# Testing communicating systems : a model, a methodology and a tool

Ismail Berrada, Richard Castanet, Patrick Felix

# Testing Communicating Systems: a Model, a Methodology, and a Tool *

Ismaïl Berrada Richard Castanet Patrick Félix

LaBRI - CNRS - UMR 5800 Université Bordeaux 1
33405 Talence cedex, France
{berrada | castanet | felix}@labri.fr

**Abstract.** This paper follows two main lines of research. The first line is related to the study of models for the description of systems. For this line, we introduce the model of *Communicating Systems* (CS), which defines a set of common resources, a set of entities, and a topology of communication. The second line focuses on testing methodologies adapted to protocol testing. For this line, we give a formal definition of a generic generation algorithm (GGA). We demonstrate that the CS model with a GGA supports various 1) test architectures, 2) test types: conformance, interoperability, embedded, component testing, and 3) test approaches: passive and active testing. The paper presents also the main characteristics of the TGSE tool (Test Generation, Simulation, and Emulation). TGSE is made-up of a test case generator, based on the CS model and implementing a GGA, a graphic simulator of the execution of a sequence generated by TGSE, and a real-time emulator of communicating specifications. In its current version, TGSE supports the passive and active testing of one or several components with data and temporal constraints.

## 1 Introduction

Protocol specifications are used to develop products and services. To ensure correctness of such products (implementations), testing, the process of checking that a system possesses a set of desired properties and/or behaviors, is one of the most used validation techniques.

Testing process is a hard work that is long, repetitive and which represents a potential source of errors. The use of formal specifications provides support for automating this process. Different models (FSM, EFSM, CEFSM, LTS, IOLTS, TIOA,...) and languages (SDL, Lotos, IF, UML,...) have been proposed to describe protocols and the desired behaviors about them in a formal way. Due to the nature of protocols/functions being tested, various test types are required. For example, in conformance testing, a single implementation is compared to relevant standards. In interoperability testing, two or more implementations are tested directly against each other, with the standard used primarily as a reference to adjudicate problems and incompatibilities, and secondarily as a guide

* This research has been supported by the French RNTL project Avérroes and the Marie Curie RTN TAROT (MCRTN 505121).

to the functions to be tested. Embedded testing considers an implementation communicating through its environment. The ways to test communicating systems can be classified into two basic groups. The most natural way, namely the active testing approach [1–17], consists in carrying out the test derivation starting from specifications. Another possibility is the passive testing approach [21]. The absence of observations allows only the validation of traces, and thus this approach checks that a trace of an implementation is a valid execution of the specification.

From our point of view, this diversity of types, models and approaches points only to the specificity of requirements. Indeed, the different types of test are a consequence of the composition of the systems to be tested : conformance testing considers only one entity while interoperability and embedded testing consider several communicating components interacting according to a test architecture. The considered model is justified by needs of system description: systems have behaviors and can handle data and temporal constraints. With either the passive testing or the active testing, we are confronted with the same problem: the accessibility problem of states or transitions.

Thus, the aim of this paper is not to introduce a new test generation technique, but rather to show that it is possible to treat the different types (conformance, interoperability, embedded, component) and approaches (passive and active) of testing in a unified manner. Our main contributions are the following:

First, we introduce the model of communicating systems (CS). This model defines (i) a set of communicating entities (components), (ii) a set of common resources (variables and parameters) shared by these entities, and (iii) a topology of communication, inspired by [19, 20], which specifies the different possible synchronizations in a global state of the system. We have chosen to model entities by extended timed automata but other models may be used.

Second, we demonstrate that the CS model is a generic model for testing in the sense that (i) it offers mechanisms for modeling different types of communications and test architectures, and (ii) it allows the possibility of applying the same generic generation algorithm (GGA) with different test types and approaches. As we will see, these results have a consequence on the classical test activities in the sense that the specification modeling and the use of a test approach and a test architecture are not two separate steps.

Finally, by presenting the TGSE tool (Test Generation, Simulation and Emulation), we show that our framework is usable in practical tools. TGSE implements an on-the-fly GGA, and supports the passive and active testing (with test purpose) of one or several components with data and temporal constraints.

The paper is organized as follows. Section 2 introduces the CS model. The generic character of this model and its suitability for protocol testing are discussed in section 3. Section 4 gives some elements of the implementation of the test generator tool TGSE. Section 5 reports our experimental results on CSMA/CD protocol. Finally, we conclude and draw some perspectives in section 6.

## 2 Model and Methodology

The behavior of a communication protocol can be described by means of formal models such as communicating systems (CS). In this paper, $\mathbb{R}$ will denote the set of reals, and $\mathbb{R}^+$ will denote the set of positive reals.

### 2.1 Preliminaries

**Clocks and Constraints.** A clock is a variable that allows to record the passage of time. It can be set to a certain value and inspected at any moment to see how much time has passed. In the Alur-Dill model [18], clocks increase at the same rate, they are ranged over $\mathbb{R}^+$, and the only assignments allowed are clock resets of the form $x := 0$. For a set $C$ of clocks, a set $P$ of parameters, and a set $V$ of variables, the set of clock constraints $\Phi(C, P, V)$ is defined by the grammar:

$$\phi := \phi_1 \; |\phi_2| \; \phi_1 \wedge \phi_2, \;\; \phi_1 := x \leq f(P, V), \;\; \phi_2 := f(P, V) \leq x$$

where $x$ is a clock of $C$, and $f(P, V)$ is a linear expression of $P$ and $V$. For two sets $L_1$ and $L_2$, $L_1 \backslash L_2$ will denote the set $L_1 \backslash L_2 = \{a \,|\, a \in L_1 \wedge a \notin L_2\}$.

**Definition 1 (ETIOA).** *An extended timed input/output automaton (ETIOA) is a 10-tuple* $M = (S, L, C, P, V, V_0, Pred, Ass, s_0, \rightarrow)$ *where :*

- $S$ *is a finite set of states.*
- $s_0$ *is the initial state.*
- $L$ *is a finite alphabet of actions,* $L = L_i \cup L_o \cup I$.
- $C$ *is a finite set of clocks.*
- $P$ *is a finite set of parameters.*
- $V$ *is a finite set of variables.*
- $V_0$ *is a finite set of the initial values for variables of* $V$.
- $Pred = \Phi(C, P, V) \cup \tilde{P}[P, V]$, *where* $\tilde{P}[P, V]$ *is a set of linear inequalities on* $V$ *and* $P$.
- $Ass = \{x := 0 \,|\, x \in C\} \cup \{v := f(P, V) \,|\, v \in V\}$ *is a set of updates on clocks and variables.*
- $\rightarrow \subseteq S \times L \times Pred \times Ass \times S$ *is a set of transitions.*

The alphabet $L$ is partitioned into three sets: $L_i$ (resp. $L_o$) is the input (resp. output) alphabet, and $I$ is the alphabet of internal actions. $t = (s, a, pred, ass, s') \in \rightarrow$ represents an edge from state $s$ to state $s'$ on symbol $a$. $pred \subseteq Pred$ is a set of constraints, and $ass \subseteq Ass$ is a set of updates.

*Example 1.* Fig. 1 illustrates an example of an ETIOA.

- $S = \{s_0, s_1, s_2, s_3\}$ and $s_0$ the initial state.
- $L = \{!a, ?b, !c, ?d\}$, $C = \{x, y\}$, $P = \{\beta, \lambda\}$, $V = \{v1\}$ and $V_0 = \{\beta\}$.
- The variable $v_1$ has the initial value $\beta$.
- $Pred = \{y \geq \lambda, x \leq 1, v1 \leq 4\}$, $Ass = \{x := 0, y := 0, v1 := v1 + 1\}$.
- The transition $t$ from $s_2$ to $s_3$ is: $t = (s_2, !c, \{x \leq 1\}, \{v1 := v1 + 1\}, s_3)$.

*Remark 1.* For an ETIOA $M = (S, L, C, P, V, V_0, Pred, Ass, s_0, \rightarrow)$:

- When $P = \emptyset$ and $V = \emptyset$, then we find the usual definition of a timed i/o automaton (TIOA). In this case, $M$ will be simply noted $(S, L, C, s_0, \rightarrow)$.
- When $C = \emptyset$, $P = \emptyset$ and $V = \emptyset$, then we find the usual definition of an i/o automaton (IOA). In this case, $M$ will be simply noted $M = (S, L, s_0, \rightarrow)$.

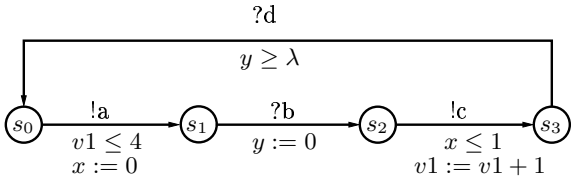



**Fig. 1.** ETIOA.

### 2.2 Topologies of communication and Communicating Systems

A topology of communication **Top** of a set of processes is a synchronization model of the different processes. It describes the dynamic configurations of processes, and the possible synchronizations in a given configuration. The definition of **Top** is inspired by [19, 20]. It defines a set of global actions, a set of sets of actions, and a Transducer (this terminology is borrowed from [20]) modeled by an automaton.

**Definition 2 (Topology).** *The topology of communication $Top$ of a set of $n$ processes is a 3-tuple $(G, I, Tr)$, with $G$ a finite set of global actions, $I = \{I_i\}_{1 \leq i \leq n}$ a finite set of sets, and $Tr = (S_{tr}, L_{tr}, s_{0tr}, \rightarrow_{tr})$ an automaton such that the events of $L_{tr}$ are vectors $\overrightarrow{v}$ of $n+1$ elements, and $\forall \overrightarrow{v} \in L_{tr}$, $\overrightarrow{v} =< a_g, a_1, ..., a_n >$ with $a_g \in G$ and $\forall i \in [1, n]$, $a_i \in I_i \cup \{idle\}$.*

A vector $\overrightarrow{v} =< a_g, a_1, ..., a_n >$ of $L_{tr}$ describes the action $a_i$ that the process $i$, $i \in [1, n]$, has to perform. The synchronization of the actions $(a_i)_{i \in [1,n]}$ gives place to the global action $a_g$. When a vector $\overrightarrow{v} =< a_g, idle, ..., a_i, ..., idle >$ defines only one action, the process $i$ executes lonely $a_i$, and changes its state. For a topology $Top = (G, I, Tr)$, when the number of states of $Tr$ is equal to 1 then $Top$ is called a *static* topology.

A topology offers the possibility of modeling communications between one, two or several processes: unicast, multicast, and broadcast. It can be used, in certain cases, as a kind of controller on actions allowed by processes in a given configuration of the global system. Note that, in order to describe inter-component communications, a process algebra can be more expressive than the topology, however, this latter offers suitable modeling mechanisms and algorithms usable in practical tools.

**Definition 3 (Communicating System).** *A communicating system $CS$ is a 5-tuple $(SP, SV, SV_0, (M_i)_{1 \leq i \leq n}, Top)$ where:*

- $SP$ *is a set of shared parameters.*
- $SV$ *is a set of shared variables.*
- $SV_0$ *is a set of the initial values for variables of* $SV$.
- $Top = (G, \{I_i\}_{1 \leq i \leq n}, Tr)$ *is a topology.*
- $M_i = (S_i, L_i, C_i, P_i, V_i, V_{0i}, Pred_i, Ass_i, s_{0i}, \rightarrow_i)$ *is an ETIOA such that* $I_i \subseteq L_i$, $\forall i \in [1, n]$.

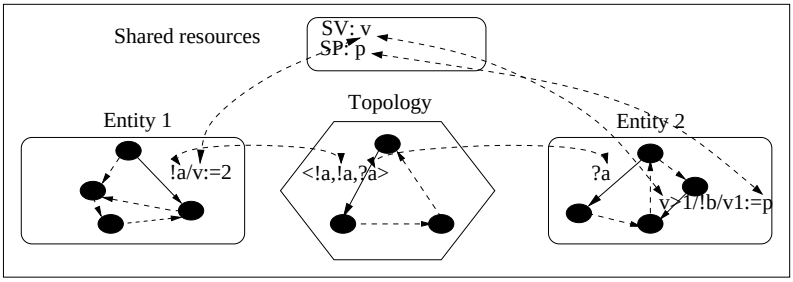



**Fig. 2.** CS model.

Fig. 2 illustrates an example of a CS. Entities represent processes. They are modeled by ETIOAs. The topology of communication describes the different possible synchronizations between the entities. We assumed in the definition of entities that $\forall i \in [1, n]$, $I_i \subseteq L_i$. This enables the definition of partial topologies in which only allowed synchronizations are reported (in the next section, we will give some examples of such topologies). The common resources represent the shared data of the CS. We will restrict the shared data to variables and parameters. The parameters (resp. variables) can be read (resp. read and modified) by the CS entities [1]. The semantics of a CS is defined by an ETIOA. To simplify, we will assume that the names of parameters and variables of entities are different, and different from those of the CS.

**Definition 4 (Semantics).** *The semantics of a communicating system* $S = (SP, SV, SV_0, (M_i)_{1 \leq i \leq n}, Top)$, *with* $M_i = (S_i, L_i, C_i, P_i, V_i, V_{0i}, Pred_i, Ass_i, s_{0i}, \rightarrow_i)$ *and* $Top = (G, I, (S_{tr}, L_{tr}, s_{0tr}, \rightarrow_{tr}))$, *is defined by the ETIOA* $\zeta(S) = (S, L, C, P, V, V_0, Pred, Ass, s_0, \rightarrow)$ *such that :*

- $S = \{s = (s_{tr}, s_1, ..., s_n) \mid s_{tr} \in S_{tr}, \forall i \in [1, n], s_i \in S_i\}$
- $s_0 = (s_{otr}, s_{01}, ..., s_{0n})$.
- $L = G$, $C = C_1 \cup ... \cup C_n$, $P = SP \cup P_1 \cup ... \cup P_n$.
- $V = VP \cup V_1 \cup ... \cup V_n$, $V_0 = VP_0 \cup V_{01} \cup ... \cup V_{0n}$.
- $Pred = Pred_1 \cup ... \cup Pred_n$, $Ass = Ass_1 \cup ... \cup Ass_n$.

[1] Shared parameters and variables can appear in the definition of a transition of an entity.

– $\rightarrow = \{(s_{tr}, s_1, ..., s_n) \xrightarrow{a,pred,ass} (s'_{tr}, s'_1, ..., s'_n) \mid \exists \overrightarrow{v} = <a, a_1, ..., a_n> \in L_{tr}, s_{tr} \xrightarrow{\overrightarrow{v}}_{tr} s'_{tr}, \forall i \in [1,n], (((a_i = idle) \wedge (s_i = s'_i)) \parallel ((a_i \neq idle) \wedge (s_i \xrightarrow{a_i,pred_i,ass_i}_i s'_i))), pred = pred_1 \wedge ... \wedge pred_n, ass = ass_1 \wedge ... \wedge ass_n\}$.

The alphabet $L$ of $\zeta(S)$ is the set $G$ of global actions of $Top$. A state of $\zeta(S)$ consists of a state of $Top$ and states of $(M_i)_{i\in[1,n]}$. A transition $(s_{tr}, s_1, ..., s_n) \xrightarrow{a,pred,ass} (s'_{tr}, s'_1, ..., s'_n)$ of $\zeta(S)$ is conditioned by the existence of a transition of $Top$ from $s_{tr}$ to $s'_{tr}$ on a vector having the global action $a$.

Thus, the semantics of a CS allows the possibility of the synchronization with other CSs, which gives a hierarchical definition for CSs. A possible extension of the CS model consists in the definition of extended topologies: the transducer modeled by an ETIOA (could be useful for modeling network latencies). Note that, the size (number of transitions) of the semantics automaton is linear in the size of entities times the size of the topology. In practice, however, this size is orders of magnitude less. For example, the size of a CS, such that its topology is a tree, is linear in the size of its topology.

### 2.3 Methodology of Generic Generation Algorithms.

The majority of test generation algorithms are based on a depth-first search of a target state or transition in the accessibility graph. It is then possible to define generic generation algorithms for various test types. In this part, we show how to define such algorithms.

**Definition 5.** *A communicating system under test (CSUT) is a communicating system $S = (SP, SV, SV_0, (M_i)_{1\leq i\leq n}, Top)$, such that there is at least one entity $M_i$, $i \in [1,n]$, defining one or several states labeled by $ACCEPT$.*

States labeled by $ACCEPT$ define the behaviors to be tested. Our definition of CSUT considers only states labeled by $ACCEPT$, but it is possible to define transitions labeled by $ACCEPT$. This last case is not treated in this paper, but the approach remains the same. Let us note by $\mathcal{CSUT}$, the set of all CSUTs.

**Definition 6.** *For a $S \in \mathcal{CSUT}$, a state $s = (s_{tr}, s_1, ..., s_n)$ of $\zeta(S)$ and $\rho = t_0...t_n$ a sequence of transitions in $\zeta(S)$ from the initial state:*

- *$s$ is an accepting state of $\zeta(S)$ if there exists $i \in [1,n]$ such that $s_i$ is a state labeled by $ACCEPT$.*
- *$\rho$ is an accepting path of $\zeta(S)$, if*
  1. *$\rho$ is an executable path.*
  2. *The target state of the last transition $t_n$ is an accepting state of $\zeta(S)$.*

A state $s$ of the ETIOA $\zeta(S)$, the semantics of $S$, is an accepting state of $\zeta(S)$, if one of the states that compose it, is a state labeled by $ACCEPT$. A path $\rho = t_0...t_n$ of $\zeta(S)$ from the initial state is an accepting path of $\zeta(S)$ if 1) the state $s_n$ of the last transition $t_n = (s_{n-1}, a, pred, ass, s_n)$ is an accepting state of $\zeta(S)$ and 2) $\rho$ is an executable (feasible) path, i.e, the different constraints on the transitions are all satisfied. The executability of a path is treated in [21, 22].

**Definition 7.** *A generic generation algorithm (GGA) for $\mathcal{CSUT}$ is an algorithm that computes, for all $S \in \mathcal{CSUT}$, all accepting paths of $\zeta(S)$.*

An algorithm $gga$ is a GGA, if $gga$ applied to $\zeta(S)$ returns a set $PATH(S)$ containing all accepting paths of $\zeta(S)$. Examples of GGA can be found in [21, 23, 22]. Note that the Hit-or-Jump algorithm [23] does not deal with the temporal aspect of systems and considers $ACCEPT$ transitions.

Finally, an algorithm $gga$ does not depend on a CSUT. It can be applied to any ETIOA and it is exhaustive in the sense that all accepting paths are returned by $gga$. Its complexity depends on the size of entities and the size of the topology used. We have chosen the state coverage criterion for defining $gga$ but the transition (or other) coverage criterion can also be chosen [22].

## 3 CS : A Generic Model for Testing

In this section we present the expressivity and the generic character of CSs for describing and testing protocols. Modeling specifications is presented in 3.1. Testing with different types and approaches is presented in 3.2.

In the remainder of this section, we will consider two specifications $S_A$ and $S_B$, sharing the set of parameters $SP$, and the set of variables $SV$, such that $SV_0$ is a finite set of the initial values for variables of $SV$. We model $S_A$ (resp. $S_B$) by the ETIOA $A = (S_A, L_A, C_A, P_A, V_A, V_0, Pred_A, Ass_A, s_0, \rightarrow_A)$ (resp. $B = (S_B, L_B, C_B, P_B, V_B, V_0', Pred_B, Ass_B, s_0', \rightarrow_B)$). $L_{AB}$ (resp. $L_{BA}$) will denote the set of events of $L_A$ (resp $L_B$) which synchronize with an event of $L_B$ (resp. $L_A$). For example, if $L_A = \{?a_1, ?a_2, !a_3\}$ and $L_B = \{!a_2, ?a_3, !a_4\}$ then $L_{AB} = \{?a_2, !a_3\}$, $L_{BA} = \{!a_2, ?a_3\}$, and $?a_2$ (resp. $!a_3$) synchronizes with $!a_2$ (resp. $?a_3$). To simplify, we will assume that $\forall a \in L_{AB}$, there is a unique $b \in L_{BA}$ such that $a$ synchronizes with $b$.

### 3.1 CS as a Specification Model

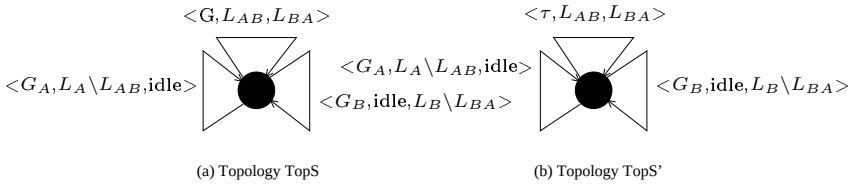



**Fig. 3.** Different topologies

**Observable events.** Suppose that $S$ is the specification made up of specifications $S_A$ and $S_B$. A CS modeling of $S$ is : $CS_1 = (SP, SV, SV_0, (A, B), TopS)$, with $TopS$ the automaton of Fig. 3 (a). $TopS$ is a static topology. Vector $< G, L_{AB}, L_{BA} >$ denotes the vectors $< g_{ab}, a, b >$ such that $a \in L_{AB}$ synchronizes with $b \in L_{BA}$, and their synchronization gives place to an observable

action $g_{ab}$. An example of $g_{ab}$ can be $a$ (resp. $b$) if $a$ (resp. $b$) is an emission (the visible action of an emission and a reception is an emission). In the same way, $< G_A, L_A \backslash L_{AB}, idle >$ denotes the vectors $< g_a, a, idle >$ such that $a \in L_A \backslash L_{AB}$. In $< g_a, a, idle >$, the ETIOA $A$ performs the action $a$ giving place to the observable action $g_a$, and the ETIOA $B$ remains in the same state ($idle$). The set $G_A$ corresponds, in general, to the set $L_A \backslash L_{AB}$. Finally, $TopS$ allows the application of each vector (if it is possible) in a global state of $S$.

**Non-observable events.** Now, suppose that the synchronizations of $L_{AB}$ events with $L_{BA}$ events are non-observable (as it is the case of the black-box test architecture), then modeling $S$ in CS is: $CS_2 = (SP, SV, SV_0, (A, B), TopS')$, with $TopS'$ the automaton of Fig. 3 (b). In $< \tau, a, b >$ of $< \tau, L_{AB}, L_{BA} >$, the synchronization of $a$ with $b$ gives place to an internal action $\tau$. Generally, we can describe the synchronization on internal actions only for a part of the synchronization events as it is the case of a test architecture.

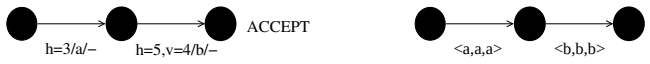
Thus, from a testing standpoint, the CS model is not only a formal model allowing the description of inter-component communications, but also a model that is able to incorporate the test architectures.

### 3.2 CS as a Test Generation Model

Two major approaches were used for protocol testing: *Active Testing* and *Passive Testing*. In active testing, the derivation is made from specifications. The derivation can consider only a part of the specification with the aim of limiting the state space explosion which occurs during the system composition and analysis. This approach is known as the test purpose technique. Active testing can deal with one or several communicating entities [1–17]. On the other hand, passive testing considers execution traces of an implementation, which can contain values of variables and clocks, and checks the validity of these traces with respect to the specification. In the works relating to passive testing [21], the authors consider only one untimed specification.

To simplify, let us call *one-component testing* the test of one specification (conformance testing) and *several-component testing* the test of several specifications (interoperability, embedded, component testing). In the rest of this section, we consider that $gga$ is a GGA. This section shows that the test activities amount to a CS modeling, by deferring the different characteristics of a test to the topology of communication, and the application of the algorithm $gga$ to validate a trace (passive testing) or to generate traces (active testing).

**Passive testing.** Suppose that $I$ is an implementation of the specification $S_A$, and the trace modeled by the ETIOA of Fig. 4 (a) is a trace of $I$. This trace reports that $I$ has executed $a \in L_A$ (we recall that $A$ is the ETIOA of $S_A$) at moment 3, followed by $b \in L_A$ at moment 5 such that the shared variable $v \in SV$ is equal to 4. Checking the validity of this trace consists in modeling a CS $CS_3 = (SP, SV, SV_0, (A, PTrace), PTop)$, with $PTrace$ the ETIOA of Fig. 4 (a) and $PTop$ the automaton of Fig. 4 (b).
The topology $Ptop$ is partial, i.e, it defines only the synchronizations on $PTrace$




(a) PTrace (b) PTop

**Fig. 4.** Passive testing.

events. $< a, a, a >$ (resp. $< b, b, b >$) considers that $A$ and $PTrace$ synchronize on $a$ (resp. $b$), and the visible action will be $a$ (resp. $b$). We have labeled the state reached by the action $b$ in $PTrace$ by $ACCEPT$ in order to make $CS_3$ a CSUT and to be able to apply $gga$. Consequently, $gga$ allows to decide if $PTrace$ is a valid trace of $A$: if $gga$ returns an empty set ($PATH(CS_3) = \emptyset$) then $PTrace$ is not a valid trace of $A$ (we recall that $gga$ is applied to the semantics of $CS_3$).

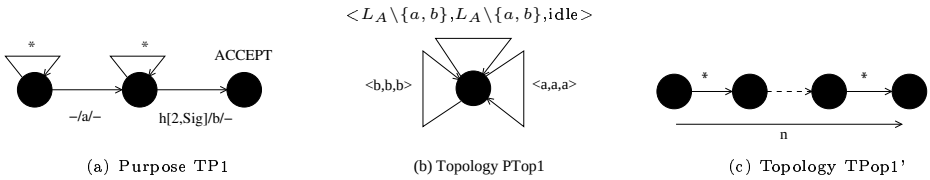
*Remark 2.* -Generally, the construction of $PTop$ depends strongly on $PTrace$. It should define only synchronizations on $PTrace$ events and in the same order.
-The trace $PTrace$ is considered as an entity of $CS_3$ without any distinction compared to the other entities. This allows to enlarge the form of the considered traces to any traces modeled by an ETIOA defining some accepting states.
-The example of the passive testing of the specification $S_A$ is one-component testing, but the approach remains the same in the case of several-component testing. In this setting, the difficulty is to reorder various traces from the different components to construct only one trace. We think that the stamp mechanisms, and especially the stamp process presented in [24], could be used. This subject goes beyond the framework of this paper and needs more investigation.

**Active testing.** For a CSUT $S$, the paths $PATH(\zeta(S))$ generated by $gga$ can be used to derive test cases that cover, for example, all $S$ states. This amounts to define all $S$ states as being accepting states ($gga$ could be an adaptation of the TT/UIO/Wp methods for untimed systems). Thus, we consider here only the test purpose technique.

**Definition 8.** *A test purpose (TP) is an ETIOA* $(S, L, C, P, V, V_0, Pred, Ass, s_0, \rightarrow)$ *having two sets of states* $ACCEPT$ *and* $REJECT$ *characterizing the behaviors to be tested.*

A TP is a property that one would like to check on implementation behavior. $TP1$ of Fig. 5 (a) illustrates an example of a TP for the specification $S_A$. $TP1$ tests that an implementation $I$ of $S_A$ can execute $a$ followed by $b$ at an instant between $[2, Sig]$ according to the clock $h$ ($Sig \in SP$ is a shared parameter of $S_A$). The label '*' denotes the alphabet $L_A$ of $A$. We assume here that $a, b, c \in L_A$.

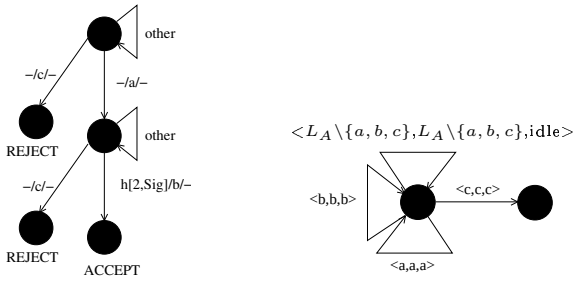
*One-component testing.*
Suppose that $TP1$ is a TP for specification $S_A$. A modeling of this test in CS is: $CS_4 = (SP, SV, SV_0, (A, TP1), TPop1)$, with $TPop1$ the topology of Fig. 5 (b). The vectors $< L_A \backslash \{a, b\}, L_A \backslash \{a, b\}, idle >$ denote free evolutions of specification $A$ on events other than $a$ and $b$. $CS_4$ is a CSUT and thus $gga$ will generate paths checking $TP1$.




(a) Purpose TP1 (b) Topology PTop1 (c) Topology TPop1'

**Fig. 5.** Active testing: one-component testing (1).

Note that for the same TP, several CS modelings can be formulated, considering different topologies. In fact, the definition of the topology gives more expressivity to the behaviors awaited by a TP. A typical example of this expressivity is as follows: since paths generated by $gga$ for $CS_4$ are of arbitrary lengths (the number of transitions), one can wish to generate only paths of lengths less than $n \in \mathbb{N}$. This wish cannot be expressed by a TP (there is no mechanism to count the event occurrences). Now, let us consider the CS $CS_5 = (SP, SV, SV_0, (A, TP1), TPop1')$, with $TPop1'$ the topology of Fig. 5 (c). The label '*' in $TPop1'$ denotes vectors $< a, a, a >$, $< b, b, b >$, and $< L_A\backslash\{a,b\}, L_A\backslash\{a,b\}, idle >$ (a transition '*' is then the set of transitions on these vectors). With $TPop1'$ the semantics of $CS_5$ is a tree of depth less than $n$ and thus the lengths of paths generated by $gga$ are less than $n$.




(a) Purpose TP2 (b) Topology PTop2

**Fig. 6.** Active testing: one-component testing (2).

To close the part of one-component testing, let us take the TP $TP2$ of Fig. 6 (a). $TP2$ tests the same functionalities as $TP1$, but prohibits the appearance of $c$ in the two first states of $TP2$. The label *'other'* in $TP2$ denotes the events $L_A\backslash\{c\}$. Note that the definition of $REJECT$ states is only a manner of prohibiting synchronizations on a set of events. This prohibition can be formulated in the topology instead of the test purpose. In this case, we can use $TP1$ instead of $TP2$. Indeed, the active testing of $S_A$ with test purpose $TP2$ can be modeled by the CS $CS_6 = (SP, SV, SV_0, (A, TP1), TPop2)$, with $TPop2$ the topology of Fig. 6 (b). In $TPop2$, when a synchronization on $c$ occurs, the communicating system evolves/moves to a blocking/deadlock state and thus during the application of $gga$ to $CS_6$, $gga$ is forced to dequeue this synchronization. Finally, note that we have used $TP1$ in $CS_6$ to test $TP2$, and therefore a test purpose can

contain only $ACCEPT$ states.

*Several-component testing.*
Suppose that $S$ is the specification made up of specifications $S_A$ and $S_B$, and $TP1$ (Fig. 7 (a)) is a TP for $S$. To simplify, we assume here that $a \in L_A$, $a \notin L_{AB}$ ($a$ is not a synchronization event), and $b \in L_{AB} \cap L_{BA}$. A modeling of this test in CS is as follows: $CS_7 = (SP, SV, SV_0, (A, B, TP1), Top)$, with $Top$ the topology of Fig. 7 (b). The vector $< b, b, b, b >$ considers that $A$, $B$ and $TP1$ synchronize on $b$. The vector $< a, a, idle, a >$ considers that only $A$ and $TP1$ synchronize on $a$. Again, the application of $gga$ allows generating paths checking $TP1$.

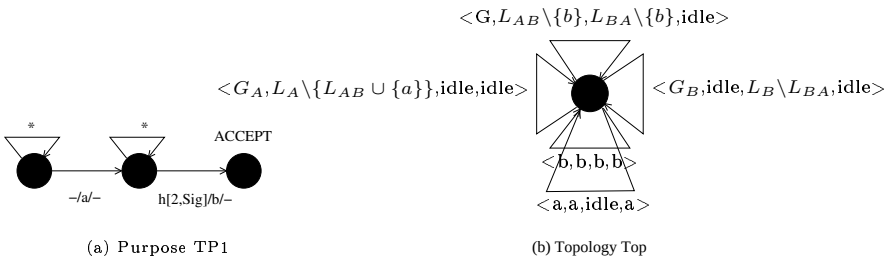



**Fig. 7.** Active testing: several-component testing.

To summarize this section, Fig. 8 presents the test activities (without the implementations). Three main steps are identified. Firstly, from (i) an informal specification(s), (ii) a test approach (passive or active testing), and (iii) a test architecture, a description $S$ in the CS model is elaborated. Secondly, a GGA algorithm (with a coverage criterion) is applied to $\zeta(S)$ to generate a set of executable paths $PATH(\zeta(S))$. Finally, this set is interpreted according to the test approach: for passive testing, if $PATH(\zeta(S)) = \emptyset$ then the implementation(s) is (are) incorrect. For active testing, test cases are generated from this set. In the opposite, the classical test activities involve the specification, the generation algorithm and the test architecture in three separate steps and thus looses the generic character of our framework.

Thus, there is no reason to make distinction between these types and approaches of protocol testing.

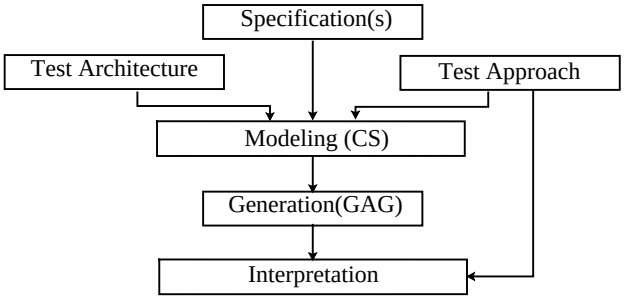



**Fig. 8.** Test activities.

## 4 TGSE: A Generic Test Generation Tool

The automation of the test generation becomes a need faced with the growth of the complexity of the protocols being tested. This section describes the main characteristics of the implementation of the TGSE tool (Test Generation, Simulation and Emulation) based on the CS model. Due to space restrictions, we present only the test case generator. The interested readers are referred to [22].

**TGSE Interfaces.** The French RNRT project Calife and its successor Averroès are an academic and industrial projects gathering France Telecom R&D, CRIL Technology, LaBRI, LSV, Loria, LRI. The goal of this project is to define a generic platform (Open Source) able to interface verification and test generation tools. The Calife platform [25] comprises an editor and a simulator. The editor provides a pleasant and easy-to-handle graphical user interface of various types of automata (timed, hybrids, and extended automata). The simulator allows the graphical execution of automata.

The input of TGSE is the description of a CS following a simple syntax. Each ETIOA of a CS is defined in a separate file. A system file describes the access paths to each component, as well as the shared data and the topology of communication. The output of TGSE is an XML (eXtensible Markup Language) file according to a Calife DTD defining a test sequence.

TGSE can also be used in a graphical mode through Calife. In this case, the description of a specification is done through the Calife editor that allows the automatic generation of the synchronization vectors. Many synchronization modes are offered: rendez-vous, broadcast, identical labels and the Uppaal binary synchronization. The call to TGSE is done through the editor that generates the input files of TGSE. TGSE produces a test case to be simulated in Calife

**Generation Techniques.** TGSE implements an on-the-fly algorithm $gga$. It is based on a depth-first traversal of the CS semantics. The traversal is parameterized by the maximum number allowed for a transition to appear in the generated sequence. The choice of a transition, a synchronization vector and the automaton that performs an action is parameterized (RANDOM or FIFO access). The algorithm $gga$ computes an accepting path for a given CS. During the traversal, several computations are performed:

*Step 1: Successor Computation.* From the current state $s$ of the semantics automaton, the synchronization vectors are evaluated in a parameterized way to compute a successor state $s'$. The API $SymbolicTrace()$ is then called.

*Step 2: Symbolic Trace.* $SymbolicTrace()$ calculates the symbolic trace of the new fired transitions and updates the predicates and the context (assignments and resets of the new transitions,see annex).

*Step 3: Constraint Resolution.* Once the symbolic trace is calculated, the API $feasible()$ is called. In the case of a parameterized trace, $feasible()$ calls $checkParams()$. This latter interacts with the linear programming tool $lp\_solve$ $v4$ for instancing parameters. In the opposite case, $checkClocks()$ is carried out for computing the fastest/slowest timed executions [11, 22].

*Step 4: Test Case Computation.* If during the traversal an accepting state is met, the search ends by a call to the $writeTrace()$ to decorate the path obtained by the different verdicts. The output is an XML file according to a Calife DTD.

The algorithm *gga* is explained in more detail in the annex and in [22]. Its complexity is linear in the size of the CS times the complexity for solving linear programming problems. Finally, if no accepting state is met, *gga* is automatically started for a new attempt (the launching is parametrized). Moreover, it is possible to generate a test case that has the minimal number of transitions for a given number of attempts. We point out that TGSE is based on a conformance relation (traces inclusion) taking into account data and clocks [22].

## 5 Case Study: CSMA/CD Protocol

The CSMA/CD Protocol (Fig. 9) is made up of a bus (medium of communication) and one or more senders (transmitting stations). We do not model here the receivers. When two or several senders transmit simultaneously data on the bus ($!begin$), a collision event ($!CD$) is sent by the bus to all senders. These latter have to retransmit later. Thereafter, $Senders$ (resp. $Bus$) will denote the ETIOA representing the specification of the sender (resp. bus) (Fig. 9).

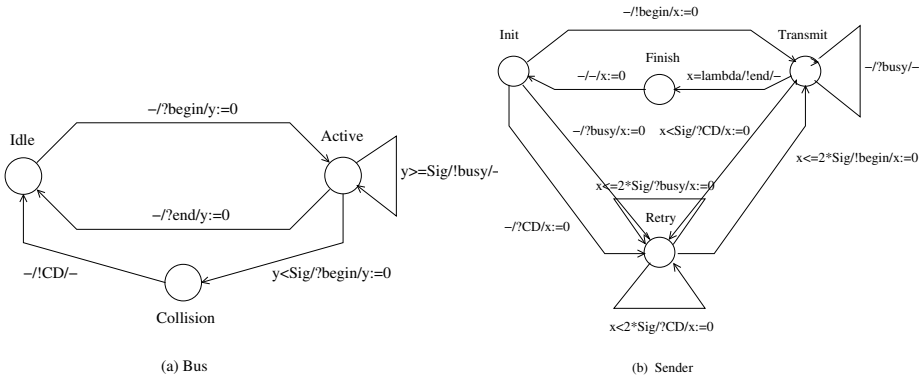
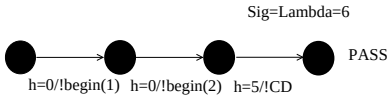



**Fig. 9.** CSMA/CD specifications

The next table reports the experimental results of applying TGSE to CSMA/CD with one bus, several senders and the TP: $1 \xrightarrow{-/!begin/-} 2 \xrightarrow{h=5/!CD/-} 3$. TP checks that a sender sends data ($!begin$), and the bus detects a collision at instant 5 ($!CD$). A test case generated by TGSE for this test appears bellow. In this case, Sender 1 transmits at 0, Sender 2 transmits at 0 and the Bus detects a collision at 5.




(b) PTop

The experience is run on a INTEL P4 DELL INSPIRON 5100 PC, with 256Mo of RAM running Mandrake 10.0. Each input of the table was launched 2000 times. Lock represents the number of times that a transition can appear in a path, Size TC the average size of a test case, Nb Sender the number of senders considered and CPU Time the average generation time. The reader can notice that the generation with Lock equal to 1 takes more time. In fact, with Lock equal to 1, *gga* moves to locked states and thus dequeues several times.

| Lock | Nb Sender | TC Size | CPUs Time (s) |
|---|---|---|---|
| 1 | 5 | 3 | 0.303 |
| 1 | 10 | 3 | 0.621 |
| 1 | 20 | 3 | 0.914 |
| $10^3$ | 5 | 55 | 0.098 |
| $10^3$ | 10 | 79 | 0.234 |
| $10^3$ | 20 | 130 | 0.793 |

Although the CSMA/CD protocol is of a reduced size, the use of several senders increases its complexity. The obtained results are encouraging and improvements are at hand.

## 6 Conclusion

The aim of this paper is to show that different types (conformance, interoperability, embedded, component) and approaches (passive and active) of protocol testing can be treated in a unified manner. To achieve this aim, we have presented a testing framework based on the generic model of communicating systems (CS) and the methodology of generic generation algorithms (GGA). The CS model defines a set of communicating entities (components) modeled by extended timed input/output timed automata (ETIOAs), a set of common resources (variables and parameters) shared by these entities, and a topology of communication specifying the different synchronizations allowed in a system configuration. We showed that the test activities consist then in modeling a CS and applying an algorithm GGA. To our knowledge, this is the first framework that can fully handle various test types and approaches. Our framework was implemented in TGSE tool (Test Generation, Simulation and Emulation). The current version of TGSE can be used both for passive and active testing of one or several components but supports only deterministic ETIOAs and the definition of a test purpose.

Regarding future work, our intention is to study the impact of a coverage criterion on the definition of GGA, and to realize a realistic performance evaluations

of TGSE on complex protocols. Finally, until very recently, research had been carried out with almost no interactions between the software and protocol testing communities. So, our framework might bring the two communities together, since object-oriented programming languages and component-based approaches (code testing) are now widely used in software development, and these lead to the need of state-based test techniques.

**Acknowledgments.** We would like to thank the members of the specific action AS 32 carried out by Ana Cavalli for their fruitful remarks. We would like also to thank the ENSEIRB students Dimitri Kandassamy, Jamel Semeh, David Dogoh and Carine Beduz for their participations in the realization of TGSE.

## References

1. Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence, *Software - Concepts and Tools* 17(3): 103-120 (1996).
2. Laura Brandán and Ed Brinksma. A test generation framework for quiescent real-time systems. *FATES2004*, Linz, Austria September 21 2004.
3. S. Seol, M. Kim, S. Kang, J. Ryu. Fully automated interoperability test suite derivation for communication protocols, *Computer Networks* Volume 43, Pages 735 - 759, December 2003.
4. Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata Specifications, *Formal Aspects of Computing*, 12(5):350-371,2000.
5. Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE RTSS*, 1997.
6. A. En-Nouaary, R. Dssouli, F. Khenedek, and A. Elqortobi. Timed test cases generation based on state characterization technique, *In 19th IEEE RTSS*, Madrid, Spain, 1998.
7. T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating Test Cases for a Timed I/O Automaton model, *TESTCOM99*, Budapest, Hungary, September 1999.
8. A. Koumsi, M. Akalay, R. Dssouli, A. En-Nouaary, L. Granger. An approach for testing real time protocols, *TESTCOM*, Ottawa, Canada, 2000.
9. Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications, *ACM Transactions on Computer Systems*, 13(4):365-398, 1995.
10. Jan Springintveld, Frits Vaandrager, Pedro R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 252(1-2):225-257, March 2001.
11. I. Berrada, R. Castanet, P. Félix. From the Feasibility Analysis to Real-Time Test Generation, *Studia Informatica Universalis* Volume 3 (2) pp.203-230 2004.
12. K. Larsen, M. Mikucionis, and B. Nielsenn. Real-time system testing on-the-fly. *In the 15th Nordic Workshop on Programming Theory (NWPT)*, 2003.
13. M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. *In SPIN 2004* (2004), Spring-Verlag Heidelberg, pp. 109-126.
14. Abdeslam En-Nouaary, Rachida Dssouli: A Guided Method for Testing Timed Input Output Automata. TestCom 2003: 211-225
15. Ahmed Khoumsi, Thierry Jéron, Hervé Marchand. Test Cases Generation for Nondeterministic Real-Time Systems. FATES 2003: 131-146
16. K. El-Fakih and N. Yevtushenko. Fault Propagation by Equation Solving. *Proceeding of FORTE*, Madrid, Spain. LNCS 3235, September 2004.

17. S. Boroday, A. Petrenko, R. Groz and Y.M. Quemener. Test Generation for CEFSM Combining Specification and Fault Coverage. *TESTCOM02*, Berlin, Germany, March 2002.
18. R. Alur and D. Dill. A theory of timed automata, *Theoretical Computer Science*, 126:183-235, 1994.
19. André Arnold et M. Nivat. Comportements de processus. *In Colloque AFCET "Les mathématiques de l'Informatique"*, pages 35-68, 1982.
20. Tomás Barros, Rabéa Boulifa and Eric Madelaine. Parameterized Models for Distributed java Objects. *FORTE*, Madrid, Spain. LNCS 3235, September 2004.
21. Baptiste Alcalde, Ana Cavalli, Dongluo Chen, Davy Khuu, and David Lee. Network Protocol System Passive Testing for Fault Management: A Backward Checking Approach. *Proceeding of FORTE*, Madrid, Spain. LNCS 3235, September 2004.
22. Ismail Berrada, Richard Castanet and Patrick Félix. Techinques de Test d'Interopérabilité. Fourniture Calife, 2005.
23. Ana Cavalli, David Lee, Christian Rinderknecht and Fatiha Zaïdi. Hit-or-Jump: An algorithm for embedded testing with applications to IN services. *FORTE/PSTV'99*, Beijing, China. October 1999.
24. Claude Jard, Thierry Jéron, Lénaïck Tanguy and César Viho. Remote testing can be as powerful as local testing. *FORTE/PSTV'99*, Beijing, China. October 1999.
25. http://www.cril-technology.fr.

## Annex

**Description.** The generation algorithm $gga$ applied to a CS $S$ performs a depth-first traversal of $\zeta(S)$. During the traversal, $gga$ computes the symbolic trace and checks the feasibility of the new fired transitions. When an accepting state is met (the function $AcceptStates()$), a backtracking in the synchronization path is performed to decorate this latter with verdicts (function $writeTrace()$). Due to the space limit, we will present only the $gga()$ and $SymbolicTrace()$ functions.

**Data Structure.**

$States$: a n+1-tuple $(s_{tr}, s_1, ..., s_n)$.
$Context$: records the values of variables and last resets for clocks.
$Transition$: a n+1-table of pointers on the current transitions.
$Element$ is a structure composed of a $States$, a $Context$ and a $Transitions$.
$Path$: a stack of Elements. It managed by the operations "push", "top" and "pop".

**Other functions.**

$SynchronizationOnEvents()$: chooses a synchronization vector from the current state and returns a structures Element composed of the new transitions and states reached.
$getSuccessors()$: returns a successor state of the current state.
$getInitStates$: returns the initial state of $\zeta(S)$.

**Function gga():**

```
1.  Begin
2.      States:= getInitialStates(), Element := NULL, Path := ∅;
3.      Do
4.          Element := SynchronizationOnEvents(States);
5.          If (Element ≠ NULL) then
6.            push(Element,Path);
7.            SymbolicTrace(Path);
8.            If(!feasible(Path)) then
9.              pop(Element,Path);
10.           States := getSuccessors(Element);
```

11. Else
12. pop(Path); States := getSuccessors(top(Path)) ;
13. If(AcceptStates(States))
14. writeTrace(Path);
15. **While**(Path $\neq \emptyset$);
16. **End**

**Symbolic Trace.** Let us assume that $M = (S, L, C, P, V, V_0, Pred, Ass, s_0, T)$ is an ETIOA such that $C = \{c_1, ..., c_k\}$, $V = \{v_1, ..., v_m\}$, $V_0 = \{v_{01}, ..., v_{0m}\}$ and $\rho = t_1...t_n$ is a suite of transitions of $M$ from the initial state. The symbolic trace of $\rho$ is $\rho$ such that $\forall\ t_i = (s_{i-1}, a, pred, ass, s_i)$ of $\rho$, and $\forall v \in V$, $v$ is replaced in $pred$ by its last value before $t_i$ (see [22]). **SymbolicTrace()** uses two vectors: $V1$ contains the current values of variables (may depend on P parameters). $V2$ is a vector of natural numbers. $V2[q]$ stocks the index of transition where the clock $x_q \in C$ was last reseted.

**Function SymbolicTrace():**

1. **Input/output:** $\rho = t_1...t_n$, with $t_i = (s_{i-1}, a, pred_i, ass_i, s_i)$,
2. **Temporary Data** Two vectors: $V1$ of size $m$ and $V2$ of size $k$.
3. **Begin**
   /*Initialization */
4. For $i := 1$ to $m$ Do $V1[i] \leftarrow v_{0i}$;
5. For $i := 1$ to $k$ Do $V2[i] \leftarrow 0$;
   /*Updating */
6. For $i := 1$ to $n$ Do
7. $pred_i \leftarrow updatePredicates(pred_i, V1, V2, i)$;
8. $updateContext(ass_i, V1, V2, i)$;
9. **End**

The function **UpdatePredicates** replaces the variables with their current values from $V1$. For clocks, if $c_p$ is last reseted in the $t_j$ and $i$ is the index of the current step then $c_k$ is replaced by $h_i - h_j = h_i - h_{V2[p]}$ (line 4).

**Function UpdatePredicates():**

1. **Input:** A predicate $pred$, an index $i$, and two vectors $V1$ and $V2$.
2. **Output:** A predicate $predUpdated$.
3. **Begin**
4. $predUpdate \leftarrow pred[h_i - h_{V2[1]}, ..., h_i - h_{V2[k]}, V1[1], ..., V[m], P]$;
5. **End**

The function **UpadateContext** updates 1) the current values of variables in $V1$ from the new assignments (lines 4 and 5), and 2) the clock resets $V2$ by assigning the index of the current step (lines 6 et 7).

**Function UpdateContext():**

1. **Input:** An assignment $ass$, and an index $i$.
2. **Input/Output:** Two vectors $V1$ and $V2$.
3. **Begin**
4. For $j := 1$ to $m$ Do
5. If $v_j := f(v_1, .., v_m, P) \in ass$ then $V1[j] := f(V[1], .., V[m], P)$;
6. For $j := 1$ to $m$ Do
7. If $c_j := 0 \in ass$ then $V2[j] := i$;
8. **End**