input (ack). The second test case is generated from the complete test graph of the reception module of RT-EP and the observer related to the second property.

6 Conclusion

We have presented a method to test selection for real-time systems. Some ideas in our test design process are inspired by techniques used, in other respects, in different research fields. The forward symbolic analysis was used for model-checking of timed systems. The backward analysis was a technique used in fault tolerance analysis to track the cause of a failure in the past of system execution. Combining such techniques for the purpose of testing is - to our knowledge - a new contribution for test selection improvement.

The current work is concerned with functional conformance requirements, and it does not address the full features of the real-time observers actually to be used in the current project. These observers are expected to model extra features like failures of the run-time environment, etc. For future work, our method and its implementation within the `OOTEST` tool will upgrade in a short term, to take some aspects of robustness testing into account.

References

1. R. Alur and D. Dill, *A Theory of Timed Automata.* Theoretical Computer Science 126:183-235, 1994.
2. *Analysis and VERification for the Reliability Of Embedded Systems.*(www.education.gouv.fr/rntl)
3. R.Bouaziz, O.Koné. *Design principles and applications of the* `OOTEST` *tool.* Technical Report, CNRS University of Toulouse 2006.
4. Lori Clarke and Debra Richardson. *Symbolic evaluation methods for program analysis. In Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice-Hall, Englewood Cliffs, NJ, 79–101. 1981
5. Ph. Clauss and V. Loechner *PolyLib: A Library for Manipulating Parameterized Polyhedra.* Technical Report, University of Strasbourg, 1999.
6. R. Castanet, O. Koné and P. Laurencot, *On-the-Fly Test Generation for Real-Time Protocols.* IEEE International Conference on Computer Communication and Networks. Lafayatte, 1998.
7. 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 (ISSRE), Rennes, France, November 2004.
8. Grabowski J., Hogrefe D., Nahm R. *Test case generation with test purpose specifications by MSCs.* 6th SDL Forum. Elsevier Science, North Holland, 1993. Pages 253-266.
9. K. Larsen, M. Mikucionis, and B. Nielsen, *On line Testing of Real-Time Systems.* Formal Approaches To Testing of Software, Link2, Austria. September 2004.
10. T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli, *Generating Test Cases for a Timed I/O Automaton Model.* IFIP (IWTCS'99) Budapest, 1999.
11. P.Gaufillet. *The TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEm Design* ERTS2006 - 3rd Embedded Real Time Software Conference - Toulouse January 2006. http://www.topcased.org
12. Ph.Dhaussy, JC.Roger, H.Bonin, E.Saves and J.Honnoré. *Experimentation of Timed Observers for Validation of an Avionics Software.* Toulouse, January 2006.
13. William E. Howden. *Methodology for the Generation of Program Test Data.* IEEE Trans. Computers, 24(5): 554-560, 1975
14. T. Hinzinger, X. Nicollin, J. Sifakis, and S. Yovine. *Symbolic Model Checking for Real-Time Systems. Information and Computation.* 111(2): 193-244, June 1994.
15. O.Koné. A local approach to the testing of real-time systems. *The Computer Journal*, British Computer Society, Oxford Press. Vol. 44 N.5, 2001.
16. J. M. Martinez, M. G. Harbour, and J. J Gutierrez, *RT-EP : Real-Time Ethernet for analyzable distributed application an a minimum real-time POXIS-kernel.* 2nd International Workshop on Real-Time LANs in the Internet Age. RTLIA 2003.
17. K.L.McMillan. *Symbolic model-checking: An approach to the state explosion problem.* Kluwer Academic, 1993.