DEFINING A FORMAL SEMANTICS FOR REAL-TIME TEST SPECIFICATION WITH TTCN-3[1]

Diana Alina Serbanescu[1] and Ina Schieferdecker[2]

[1]FOKUS Fraunhofer Institute for Open Communication Systems,
MOTION Department, Berlin, Germany
diana.serbanescu@fokus.fraunhofer.de
[2]Free University of Berlin, Department of Mathematics and Computer Science,
Software Engineering, Berlin, Germany
ina.schieferdecker@fu-berlin.de

ABSTRACT

Real-time software is usually used in circumstances where safety is important and the margin for errors is narrow. These kinds of systems have applicability in a broad band of domains as: automotive, avionics, air traffic control, nuclear power stations, industrial control, etc. As the name denotes, the main feature of "real-time" applications is the criticality of their timeliness. Guaranteeing certain timeliness requires appropriate testing. As manual testing is burdensome and error prone, automated testing techniques are required. Although the importance of having a standard environment for automatic testing is high, the technologies in this area are not sufficiently developed. This paper reviews the standardized test description language "Testing and Test Control Notation version 3 (TTCN-3)" as a mean for real-time testing and proposes extensions to enable real-time testing with TTCN-3. The aim is to provide a complete testing solution for automatic functional and real-time testing, built around this already standardized testing language. The solution includes an environment for designing and running the tests written in the extended language. As a proof of concept, test examples, designed using the enhanced TTCN-3, are mapped to real-time platform implementations and the timeliness of each implementation is analyzed.

KEYWORDS

Real-time Software, Testing, Formal Semantics, Timed Automata, TTCN-3

1. INTRODUCTION

As the name denotes, the main feature of real-time applications is the criticality of their timeliness. Testing of real-time software represents a challenge due to their special nature which requires not only testing the functional aspects, but also timing aspects of the computation. The challenge is even greater when the complexity and diversity that prevails in the world of real-time applications has to be handled. An automatized and standardized testing process will increase the efficiency of testing and will be more suitable for covering and testing requirements of such complex systems than traditional testing methods, which are involving a lot of manual testing and are, therefore, more prone to errors. An automatized testing framework is ought to guarantee reproducible tests. Furthermore, the reproducibility of the testing process will better lead to the discovering of reproducible errors and failures in the time behavior. Without automation, searching for errors in a timed behavior would be lumpish and totally unreliable. Also, a standardized testing framework would provide a common basis of usage for different stakeholder.

[1] Testing and Test Control Notation version 3 – defined by the European Telecommunications Standards Institute (ETSI). The TTCN-3 related standards can be found at: http://www.ttcn-3.org/

Different manufacturer could test their products using a common standardized set of tests and they could interchange and compare their results.

The solution provided in this paper is concerned with the design of a testing framework for real-time embedded systems. The aim is to provide a test technology that can be successfully used for automating the test procedures, especially with regards to the real-time aspects, in domains with rapid development process and high quality demands as, for example in the automotive industry. The real-time testing methodology and framework presented here is based on a standardized test language that was proven to be popular and successful in the industry, in areas as mobile and broadband telecommunications, medical system, IT systems and, since recently, also in the automotive. The referred language is Testing and Test Control Notation version 3[2] (TTCN-3), developed and maintained by European Telecommunications Standards Institute (ETSI).

Having the advantages of being a well modularized, test-oriented, user friendly and popular, TTCN-3 has also the downside of not being developed with real-time focus in mind. Thus, it lacks some mechanism for dealing with real-time specific test situation. The insufficiencies of TTCN-3 language towards real-time were first being analysed in [3]. Therefore, several extensions were developed for this language in order to make it suitable for real-time.

The new proposed concepts are integrated into the syntactical structure of the TTCN-3 language, by means of clear syntactic rules, based on extended Backus-Naur Form (BNF) notation. The semantics of the real-time test system, realized on the basis of enhanced TTCN-3 is further defined by means of timed automata [4]. The focus of this paper will be on presenting these time automata defining the semantic of the enhancements.

This approach, using timed automata, is new and different from the way semantics of TTCN-3 was previously defined into the standard. The motivation for choosing timed automata is that they are mathematical instruments specialized in modelling timed behaviour in a formal way.

2. RELATED WORK

There exist already several approaches that have been proposed in order to extend TTCN (Tree and Tabular Combined Notation, an earlier version of TTCN-3) and TTCN-3 for real-time and performance testing. Among these RT-TTCN, PerfTTCN, TimedTTCN-3, Continuous TTCN-3 should be mentioned. A more comprehensive state of the art, describing those enhancements is being presented in [3].

The work presented in this paper began as collaboration with the TEMEA project[2]. Therefore, the basic real-time concepts that are introduced in this document are based on the ones developed in the context of TEMEA project.

In the context of TEMEA project, a new paradigm and a new extension for real-time testing based on the TTCN-3 notation was developed. The ambitious goal was to use the experience of the past and to develop the language with new meaningful concepts, which are more powerful and more oriented towards the embedded systems used in the automotive domain than the attempts made in the past. The aims of the TEMEA project can be summarized in the following:

- Support for integrated testing of discrete and continuous behaviour.
- Exchange of test definitions between different test- and simulation platforms e.g. Model in the Loop (MIL) platforms, Software in the Loop (SIL) platforms and Hardware in the Loop (HIL) platforms.

[2] TEMEA (TEst specification and test Methodology for Embedded systems in Automobiles)", http://www.temea.org, Last verified on July 2012

- Support over the entire process of software integration and hardware integration.
- Analysis of real-time and reliability requirements.
- Testing distributed components according to AUTOSAR architecture.
- Analysis of the quality of tests.

The basic concepts introduced by TEMEA can be organized in the following categories:

- Representation of time: In order to ease the manipulation of time values, two new abstract data types are introduced. *datetime* designates global time values and *timespan* designates time distances or intervals between different points in time. It is used to represent the amount of time that passed between events. In order to give the tester the right instruments of detecting relative time, some predefined symbols of the type *datetime* are introduced. *testcasestart* returns the time point when the test case execution started and *testcomponentstart* returns the point in time when the test component execution started.

- Measurement of time: The observation of time is directly connected to the reception and provisioning of messages at communication ports. A new construct is introduced for automatically registering the time value at which a receive-event, or send-event has occurred. The saving is indicated by redirect symbol '->' followed by the *timestamp* keyword.

- Control of application: The '*at'* operator is introduced to send certain messages at fixed points in time. It is associated with a *datetime* value, representing the point in time when the sending of a message should be performed.

- Time verification: In order to verify whether certain messages were received in time, the *within* operator is introduced. The operator is associated with an interval of *datetime* values that represent the range for the allowed times of reception.

A more comprehensive presentation of those concepts can be found in the list of following publications: [6], [1] and [7].

Some of the concepts listed above have gained concrete ground and became part of a new standard from ETSI. This standard is intended to be an extension for performance and real-time testing, and is regarded as an additional package to TTCN-3. TTCN-3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional as part of a package which is suited for dedicated applications and/or usages of TTCN-3.

To fulfil the requirements for testing real-time system, the following TTCN-3 core language extensions are being standardized in [5].

- A test system wide available test system clock, that allows the measurement of time during test case execution.

- Means to directly and precisely access the time points of the relevant interaction events between the test system and the system under test.

As previously mentioned, the work presented here was developed in parallel with the evolution and definition of the real-time concepts from TEMEA. Therefore, it undertakes the basic set of concepts defined within this context, but it is not limited to it. In the following section the concepts that are focus of this work are presented and briefly discussed. For a more detailed presentation of these concepts, one should look in [8].

3. SUMMARY OF THE REAL-TIME EXTENSIONS FOR TTCN-3

In Table 1 are listed the new concepts, divided in four categories, comprised into four columns: the first column introduces the special operations; the second column introduces the temporal predicates, the third column introduces the predefined constants used to build `timespan` values, and the last column lists the new introduced data types.

Now	***at***	***hour***	***datetime***
Wait	***within***	***min***	***timespan***
Break	***before***	***sec***	***tick***
Timestamp	***after***	***millisec***	
testcasestart	***not***	***microsec***	
testcomponentstart		***nanosec***	
testcomponentstop			

Table 1: List of RT-TTCN-3 Terminals Which Are Reserved Words

4. SEMANTIC DEFINITIONS OF CONCEPTS FOR REAL-TIME TTCN-3

The ways to address different timing requirements and transform them into reliable timed test specification, using the standardized language TTCN-3, had been presented already in publications as [6], [1] and [7]. A new set of concepts had been introduced in TTCN-3 in order to enhance the power of the language with regards to testing real-time applications. In order to properly show the power of those concepts and to understand how timed test specification can be converted into timed test behavior a clear understanding of the underlying semantics is required. Therefore, a semantic formalism for of all these concepts is presented in the following.

An example of a simple real-time test system is presented in Figure 1.The test system is illustrated there together with the correspondent system under test (SUT). At a closer look, one can identify two critical sections with regard to time, contained in the test system's behavior: first, there is t_{max}, a timed requirement for the SUT, that indicates the interval in which the reaction to the first stimulus should be received by the test component; the second is t_{wait}, which indicates the time that should elapse at the test component side, between the receiving of the first reactions from the SUT and the sending of the second stimulus to the SUT. These are simple examples of timing requirements that should be added to a test system in order to make it appropriate for testing timing aspects of real-time applications.

In the next subsection, a real-time test system is formally defined in terms of timed automata. Each feature particular of a test system is being mathematically described, and where necessary, tagged with time stamps.

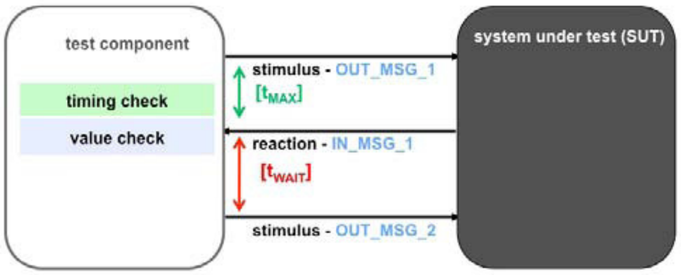

Figure 1: Black-box Test with Time Restrictions [1]

4.1. Defining The Real-time Test System (RTTS) As A Timed Automata

The semantics for the new concepts will be presented in the following context. We consider our test system to be one type of timed automata described by the set: $TS = \{L, X, A, G, I, U, E\}$ where:

- L is a set of location or states from which we use a subset to define the semantics of the new introduced concepts. This subset is:
  $S = \{S_{now}, S_{clock}, S_{wait}, S_{next}, S_{error}, S_{send}\} \bigcup \{S^{i}_{comp\_start}, S^{i}_{comp\_stop} \mid i = 1..n_{comp}\}$
  $\bigcup \left\{S^{i}_{snd\_timestamp} \mid i = 1..n_{ports}\right\}$
- $X = Clocks \bigcup VarList$
  - $Clocks$ is considered to be a set of $\Re^{\geq 0}$-valued variables called clocks, where C_0 is the general clock of the system, visible from every state and which increments its value at fixed intervals; the general clock of the system cannot be reset or have its value changed; all the other clocks are available to be used and initialized in any state. They are useful for calculating relative timings, as for measuring the time spent in S_{wait} state, for example.
  - $VarList$ is the list of all variables from the TS .
  - $VarList \supset VarList' = Messages \times Timestamps$, where:
    - $Messages$ represents the set of all messages that enter or leave the system at runtime.
    - $Timestamps = \{timestamp \mid timestamp \in \Re^{\geq 0}\}$ represents the set of timestamps for the messages that enter or leave the system at runtime. The timestamp variable represents time values in seconds.
    - $VarList'$ is organized into queues in the following way:

$$VarList' = \bigcup_{i=1}^{n} queue_i$$

$$= \bigcup_{i=1}^{n} \bigcup_{j=1}^{m_i} \{(msg_{i,j}, timestamp_{i,j}) \mid msg_{i,j} \in Messages, \; timestamp_{i,j} \in Timestamps\}$$

  - $VarList \supset Components$, where $Components = \{Comp_i \mid i \in \aleph^{+}, i = 1..n_{comp}\}$ is the set of all components that were created in the TS .

- $A = Chans(A_{in}) \cup Chans(A_{out}) \cup A'; A_\tau = A \cup \{\tau\}$. We assume a given set of actions A, mainly partitioned into three disjoint set of output actions:
    - $Chans(A_{in}) = \bigcup_{i=1}^{n} Ch_i(A_{in}^i) = \bigcup_{i=1}^{n} \{ch_i(a?) \mid a \in A_{in}^i\}$, where ch_i are channels attached with communication ports of the TS, n is a natural number representing the number of ports, and $Ch_i(A_{in}^i)$ is the set of input events that can enter on that port into the TS.
    - $Chans(A_{out}) = \bigcup_{i=1}^{n} Ch_i(A_{out}^i) = \bigcup_{i=1}^{n} \{ch_i(a!) \mid a \in A_{out}^i\}$, where ch_i are channels attached with communication ports of the TS, n is a natural number representing the number of ports, and $Ch_i(A_{in}^i)$ is the set of input events that can leave the TS through the ith port.
    - A' is the set of special internal events, used for synchronizing different parts of the TS which are running in parallel. The used set of internal events is:

      $$\begin{aligned} A' = &\{tick!, tick?, now!, now?, break!, break?\} \\ &\bigcup \{received_i!, received_i?, queue_i!, queue_i?, received_{ij}!, received_{ij}?, \\ &send_i!, send_i?, stop\_send_i!, stop\_send_i? \mid i = 1..n_{ports}, j = 1..n_{tmpl_i}\} \\ &\bigcup \{start_i!, start_i?, stop_i!, stop_i?, stop\_comp\_start_i!, stop\_comp\_start_i?, \\ &stop\_comp\_stop_i!, stop\_comp\_stop_i? \mid i = 1..n_{comp}\} \end{aligned}$$

    - In addition, it is assumed that there is a distinguishable unobservable action $\tau \notin A$
- $G = Guards(Clocks) \bigcup Templates(Messages)$,where
    - $Guards(Clocks)$ denotes the set of guards on clocks, being conjunctions of constraints of the form $c \propto t$, where $\propto \in \{\leq, <, ==, >, \geq\}$, $c \in Clocks$ and $t \in \Re^{>0}$
    - $Templates(Messages)$ is a set of constraints on messages which splits the message set into equivalence classes of the form $EqMsgs = \{msg \mid msg \approx tmpl, msg \in Messages\}$ and $tmpl \in Templates(Messages)$, where $Templates(Messages)$ is the set of all applicable templates that are available in TS.
- $I : L \to Guards(Clocks)$ assigns invariants to locations
- $U(X) = U(Clocks \bigcup VarList) = U(Clocks) \bigcup U(VarList)$ is the set of updates of clocks corresponding to sequences of statements of the form $c := t$, where $c \in Clocks$ and $t \in \Re^{>0}$, represents the time in seconds.
- E is a set of edges such that $\mathrm{E} \subseteq L \times G \times A_\tau \times U(X) \times L$

- $Q : Chans \rightarrow Queues$ is a bijection which assigns queues to channels. For each channel ch_i there is a queue $queue_i$ associated with it so that $Q(ch_i) = queue_i$, where $i \in \aleph, i = 1..n$ with n designating the number of channels used by the TS .
- We consider $Part$ to be the set of all partitions over the set $Guards \times Templates$. The function $R : Queues \rightarrow \bigcup_{P \in Part} P$ assigns a set of time restrictions and message templates to each incoming queue. $R(Queue_i) = \bigcup_{j=1}^{n_{tmpl_i}} g_{ij} \times tmpl_{ij}$, with $g_{ij} \in Guards, tmpl_{ij} \in Templates$, $i, j \in \aleph$ and n_{tmpl_i} is the number of templates associated with the queue.
- We consider $Brunches_{Alt}$ being a subset of branches associated with an alternative. Then the function $B : Queues \times Guards \times Templates \rightarrow Brunches_{Alt}$ represents a bijection which associates a queue with time guards and a template to a branch of the alternative. $B(queue_i, g_{ij}, tmpl_{ij}) = branch_k$, where $i = 1..n_{ports}, j = 1..n_{tmpl_i}, k = 1..|\,Brunches_{Alt}\,|$.

We will consider in the following that the test system TS is a timed automaton with the above presented structure. The semantics of the new introduced instructions is represented accordingly, also as timed automata that represent subsections of the test system whose behavior they are forming. In this context, the TS will be a composition of those smaller time automata which may run sequentially or in parallel.

4.2. Semantics Of Special Operations Relaying On Time

At the core of every timed TS there is a `clock` which keeps the track of time from the beginning of the TS execution. In

Figure 2 is presented the time automaton associated with the global clock. The clock is considered to be periodic, with the period δt , this being a characteristic of the used processor, $\delta t = \frac{1}{f}$ where f is the frequency of the processor and $t, f \in \Re^+$.

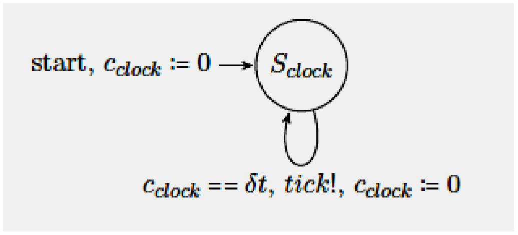

Figure 2: Logical Clock Timed Automaton

Complementary to the timed automaton for the logical clock, there is a timed automaton associated with the `now` instruction, which gives the total time from the beginning of the TS execution, until the current moment when the instruction is invoked (

Figure 3). When the period of time characteristic to the logical clock expires, the `clock` automaton emits a $tick!$ signal. This signal is used for synchronization between the two automata.

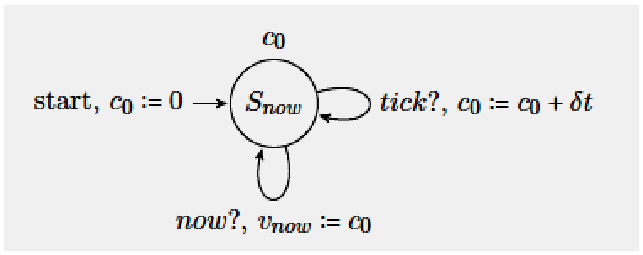

Figure 3: Now State Timed Automaton

When signal $tick!$ is received by the `now` automaton, the local variable from the S_{now} state increases its value with the value of the period δt. The `clock` and `now` automata are going to be active for the whole life of the TS. The now automata uses the $now?$ signal for synchronization with other automata existing in the system. When some other automata need to access the current time value, it will emit a complementary $now!$signal. This will be intercepted by the `now` automata. Every time when this signal is generated and intercepted, the value of the local clock c_0 is copied in the global variable $v_{now} \in VarList$. This variable can be then accessed by the automata that requested the current time.

A trace for the `clock` automata would look like: $(\delta t \cdot tick!)^*$, while a trace for the `now` automata might look like: $(tick\,?^n \cdot now\,?^m)^*$, where $n, m \in \aleph$ are naturals.

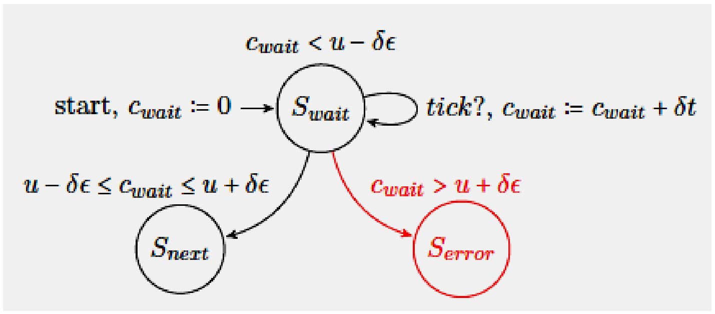
$c_{wait} < u - \delta\epsilon$
start, $c_{wait} := 0$
S_{wait}
$tick?$, $c_{wait} := c_{wait} + \delta t$
$u - \delta\epsilon \leq c_{wait} \leq u + \delta\epsilon$
$c_{wait} > u + \delta\epsilon$
S_{next}
S_{error}

Figure 4: Wait State Timed Automaton

In Figure 4 is presented the timed automaton associated with the `wait` instruction. It can be observed that this timed automaton is approximately similar to the `now` automaton, with the difference that when the local clock, c_{wait}, reaches a first threshold but not overpass a second threshold, there will be a transition to state S_{next}. If the second threshold is stepped over, then the transition will lead to an error state. The state S_{wait} has therefore the invariant $c_{wait} < u - \delta t$ associated with it. The states S_{next} and S_{error} represent generic states, used to designate the transition to other parts from the TS, possibly represented by other automata.

S_{error} represents a final state indicating time inaccuracy of the `wait` instruction. This type of errors should be signalized because otherwise the system would lose its propriety of being time deterministic.

A trace for the wait automata would look like $(tick\ ?)^n \cdot t_{wait}$, where t_{wait} might be $t_{wait} \in [(u - \delta e), (u + \delta e)]$, or $t_{wait} > (u + \delta e)$.
The automata represented in
Figure **5** and
Figure **6** illustrate the times tamping process that takes place at the beginning of a test case, at the beginning of the execution of a test component and at the end of the execution of a test component, respectively. The next state is not reached before interrogating the now automaton for the current time. The current time is the stored in a variable which is associated either with the beginning of the test case, either with the beginning or the end of a test component. If we assume that there are n_{comp} test components in the current test case, then there will be $2 * n_{comp}$ variables for registering starting and ending time of each component.

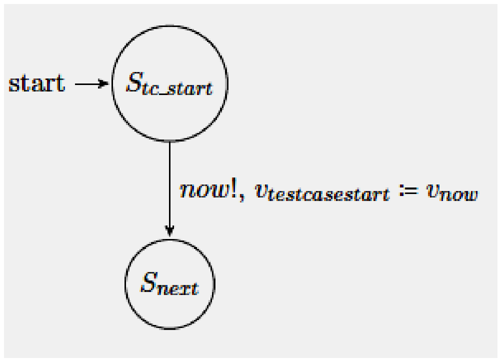

Figure 5: Test Case Start Timed Automaton

As we assume that our TS runs on a single machine, we consider a sequential execution of the test cases and we keep a global variable for recording the starting point of the current test case.

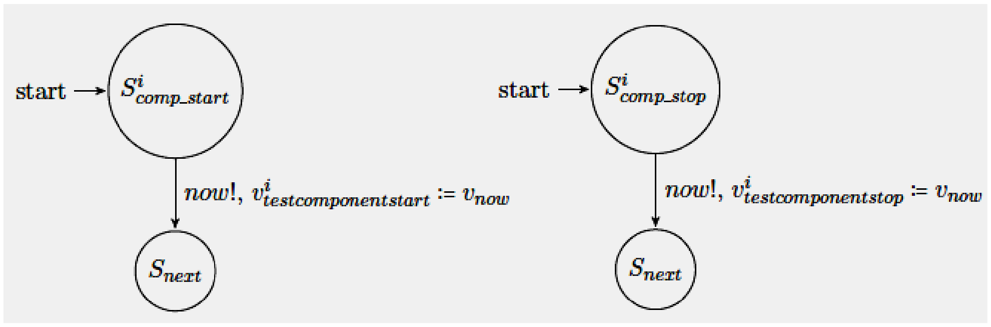
start
$S^i_{comp\_start}$
$now!,\ v^i_{testcomponentstart} := v_{now}$
S_{next}
start
$S^i_{comp\_stop}$
$now!,\ v^i_{testcomponentstop} := v_{now}$
S_{next}

Figure 6: Test Component Start & Test Component Stop Timed Automata

4.3. Semantics For Receive With Timestamp

The timed automaton from
Figure **7** in composition with the timed automaton from

Figure **8** illustrate the `receive` mechanism with automate time stamping at the message arrival. The two automata should not be regarded separately but in relationship to each other. The functionality is split between those two automata so that to increase precision of the time stamp value. The `receive` automaton presented in

Figure **7** has a functionality that should be simpler and therefore faster than the `match` automaton presented in

Figure **8**.

The `receive` automata is triggered by the receiving of a message on the input channel. When this happens, some basic actions as extracting the message and saving the current time are being performed. The signal $queue_i!$ is emitted for waking up the match automata which takes further the task of verifying whether the freshly arrived message is conforming with the template associated with that port or not. The operations performed by the `receive` automaton should be fast enough (and executed within predictable time bounds) and after they are accomplished, the `receive` automaton is back to the $S_{receive}$ state, where is free to receive other incoming messages, while the `match` automaton may continue performing checking operations which are usually much more time consuming and also more difficult to time bound due to different lengths of messages and templates.

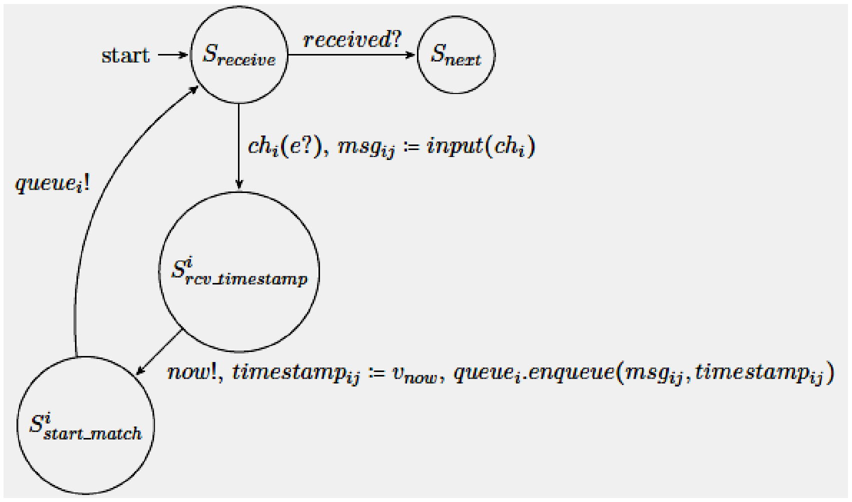
start
$S_{receive}$
$received?$
S_{next}
$ch_i(e?),\ msg_{ij} := input(ch_i)$
$queue_i!$
$S^i_{rcv\_timestamp}$
$now!,\ timestamp_{ij} := v_{now},\ queue_i.enqueue(msg_{ij}, timestamp_{ij})$
$S^i_{start\_match}$

Figure 7: Receive With Timestamp Timed Automaton

In the present context from

Figure **7** $ch_i(e?)$ means that one input event is expected on the channel ch_i. The channels are associated to ports and are indexed according to the order in which the ports are used inside a test case. If it's assumed that there are n_{ports} ports, then $i = 1..n_{ports}$.

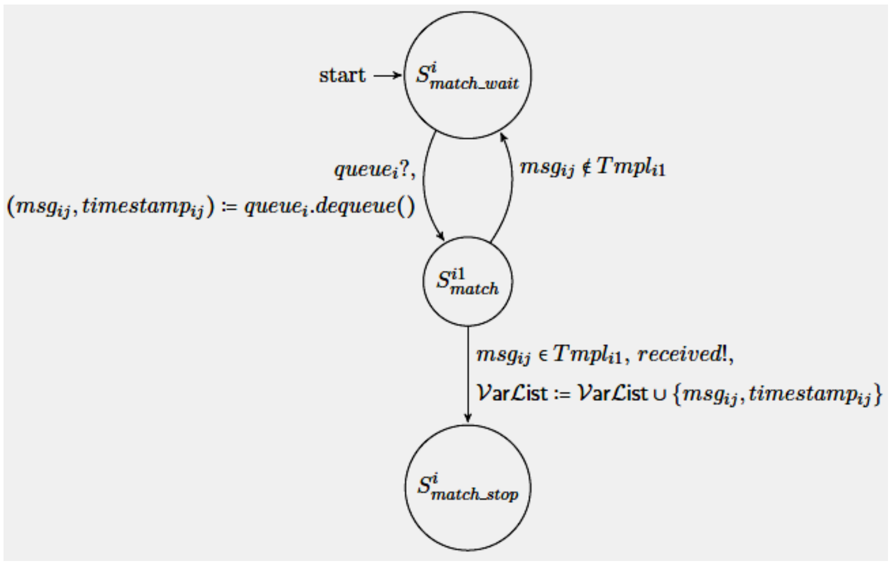
start
$S^i_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
$msg_{ij} \notin Tmpl_{i1}$
S^{i1}_{match}
$msg_{ij} \in Tmpl_{i1}$, $received!$,
$VarList := VarList \cup \{msg_{ij}, timestamp_{ij}\}$
$S^i_{match\_stop}$

Figure 8: Match Timed Automaton

$msg_{ij} = input(ch_i)$ means that when a message arrives on the channel ch_i it is extracted and saved in the variable msg_{ij}, where j indicates that this is the j-th message received on this channel, for this `receive` instruction. The timestamp associated with the arrival of msg_{ij} is saved in the variable $timestamp_{ij}$, which takes its value from the variable v_{now} after sending a time refresh request to the `now` automaton which is active in the background. Signal $queue_i!$ is being sent to the `match` automaton for indicating that a new message is available for the check. The relationship between receive and match can be described also in terms of the classical producer – consumer.

We consider that each channel ch_i has a queue $queue_i$ associated with it, where the incoming messages for that channel are going to be stored. When `match` automata is waked up by the $queue_i!$ signal, it extracts the newest message from the $queue_i$ queue and starts the comparison against the given template. We are going to see in the following sections that there can be more than one template for messages associated with each queue. But in this case, for this instruction, there is only one template to be matched. If the matching succeeds, both the value of the message and the time stamp for the message are saved into the *VarList* and the *receive*! signal is sent from the `match` automaton to the `receive` automaton to indicate that the right message was receiver and that it can move forward to the next state (see
Figure **7**). As we know, the receive statement is a blocking operation which returns only when the expected message is received. If the message could not be matched, the match automaton goes back in the waiting state until the next awakening signal (see
Figure **8**).

In the presented timed automata, $S_{receive}$ and S_{next} are symbolic states designating any generic `receive` statement and all the possible states that come next to it. Also $S^i_{rcv\_timestamp}$, $S^i_{start\_match}$, $S^i_{match\_wait}$ and $S^i_{match\_stop}$ are generic states which are indexed after the number of

port they are associated with. S_{match}^{i1} state is indexed after the number of port and the number of the template for incoming communication on the port the state is associated with.
One possible trace for `receive` automaton might look like: $(ch_i(e?) \cdot now! \cdot queue_i!)^k \cdot received?$, while one possible trace for `match` should be a complementary trace of the form: $(queue_i\,?)^k \cdot received!$, where $k \in \aleph$ represents the number of messages received until the last one matched.

4.4. Semantics For Send With Timestamp

The semantics of `send` with `timestamp` is simpler than that of the `receive` with `Timestamp` and it can be represented with a single timed automaton, as in Figure 9. The logic starts from the initial state S_{send}. After the message is sent out through the indicated port, the global time is requested using the signal $now!$ and the returned value is saved in the list of global variables. S_{send} is a generic state, indicating the initial state of any `send` operations. $S^i{}_{snd\_timestamp}$ is a generic state indicating an intermediary step for the time stamping procedure, associated with a specific port; the association between the port and the state is realized through the index i, where $i \in 1..n_{ports}$. The index j in $s\_timestamp_{ij}$, states that the j-th message is being sent from port i, where $j \in 1..n_{messages\_on\_port\_i}$. S_{next} represents a generic state, indicating the next flow of instructions.

A trace for the send automata would simple look like: $ch_i(e!) \cdot now!$.

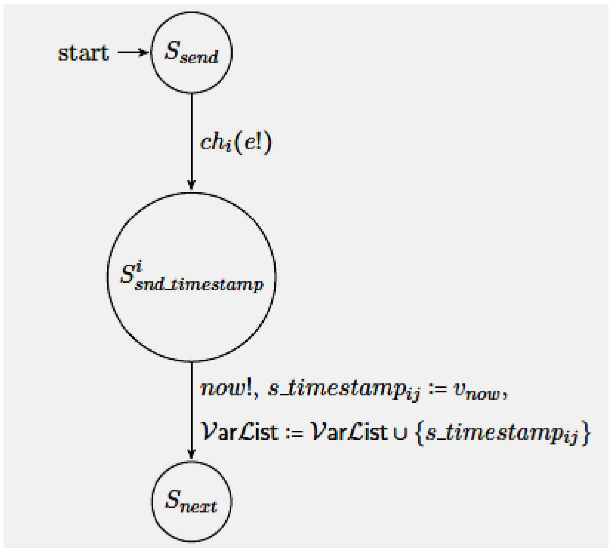
start
S_{send}
$ch_i(e!)$
$S^i_{snd\_timestamp}$
$now!,\ s\_timestamp_{ij} := v_{now},$
$\mathcal{V}ar\mathcal{L}ist := \mathcal{V}ar\mathcal{L}ist \cup \{s\_timestamp_{ij}\}$
S_{next}

Figure 9: Send With Timestamp Timed Automaton

4.5. Semantics For The Receives Which Verify Incoming Communication

The semantics of the `receive` instructions which verify incoming communication using time predicates is going to be expressed by enhancing the `receive` and `match` timed automata that were introduced in 4.3. The intention behind using time predicates in combination with the receive statement is to impose time restrictions for the arrival of messages. For verifying a real-time system is not sufficient to verify the functionality aspects, reflected in a black-box test system by the accuracy of the responses of the SUT to certain stimuli, but also the timing of the responses from the SUT. That implies that there are two matching, performed when a new message is received: message matching against a structural template and a temporal matching. The temporal matching verifies whether the time predicate associated with the receive instruction is satisfied by the time of the message receive.

The `receive` timed automaton presented in Figure 10 is similar to the `receive` timed automaton presented in
Figure **7**, excepting the S_{error} state which is newly introduced. This state is added in order to avoid the situation when a receive operation becomes blocking for an indefinite period of time. If the expected message never arrives, this blocking behavior might compromise the well-functioning of the whole test system. Introducing time restrictions for the incoming messages helps in avoiding this situation. If the time interval when a valid message is expected is overstepped, then it becomes clear that the time predicate could not be further satisfied. In this case, the `receive` returns from the waiting state and enters an error state, where the failure of the SUT can be acknowledged.

The two new `receive` and `match` timed automata from Figure 10 and
Figure **11** are also complementary to each other, in the same way that it was shown for their predecessors in
Figure **7** and
Figure **8**. Therefore, their behavior should be understood in relation to each other and together they can form as a composite timed automaton.
The `match` timed automata from
Figure **11** is also enhanced with an additional state, $S^{i1}_{match\_time}$, which performs the second matching, the time matching. As illustrated here, time matching is performed before the structural matching.

If the time does not correspond to the time predicate, then there is no reason for continuing with the structural matching. A $stop\_receive!$ signal is emitted towards the receive automaton. Then both automata are entering an error state, symbolized by the generic S_{error}.

If the time predicate is still valid then the structural matching is performed. If the message does not match, then the `match` automaton returns to the waiting state, $S^{i}_{match\_wait}$. If both the temporal match and the structural match are fulfilled, the `match` automaton signals the `receive` automaton that the expected message has just been received, by emitting a $receive!$ signal. Afterwards, the `match` automata moves to a successful terminal state, $S^{i}_{match\_stop}$.The `receive` automaton, which was blocked in the $S_{receive}$ state, moves its execution to S_{next}.

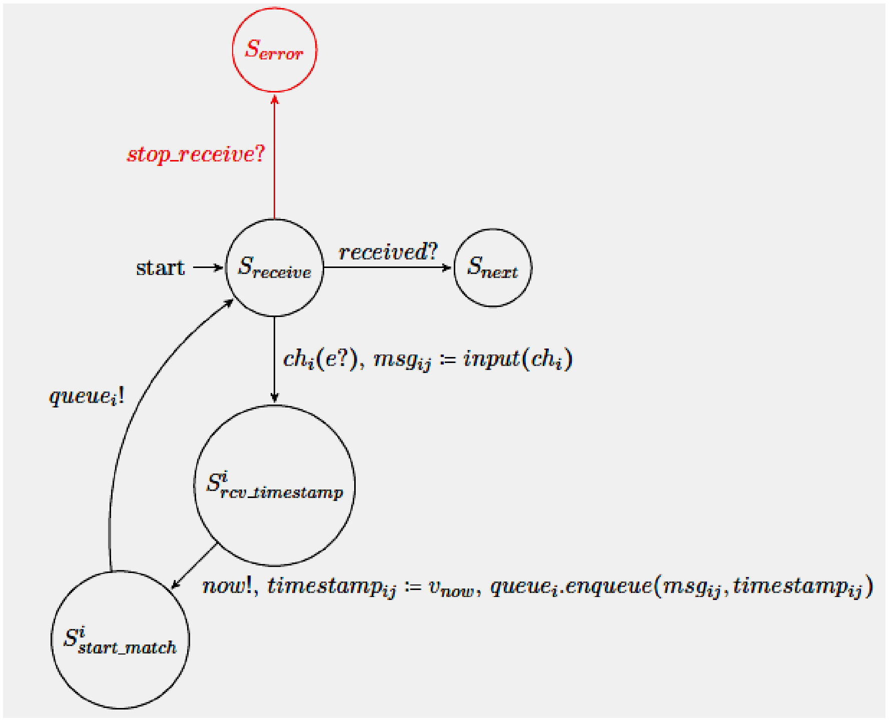

Figure 10: Receive Timed Automaton

The indexes used in both Figure 10 and
Figure **11** have the following meaning: $i \in 1..n_{ports}$ is the index of the port on which the message is expected; $j \in 1..n_{messages\_on\_port_i}$. Also, the timed automaton presented in
Figure **11** is a simplified version, with only one time predicate and only one template. This is indicated by the index 1 associated with the states $S^{i1}_{match\_time}, S^{i1}_{match}$. A receive operation, waiting on one port can have associated a set with more than one pair of the type: time predicate, template. Therefore, as a generalization, for modeling this situation one should add pairs of states $S^{i1}_{match\_time}, S^{i1}_{match}$ equal in number with the cardinal of the set. Each transition from a $S^{i}{}_{match\_wait}$ a $S^{ix}_{match\_time}$ state will be tried, listening to a priority rule.
One possible trace for the `receive` automaton from Figure 10 would be: $(ch_i(e?)now!queue_i!)^{k_1}\ received?$ and the complementary trace based on the `match` automaton would be: $(queue_i\ ?)^{k_1}\ received!$, where $k_1 \in \aleph^{>0}$ represents the number of messages received on the port and handled by this receive instruction until one of them matched. This trace indicates a situation when the SUT passed the verification that regarded both time and structure of the message.

Other possible couple of traces, this time corresponding to a failure of the SUT, might look, on the `receive` automaton side, like: $(ch_i(e?)now!queue_i!)^{k_2}\ stop\_receive?$ and on the `match` automaton side: $(queue_i\ ?)^{k_2}\ stop\_receive!$, where $k_2 \in \aleph^{\geq 0}$ represents the number of messages received on the port and handled by this receive instruction until the valid time frame for time constraints on the message receive finally expires.

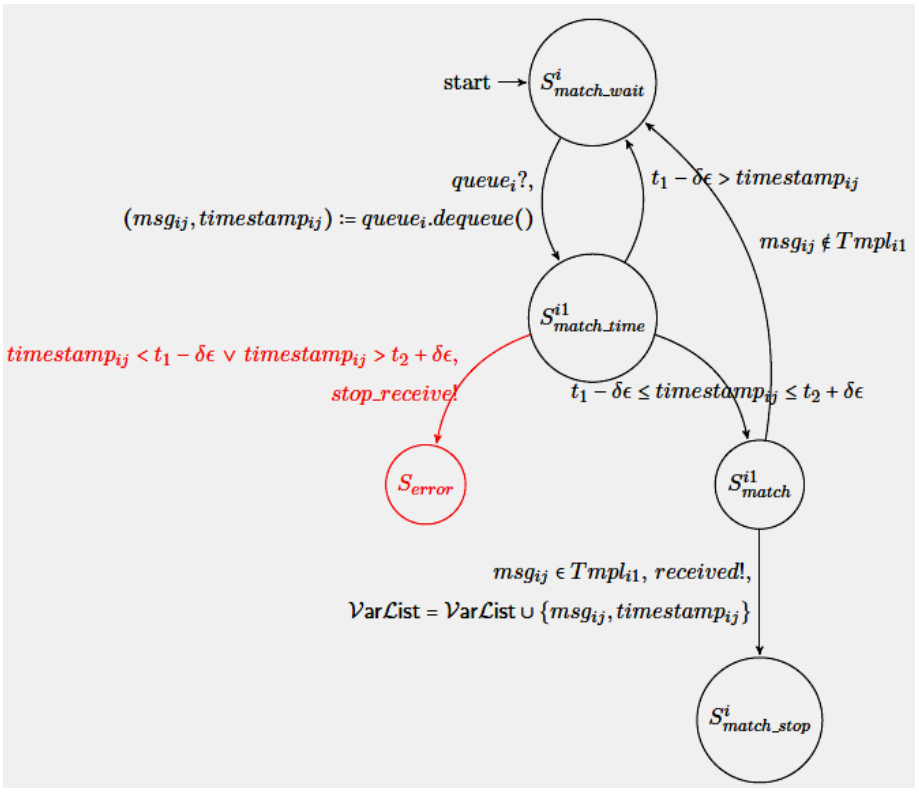
start
$S^i_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
$t_1 - \delta\epsilon > timestamp_{ij}$
$msg_{ij} \notin Tmpl_{i1}$
$S^{i1}_{match\_time}$
$timestamp_{ij} < t_1 - \delta\epsilon \vee timestamp_{ij} > t_2 + \delta\epsilon$,
$stop\_receive!$
$t_1 - \delta\epsilon \leq timestamp_{ij} \leq t_2 + \delta\epsilon$
S_{error}
S^{i1}_{match}
$msg_{ij} \in Tmpl_{i1}$, $received!$,
$VarList = VarList \cup \{msg_{ij}, timestamp_{ij}\}$
$S^i_{match\_stop}$

Figure 11: Match Timed Automaton

4.6. Semantics For Sends Which Control Outgoing Communication

For illustrating the semantics of send operation with time constraints for the outgoing of the message, the timed automaton in Figure 12 is being introduced. The `send` timed automaton starts in state S_{send} where is waiting for the time point given as parameter to the "`at` statement" to be reached. This `send` automaton has a behavior which is similar to the `wait` automaton presented in Figure 4. The time point, representing the time constraint, would be expanded to a time interval in the vicinity of the given time point, $[t_{\max} - \delta\varepsilon, t_{\max} + \delta\varepsilon]$, in order to introduce some tolerated error, inherent to the real world. If there are scheduling problems, due to overloading of the system, or other causes, and the time interval for sending the message is missed, the `send` automaton enters a terminal error state. This state indicates that the TS itself had a malfunction.

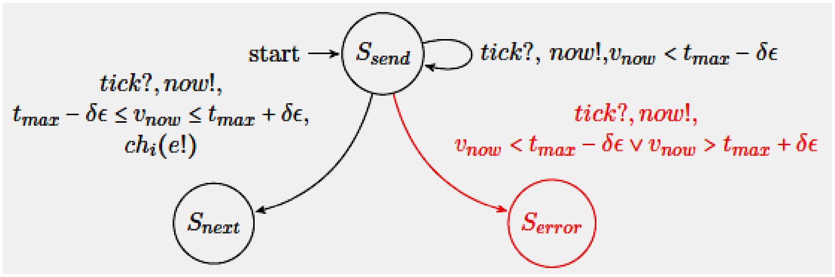
start
S_{send}
$tick?$, $now!$,$v_{now} < t_{max} - \delta\epsilon$
$tick?, now!$,
$t_{max} - \delta\epsilon \leq v_{now} \leq t_{max} + \delta\epsilon$,
$ch_i(e!)$
$tick?, now!$,
$v_{now} < t_{max} - \delta\epsilon \vee v_{now} > t_{max} + \delta\epsilon$
S_{next}
S_{error}

Figure 12: Send With Time Constraint Timed Automaton

One possible trace for the send with time constraints timed automata would look like:

$(tick?\cdot now!)^k \cdot ch_i(e)!$, where $k \in \aleph$ is a natural number indicating the number of clock ticks that passed before sending the message.

Other possible trace, this time indicating an error would be: $tick?\cdot now!$ which indicates that the specified moment of time had already expired.

4.7. Semantics For Alt Blocks Which Control Incoming Communication

The semantics of an alt statement is more complex than the semantics of the other instructions which extensions were presented in this paper. It involves the collaboration of the three timed automata that are illustrated here in
Figure **13**,
Figure **14** and
Figure **15**.

Due to its complexity, in
Figure **13** is shown only an excerpt from the `alt` automaton. The excerpt contains the semantics associated with one representative branch of a generic `alt` statement. We assume that the generic `alt` statement considered here has a number of a_n receiving branches. Each receiving branch is waiting for input on one specific port of the TS .

We consider to have a function $B : Queues \times Guards \times Templates \rightarrow Branches_{Alt}$ that represents a bijective relation, associating one queue with time guards and templates to each branch of the alternative. $B(queue_i, TP_{il}, Tmpl_{il}) = branch_j$,
where $i = 1..n_{ports}, l = 1..n_{tmpl_i}, j = 1..|Branches_{Alt}| = a_n$. In
Figure **13**, representing the main functionality of an alternative, we can see that the automaton is able to receive input messages on different channels or ports. The flow corresponding to a `receive` on a port is similar for the `receive` automaton that was already presented in Figure 10. If one message arrives while inside an alternative, the transition associated with the channel on which the message is received is going to be taken. This will lead to a state that will take the time stamp for the message arrival and then, the associated match automaton for that channel will be woken up with the corresponding signal, $queue_i$. Then the automaton enters back in the listening state. S_{alt} state might be regarded as the state where the automata is listening to all the ports on which it expects to receive messages inside the alternative.
Once received and time-stamped, the message is passed to the extended `match` automaton, to be verified whether it respects the temporal and structural constraints associated with that port. The `match` automaton defined here, represents a generalization for the automaton presented in
Figure **11**. In an `alt` statement one port might be used in more than one `receive` branches. On each of the receive branches, there might be temporal and structural constraints expressed as time predicates and message templates. We consider one `match` automaton to be responsible for verifying all the constraints associated with the port correspondent to that automaton. The matching will be performed in the order in which the `receive` branch is encountered inside the `alt` statement, from top to the bottom. The first (l^{th}) matching state with the pair - time constraint (TP_{ij}) and the structural constraint $(Tmpl_{ij})$ - that is satisfied by the received message is going to trigger the $receive_{il}!$ signal to indicate to the alt automata that one of the branches was satisfied. This is a success scenario when the `alt` is satisfied, the extended `match` automaton reaches a successful terminal state and the execution of the test system moves to the next state.

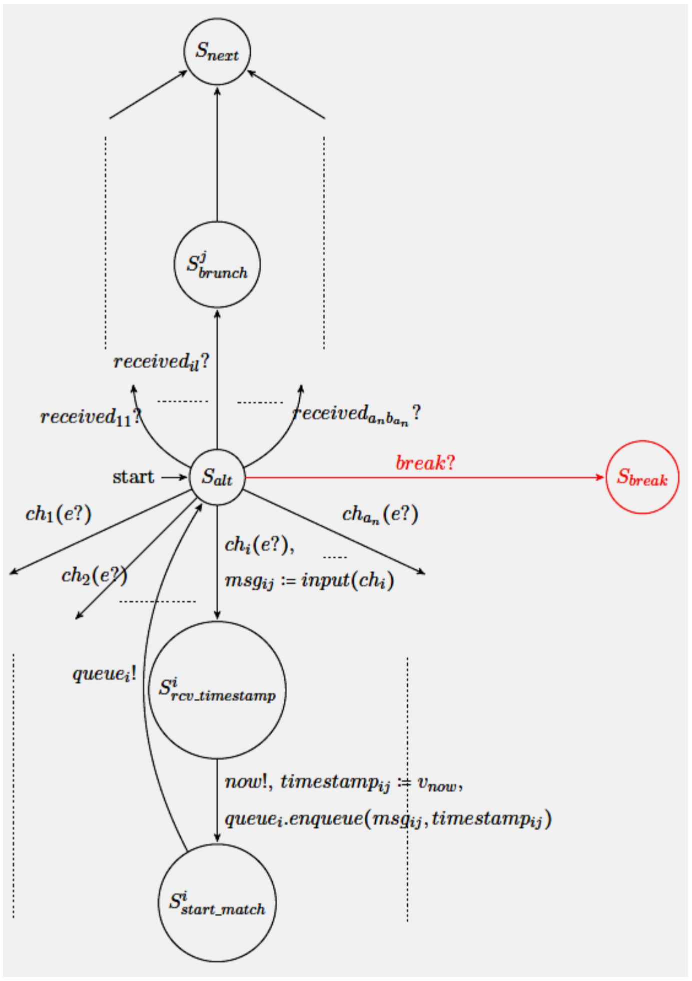

Figure 13: Alt Block Timed Automaton

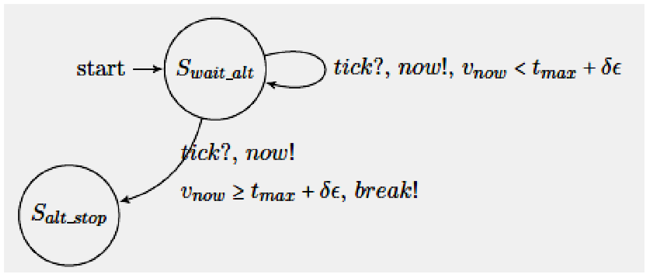
start
S_{wait_alt}
$tick?$, $now!$, $v_{now} < t_{max} + \delta\epsilon$
$tick?$, $now!$
$v_{now} \geq t_{max} + \delta\epsilon$, $break!$
S_{alt_stop}

Figure 14: Timer For The Alt State

If the `match` is not successful, despite of trying all the constraints associated with the port, the extended `match` automaton associated with that port moves into a waiting state. If none of the branches of the alt statement was satisfied in the requited amount of time indicated by the parameter of the `alt` statement, then the execution of the `alt` automaton is interrupted by a signal given by the `wait_alt` automaton. This signalizes that an alternative error behavior should be run in this situation. This alternative behavior is the one associated with the `break` instruction.

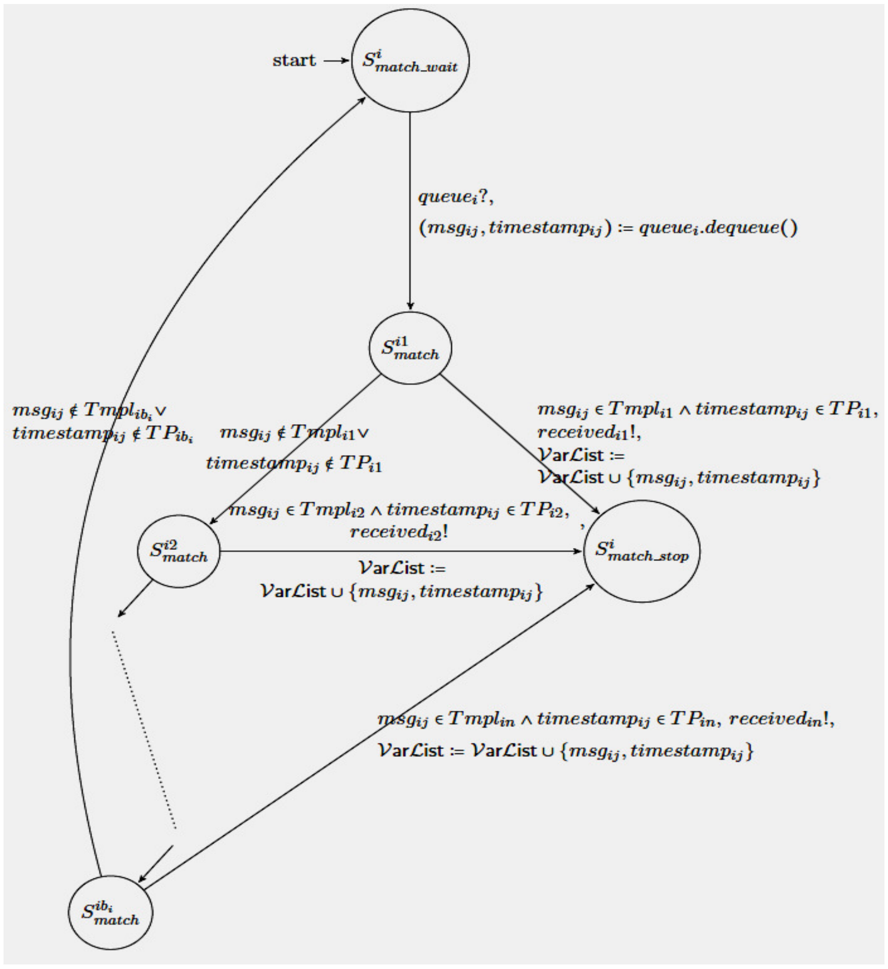
start
$S^i_{match\_wait}$
$queue_i?$,
$(msg_{ij}, timestamp_{ij}) := queue_i.dequeue()$
S^{i1}_{match}
$msg_{ij} \notin Tmpl_{ib_i} \vee$
$timestamp_{ij} \notin TP_{ib_i}$
$msg_{ij} \notin Tmpl_{i1} \vee$
$timestamp_{ij} \notin TP_{i1}$
$msg_{ij} \in Tmpl_{i1} \wedge timestamp_{ij} \in TP_{i1}$,
$received_{i1}!$,
$VarList :=$
$VarList \cup \{msg_{ij}, timestamp_{ij}\}$
$msg_{ij} \in Tmpl_{i2} \wedge timestamp_{ij} \in TP_{i2}$,
$received_{i2}!$
$VarList :=$
$VarList \cup \{msg_{ij}, timestamp_{ij}\}$
S^{i2}_{match}
$S^i_{match\_stop}$
$msg_{ij} \in Tmpl_{in} \wedge timestamp_{ij} \in TP_{in}$, $received_{in}!$,
$VarList := VarList \cup \{msg_{ij}, timestamp_{ij}\}$
$S^{ib_i}_{match}$

Figure 15: Extended Match Timed Automaton

One possible trace for the `alt` automaton will be:

$$((ch_1 ?\cdot now!\cdot queue_1!)^* \cdot (ch_2 ?\cdot now!\cdot queue_2!)^* \cdot \ldots \cdot (ch_i ?\cdot now!\cdot queue_i!)^* \cdot \ldots \cdot (ch_{a_n} ?\cdot now!queue_{a_n}!)^*)^* received_{il} ?$$

, with the complementary trace for the extended `match` automaton:

$$(queue_1 ?^* \cdot queue_2 ?^* \cdot \ldots \cdot queue_i ?^* \cdot \ldots \cdot queue_{a_n} ?^*)^* \cdot received_{il}!,$$

And the complementary trace for the `wait_alt` automaton:

$$(tick ?\cdot now!)^* \cdot break!'$$

5. CONCLUSIONS

A standardized testing language is needed for providing the possibility of describing tests in an easy way, a way that is used and understood without difficulty among different stakeholder in the industry. We chose TTCN-3 as a well-designed testing language with a high degree of popularity and usage in the real world. But even though perfectly suitable for attesting conformance and performance, it was proved that the language lacks methods of expressing temporal requirements.

One of the big challenges of this work was to identify precisely the aspects regarding the real-time testing where TTCN-3 was not expressive enough and to cover these situations, by introducing a minimal set of concepts that will not burden, but enhance the language.

The highlight of this work is that it presents each new concept endowed with a clear semantic defined by a well-constructed mathematical formalism. Timed automata were used to model the introduced concepts and the test system itself. The semantic developed here establishes mappings between TTCN-3 abstract test specifications and timed automata, the conversion being made possible back and forth. This approach opens also new interesting possibilities, as, for example, semantic validation of the timed TTCN-3 code, based on model-checking methods developed for timed automata. Further on, based on the description of behaviour in the previously defined formalism, the mapping to real-time operating system's mechanisms can be realized.

As the semantics of the new real-time features added to the TTCN-3 language is expressed using timed automata, an interesting idea would be to implement a translator from a real-time test specification to a network of timed automata. That would open the possibility of applying model-checking verification techniques to semantically validate the real-time test specification. If an automatic translation from TTCN-3 to timed automata can be performed, the generated timed automata model can be used as input for an already existing verification tool, as Uppaal [9] for example, that can be used to perform this checking.

Real-time testing is a hot topic now days and has a huge potential of applicability in a wide range of domains. This work brings its contribution in the field of standardized and automatized testing for real-time by defining a thorough methodology and a specialized set of instruments and examples for building a framework for this type of testing. Thus, the goal presented in the starting chapter of this thesis was achieved, but the greatest aim of this work is to be continued, extended and, most importantly, applied, in all types of industrial situations.

REFERENCES

[1] D. S. a. I. S. Juergen Grossmann, "Testing embedded real time systems with TTCN-3," in International Conference on Software Testing Verification and Validation, Washington DC, USA, 2009.

[2] E. E. 2. 8.-1. V3.2.1, "Methods for Testing and Specification (MTS). The Testing and Test Control Notation Version 3, Part 1: TTCN-3 Core Language," European Telecommunications Standards Institute, Sophia Antipolis, France, 2007.

[3] R. Sinnott, "Towards More Accurate Real-Time Testing," in The 12th International Conference on Information Systems Analysis and Synthesis (ISAS 2006), Orlando, Florida, 2006.

[4] R. A. a. D. L. Dill, "A Theory of Timed Automata," Theor. Comput. Sci., 1994.

[5] E. E. 2. 782, "TTCN-3 Language Extensions: TTCN-3 Performance and Real Time Testing," European Telecommunications Standards Institute, Sophia Antipolis, France, 2010.

[6] V. M. G. D. I. S. a. I. R. Diana Alina Serbanescu, "Real-Time Testing with TTCN-3," in TestCom/Fates, Tokyo, Japan, 2008.

[7] D. S. P. D. I. S. Jürgen Grossmann, "Test Specification and Test Methodology for Embedded Systems in Automobiles," in AutoTest, Forschungsinstitut für Kraftfahrwesen und Fahrzeugmotoren Stuttgart (FKFS), Stuttgart, Germany, 2008.

[8] D. A. a. S. I. Serbanescu, "Testing Environment for Real-Time Communications Intensive Systems," in INTENSIVE , Cancun, Mexico, 2010.

[9] K. G. L. a. P. P. a. W. Yi, "UPPAAL in a nutshell," Int. Journal on Software Tools for Technology Transfer, vol. 1, pp. 134--152, 1997.

[10] M. K. a. S. Tripakis, "Real-time Testing with Timed Automata Testers and Coverage Criteria," Verimag Technical Report, 2004.

[11] S. Voget, "Future Trends in Software Architectures for Automotive Systems," Microsystems for Automotive Applications, Berlin, Germany, 2003.

[12] M. Broy, "Challenges in automotive software engineering," in ICSE '06: Proceedings of the 28th international conference on Software engineering, Shanghai, China, 2006.

Authors

Dipl.-Ing. Diana Alina Serbanescu
Short Biography:

Dipl. -Ing. Diana Alina Serbanescu studied computer science at “Poitehnica” University of Bucharest. She completed her reasearch for her final dissertation for the engineering degree at the FOKUS Fraunhofer Institute for Open Communication Systems in Berlin, in 2005. Ever since she continued working by the FOKUS institute as researcher with her main interests in testing, TTCN-3, UML modeling, real-time and embedded systems. She is currently working at her PhD with the title “Real-time Testing Framework For Embedded Systems”.

Prof. Dr. Ina Schieferdecker
Short Biography:

Prof. Dr.-Ing. Ina Schieferdecker studied Mathematical Computer Science at the Humboldt University Berlin and did her PhD at the Technical University Berlin about performance-extended specifications and analysis of QoS characteristics in 1994. From 1997 to 2004, she was Head of Competence Center for Testing, Interoperability and Performance (TIP) and is currently Head of the Competence Center for Modeling and Testing (MOTION) at the Fraunhofer Institute for Open Communication Systems (FOKUS) in Berlin