# On Time Actors

Marjan Sirjani[1] and Ehsan Khamespanah[1,2]

[1] School of Computer Science, Reykjavik University - Iceland
[2] School of Electrical and Computer Engineering, University of Tehran - Iran

**Abstract.** Actor model is a concurrent object-based computational model in which actors are the units of concurrency and communicate via asynchronous message passing. Timed Rebeca is an actor-based modeling language which is designed for modeling and analyzing of event-based and asynchronous systems with time constraints. Timed Rebeca is equipped with analysis techniques based on the standard semantics of timed systems, and also an innovative event-based semantics that is tailored for timed actor models. The developed techniques are applied on different applications using Afra toolset, the integrated development environment of Timed Rebeca. This paper is a survey on the published work on Timed Rebeca, its semantics, supporting tools, and applications.

**Keywords:** Actor model, Timed Rebeca, Verification, Reduction Technique, Floating Time Transition System

## Foreword

Frank is fun and frustration! This is what I told him 13 years ago and I still can hold to it. He is full of positive energy, and he works hard, although he has a rule: it's stupid to work on a sunny day! Frank is always full of great ideas, half of those I never understood! I listened to his Hoare logic presentation three times, without much success. He is a fantastic leader. He has the reasoning mind of a logician, and the wit of a philosopher, while he can understand Java like an experienced programmer.

We started working on timed actors in 2006, and we had our first paper on Timed Rebeca at the Nordic Workshop on Programming Theory in 2007. So, ten years have passed ... but if we look at our relative age, then nothing has changed. Congratulations Frank! For your 60th birthday! Wish you yet more success and happiness for the next 60 years to come!

## 1 Introduction

Modeling is crucial, both in science and engineering. We build models to be able to do analysis without having to deal with the details of a system's implementation. Edward Lee [1] emphasizes on the difference between engineers and scientists when they build and use a model. Engineers build a model to explore the design space and construct a system based on the model; and scientists build

a model of an existing system to be able to analyze it. So, engineers do their best to build the system just like the model, and scientists do their best to build the model similar to the existing system. No matter we use a model as an engineer or a scientist, we need to have a faithful model in order to perform a valid analysis and/or design exploration.

We may hear the following question, mostly in more theoretical communities: "why yet another modeling language?" This question is usually asked if you mainly focus on the expressibility of the modeling languages. But usability and fidelity are also two crucial features of a modeling language, and their importance is very well acknowledged from a more practical point of view. Models need to be able to capture the characteristics of the system which affect the properties of our interest (fidelity), and we need to be able to understand and build a model with the least possible effort (usability). For example, object-oriented approaches were introduced with the philosophy of reducing the semantic gap between the real world problems and the models representing those problems; and their success is undeniable. With the growing need for various software applications, and fast changes in hardware and network infrastructures, the answer to the above question is simple: because we are not there yet! And with "change" being the only constant in our software world, we will possibly never be there!

The non-functional properties of different nature are becoming more crucial in correctness of a software system, demanding for new models and/or extensions of existing languages. Timing features are no more just performance concerns. In many software systems, nowadays, timing features are part of correctness properties. So, the so called non-functional properties are becoming first-class characteristics of a system like the functional ones.

The modeling language Rebeca (*Reactive Objects Language*) [2, 3], is an operational interpretation of the actor model [4, 5] provided with formal semantics and supported by model checking tools [6]. Rebeca is designed to be a usable and analyzable modeling language to bridge the gap between software engineers and formal method community. The application domain targeted by Rebeca is where we have event-driven systems, with asynchronous message passing. In Rebeca, we have non-blocking sends, no explicit receive, no shared variables, and non-preemptive method execution.

In this paper, we will provide a brief overview on Time Rebeca [7, 8], the timed extension of Rebeca which is much praised by Frank de Boer. In the following sections, we will show how Floating Time Transition System is a natural semantics for event-based actor languages based on the work of [9]. Then we will have a short survey on state-based model checking of Timed Rebeca based on [10] and [11]. Finally, we will conclude by showing how Timed Rebeca is used for analysis and design exploration in real world case studies which were studied in [12] and [13].

## 2 Timed Rebeca

Timed Rebeca [8, 7] is an extension of Rebeca [2, 3] with time features for modeling and verification of time-critical systems. These primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events*. In a Timed Rebeca model, each actor has its own local clock. The local clocks can be considered as synchronized distributed clocks. Methods are still executed atomically like in Rebeca, however passing of time while executing a method can be modeled. In addition, instead of having a queue for the messages, there is a bag of messages where messages are stored together with their time tags. The time tag of a message represents the time that the message arrived in the bag and can be taken to be served. The model is based on discrete events and discrete time.

$$
\begin{array}{rl}
\mathit{Model} ::= & \mathit{Class}^{*}\ \mathit{Main} \\
\mathit{Main} ::= & \textbf{main}\ \{\ \mathit{InstanceDcl}^{*}\ \} \\
\mathit{InstanceDcl} ::= & \mathit{className}\ \mathit{rebecName}(\langle \mathit{rebecName} \rangle^{*}) : (\langle \mathit{literal} \rangle^{*}); \\
\mathit{Class} ::= & \textbf{reactiveclass}\ \mathit{className}\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{MsgSrv}^{*}\ \} \\
\mathit{KnownRebecs} ::= & \textbf{knownrebecs}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{Vars} ::= & \textbf{statevars}\ \{\ \mathit{VarDcl}^{*}\ \} \\
\mathit{VarDcl} ::= & \mathit{type}\ \langle v \rangle^{+}; \\
\mathit{MsgSrv} ::= & \textbf{msgsrv}\ \mathit{methodName}(\langle \mathit{type}\ v \rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \} \\
\mathit{Stmt} ::= & v = e;\ |\ v = ?(e, \langle e \rangle^{+});\ |\ \mathit{Call};\ |\ \mathit{if}\ (e)\ \{\ \mathit{Stmt}^{*}\ \}[\mathit{else}\ \{\ \mathit{Stmt}^{*}\ \}]\ | \\
& \textbf{delay}(t); \\
\mathit{Call} ::= & \mathit{rebecName}.\mathit{methodName}(\langle e \rangle^{*})\ [\textbf{after}(t)]\ [\textbf{deadline}(t)]
\end{array}
$$

**Fig. 1.** Abstract syntax of Timed Rebeca (from [9]). Angled brackets $\langle ... \rangle$ are used as meta parenthesis, superscript $+$ for repetition at least once, superscript $*$ for repetition zero or more times, whereas using $\langle ... \rangle$ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. Identifiers *className*, *rebecName*, *methodName*, $v$, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and $e$ denotes an (arithmetic, boolean or nondetermistic choice) expression.

Two major semantics are considered for Timed Rebeca: Floating Time Transition System (FTTS) [9] which is a natural event-based semantics for actors, and Timed Transition System (TTS) which is a standard state-based semantics for timed models. In the FTTS semantics, in each state, the local time of each actor

can be different from the others, i.e., the execution of actors is not synchronized over their local times. In the TTS semantics the local time of all actors is the same. Note that when we talk about synchronized local clocks we are explaining the concept of time in the model, while TTS semantics respects this synchrony, in FTTS we relax the time synchronization constraint. Comparing to TTS, FTTS can be considered as a reduced state transition system where the event-based properties are preserved. A more detailed description is in Section 3. The syntax of Timed Rebeca is shown in Figure 1 and we illustrate Timed Rebeca language constructs using a simple Network on Chip (NoC) example in Figure 2 [12, 14]. NoC is a promising architecture paradigm for many-core systems. As an example of a NoC, we modeled and analyzed ASPIN (Asynchronous Scalable Packet switching Integrated Network), which is a fully asynchronous two-dimensional NoC design with XY routing algorithm [15]. In the two-dimensional NoC design, each node has four adjacent nodes and four internal buffers for storing the incoming packets (one for each direction). Using XY routing algorithm, packets are moving along X direction first, then along Y direction, to reach their destination nodes. In ASPIN, packets are transferred through channels, using four-phase handshake communication protocol. The protocol uses two signals, namely *Req* and *Ack*, to implement four-phase handshaking protocol. This way, to transfer a packet, first the sender sends a request by raising *Req* signal, and waits for an acknowledgment which is the raising of *Ack* signal by the receiver. In the third phase, data is sent. Finally, after a successful communication all of the signals return to zero.

A Timed Rebeca model consists of a number of *reactive classes*, each describing the type of a certain number of *actors* (called *rebecs* in Timed Rebeca)[3]. As shown in Figure 2, two different reactive classes, `Manager` and `Router`, are developed in the NoC model. `Manager` is the traffic generator of this model and `Router` is the model of a node in an ASPIN design. The local state of each actor is defined by the contents of its message bag and the values of its state variables. A composite id, using X-Y position (line 12), and buffer variables which show that the buffers are enable or busy (lines 13 and 14) are state variables of a Router, defined in a `statevars` block. `Manager` does not have any state variables in this model. The communication in Timed Rebeca takes place by asynchronous message passing among actors. Each actor has a set of *known rebecs* to which it can send messages. `Manager`, as the traffic generator of the model, may send message to any of the nodes; so, all the routers from `r00` to `r33` are its known rebecs. Contrarily, a router is only allowed to communicates with its neighbors, named `North`, `East`, `South`, and `West` (line 9). The actors instantiation and binding the known rebecs of actors are in the `main` block (lines 22-28). In this NoC model, a mesh of 16 `Routers` is created and known rebecs are set based on the topology of the mesh (e.g., as shown in line 25, router `r13` is the north neighbor of `r10`, router `r20` is the east neighbor of `r10`, router `r11` is the south neighbor of `r10`, and router `r00` is the west neighbor of `r10`).

[3] In this paper we use rebec and actor interchangeably.

The same as other actor models, reactive classes of Timed Rebeca declare the messages to which they can respond, defining *message servers*. As shown in Figure 2, `Manager` has only `generateTraffic` message server and `Router` has four different message servers, `init`, `getAck`, `reqSend`, and `giveAck` (lines 17-20). The definition of a message server is the same as the definition of class methods of Java except that it starts with `msgsrv` keyword and it does not have return value. To develop the Timed Rebeca model of ASPIN, four phase handshaking protocol is modeled using three message servers: `reqSend`, `giveAck`, and `getAck`. A router calls its `reqSend` message server to route a packet to its neighbors. A part of `reqSend` and `giveAck` message servers is shown in Figure 3.

```
1  reactiveclass Manager {
2    knownrebecs {
3      Router r00, r10, ..., r33;
4    }
5    msgsrv generateTraffic() { ... }
6  }
7  reactiveclass Router {
8    knownrebecs {
9      Router North, East, South, West;
10   }
11   statevars {
12     byte Xid, Yid;
13     byte[4] bufNum;
14     boolean[4] full, enable, outMutex;
15   }
16   Router(byte X, byte Y) { ... }
17   msgsrv init() { ... }
18   msgsrv getAck() { ... }
19   msgsrv reqSend(byte Xtarget, byte Ytarget) { ... }
20   msgsrv giveAck(byte Xtarget, byte Ytarget) { ... }
21 }
22 main {
23   Manager m(r00,r10, ... ,r33): ();
24   Router r00(r03,r10,r01,r30): (0,0);
25   Router r10(r13,r20,r11,r00): (1,0);
26   ...
27   Router r33(r32,r03,r30,r23): (3,3);
28 }
```

**Fig. 2.** The Timed Rebeca model of ASPIN Network on Chip

As shown in Figure 3, an actor can change its state variables through assignment statements (lines 6 and 7), make decisions through conditional statements

(line 2), and communicate with other actor by sending messages (line 5). Recurrent and periodic behavior can be modeled by actors sending messages to themselves (line 9). Timed Rebeca adds three primitives to Rebeca to address timing issues: *delay*, *deadline* and *after*. A *delay* statement models the passing of time of an actor during the execution of a message server (line 12). Note that all other statements are assumed to execute instantaneously. The keywords *after* and *deadline* can be used in conjunction with a method call. The term `after` $n$ indicates that it takes $n$ units of time for the message to be delivered to its receiver (line 5). The ordering of messages in a message bag is based on the delivery times of messages. Each actor takes the first message from its message bag (the message with the earliest time tag), executes the corresponding message server, and then takes the next message (or waits for the next message to arrive), and so on. Messages are executed in a non-preemptive way (atomically). The term `deadline` $n$ is used to show that if its related message is not taken in $n$ units of time, it will be purged from the receiver's message bag automatically (line 21).

```
1  msgsrv reqSend(byte Xtarget, byte Ytarget) {
2    if(Xtarget > Xid) {
3      byte leavingDirection = ...;
4      if(outMutex[leavingDirection]) {
5        East.giveAck(Xtarget, Ytarget) after(50);
6        outMutex[leavingDirection] = false;
7        enable[leavingDirection] = false;
8      } else
9        self.reqSend(Xtarget, Ytarget) after(100);
10   } else if(Xtarget < Xid) { ... }
11   ...
12   delay(50);
13 }
14
15 msgsrv giveAck(byte Xtarget, byte Ytarget) {
16   if(Xtarget == Xid && Ytarget == Yid) {
17     //Consume the packet
18   } else if(!(Xtarget == Xid && Ytarget == Yid)) {
19     byte enteranceDirection = ...;
20     bufNum[enteranceDirection]++;
21     ((Router)sender).getAck() deadline (50);
22     self.reqSend(Xtarget, Ytarget) after(100);
23   }
24 }
```

**Fig. 3.** The body of two message servers of ASPIN model

The XY-routing algorithm is implemented inside `reqSend` (lines 1-13). lines 2 to 9 shows that how a packet is routed to its east neighbor. If the packet must be sent to the router's east neighbor (line 3) and its east outgoing buffer is free (line 4), message `giveAck` is sent to the east neighbor and the internal state of the sender router is changed to the condition after sending a message. Upon processing `giveAck`, first the destination address of the newly received packet is checked and the packet is consumed if its target address is set to that node. Otherwise, the packet is stored in the buffer of the receiver (line 20), acknowledgment is sent to the sender router by sending `getAck` message (line 21), and message `reqSend` is sent to itself to route the newly received packet toward its destination (line 22).

## 3 Event-based Semantics: Floating Time Transition System

FTTS is defined in [16] as the natural semantics of Timed Rebeca. FTTS exploits the key features of actor models to generate very compact state transition systems. Having single threaded actors, with no shared variables, and no blocking send or receive, along with non-preemptive execution of each message server, ensures that the execution of a message server does not interfere with the execution of message servers of other actors. Therefore, all the statements of a given message server of an actor can be executed in isolation (even delay statements) during a single transition without considering the behavior of the other actors. This way, after performing a transition from one state to another state, different actors may be in different local times. The way the states of FTTS are generated handles the differences between the local times of actors. Such a semantics is reasonable when one is only interested in the order of the events of a model. FTTS may not be appropriate for reasoning about the synchronized global states of an actor model [9].

The operational semantics of a Timed Rebeca model $\mathcal{M}$ with the set of actors $\mathcal{I}$ is defined as Floating Time Transition System $FTTS = (S, s_0, Act\ , \hookrightarrow)$ where $S$ is the set of states, $s_0$ is the initial state, $Act$ is the set of actions, and $\hookrightarrow$ is the transition relation, as described below (from [17]).

**States.** A state $s \in S$ of the Timed Rebeca model $\mathcal{M}$ consists of the local states of actors plus their current time. The local state of an actor $i$ in (the global) state $s$ is the pair of the valuation of its state variables (shown by $V_{s,i}$) and the bag of its received messages (shown by $B_{s,i}$). The local time of the actor $i$ is denoted as $now_{s,i}$. So, the state $s \in S$ is defined as $s = \prod_{i \in \mathcal{I}} (V_{s,i}, B_{s,i}, now_{s,i})$.

**Initial State.** In the initial state $s_0$ of the Timed Rebeca model $\mathcal{M}$, the state variables of the actors are set to their initial values (according to their types), the `initial` message is put in the bag of actors (their arrival times are set to zero), and the current times of all the actors are set to zero.

**Actions.** There is only one kind of action in FTTS, which is taking a message from the message bag and executing the corresponding message server entirely.

The message $tmsg$ is denoted by a tuple $((sid, rid, mid), ar, dl)$ where $sid$ is the id of its sender actor, $rid$ is the id of the receiver actor, $mid$ is the id of its corresponding message server, $ar$ is its arrival time, and $dl$ is its deadline. This way, the set of actions, $Act$, is defined as $Act = \bigcup_{i \in \mathcal{I}} ((\mathcal{I} \times \{i\} \times \mathcal{M}_i) \times \mathbb{N} \times \mathbb{N})$ where $\mathcal{M}_i$ is the set of message servers of actor $i$.

**Transition Relations.** We first define the notion of *release time* of a message. An actor $a_i$ in a state $s \in S$ has a number of timed messages in its bag. The release time of the message $tmsg = ((sid, rid, mid), ar, dl) \in B_{s,i}$ is defined as $rt_{tmsg} = \max(now_{s,i}, ar)$ (Note that $ar < now_{s,i}$ means that $tmsg$ has arrived at some time when $a_i$ has been busy executing another message server. Hence, $tmsg$ is ready to be processed at $now_{s,i}$). Consequently, the set of enabled messages of actor $a_i$ in state $s$ is $E_{s,i} = \{tmsg \in B_{s,i} \mid \forall tmsg' \in B_{s,i} \cdot rt_{tmsg} \leq rt_{tmsg'}\}$ which are the messages with the minimal release time. For a set of enabled messages $E_{s,i}$, enabling time $ET_{s,i}$ is defined as the release time of members of $E_{s,i}$.

Now we define the transition relation $\hookrightarrow \subset S \times Act \times S$ such that for every pair of states $s, t \in S$, we have $(s, tmsg, t) \in \hookrightarrow$ for every $tmsg \in E_{s,i} \wedge (\forall j \in \mathcal{I} \cdot ET_{s,i} \leq ET_{s,j})$. All the transitions of FTTS are called taking-event transitions and as a result of a taking-event transition labeled with $tmsg$, $tmsg$ is extracted from the bag of $a_i$, the local time of $a_i$ is set to $ET_{s,i}$, and all the statements in the message server corresponding to $tmsg$ are executed sequentially. Here, $a_i$ is called *enabled actor*. The effect of executing non-delay statements is changing the state variables of $a_i$ and sending some messages to $a_i$ or other actors. The effect of executing a delay statement with parameter $d \in \mathbb{N}$ is increasing the local time of $a_i$ by $d$ units of time.

```
1  reactiveclass Ping(3) {
2    knownrebecs { Pong po; }
3    Ping() {
4      self.ping();
5    }
6    msgsrv ping() {
7      po.pong() after(1);
8      delay(2);
9    }
10 }
11
```

```
12 reactiveclass Pong(3) {
13   knownrebecs { Ping pi; }
14   msgsrv pong() {
15     pi.ping() after (1);
16     delay(1);
17   }
18 }
19 main {
20   Ping ping(pong):();
21   Pong pong(ping):();
22 }
```

**Fig. 4.** The Timed Rebeca model of ping pong example

To illustrate how FTTS is created for a Timed Rebeca model, we prepared a very simple model in Figure 4, the *ping pong* example. In this example, there are two actors, `Ping` and `Pong`, which send messages to each other periodically. Without loss of generality, we assumed that the actors of this model do not

have state variables. Figure 5 shows the beginning part of the FTTS of the ping pong example. The first enabled actor of the model is `Ping` (its constructor puts message `ping` in its bag, line 4), so the first executed message is `ping`. As shown in the detailed contents of the second state (the gray block), execution of the message `ping`, actor `Ping` is at time 2 (because of executing the `delay` statement in line 8); however, actor `Pong` is at time 0 as it does not execute any messages. Also, message `pong` is put in the bag of actor `Pong` which its release time is 1 (because of the value of `after` in line 7). The deadline of this message is $\infty$ as no specific value is set as the deadline of this message in line 7.

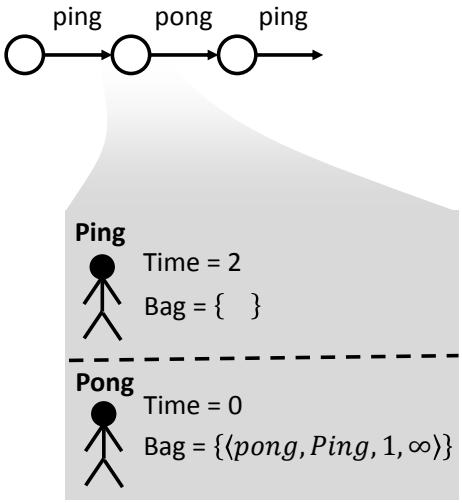


**Fig. 5.** The beginning part of the FTTS of ping pong example

There is no explicit reset operator for the time in Timed Rebeca; so, progress of time results in an infinite number of states in FTTS. However, Timed Rebeca models are models of reactive systems which generally show periodic or recurrent behaviors. It means that, if the absolute time of the states are ignored, usually finite number of untimed traces are generated for Timed Rebeca models. Based on this fact, in [9] we presented a new notion for equivalence relation between two states to make FTTS finite, called *shift equivalence relation*. In the shift equivalence relation two states are equivalent if and only if they are the same except for the value of parts which are related to the time (i.e. the local times of actors, the arrival times of messages, and the deadlines of messages) and shifting the value of parts which are related to the time in one state makes it the same as the other one. This way, instead of preserving absolute value of time, only the relative difference of timing parts of states are preserved. As discussed in [9], for most systems of interest, shift equivalence relation succeeds to make their transition systems finite.

In FTTS, we have to make sure that the models are Zeno-free because a timed system with Zeno behavior does not exist in the real-world. As the model of time in Timed Rebeca is discrete, the execution of an infinite number of message servers in zero time is the only scenario resulting in Zeno behavior. So, the Zeno behavior happens if and only if there is a cycle of message server invocations among different actors without progress of time. This can be detected by static analysis of the Timed Rebeca model.

FTTS can be used for reasoning about event-based properties, i.e., the relations among actions of systems and the time where they are triggered (messages are taken from bags). The most expressive action-based logic which can be evaluated using FTTS is defined in [17]. As proved in [17], FTTS can be used for verification of properties in a form of the modal $\mu$-calculus with weak modalities (a superset of event-based LTL properties). The weak modal $\mu$-calculus has the same syntax as the modal $\mu$-calculus, where we assume that the diamond ($\langle a \rangle \varphi$) and box ($[a]\varphi$) modalities are restricted to observable transitions, i.e., action $a$ must be a taking-event transition. The semantics of this logic is identical to that of the $\mu$-calculus, except for the semantics of the diamond and box operators — a state $s$ satisfies $\langle a \rangle \varphi$ if there is an execution starting from state $s$ to $t$, such that $a$ is the only visible action, and $t$ satisfies (inductively) $\varphi$. The semantics of box is defined dually.

## 4 The Standard Semantics: Time Transition System

FTTS can be used for efficient event-based analysis of Timed Rebeca models; however, it can not be used for the analysis against timed state-based properties. To be able to analyze Timed Rebeca models against timed state-based properties, a few mappings and techniques are developed based on TTS.

### 4.1 Time Transition System of Timed Rebeca

Time Transition System (TTS) of a Timed Rebeca model $\mathcal{M}$ is defined as a tuple $TTS = (S, s_0, Act, \rightarrow)$ where $S$ is the set of states, $s_0$ is the initial state, $Act$ is the set of actions, and $\rightarrow$ is the transition relation as described below (from [17]).

**States.** A state $s \in S$ of the Timed Rebeca model $\mathcal{M}$ consists of the local states of the actors, together with the current time of the state. The local state of actor $a_i$ in (the global) state $s$ is defined as the tuple $(V_{s,i}, B_{s,i}, pc_{s,i}, res_{s,i})$, where $V_{s,i}$ and $B_{s,i}$ are defined as the valuation of state variables and the bag of messages respectively (the same as in FTTS), $pc_{s,i} \in \{null\} \cup (\mathcal{M}_i \times \mathbb{N})$ is the program counter, and $res_{s,i} \in \mathbb{N}_0$ is the resuming time for actor $a_i$ which executes a `delay` in $s$. The program counter tracks the execution of the current message server and is *null* if actor $i$ is idle in $s$. So, state $s \in S$ can be defined as $\left(\prod_{i \in \mathcal{I}} (V_{s,i}, B_{s,i}, pc_{s,i}, res_{s,i}), now_s\right)$ where $now_s \in \mathbb{N}$ is the global current time of $s$.

**Initial State.** $s_0$ is the initial state of the Timed Rebeca model $\mathcal{M}$ where the state variables of the actors are set to their initial values (according to their types), the `initial` message is put in the bag of all actors having such a message server, the program counters of all actors are set to *null*, and the time of the state is set to zero.

**Actions.** There are three possible types of actions: taking a message $tmsg = ((sid, rid, mid), ar, dl)$, executing a statement by an actor (which we consider as an internal transition $\tau$), and progress of $n \in \mathbb{N}$ units of time. Hence, the set of actions is $Act = \bigcup_{i \in \mathcal{I}} ((\mathcal{I} \times i \times \mathcal{M}_i) \times \mathbb{N} \times \mathbb{N}) \cup \{\tau\} \cup \mathbb{N}$.

**Transition Relations.** Before defining the transition relation, we introduce the notation $E_{s,i}$ which denotes the set of *enabled messages* of actor $a_i$ in state $s$ which contains the messages whose arrival time is less than or equal to $now_s$. The transition relation $\rightarrow \subset S \times Act \times S$ is defined such that $(s, act, t) \in \rightarrow$ if and only if one of the following conditions holds.

1. **(Taking a message for execution)** In state s, there exists actor $a_i$ such that $pc_{s,i} = null$ and there exists $tmsg \in E_{s,i}$. Here, we have a transition of the form $s \xrightarrow{tmsg} t$. This transition results in extracting $tmsg$ from the message bag of $r_i$, setting $pc_{t,i}$ to the first statement of the message server corresponding to $tmsg$, and setting $res_{t,i}$ to $now_t$ (which is the same as $now_s$). Note that $V_{t,i}$ remains the same as $V_{s,i}$. These transitions are called *taking-event transitions* and $a_i$ is called *enabled actor*.
2. **(Internal action)** In state $s$, there exist $a_i$ such that $pc_{s,i} \neq null$ and $res_{s,i} = now_s$. The statement of message server of $a_i$ specified by $pc_{s,i}$ is executed and one of the following cases occurs based on the type of the statement. Here, we have a transition of the form $s \xrightarrow{\tau} t$.
   (a) Non-delay statements: the execution of such a statement may change the value of a state variable of actor $a_i$ or send a message to another actor. Here, $pc_{t,i}$ is set to the next statement (or *null* if there is no more statements). All other elements of $t$ are the same as those of $s$.
   (b) The statement is a non-deterministic assignment: the execution of a non-deterministic assignment $a = ?(e_1, ..., e_n)$ results in n different transitions from s to states $s_{v_1}, s_{v_2}, ..., s_{v_n}$, where $a = e_i$ in state $s_{v_i}$. The action is $\tau$, and the execution of $\tau$ results in $s_{v_i}$ $(1 \leq i \leq n)$.
   (c) Delay statement with parameter $d \in \mathbb{N}$: the execution of a delay statement sets $res_{t,i}$ to $now_s + d$. All other elements of the state remain unchanged. Particularly, $pc_{t,i} = pc_{s,i}$ because the execution of delay statement is not yet complete. The value of the program counter will be set to the next statement after completing the execution of delay (as will be shown in the third case).

   These transitions are called *internal transitions*.
3. **(Progress of time)** If in state $s$ none of the conditions in cases 1 and 2 hold, meaning that $\nexists a_i \cdot ((pc_{s,i} = null \wedge E_{s,i} \neq \emptyset) \vee (pc_{s,i} \neq null \wedge res_{s,i} = now_s))$, the only possible transition is progress of time. In this case, $now_t$ is set to $now_s + d$ where $d \in \mathbb{N}$ is the minimum value which makes one of the

aforementioned conditions become true. The transition is of the form $s \xrightarrow{d} t$. For any actor $a_i$, if $pc_{s,i} \neq null$ and $res_{s,i} = now_t$ (the current value of $pc_{s,i}$ points to a delay statement), $pc_{t,i}$ is set to the next statement (or to *null* if there are no more statements). These transitions are called *time transitions*. Note that when such a transition exists, there is no other outgoing transition from $s$.

We reuse ping pong example of Figure 4 to illustrate how TTS is generated and be able to compare the FTTS and the TTS of this example. Figure 6 presents the beginning part of the TTS of the ping pong example. In this figure, the details of the fourth state is shown. As only one timed transition is in the path to the fourth state, the global time of the state is 1. Also, the execution of both `Ping` and `Pong` actors are suspended by `delay` statements until reaching time 2 (based on the value of the program counters and the resuming times). Executing the first statement of `pong` message server, a message is scheduled for `Ping` actor (line 15), shown as the only content of the bag of `Ping`.

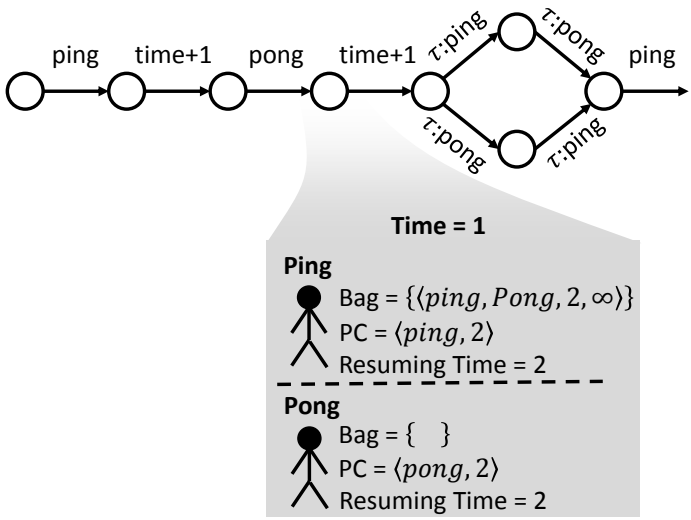


**Fig. 6.** The beginning part of the TTS of ping pong example

Based on TTS semantics, different mappings to existing languages and tools are created for Timed Rebeca. Recently a dedicated efficient tool is developed for TCTL model checking of Timed Rebeca.

### 4.2 Mapping Timed Rebeca to Timed Automata

We developed a mapping technique from Timed Rebeca models into timed automata [18, 9] for generating the TTS of Timed Rebeca models and supporting

state-based model checking. Timed automata [19] model is one of the most widely used modeling languages for modeling of realtime systems. UPPAAL toolset supports TCTL model checking of timed automata. In the proposed mapping, each actor is mapped into two timed automata, called *actor-behavior* automaton and *actor-bag* automaton. Additionally, one time automaton is defined to handle the behavior of *after* primitive for all actors, called *after-handler* automaton.

The actor-behavior automaton models the behavior of an actor according to the statements of its message servers and valuations of state variables. The state variables of each actor are mapped into variables of its corresponding actor-behavior automaton. The actor-behavior automaton, after receiving a message, moves to a state which represents the beginning of the corresponding message server. To model the behavior of a message server, its statements are mapped to the transitions of timed automata. The details of this mapping are presented in [9]. The actor-bag automaton handles the behavior of the message bag of each actor using an internal buffer called *messageQ*. The actor-bag accepts messages which are sent to its corresponding actor asynchronously, regardless of the state of the corresponding actor-behavior automaton. Then, actor-bag automaton delivers received messages upon the requests of its corresponding actor-behavior automaton. The after-handler automaton handles the messages which should be delivered to the actor-bag automata in the future (messages which are sent by the *after* primitive). The after-handler automaton accepts messages and put them into its buffer until the release time of the messages. When a message in the buffer of after-handler is released, it is sent to its corresponding actor-bag automaton.

The parallel composition of the resulting timed automata and the schedulability analysis of the model is done using UPPAAL [20]. Modeling of asynchronous message passing between actors using synchronous communication of timed automata increases the number of states dramatically [21]. We can apply some techniques, like using *committed states* and techniques that are presented in [22–24], to reduce the number of states of the resulting region transition system. However, as shown in [9], the technique stays inefficient for modeling asynchronous communication.

### 4.3 Mapping Timed Rebeca to Realtime Maude

Timed Rebeca is mapped into Realtime Maude [25, 26] to support timed analysis of Timed Rebeca models with dynamic actor creation. Realtime Maude is a specification formalism and analysis tool for realtime systems based on rewriting logic [27]. Realtime Maude is highly expressive and is particularly suitable for formally specifying distributed realtime systems in an object-oriented way.

In the Realtime Maude model, a multiset of actor objects and messages represents the state of a Timed Rebeca model, where each actor object represents a rebec and each message in the multiset represents a message in the set of undelivered messages of the Timed Rebeca model. Communication between actors takes place by putting a message into the multiset of undelivered messages. This message is remained in the undelivered messages until its message delivery delay

ends (i.e. the parameter specified by *after* keyword). The instantaneous actions of a rebec are formalized using rewrite rules, as shown in [10].

The "standard" object-oriented tick rule [25] is used to model time advance until the next time when something must "happen". The effect of time elapse on an actor is that the remaining time for a delay statement is decreased according to the elapsed time. For messages traveling between actors, their remaining delivery delays and deadline are decreased according to the elapsed time. In both cases, if the deadline expires before the message is served, the message is purged.

Using this mapping, we analyzed several Timed Rebeca models using the bounded-TCTL model checker engine of Realtime Maude. This mapping supports dynamic actor creation in the model, which is not supported by other approaches. Realtime Maude performs *bounded* model checking and needs high expertise to work with.

### 4.4 Direct TCTL Model Checking of Timed Rebeca Models

To overcome the inefficiencies of using back-end model checkers, we developed a dedicated TCTL model checking toolset for Timed Rebeca models. To this end, we directly generated the TTS of Timed Rebeca models and applied a modified version of the model checking algorithm of [28] for analysis against $\text{TCTL}_{\leq,\geq}$ properties. As shown in [11], the modified version of the algorithm analyzes a TTS with $V$ states and $E$ transitions against property $\Phi$ in $O((V \lg V + E)|\Phi|)$ which is the best possible $\text{TCTL}_{\leq,\geq}$ model checking algorithm for dense transition systems.

In [11] we also showed that for the majority of the timed actors, the proposed algorithm cover model checking against complete TCTL properties in pseudo polynomial time. However, UPPAAL only supports model checking for a fragment of TCTL and realtime Maude supports bounded-model checking of TCTL properties.

## 5 Timed Rebeca in Practice

Timed Rebeca is used in several applications such as modeling and analysis of routing algorithms for Network on Chips (NOCs) [12, 14], and schedulability analysis of wireless sensor and actuator network applications (specifically, real-time continuous sensing application for structural health monitoring) [29]. Our NoC example is the basis and the reference model of the work of Din *et al.* in [30] which proposes a scalable verification technique for generic NoC models.

### 5.1 Analyzing NoCs

As mentioned in details in Section 2, Network on Chip (NoC) has emerged as a promising architecture paradigm for many-core systems. Asynchronous communication has become conspicuous in NoC design to overcome problems of clock skew and clock tree distribution of fully synchronous design. Thereby Globally

Asynchronous Locally Synchronous (GALS) NoC has gained attention in design of such systems. In GALS NoCs, a sent packet might be delayed by a number of disrupting packets, which creates various end-to-end latencies. Thus, for analysis of such systems it is essential to consider all possible behaviors of the systems (at least for specific scenarios) and consider the whole state spaces. Simulation techniques cannot be applied to exhaustive search. As complexity grows in NoCs, functional verification and performance prediction in the early stages of the design process are suggested as ways to reduce the fabrication cost. Formal methods have gained more attention as alternative ways for analyzing NoC designs. Timed Rebeca is used in [12] to model two-dimensional mesh GALS NoCs with a four-phase handshake communication protocol, and functional and timing behaviors, the routing algorithm and communication protocol are captured in the model. Deadlock freedom, message arrival, and end-to-end packet latency are checked and the verification results are compared and matched to the simulation results of HSPICE[4] using 32nm technology. This work is extended in [14] for comparing different routing algorithms in GALS NoCs.

**Comparing Routing Algorithms in NOC.** In [14], Timed Rebeca models for the three following routing algorithms on GALS NoCs are developed: XY, Odd-Even, and Dynamic Adaptive Deterministic (DyAD). In XY routing algorithm, as the first step, packets move along the X direction until they reach the column of the destination. Then they move along the Y direction to reach their destinations. The Odd-Even routing algorithm works based on the Odd-Even turn model [31]; north-to-west and south-to-west turns are prohibited in routers located in an odd column, and east-to-south and east-to-north turns are prohibited in routers located in an even column. The odd-Even turn model restricts the turns in the packet path to ensure deadlock freedom. Finally, DyAD routing is a dynamic algorithm that uses a deterministic or adaptive routing based on different network congestion conditions. Each router monitors the occupation ratio of its input buffers; whenever one of the buffers passes the congestion threshold the corresponding neighboring routers are informed about the congestion. Routers priodically check their neighbors to change their routing algorithm into adaptive routing in case of congestion.

FTTS-based model checking of Timed Rebeca is used for comparing the performance of XY, Odd-Even, and DyAD algorithms. The NoC size in these comparisons is set to 4x4. The size of input buffers is set to 3 packets and congestion threshold is set to %33. To compare the three algorithms, six different scenarios describing different network traffics, are used. The selected scenarios are representers of widely occurring traffic patterns. As illustrated in Figure 7 in the first three scenarios, DyAD and Odd-Even show less end-to-end packet latency as these algorithms are designed to avoid congestion. In the second three scenarios, there are disrupting packets in all possible directions. These scenarios investigate the impact of low latency of XY and Odd-Even algorithms which is

[4] HSPICE provides the lowest level simulation for hardware designs. All the details of transistors and wires of hardware designs are considered in HSPICE simulator.

the result of their simplicity in contrast to DyAD. As shown in Figure 7, XY shows the best performance indicating that it works better in a fully chaotic situation.

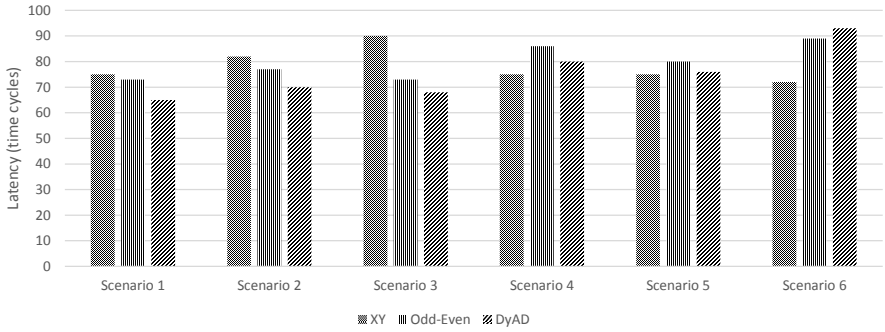


**Fig. 7.** Comparison among latencies of routing algorithms in six different scenarios

**Design Exploration for ASPIN Architecture on GALS NoC.** ASPIN is a fully asynchronous two-dimensional mesh NoC with physically distributed routers in each core. ASPIN uses the storage strategy of input buffering, and each input port is provided by an independent FIFO buffer. Packets arrived from different sides (from neighboring routers on four sides and the local core), are stored in the FIFO buffer of the input port. If there is more than one request for an output port, a round robin policy is used for the arbitration. ASPIN uses XY routing algorithm to route packets from output port of the source router to the input port of the destination router. Communications between routers are established using four-phase handshake protocol. The protocol uses two signals namely *Req* and *Ack*. To transfer a packet, first, the sender sends a request by rising the *Req* signal, and waits for an acknowledgment from the receiver. All signals must return to zero before the next packet could be sent.

Traditionally, simulating the ASPIN design using HSPICE is used for the analysis of these systems. HSPICE provides precise simulation results, and for that all the details of an ASPIN design must be specified prior to performing simulation. In addition, the time consumption of HSPICE simulation of ASPIN is very high. In [12], Timed Rebeca is used for modeling and Afra is used for the analysis of ASPIN designs. The comparison among the latencies which are measured using HSPICE simulation and Afra model checker is shown in Figure 8. As shown in Figure 8, three different packet generation scenarios (i.e., different traffics) are used in this comparison. As a result, having similar trends show that despite the fact that we captured much less details in Timed Rebeca comparing to HSPICE, Timed Rebeca analysis provides the same results in design exploration, and hence can be used for the required measurements. This comparison

shows that using Timed Rebeca in the early stages of design helps designers in making suitable architectural decisions according to the desired performance of the systems.

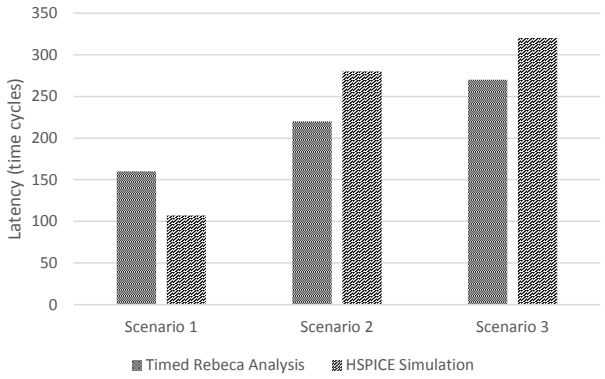


**Fig. 8.** Comparison among computed latencies for three different scenarios, using HSPICE simulation, and Timed Rebeca model checking

To compare the new approaches with the simulation technique using HSPICE, the model of ASPIN for 32nm CMOS technology was considered. Running each scenario in HSPICE for the analysis of one path took more than 24 hours on a system with Core i7, 2.6 GHz processor and 24GB of memory. In contrast running the scenarios and analyzing all the paths using the new approaches took less than 5 seconds.

### 5.2 Analyzing WSAN Applications

Distributed Wireless Sensor and Actuator Networks (WSANs) have become an attractive method of providing low-cost continuous monitoring in different applications. However, because of the complexity of concurrent and distributed programming, networking, and real-time and embedded requirements, building WSAN applications can be particularly challenging. In WSAN applications, coordination among distributed sensors must be well configured to achieve the optimum point which satisfies several constraints, including low power constraints, realtime deadlines of physical processes, and constraints on scheduling and resource utilization. Programmers often use informal worst-case analysis and debugging to ensure schedules that satisfy these requirements. Not only can this process be tedious and error-prone, it is inherently conservative and thus likely to lead to an inefficient use of resources. Moreover, the process fails to provide any safety guarantees for the resulting configuration.

Timed Rebeca is used to model a case study involving real-time continuous data acquisition for structural health monitoring and control (SHMC) of

civil infrastructure [29]. This system has been implemented on the Imote2 wireless sensor platform, and has been deployed for long-term monitoring of several highway and railroad bridges [32]. Ensuring safe execution requires modeling the interactions between the CPU, sensor and radio within each node, as well as interactions between nodes. Moreover, the application tasks are not isolated from other aspects of the system: they execute alongside tasks belonging to other applications, middleware services, and operating system components. In this application, all periodic tasks (sample acquisition, data processing, and radio packet transmission) are required to complete before the next iteration starts. The results show that a safe configuration can be found which improves resource utilization compared to the previous informal schedulability analysis used in [32]. The sampling rate of the system can be increased by 7% without encountering safety hazards.

## 6 Discussion and Related Work

Different approaches have been proposed for modeling and analysis of realtime systems. Timed automata [19], realtime Maude [25], and TCCS [33] are examples of modeling formalisms for design and analysis of realtime systems. For designing Timed Rebeca we looked into all the above languages and used the same basic ideas and concepts, we also have mappings to and extensive comparisons with timed automata [18] and realtime Maude [10].

Apart from these well-known and general purpose modeling formalisms, high level modeling languages are adopted for the realtime requirements. Actor model as an example of such languages is extended with timing features to address the functional behaviors of actors and the timing constraints on patterns of actor invocations. A realtime actor model, RT-synchronizer, is proposed in [34] as an example of actor model which enforces realtime relations between events. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, Timed Rebeca allows us to work at a lower level of abstraction. Using Timed Rebeca, a modeler can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various points of view.

Creol [35] is a concurrent object based language which is designed in parallel with Rebeca. Concurrent objects of Creol can be checked for schedulability using the approach of [35], which is developed based on the same idea presented for Timed Rebeca in [36]. ABS [37] is an extension of Creol in multiple ways. While in Creol and its descendent, ABS, the focus has been on different modeling features, for Rebeca we kept the core of the language simple and avoided adding any complexity. Our focus has been on analysis and formal verification of Rebeca and its extension. Recently, Timed Rebeca is extended to capture probabilistic behaviour, the language is presented in [38].

## References


1. Ptolemaeus, C.: System Design, Modeling, and Simulation using Ptolemy II. Ptolemy.org (2014)
2. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and Verification of Reactive Systems using Rebeca. Fundam. Inform. **63**(4) (2004) 385–410
3. Sirjani, M., de Boer, F.S., Movaghar-Rahimabadi, A.: Modular verification of a component-based actor language. J. UCS **11**(10) (2005) 1695–1717
4. Hewitt, C.: Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science, MIT (April 1972)
5. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press series in artificial intelligence. MIT Press (1990)
6. Sirjani, M., Jaghoori, M.M.: Ten Years of Analyzing Actors: Rebeca Experience. In Agha, G., Danvy, O., Meseguer, J., eds.: Formal Modeling: Actors, Open Systems, Biological Systems. Volume 7000 of Lecture Notes in Computer Science., Springer (2011) 20–56
7. Reynisson, A.H., Sirjani, M., Aceto, L., Cimini, M., Jafari, A., Ingólfsdóttir, A., Sigurdarson, S.H.: Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. Sci. Comput. Program. **89** (2014) 41–68
8. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In Mousavi, M.R., Ravara, A., eds.: FOCLASA. Volume 58 of EPTCS. (2011) 1–19
9. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed Rebeca Schedulability and Deadlock Freedom Analysis using Bounded Floating Time Transition System. Sci. Comput. Program. **98** (2015) 184–204
10. Sabahi-Kaviani, Z., Khosravi, R., Sirjani, M., Ölveczky, P.C., Khamespanah, E.: Formal Semantics and Analysis of Timed Rebeca in Real-Time Maude. In Artho, C., Ölveczky, P.C., eds.: FTSCS. Volume 419 of Communications in Computer and Information Science., Springer (2013) 178–194
11. Khamespanah, E., Khosravi, R., Sirjani, M.: Efficient TCTL Model Checking Algorithm for Timed Actors. In Boix, E.G., Haller, P., Ricci, A., Varela, C., eds.: Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014, ACM (2014) 55–66
12. Sharifi, Z., Mosaffa, M., Mohammadi, S., Sirjani, M.: Functional and Performance Analysis of Network-on-Chips using Actor-Based Modeling and Formal Verification. ECEASST **66** (2013)
13. Mechitov, K.A., Khamespanah, E., Sirjani, M., Agha, G.: A Model Checking Approach for Schedulability Analysis of Distributed Real-Time Sensor Network Applications. In: Submitted for Publication. (2015)
14. Sharifi, Z., Mohammadi, S., Sirjani, M.: Comparison of NoC Routing Algorithms using Formal Methods. In proceedings of PDPTA'13 (2013)
15. Sheibanyrad, A., Greiner, A., Panades, I.M.: Multisynchronous and fully asynchronous nocs for GALS architectures. IEEE Design & Test of Computers **25**(6) (2008) 572–580

16. Khamespanah, E., Sabahi-Kaviani, Z., Khosravi, R., Sirjani, M., Izadi, M.J.: Timed-Rebeca Schedulability and Deadlock-Freedom Analysis using Floating-Time Transition System. In Agha, G.A., Bordini, R.H., Marron, A., Ricci, A., eds.: AGERE!@SPLASH, ACM (2012) 23–34
17. Khamespanah, E., Sirjani, M., Viswanathan, M., Khosravi, R.: Floating Time Transition System: More Efficient Analysis of Timed Actors. In: Formal Aspects of Component Software - 12th International Symposium, FACS 2015, Rio de Janeiro, Brazil, October 14-16, 2015. (2015)
18. Izadi, M.J.: An Actor Based Model for Modeling and Verification of Real-Time Systems. Master's thesis, University of Tehran, School of Electrical and Computer Engineering, Iran (2010)
19. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science **126**(2) (1994) 183–235
20. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In Alur, R., Henzinger, T.A., Sontag, E.D., eds.: Hybrid Systems. Volume 1066 of Lecture Notes in Computer Science., Springer (1995) 232–243
21. Lamport, L.: Real-Time Model Checking Is Really Simple. In Borrione, D., Paul, W.J., eds.: CHARME. Volume 3725 of Lecture Notes in Computer Science., Springer (2005) 162–175
22. Bengtsson, J., Jonsson, B., Lilius, J., Yi, W.: Partial Order Reductions for Timed Systems. In Sangiorgi, D., de Simone, R., eds.: CONCUR. Volume 1466 of Lecture Notes in Computer Science., Springer (1998) 485–500
23. Minea, M.: Partial Order Reduction for Model Checking of Timed Automata. In Baeten, J.C.M., Mauw, S., eds.: CONCUR. Volume 1664 of Lecture Notes in Computer Science., Springer (1999) 431–446
24. Håkansson, J., Pettersson, P.: Partial Order Reduction for Verification of Real-Time Components. In Raskin, J.F., Thiagarajan, P.S., eds.: FORMATS. Volume 4763 of Lecture Notes in Computer Science., Springer (2007) 211–226
25. Ölveczky, P.C., Meseguer, J.: Semantics and Pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation **20**(1-2) (2007) 161–196
26. Ölveczky, P.C., Meseguer, J.: The Real-Time Maude Tool. In: Proc. TACAS'08. Volume 4963. (2008) 332–336
27. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. Theoretical Computer Science **96**(1) (1992) 73–155
28. Laroussinie, F., Markey, N., Schnoebelen, P.: Efficient Timed Model Checking for Discrete-Time Systems. Theoretical Computer Science **353**(1-3) (2006) 249–271
29. Mechitov, K., Khamespanah, E., Sirjani, M., Agha, G.: Schedulability Analysis of Distributed Real-Time Sensor Network Applications using Actor-Based Model Checking. Technical Report (2015)
30. Din, C.C., Tarifa, S.L.T., Hähnle, R., Johnsen, E.B.: History-Based Specification and Verification of Scalable Concurrent and Distributed Systems. In Butler, M., Conchon, S., Zaïdi, F., eds.: Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings. Volume 9407 of Lecture Notes in Computer Science., Springer (2015) 217–233
31. Chiu, G.: The Odd-Even Turn Model for Adaptive Routing. IEEE Trans. Parallel Distrib. Syst. **11**(7) (2000) 729–738
32. Linderman, L., Mechitov, K., Spencer, B.F.: TinyOS-Based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control. Structural Control and Health Monitoring (2012)

33. Yi, W.: CCS + time = an interleaving model for real time systems. In Albert, J.L., Monien, B., Rodríguez-Artalejo, M., eds.: Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings. Volume 510 of Lecture Notes in Computer Science., Springer (1991) 217–228
34. Ren, S., Agha, G.: RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems. In Gerber, R., Marlowe, T.J., eds.: Workshop on Languages, Compilers, & Tools for Real-Time Systems, ACM (1995) 50–59
35. de Boer, F.S., Chothia, T., Jaghoori, M.M.: Modular Schedulability Analysis of Concurrent Objects in Creol. In Arbab, F., Sirjani, M., eds.: Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers. Volume 5961 of Lecture Notes in Computer Science., Springer (2009) 212–227
36. Jaghoori, M.M., de Boer, F.S., Sirjani, M.: Task Scheduling in Rebeca. In: NWPT. (2007) 16–18
37. Albert, E., de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Tarifa, S.L.T., Wong, P.Y.H.: Formal modeling and analysis of resource management for cloud architectures: an industrial case study using Real-Time ABS. Service Oriented Computing and Applications **8**(4) (2014) 323–339
38. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H.: Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors. ECEASST **70** (2014)