Timed Rebeca Schedulability and Deadlock Freedom Analysis Using Bounded Floating-Time Transition System

Ehsan Khamespanah[a,**], Marjan Sirjani[b,*], Zeynab Sabahi Kaviani[a,*], Ramtin Khosravi[a,*], Mohammad-Javad Izadi[a]

[a] *School of Electrical and Computer Engineering, University of Tehran*
[b] *School of Computer Science, Reykjavik University*

Abstract

Timed Rebeca is an extension of the actor-based modeling language Rebeca that supports timing features. Rebeca is purely actor-based with no shared variables and asynchronous message passing with no explicit receive. Both computation time and network delays can be modeled in Timed Rebeca. In this paper, we propose a new approach for checking schedulability and deadlock freedom of Timed Rebeca models. The key features of Timed Rebeca, asynchrony of actors and absence of shared variables, and the focus on events instead of states in the selected properties, led us to a significant reduction in the state space. In the proposed method, there is no unique time value for each state, instead of that we store the local time of each actor separately. We prove the bisimilarity of the generated transition system, called floating-time transition system, and the state space generated from the original semantics of Timed Rebeca. In addition, to avoid state space explosion because of time progress, we define a type of equivalency among states called shift equivalency. The shift equivalence relation between states can be used for Timed Rebeca as the timing features are based on relative values. We developed a tool, and the experimental results show that our approach mitigates the state space explosion problem of the former methods and allows model-checking of larger systems.

Keywords: Actor model, Timed Rebeca, Verification, Realtime systems, Schedulability, Deadlock Freedom

1. Introduction

A well-established paradigm for modeling concurrent and distributed systems is the *actor* model. Actor model is originally introduced by Hewitt [18] as an agent-based language and is later developed by Agha [6] as a mathematical model of concurrent computation. *Actors* are seen as the universal primitives of concurrent computation [6]. Each actor provides a certain number of services which can be requested by other actors by sending messages to the provider. Messages are put in the message buffer of the receiver, the receiver takes the message and executes the requested service, possibly sending messages to some other actors. There are some extensions on the actor model such as RT-synchronizer [39] and Timed Rebeca[5] for modeling timed systems.

Timed Rebeca [5] is proposed as an extension of *Reactive Objects Language*, *Rebeca* [43]. *Rebeca*, is an operational interpretation of the actor model with formal semantics, supported by model checking tools [42]. Rebeca is designed to bridge the gap between formal methods and software engineering. The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification,

[*] Corresponding author
[**] Principal corresponding author
Email addresses: e.khamespanah@ece.ut.ac.ir (Ehsan Khamespanah), marjan@ru.is (Marjan Sirjani), z.sabahi@ece.ut.ac.ir (Zeynab Sabahi Kaviani), r.khosravi@ut.ac.ir (Ramtin Khosravi), m.j.izadi@ece.ut.ac.ir (Mohammad-Javad Izadi)

abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and is embodied in verification tools [42, 23, 40, 41]. With its simple, message-driven and object-based computational model, Java-like syntax, and a set of verification tools, Rebeca is an interesting and easy-to-learn model for practitioners. In Timed Rebeca, timing primitives are added to Rebeca to address *computation time*, *message delivery time*, *message expiration*, and *period of occurrence of events* [5].

The advantage of Timed Rebeca from modeling point of view is its intuitive and easy-to-use syntax and the actor-based paradigm of modeling. Comparing to other timed modeling languages like TCCS [44], Real-Time Maude [35], and Timed Automata [8], instead of using process algebra, rewriting logic, and automata (respectively), here you are using an actor-based language and there is no need for any knowledge of formal methods. A comprehensive comparison is given in [5] and [4].

In this work, our focus is on the analysis of Timed Rebeca, and how the *key features of actors* and *event-based properties* led us to a significant amount of state space reduction. We introduce the notion of *floating-time transition system* (FTTS) for model checking Timed Rebeca. Floating-time transition system is built based on the states of concurrent reactive objects, called *rebec*s (in this paper we use the words rebec and actor interchangeably). States in floating-time transition system contains the local times of each rebec, in addition to values of their state variables and the bag of their received messages. The local times of each rebec in a state can be different from other rebecs, and there is no unique value for time in each state. This is only admissible where we are not interested in the state of all the rebecs/actors at a specific point of time and instead of the synchronized states the order of events matter.

The *key features of actors* that lead us to this technique are having no shared variables, no blocking send or receive, single-threaded actors, and non-preemptive execution of each message server which give us an isolated message server execution. This means that execution of a message server of a rebec will not interfere with execution of a message server of another rebec. For an actor model and its asynchronous message passing, the property language needs to be able to reason about the timing and occurrence of the messages more than the values of some variables inside the actors. This leads us towards *event-based properties* where an event occurs each time a message is received and served by an actor, rather than a property on state propositions. Timed properties are usually one of the followings: the minimum, maximum, or exact distance between two events (where the second one can be a reaction or response to the first one), periodicity of an event, and occurrence of an event before another event. Time-out or deadline-miss is an example of an event-based property which happens by exceeding the desired distance between two events.

Generally, model checking and simulation tools for timed models, like Real-Time Maude [38], TLC [27], and McErlang [16], generate a timed transition system (TTS). In these timed transition systems the current time of the model is represented as the value of a state variable, called *now* (or clock). The passage of time is modeled by increasing the value of *now* [10]. In contrast, in floating-time transition system of Timed Rebeca there is a *now* variable for each rebec instead of a single *now* for the system.

Most of the TTS-based model checking tools use ”maximum time elapse” strategy [34] and only encounter the significant events as transitions, but they keep the states of different components (processes, modules) in sync. By relaxing this synchronization constraint, which seems unnecessary in the actor world, we gain a significant amount of state space reduction (See Section 6). Note that in a Timed Rebeca ***model*** we consider the notion of synchronized local clocks for the rebecs of the system which is similar to the notion of global time in other timed models. The novelty of our approach is building the ***state space*** as floating-time transition system where in each state the local time of rebecs can be shuffled while the order of events is preserved.

The standard timed temporal logics like TCTL [7] and MTL [25] (an extension of LTL, adding optional real-time constraints to the temporal operators) can be used as property languages when the state space is presented as a TTS. However, in reactive and distributed systems we mostly care about events. Therefore, although the states of FTTS do not represent the state of all rebecs at a specific point of time and hence TCTL and MTL model checking become limited, event based verification can be done using FTTS. In this work, we check schedulability (deadline-missed) and deadlock freedom of Timed Rebeca using FTTS. A Timed Rebeca model is schedulable if none of the rebecs miss any deadline [21, 20], so, the tool reports a failure at the first occurrence of deadline-miss for any rebec. Deadline-miss is checked per rebec and for that we do not need to have a unique value of time for all the rebecs. Deadlock happens when there is absolutely no message for any of the rebecs to handle, where again there is no need for a unique value of time in each state.

Progress of time in floating-time transition system results in unbounded number of states. Hence, we introduce bounded floating-time transition system. In bounded floating-time transition system, a new kind of equivalency between states, called shift equivalence relation, is used to bound the number of generated states. This approach is similar to time-translation symmetry relation of Lamport for TLC [28]; however, instead of a state variable *now* in the model, we have several *now* variables with different values, one per each rebec. We prove that there is a bisimulation relation between the bounded floating-time transition relation and the floating-time transition relation.

Contribution. The contributions of this paper can be summarized as follows:

– Introducing the notion of floating-time transition system as a basis for state space generation and tool development for Timed Rebeca models where the focus is on the order of events and the value of current time for each actor in a state may differ from other actors.

– Introducing a time-shift equivalency relation to obtain bounded floating-time transition system and proving the bisimilarity of the bounded floating-time transition system and the floating-time transition system.

– Implementing a tool for schedulability and deadlock-freedom analysis based on the proposed techniques and providing experimental results which very well illustrate the efficiency of our technique by means of a number of case studies.

This paper is a revised and extended version of [24]. Here, we reorganized and rewrote all the formal definitions and proofs. We added more case studies to the section on experimental results, we also added a comparison to the results of a newly developed tool for model checking Timed Rebeca using McErlang [26]. An extensive section on related work has been added to the paper.

Roadmap. The rest of this paper is structured as follows. The next section is on background knowledge, Rebeca, Timed Rebeca, and the operational semantics of Timed Rebeca. Section 3 gives the definition of floating-time transition system based on the SOS semantics of Timed Rebeca in [5]. Section 4 defines bounded floating-time transition system and provides the proof of bisimulation relation between bounded floating-time transition system and floating-time transition system. In Section 5 we explain the schedulability and deadlock freedom analysis of Timed Rebeca and the implementation issues of state space generation algorithm. Section 6 presents the experimental results. Related work has been reviewed in Section 7 and the concluding remarks are presented in Section 8.

2. Background

Here we give a brief overview of syntax and semantics of Rebeca, the extension which builds Timed Rebeca, and the semantics of Timed Rebeca given as SOS rules.

Rebeca. Rebeca [43, 42] is an actor-based language for modeling concurrent and reactive systems with asynchronous message passing. Rebeca model is similar to the actor model as it has reactive objects with no shared variables, asynchronous message passing with no blocking send and no explicit receive, and unbounded buffers for messages. Objects in Rebeca are reactive, self-contained, and each of them is called a *rebec* (reactive object). Communication takes place by message passing among rebecs. Each rebec has an unbounded buffer, called message *queue*, for its arriving messages. Computation is event-driven, meaning that each rebec takes a message that can be considered as an event from the top of its message queue and execute the corresponding message server (also called a method). The execution of a message server is an atomic execution of its body that is not interleaved with any other method execution.

A Rebeca model consists of a set of reactive classes and the main block (for the syntax of (Timed) Rebeca see Figure 1 and for an example see Figure 2). In the main block the rebecs which are instances of the reactive classes are declared. The body of the reactive class includes the declaration for its known

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

Figure 1: Abstract syntax of Timed Rebeca (a slightly revised version of the syntax presented in [5]). Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition at least once, superscript ∗ for repetition zero or more times, whereas using ⟨...⟩ 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.

rebecs, state variables, and message servers. The rebecs instantiated from a reactive class can only send messages to the known rebecs of that reactive class. Message servers consist of the declaration of local variables and the body of the message server. The statements in the body can be assignments, conditional statements, enumerated loops, non-deterministic assignment, and method calls. Method calls are sending asynchronous messages to other rebecs (or to self). A reactive class has an argument of type integer denoting the maximum size of its message queue. Although message queues are unbounded in the semantics of Rebeca, to avoid state space explosion in model checking we need a user-specified upper bound for the queue size. The operational semantics of Rebeca has been introduced in [43] in more details.

We illustrate Rebeca language with an example. Figure 2 shows the Rebeca model of a ticket service system (ignore the time primitives *delay*, *after*, and *deadline* for now). The model consists of three reactive classes: *TicketService*, *Agent*, and *Customer*. *Customer* sends the *requestTicket* message to *Agent* and *Agent* forwards the message to *TicketService*. *TicketService* replies to *Agent* by sending a *ticketIssued* message and *Agent* responds to *Customer* by sending the issued ticket.

Timed Rebeca. Timed Rebeca is an extension on Rebeca 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 rebec has its own local clock. The local clocks can be considered as synchronized distributed clocks. Methods are still executed atomically, however passing of time while executing a method can be modeled. In addition, instead of queue for messages, there is a bag of messages for each rebec.

The timing primitives that are added to the syntax of Rebeca are *delay*, *deadline* and *after*. The *delay* statement models the passing of time for a rebec during execution of a message server. The keywords *after* and *deadline* can only be used in conjunction with a method call. The value of the argument of *after* shows how long it takes for the message to be delivered to its receiver. The *deadline* shows the timeout for the message, i.e., how long it will stay valid. We illustrate the application of these keywords with an example. Figure 2 shows the Timed Rebeca model of a ticket service system. As shown in line 12 of the model, issuing the ticket takes three time units (the value of *issueDelay* passed to rebec *ts* in the main block). At line 24 the rebec instantiated from *Agent* sends a message *requestTicket* to rebec *ts* instantiated from *TicketService*,

```
1  reactiveclass TicketService {
2    knownrebecs {
3      Agent a;
4    }
5    statevars {
6      int issueDelay;
7    }
8    msgsrv initial(int myDelay) {
9      issueDelay = myDelay;
10   }
11   msgsrv requestTicket() {
12     delay(issueDelay);
13     a.ticketIssued(1);
14   }
15 }
16
17 reactiveclass Agent {
18   knownrebecs {
19     TicketService ts;
20     Customer c;
21   }
22   msgsrv requestTicket() {
23     ts.requestTicket()
24         deadline(5);
25   }
26   msgsrv ticketIssued(byte id) {
27     c.ticketIssued(id);
28   }
29 }
30
31 reactiveclass Customer {
32   knownrebecs {
33     Agent a;
34   }
35   msgsrv initial() {
36     self.try();
37   }
38   msgsrv try() {
39     a.requestTicket();
40   }
41   msgsrv ticketIssued(byte id) {
42     self.try() after(30);
43   }
44 }
45
46 main {
47   Agent a(ts, c):();
48   TicketService ts(a):(3);
49   Customer c(a):();
50 }
```

Figure 2: The model of ticket service system.

$$\frac{(\sigma_{r_i}(m), \sigma_{r_i}[rtime = max(TT, \sigma_{r_i}(now)), [\overline{arg} = \overline{v}], sender = r_j], Env, B) \xrightarrow{\tau} (\sigma'_{r_i}, Env', B')}{(\{\sigma_{r_i}\} \cup Env, \{(r_i, m(\overline{v}), r_j, TT, DL)\} \cup B) \longrightarrow (\{\sigma'_{r_i}\} \cup Env', B')}\, C$$

Figure 3: The SOS rule for the scheduler in Timed Rebeca from [5]

and gives a deadline of five to the receiver to take this message and start serving it. The periodic task of retrying for a new ticket is modeled in line 42 by the customer sending a *try* message to itself and letting the receiver to take it from its queue only after 30 units of time (by stating *after(30))*.

Structural Operational Semantics of Timed Rebeca. Now we provide an overview of the SOS rules of Timed Rebeca which has been proposed in [5]. Timed Rebeca states are pairs (Env, B), where Env is a finite set of environments and B is a bag of messages. For each rebec r of the model there exists $\sigma_r \in Env$ which stores information about the actor, including the values of its state variables and local time, as well as structural characteristics like the body of the message servers.

The bag B contains an unordered collection of all the messages of all rebecs. Each message is a tuple of the form $(r_i, m(v), r_j, TT, DL)$. Intuitively, such a tuple says that at time TT (time tag), the sender r_j sent the message to the rebec r_i requesting it to execute its message server m with actual parameters v. Moreover this message expires at time DL [5]. Note that DL specifies the latest time that the message has to be released, i.e., the messages server has to start its execution or it will expire otherwise.

The system transition relation → is defined by the rule *scheduler* of Figure 3 where the condition C is defined as follows: σ_{r_i} is not contained in Env, and $(r_i, m(\overline{v}), r_j, TT, DL) \notin B$, and $\sigma_{r_i}(rtime) \leq DL$, and $TT \leq min(B)$. The scheduler rule allows the system to progress by picking up messages from the bag and executing the corresponding methods. The third side condition of the rule, namely $\sigma_{r_i}(rtime) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The last side condition is the predicate $TT \leq min(B)$, which shows that the time tag TT of the selected message is the smallest time tag of all the messages (for all the rebecs r_i) in the bag B [5].

The τ transition shows the execution of the message server of the rebec r_i and its effects are formally defined in [5]. Intuitively, execution of a message server means carrying out its body statements atomically. The execution may change the environment of the rebec r_i by assigning new values to its state variables, modifying the *now* value of r_i, or changing the bag by sending messages to other rebecs. The new message is put in the bag with its time tag and deadline. Time tag is its relative receive time which is computed by the value of *after* statement. The $\sigma_{r_i}(now)$ (current time of the rebec) may be modified if the body of message contains the timing statement *delay*. The current time of the rebec increases by the value of the *delay* statement.

3. Floating-Time Transition System of Timed Rebeca

In this section, we describe the floating-time transition system of Timed Rebeca, an alternative semantics for Timed Rebeca which is bisimilar to SOS semantics of [5]. Although both of SOS and floating-time semantics present the same behavior for Timed Rebeca models, floating-time transition system has a more explicit representation for both states and transitions. It makes floating-time transition system more suitable as a basis for developing a toolset for analysis of Timed Rebeca models, as described in Section 5.

To define floating-time transition system, we first formalize the necessary notions and present the definitions for Timed-Rebeca models.

Definition 1 (Timed Rebeca Model)**.** *Timed Rebeca model $\mathcal{M}$ is the set of rebecs which are executing concurrently.* □

For a Timed Rebeca model $\mathcal{M}$, the function $O(\mathcal{M})$ returns all the rebec instances in the model. Each rebec of a Timed Rebeca model $\mathcal{M}$ has a state, and the collection of the states of all the rebecs of $O(\mathcal{M})$ builds the state of the model.

Definition 2 (State in a Timed Rebeca Model)**.** *A state of a Timed Rebeca model is a tuple $s = \prod_{r_i \in O(\mathcal{M})} state(r_i)$, where $state(r_i)$ is the state of rebec r_i.* □

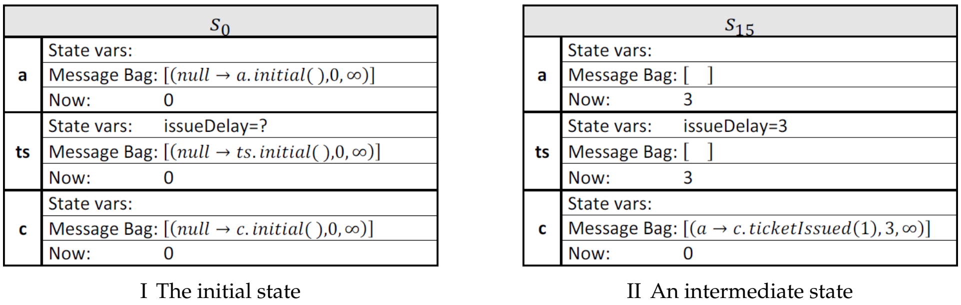
A state of a rebec consists of the values of its state variables, messages in its message bag, and its local time.

Definition 3 (State of a Rebec)**.** *A state of a rebec $r_i \in O(\mathcal{M})$ is a tuple $state(r_i) = (sv, bag, now)$ where sv is the valuation of the state variables, bag is the content of the message bag, and now is the value of the local time of the rebec r_i.* □

Functions $sv(s, r_i)$, $bag(s, r_i)$, and $now(s, r_i)$ return the value of the state variables, the content of message bag, and the current time of rebec r_i in state s, respectively.

As shown in Definition 3 the message bag of a rebec is a part of it's state. Message bag of a rebec contains messages which have been sent to the rebec. A message consists of its content (its name, the sender, the receiver, and the parameters) augmented by its arrival time and deadline. The arrival time can be considered as the time the message is put in the bag, which is the value of *now* of the sender when the *call* occurs unless the call statement has a non-zero *after* argument. In this case, the value of *after* argument will be added to the *now* of the sender to give us the arrival time. Note that the notion of *release time* of a message which is the time that the message is taken from the bag to be served is a different notion from the *arrival time*. Deadline is the time that the request will become invalid. It can be considered as a request from the sender for a release time less than the deadline.

The values of the arguments of the timing primitives, *delay*, *deadline* and *after*, in a Rebeca code are relative values. For example if you have *deadline(5)* in a call statement, it means that the deadline for the receiver rebec to take the message from its bag to serve is five time units after the current time (*now*) of the sender. The assumption of synchronized local clocks for rebecs allows us to change the arguments of



s_0		
a	State vars:	
	Message Bag:	$[(null \rightarrow a.initial(),0,\infty)]$
	Now:	0
ts	State vars:	issueDelay=?
	Message Bag:	$[(null \rightarrow ts.initial(),0,\infty)]$
	Now:	0
c	State vars:	
	Message Bag:	$[(null \rightarrow c.initial(),0,\infty)]$
	Now:	0

I The initial state

s_{15}		
a	State vars:	
	Message Bag:	[ ]
	Now:	3
ts	State vars:	issueDelay=3
	Message Bag:	[ ]
	Now:	3
c	State vars:	
	Message Bag:	$[(a \rightarrow c.ticketIssued(1),3,\infty)]$
	Now:	0

II An intermediate state

Figure 4: The initial state and one of other states of the model of ticket service system which has been depicted in Figure 2. The receiver of each message is shown in the left-most column (as a, ts, c).

deadline and *after* to absolute values when putting the message in the bag of the receiver. This can be done by adding the values of the arguments to the value of *now* of the sender. The structure of sent messages is depicted in the following definition.

Definition 4 (Message in a Timed Rebeca Model)**.** *A tuple $tmsg = (sig, arrival, deadline)$ is a message where sig is the message signature, arrival is the arrival time of the message (equals to the value of "after" argument of the call statement added to the "now" of the sender), and deadline is the deadline of the message (equals to the value of "deadline" argument of the call statement added to the "now" of the sender).* □

For $tmsg \in bag(s, r_i)$ the functions $sig(tmsg)$, $ar(tmsg)$, and $dl(tmsg)$ return the *sig*, *arrival*, and *deadline* of the message *tmsg* respectively.

Definition 5 (Message Signature)**.** *For a message tmsg, the message signature consists of its name, the sender, the receiver, and the actual parameters in the form of "sender → receiver.name(parameters)".* □

Two different states of the Timed Rebeca model of Figure 2 are depicted in Figure 4. The state in Figure 4I is an initial state. The *now* of rebecs in initial state are set to zero and the *initial* message is put in their message bags. Initial messages are special messages with no sender. Figure 4II shows one of the intermediate states of the model. In Figure 4II, rebec *c* has a message from rebec *a* which will be delivered to *a* at time 3. As shown in Figure 4II, there is no guarantee on the equality of the local times of rebecs of a state; therefore, we call Timed Rebeca states "Floating-Time State (FTS)". To ease the reading of the paper, we use the word "state" instead of FTS in the paper.

The set of the next enabled messages of each rebec is defined based on the FIFO policy of Timed Rebeca using the arrival times of the messages. Enabled messages are the messages which arrived before the others; hence, should be executed before all the other messages in the bag.

Definition 6 (Enabled Messages of a Rebec)**.** *A set of enabled messages of rebec r_i in a state s is defined as $em(s, r_i) = \{tmsg \in bag(s, r_i) | \forall tmsg' \in bag(s, r_i), ar(tmsg) \leq ar(tmsg')\}$.* □

Timed Rebeca floating-time transition system is defined based on the above definitions. As an example, Figure 5 depicts floating-time transition system of the ticket-service model in Figure 2. To make the figure simpler, transition labels from state s_0 to state s_{13} are omitted.

Definition 7 (Timed Rebeca Floating-Time Transition System)**.** *A* Timed Rebeca floating-time transition system (FTTS) *is a labeled transition system $FTTS(\mathcal{M}) = (S, s_0, Act, \hookrightarrow)$, where:*

- *S is a set of states in Timed Rebeca,*
- *$s_0 \in S$ is the initial state,*
- *Act is a set of actions, containing all possible messages in Timed Rebeca,*

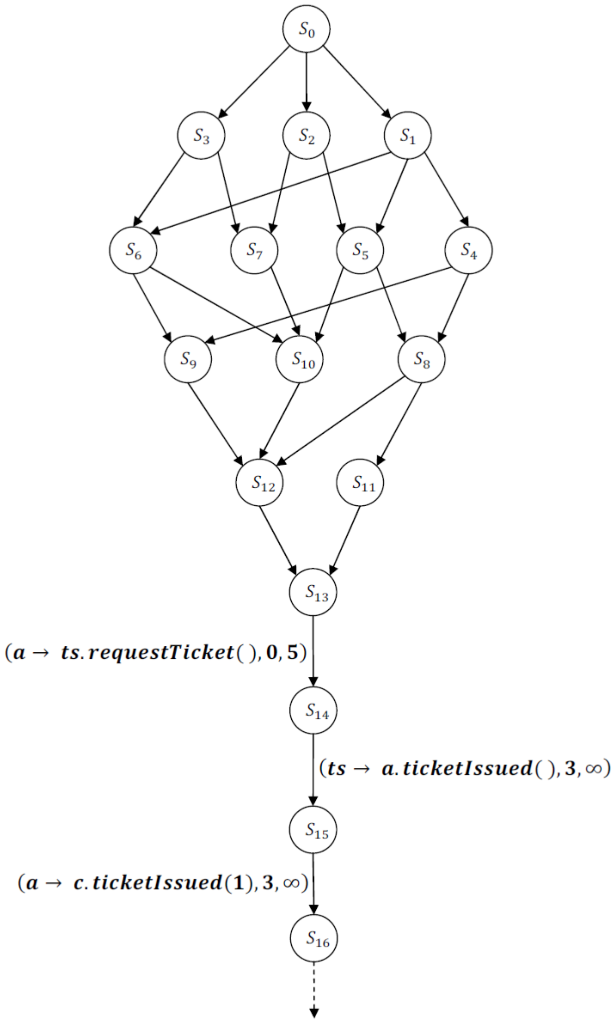

S_0
S_1
S_2
S_3
S_4
S_5
S_6
S_7
S_8
S_9
S_{10}
S_{11}
S_{12}
S_{13}
$(a \rightarrow ts.requestTicket(), 0, 5)$
S_{14}
$(ts \rightarrow a.ticketIssued(), 3, \infty)$
S_{15}
$(a \rightarrow c.ticketIssued(1), 3, \infty)$
S_{16}

Figure 5: Floating-Time transition system of the Timed Rebeca model of ticket service system which has been described in Figure 2.

- $\hookrightarrow \subseteq S \times Act \times S$ *is the set of transition relation, where* $\forall s, s' \in S, (s, tmsg, s') \in \hookrightarrow$ *iff there exists* $r_i \in O(\mathcal{M})$ *and* $tmsg \in em(s, r_i)$ *such that* $\forall r_j \in O(\mathcal{M}) \wedge \forall tmsg' \in em(s, r_j) \Rightarrow ar(tmsg) \leq ar(tmsg')$. *State* s' *results from* s *and* $tmsg$ *as follows:*
  - *tmsg is taken from the bag of* r_i,
  - *the value of now of* r_i *is set to* $\max\{now, ar(tmsg)\}$ *(i.e. the starting time for execution of tmsg),*
  - *The body of the message server corresponding to tmsg is executed. Execution of statements conforms to the Timed Rebeca SOS rules in [5] which can be modification of the state variables of* r_i*, sending messages to rebecs, and changing the value of now of* r_i *because of a delay statement.*

□

4. Bounded Floating-Time Transition System of Timed Rebeca

Unlike in Timed Automata, there is no explicit reset operator for time in Timed Rebeca. So, progress of time and hence the increasing values of *now* variables of each rebec results in an unbounded number of states in the floating-time transition system of the models. Due to the unbounded number of states, model checking of both SOS-based transition system and floating-time transition system is impossible. However, Timed Rebeca models are models of reactive systems which usually show a periodic or recurrent behavior, and hence if we abstract the absolute time away the models usually generate finite traces of computation.

As the time variables are changing in each transition, the tool can never recognize two states to be equivalent. But Observe that where the properties are only based on values of state variables, or the difference between timing primitives then the absolute value of timing primitives are no longer relevant. For example, consider a message as a request for a specific task. Catching a deadline-miss of a message in a state, is based on the difference between the value of *now* of the receiver rebec and the value of *deadline* of the message in that state. Based on this observation, we define an equivalence relation between two states called *shift equivalence relation*.

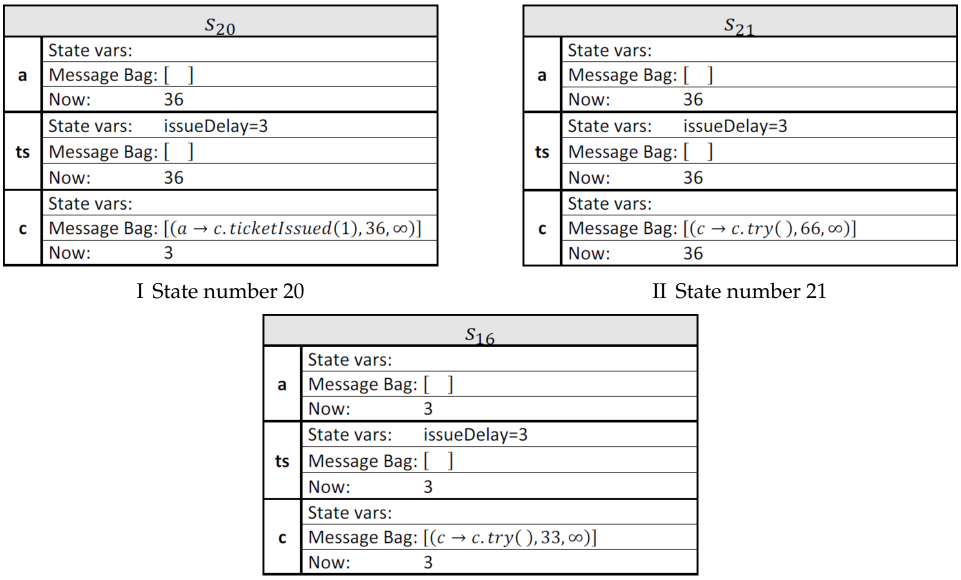
For an example see Figure 6 presenting three different states of the Timed Rebeca model of the ticket service system of Figure 2. Assume that the model is in the state shown in Figure 6I. Based on the bag of the rebecs *ts*, *c*, and *a*, $em(s_{20}, a) = \{\}$, $em(s_{20}, ts) = \{\}$, and $em(s_{20}, c) = \{(a \rightarrow c.ticketIssued(1), 36, \infty)\}$. Therefore, only rebec *c* has an enabled message whose execution results in s_{21}, shown in Figure 6II. Here, shifting back the time of s_{21} by 33 units, makes it equal to s_{16}, so s_{21} and s_{16} are shift equivalent by shifting 33 units, i.e. s_{21} projected to s_{16} by shifting 33 units, denoted by $s_{21} \cong_{33} s_{16}$.

Our method is similar to Lamport's work in [28] where he suggested that in most systems time can be modeled by an explicit variable, called *now*. In these systems, actions only depends on the passage of time, not the absolute time when the actions occur. In other words, the behavior of actions depends on their relative times not their absolute time. In Lamport model the value of *now* is incremented by a predefined *Tick* action. Then he introduced time-translation symmetry relation: two states are in time-translation symmetry relation iff they are the same except for their absolute time. The idea of shift equivalence relation is similar to time-translation symmetry relation, but in FTTS in each state we have a *now* variable for each rebec, and we have to consider the other timing primitives in the states too.

The shift equivalence relation is used to make floating-time transition system bounded. Now we proceed to present the formal definitions for shift equivalence relation between two states in FTTS.

Definition 8 (Shift Equivalence Relation between two states of FTTS with respect to a rebec)**.** *Two states* s *and* s' *are* shift equivalent *by* t *units with respect to rebec* r_i*, denoted as* $s' \cong_{i,t} s$*, if* $sv(s, r_i) = sv(s', r_i)$ *and one of the following conditions hold:*

- $now(s', r_i) = now(s, r_i)$ *and* $bag(s', r_i) = bag(s, r_i)$. *In this case* s' *and* s *are zero-time shift equivalent with respect to* r_i*, denoted by* $s' \cong_{i,0} s$.
- *There exists an integer number* t *such that* $now(s', r_i) = now(s, r_i) + t$ *and there is a bijective relation* $\leftrightarrow$ *between* $bag(s', r_i)$ *and* $bag(s, r_i)$ *such that for all* $tmsg' \in bag(s', r_i)$ *there is a message* $tmsg \in bag(s, r_i)$ *where* $sig(tmsg') = sig(tmsg) \;\wedge\; ar(tmsg') = ar(tmsg) + t \;\wedge\; dl(tmsg') = dl(tmsg) + t$.



s_{20}		
a	State vars:	
	Message Bag: [ ]	
	Now:	36
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	36
c	State vars:	
	Message Bag: $[(a \rightarrow c.ticketIssued(1), 36, \infty)]$	
	Now:	3

I State number 20

s_{21}		
a	State vars:	
	Message Bag: [ ]	
	Now:	36
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	36
c	State vars:	
	Message Bag: $[(c \rightarrow c.try(\,), 66, \infty)]$	
	Now:	36

II State number 21

s_{16}		
a	State vars:	
	Message Bag: [ ]	
	Now:	3
ts	State vars:	issueDelay=3
	Message Bag: [ ]	
	Now:	3
c	State vars:	
	Message Bag: $[(c \rightarrow c.try(\,), 33, \infty)]$	
	Now:	3

III State number 16

Figure 6: Three different states of the Timed Rebeca model of ticket service system

□

Note that the time primitives of Timed Rebeca are integer numbers, hence t is a natural number.

Two states of FTTS are in shift equivalence relation if and only if all the rebecs in these states are in shift equivalence relation.

Definition 9 (Shift Equivalence Relation between two states of FTTS)**.** *Two states s and s' are in shift equivalence relation, denoted as $s' \cong_t s$, iff there exists a $t \in \mathbb{N}$ such that $\forall r_i \in O(\mathcal{M})$ we have $s' \cong_{i,t} s$.* □

In addition to shift equivalence relation of two states, two messages $tmsg$ and $tmsg'$ are defined to be shift equivalent iff $sig(tmsg) = sig(tmsg')$ and $\exists t \in \mathbb{N} \cup \{0\}$ such that $ar(tmsg') = ar(tmsg) + t \ \wedge \ dl(tmsg') = dl(tmsg) + t$. Shift equivalence relation of messages is shown as $\cong_t$ (the same as the shift equivalence relation of states). Note that the time shift preserves the relative difference of timing primitives which results in the following lemmas.

Lemma 1 (Shift equivalence relation preserves enabled message)**.** *For two states s and s' where $s \cong_t s'$, and for each rebec r_i and message $tmsg \in em(s, r_i)$ there exists a message $tmsg' \in em(s', r_i)$ such that $tmsg \cong_t tmsg'$.*

Lemma 2 (Shift equivalence relation preserves effect of execution of message servers)**.** *If we have $s \cong_t s'$ and the execution of an enabled message* tmsg *in states s results in state q, then there exists a* tmsg$' \cong_t$ tmsg *in state s' and the execution of* tmsg$'$ *in state s' results in state q' where $q \cong_t q'$.*

Now the Timed Rebeca bounded floating-time transition system can be defined based on the above definitions. As an example, see Figure 7 which depicts the bounded floating-time transition system of the ticket-service model. To make Figure 7 simpler, transition labels from state s_0 to state s_{15} are omitted. As shown in Figure 7, a transition label is a pair consisting of the executed message and the time shift. The time shifts of all the transitions of Figure 7 are zero except for the transition from s_{20} to s_{16}, because of the equivalence relation defined in Definition 9.

Definition 10 (Timed Rebeca Bounded Floating-Time Transition System)**.** *A* Timed Rebeca Bounded Floating-Time Transition System (BFTTS) *is a labeled transition system* BFTTS$(\mathcal{M}) = (S, s_0, Act, \hookrightarrow, AP, L)$, *where:*

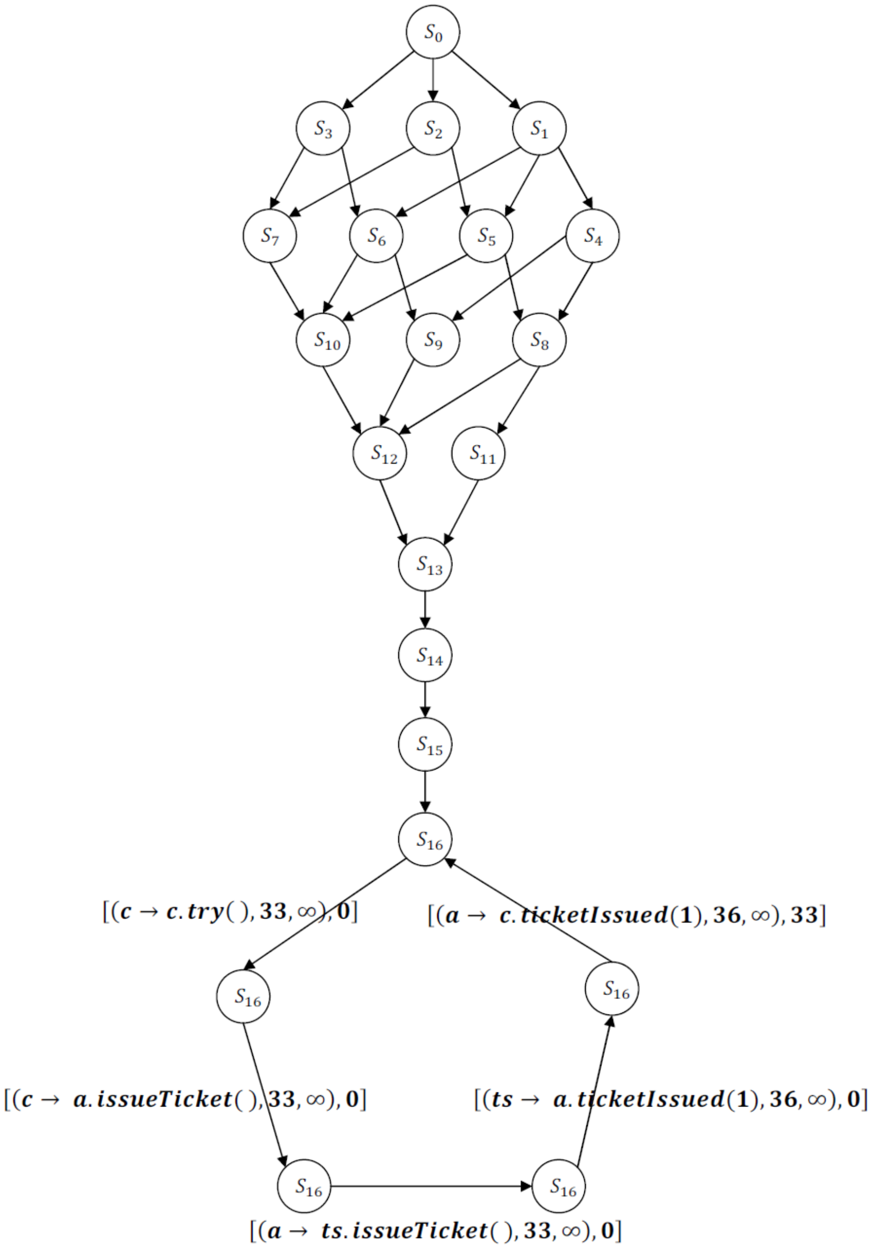

S_0
S_3
S_2
S_1
S_7
S_6
S_5
S_4
S_{10}
S_9
S_8
S_{12}
S_{11}
S_{13}
S_{14}
S_{15}
S_{16}
$[(c \to c.try(), 33, \infty), 0]$
$[(a \to c.ticketIssued(1), 36, \infty), 33]$
S_{16}
S_{16}
$[(c \to a.issueTicket(), 33, \infty), 0]$
$[(ts \to a.ticketIssued(1), 36, \infty), 0]$
S_{16}
S_{16}
$[(a \to ts.issueTicket(), 33, \infty), 0]$

Figure 7: Floating-Time transition system of the Timed Rebeca model of Figure 2.

- *S is a set of states,*
- *$s_0 \in S$ is the initial state,*
- *Act is a set of actions, containing pairs of all possible messages that can be sent in the model and a natural number as the time shift,*
- *$\hookrightarrow \subseteq S \times Act \times S$ is the set of transition relation, where $\forall s, s' \in S, (s, (tmsg, t), s') \in \hookrightarrow$ iff there exists $r_i \in O(\mathcal{M})$ and $tmsg \in em(s, r_i)$ such that $\forall r_j \in O(\mathcal{M}) \wedge \forall tmsg' \in em(s, r_j) \Rightarrow ar(tmsg) \leq ar(tmsg')$, and the execution of tmsg results in a state s'' where $s'' \cong_t s'$. State s'' results from s and tmsg as follows:*
  - *tmsg is taken from the bag of r_i,*
  - *the value of now of r_i is set to $\max\{now, ar(tmsg)\}$ (i.e. the starting time for execution of tmsg),*
  - *The body of the message server corresponding to tmsg is executed. Execution of statements conforms to the Timed Rebeca SOS rules in [5] which can be modification of the state variables of r_i, sending messages to rebecs, and changing the value of now of r_i because of a delay statement.*

□

There is a non-deterministic choice in those states with more than one enabled message.

As shown in the following theorem, for a Timed Rebeca model $\mathcal{M}$ there is a bisimulation relation between $FTTS(\mathcal{M})$ and $BFTTS(\mathcal{M})$. Therefore, analysis over BFTTS, which has bounded number of states, gives in the same results as for $FTTS(\mathcal{M})$.

Theorem 1. *For a Timed Rebeca model $\mathcal{M}$, $FTTS(\mathcal{M}) = (S, s_0, act, \hookrightarrow)$ is bisimilar to the Bounded Timed Rebeca floating-time transition system $BFTTS(\mathcal{M}) = (S', s_0', act', \hookrightarrow')$.*

Proof. *From Lemma 1 and Lemma 2 we can see that conditions I and II of bisimulation relation hold as for two states $s \in FTTS(\mathcal{M})$ and $s' \in BFTTS(\mathcal{M})$, the enabled messages of s and s' are the same and execution of the enabled messages results in the shift equivalent states.* □

5. Schedulability and Deadlock Freedom Analysis

Schedulability and deadlock freedom are two safety properties of real-time models. A Timed Rebeca model $\mathcal{M}$ is schedulable if and only if none of the messages misses their deadline. This property can be formulated as in Definition 11.

Definition 11 (Schedulability of Timed Rebeca Model). *A given Timed Rebeca model $\mathcal{M}$ is schedulable if and only if for $BFTTS(\mathcal{M}) = (S, s_0, Act, \hookrightarrow)$ there is no transition $(s, (tmsg, t), s') \in \hookrightarrow$ such that $tmsg \in em(s, r_i) \wedge dl(tmsg) > \max\{now(s, r_i), ar(tmsg)\}$.*

Deadlock freedom property for Timed Rebeca models is defined in the same way as deadlock freedom for Rebeca. Deadlock happens in a Timed Rebeca model $\mathcal{M}$ if and only if $\mathcal{M}$ reaches a state in which there is no enabled message in the bag of any of the rebecs. This state is called a deadlock state.

Definition 12 (Deadlock Freedom of Timed Rebeca Model). *A given Timed Rebeca model $\mathcal{M}$ is deadlock free if and only if for $BFTTS(\mathcal{M}) = (S, s_0, Act, \hookrightarrow)$ there is no state $s \in S$ such that $\forall r_i \in O(\mathcal{M}), em(s, r_i) = \emptyset$.*

These two properties are verified during the state space generation of models. To generate BFTTS of Timed Rebeca models we modify Rebeca model checking engine. Rebeca comes equipped with an on-the-fly explicit-state LTL model checking engine, called Modere [22]. Modere uses both the nested DFS and BFS search algorithms to explore the state space. To generate state space based on semantics of floating-time transition system, BFS search algorithm of Modere has been extended to support Timed Rebeca, incorporating detection of shift equivalent states.

Rebeca BFS State Space Generation Algorithm. The BFS state space generation algorithm, creates and explores the transition system in a level-by-level fashion. In the first level of the BFS algorithm, the initial

```
1   BFS-STATE-SPACE-GENERATOR ($\mathcal{M}$)
2     $CLQ \leftarrow \emptyset$
3     $NLQ \leftarrow \emptyset$
4     $Visited \leftarrow \emptyset$
5     ENQUEUE ($CLQ$, INITIAL_STATE($\mathcal{M}$))
6     while $CLQ \neq \emptyset$ do
7       for each state $S \in CLQ$ do
8         NewStates $\leftarrow$ SUCCESSOR($S$, $\mathcal{M}$)
9         for each State $N \in NewStates$ do
10          $ID \leftarrow$ HASH_CODE($N$)
11          if $ID \notin Visited$
12            then PUT($Visited$, $N$)
13                 ENQUEUE($NLQ$, $N$)
14          fi
15        od
16      od
17      swap($CLQ$, $NLQ$)
18      $NLQ \leftarrow \emptyset$
19    od
```

Figure 8: Pseudo code of state space generation algorithm of Modere based on the BFS search algorithm.

state of a Rebeca model is generated and marked as visited. Then, for each level, the successors of the states of that level are generated by applying the successor function, called next level states. When there are no unvisited states in the next level states, the algorithm terminates. For more details about the successor function and its formal semantics refer to [43].

The BFS state space generation algorithm is implemented using two queues. The first queue stores the states of current level (CLQ) and the second one stores the successors of the states of CLQ, called next level queue (NLQ). In each iteration, the states of the CLQ are dequeued and their unvisited successors are generated and are put in the NLQ. When all states of the CLQ are dequeued, the content of the NLQ is swapped with the CLQ and the algorithm continues until NLQ becomes empty, i.e., all successors of the states of CLQ have been visited before. Pseudo code of this algorithm is depicted in Figure 8. As shown in lines 10 and 11, the visited states are detected using a hash code generator function. Hash code generator function assigns a unique identifier (in natural number domain) to each state based on the state content, i.e. values of state variables and message signatures of message queues of all rebecs.

Timed Rebeca BFS State Space Generation Algorithm. The major difference between BFS state space generation algorithm in Modere and in Timed Rebeca analysis tool is in detecting shift equivalence among states. Shift equivalence detection is implemented by modifying *visited* data structure in algorithm of Figure 8. In BFS algorithm of Figure 8, *visited* is a hash table which maps each *ID* to a single state. In Timed Rebeca extension of BFS algorithm, *visited* maps each *ID* to a list of states. *ID*s of states in Timed Rebeca BFS algorithm are not unique because two states which are the same except in the value of *now*, *arrival* of a message, or *deadline* of a message, has the same *ID*. Hence, two shift equivalent states has the same *ID* but different values for timing primitives. If differences between all the timing primitives of two states are the same and they have the same *ID*s, these states are shift equivalent. Details of detecting shift equivalence among states are shown in lines 12 to 30 of Figure 9. Inputs of the algorithm are the model $\mathcal{M}$ and identifiers of rebecs in $\mathcal{M}$.

procedures *CHECK-FOR-DEADLINE-MISSED(N)* and *CHECK-FOR-DEADLOCK(N)* are implemented based on the definitions 11 and 12 to verify the schedulability and deadlock freedom in a given state.

Model checker of Timed Rebeca is developed and is integrated in Afra [2]. Afra is the modeling and verification IDE of Rebeca and Timed Rebeca. It is an eclipse plugin and the non-commercial distribution of Afra can be downloaded from Rebeca homepage [2].

6. Experimental Results

We provide three different case studies in different sizes to illustrate the performance of using BFTTS in comparison to UPPAAL and McErlang for model checking Timed Rebeca models. These three model checkers are installed on servers with 4 CPUs and 16GB of RAM storage, running Ubuntu 12.04 as the

```
BFS-STATE-SPACE-GENERATOR ($\mathcal{M}, r_0, ..., r_n$)
  $CLQ \leftarrow \emptyset$
  $NLQ \leftarrow \emptyset$
  $Visited \leftarrow \emptyset$
  ENQUEUE ($CLQ$, INITIAL_STATE($\mathcal{M}$))
  while $CLQ \neq \emptyset$ do
    for each state $S \in CLQ$ do
      NewStates ← SUCCESSOR($S$, $\mathcal{M}$)
      for each State $N \in NewStates$ do
        ID ← HASH_CODE($N$)
        if $ID \notin Visited$
          then PUT($Visited$, $N$)
               ENQUEUE($NLQ$, $N$)
               CHECK-FOR-DEADLINE-MISSED(N)
               CHECK-FOR-DEADLOCK(N)
        else
          $VisitedStates$ ← GET($Visited$, ID)
          for each VisitedState ∈ $VisitedStates$ do
            if VisitedState and N are shift equivalent
              then
                // discard N
              else
                PUT($Visited$, $N$)
                ENQUEUE($NLQ$, $N$)
                CHECK-FOR-DEADLINE-MISSED(N)
                CHECK-FOR-DEADLOCK(N)
            fi
          od
        fi
      od
    od
    swap($CLQ$, $NLQ$)
    $NLQ \leftarrow \emptyset$
  od
```

Figure 9: Pseudo code of state space generation algorithm of Timed Rebeca based on the BFS search algorithm.

operating system. The selected case studies are *Distributed Sensor Network*, simplified version of *Slotted ALOHA Protocol*, and *Ticket Service*. The Timed Rebeca code of the case studies are in Rebeca homepage [2].

Sensor Network The distributed sensor network is a model of a set of sensors that measure the toxic gas level of the environment. Upon sensing a dangerous level of gas, the system notifies the scientist who is working in that area. In the case that no acknowledgment is received from the scientist the system sends a rescue team to the area.

There are four reactive classes *Sensor*, *Admin*, *Scientist*, and *Rescue* in the model. *Sensor* rebecs send the measured gas level value to *Admin* rebec over the network. If *Admin* receives a report of dangerous gas levels, it notifies *Scientist* immediately and waits for the *Scientist* acknowledgement. If *Scientist* does not respond, *Admin* requests *Rescue* to reach and save *Scientist*. The main property to be checked is saving *Scientist* before the rescue deadline is missed. We have different models with different numbers of sensors to produce state spaces of different sizes. This case study is first presented in [5] and its source code is in http://www.rebeca-lang.org/wiki/pmwiki.php/Examples/SensorNetwork.

Slotted ALOHA Protocol The Slotted ALOHA protocol [3] controls access to the data link medium of computer networks. Slotted ALOHA divides the time into some slots. At each slot, one of the network interfaces, which are connected to the data link medium, is allowed to send its data via the medium. The other interfaces are sniff the medium for incoming data when some one sends data. We have modeled the Slotted ALOHA using four different reactive classes *User*, *Interface*, *Medium*, and *Controller*. To make the model more realistic, we linked rebec *User* to each *Interface* which provides the *Interface* data. We generated different sizes of models by varying the number of *User*s and *Interface*s. The source code of this case study is in http://www.rebeca-lang.org/wiki/pmwiki.php/Examples/SlottedAloha.

Ticket Service. Details of *Ticket Service* case study is explained in Section 2. Catching the deadline of issuing the ticket is the main property of this model. We achieved different size of ticket service model by varying the number of customers. The source code of this case study is in http://www.rebeca-lang.org/wiki/pmwiki.php/Examples/ TicketServiceSystem.

Problem	Size	Using BFTTS		Using Timed Automata		Using McErlang	
		#states	time	#states	time	#states	time
Ticket Service	**1 customer**	8	< 1 sec	801	<1 sec	150	<1 sec
	2 customers	51	< 1 sec	19M	5 hours	4.5k	3 secs
	3 customers	280	< 1 sec	-	>24 hours†	190K	5.1 mins
	4 customers	1.63K	< 1 sec	-	>24 hours†	> 4M‡	-
	5 customers	11K	< 1 sec	-	>24 hours†	> 4M‡	-
	6 customers	83K	2 secs	-	>24 hours†	> 4M‡	-
	7 customers	709K	3 mins	-	>24 hours†	> 4M‡	-
	8 customers	6.8M	9.7 hours	-	>24 hours†	> 4M‡	-
Sensor Network	**1 sensor**	183	< 1 sec	-	>24 hours†	> 6.5M‡	-
	2 sensors	2.4K	< 1 sec	-	>24 hours†	> 6M‡	-
	3 sensors	33.6K	1 sec	-	>24 hours†	> 6M‡	-
	4 sensors	588K	13 secs	-	>24 hours†	> 6M‡	-
Slotted ALOHA Protocol	**1 interface**	68	< 1 sec	-	>24 hours†	153K	1.8 secs
	2 interfaces	750	< 1 sec	-	>24 hours†	> 2.8M‡	-
	3 interfaces	7.84K	1 sec	-	>24 hours†	> 2.8M‡	-
	4 interfaces	45.7K	6 secs	-	>24 hours†	> 2.8M‡	-
	5 interfaces	331K	64 secs	-	>24 hours†	> 2.8M‡	-

Table 1: Model checking time and size of state space, using three different tools. The † sign on the reported time shows that model checking takes more than the time limit (24 hours). The ‡ sign on the reported number of states shows that state space explosion occurs as the model checker want to allocate more than 16GB in memory which is more than total amount of memory.

We set limit on maximum time and maximum memory consumption of each round of model checking. Each round duration should not exceed 24 hours, and consumes less than 16GB of RAM storage. The results of model checking these case studies using the three different tools are depicted in Table 1. All the case studies hit their deadlines and are deadlock free. Table 1 shows that using UPPAAL or McErlang for model checking Timed Rebeca, results in the state space explosion at the preliminary steps.

Increasing the service time and message passing delay in the case studies results in deadline-miss in some cases. For example, as shown in Table 2 the modified version of ticket service model missed its deadline when there are more than four customers. The deadline-miss happens in this case because issuing a ticket takes three time units and the response to the customers' requests should be sent in eight time units. Therefore, when more than four customers request for ticket, the last request is processed after its deadline. The same modifications are applied to the other two case studies. The results of model checking three case studies by increasing the service times is depicted in Table 2.

We also made some modifications on the case studies by adding non-determinism to sending messages as the following.

- Ticket Service: *TicketService* rebec non-deterministically discards some of the requests of a customer and does not issue a ticket in that case.
- Sensor Network: *Sensor*s stop working non-deterministically. Sensors do not send information about the environment of the scientist when they stop working.
- Slotted ALOHA Protocol: The *controller* of the *medium* stops working non-deterministically. Therefore, none of the interfaces can send data via the medium.

The Added non-determinism results in system deadlock. The state space size and time consumption of model checking the case studies to find deadlock states are depicted in Table 3.

Problem	Size	#states	time	result
Ticket Service	**1 customer**	8	< 1 sec	hit deadline
	2 customers	51	< 1 sec	hit deadline
	3 customers	280	< 1 sec	hit deadline
	4 customers	770	< 1 sec	deadline missed
	5 customers	4.92K	< 1 sec	deadline missed
	6 customers	38K	< 1 sec	deadline missed
	7 customers	316K	6 secs	deadline missed
	8 customers	3M	1 min	deadline missed
Sensor Network	**1 sensor**	206	< 1 sec	deadline missed
	2 sensors	2.8K	< 1 sec	deadline missed
	3 sensors	47.7k	2 secs	deadline missed
	4 sensors	1.14M	26 secs	deadline missed
Slotted ALOHA Protocol	**1 interface**	68	< 1 sec	hit deadline
	2 interfaces	750	< 1 sec	hit deadline
	3 interfaces	5.16K	< 1 sec	hit deadline
	4 interfaces	12K	1 sec	deadline missed
	5 interfaces	41K	1 sec	deadline missed

Table 2: Model checking the modified version of the three case studies which missed their deadlines in some cases.

Problem	Size	#states	time	result
Ticket Service	**1 customer**	5	< 1 sec	deadlock
	2 customers	25	< 1 sec	deadlock
	3 customers	180	< 1 sec	deadlock
	4 customers	1.4K	< 1 sec	deadlock
	5 customers	11.7K	< 1 sec	deadlock
	6 customers	108K	2 secs	deadlock
	7 customers	1.14	22 secs	deadlock
	8 customers	13M	7.6 mins	deadlock
Sensor Network	**1 sensor**	19	< 1 sec	deadlock
	2 sensors	147K	< 1 sec	deadlock
	3 sensors	23.7k	< 1 sec	deadlock
	4 sensors	1.14M	26 secs	deadlock
Slotted ALOHA Protocol	**1 interface**	57	< 1 sec	deadlock
	2 interfaces	277	< 1 sec	deadlock
	3 interfaces	1.2K	1 sec	deadlock
	4 interfaces	4.9K	1 sec	deadlock
	5 interfaces	20K	9 secs	deadlock

Table 3: Model checking the modified version of the three case studies which have deadlock state.

7. Related Work

Here, we give an overview of the widely used timed models and their analysis tools and techniques: real-time Maude [36], timed automata [8] and TLA+ [27]. Then we explain the existing analysis tools for Timed Rebeca.

Real-Time Maude. Maude is a high level declarative programming language supporting specification of models in rewriting logic. Maude is used for modeling nondeterministic concurrent computations and also concurrent object oriented models. Real-Time Maude language [36, 37] is an extension on Maude language [14] for modeling real-time and hybrid systems, and their simulation, and verification. The rewriting rules are divided into two categories, the instantaneous rules that model instantaneous changes done in zero time, and a predefined *tick* rule that model the elapse of time [34].

A set of tools are developed for analysis of real-time Maude. *Timed rewrite* builds a trace in the execution of system from the initial state up to a certain time. *Timed search* builds all behaviors of the system from the initial state up to a certain time and checks whether a specific state is reachable or not. *Timed model checking* checks whether all possible behaviors satisfy a temporal logic formula. Real-Time Maude extends Maude's LTL model checker to a time-bounded TLTL model checker. Recently, Real-Time Maude is equipped with a model checker for timed computation tree logic (TCTL) properties [30].

Comparing to Timed Rebeca, although real-time Maude is a powerful and flexible modeling language supported by a rich set of analysis tools, it needs intimate knowledge of the theory behind rewriting logic to adapt its computational model to actors. Timed Rebeca benefits from its similarity with other commonly used programming languages and it is more susceptible to get used by practitioners. Moreover, the actor features help in applying certain reductions in model checking as shown in this paper and in [42]. We already have simulation and model checking tools for Timed Rebeca, but standard TLTL or TCTL properties cannot be checked on bounded floating-time transition system of Timed Rebeca. Building event-based property checking tools based on BFTTS and the property language proposed in [32] is an ongoing work.

Timed Automata. Timed automata [8] model the behavior of timed systems using a set of automata that is equipped with the set of clock variables. Although clocks are the system variables, their values can only be checked or set to zero. Therefore, they can be intuitively considered as stop watches. The values of all clocks are increased in the same rate. Values of clocks can be reset on transitions of automata. Constraints over clocks can be added as enabling conditions on both states and transitions. Timed automata supports parallel composition as a convenient approach for modeling complex systems. As depicted in [10], parallel composition of timed automata is based on the handshaking actions.

There are two well-known toolsets for verification of timed automata, UPPAAL [12] and Kronos [13]. Uppaal is an integrated environment for verification of real-time systems, jointly developed by Uppsala and Aalborg Universities [11]. The tool is designed to verify systems that can be modeled as networks of timed automata, extended with data types (bounded integers, arrays, functions, etc.). UPPAAL provides both simulator and model checker for its models. Model checker of UPPAAL verifies TCTL properties. Kronos is a toolbox build with the aim of providing a verification engine to be integrated into design environment of real-time systems. Correctness criteria for Kronos verification engine can be specified in TCTL formula or timed automata. Kronos implements a symbolic verification method based on predicate transformation [15].

Modeling asynchronous message passing of actors using synchronous communication of timed automata results in a large timed automata. Therefore, the state space of large-scale practical real-time systems undoubtedly results in state spaces that cannot fit in the memory of a computer. In addition to memory limitation, the model checking time increases rapidly and makes the model checking impossible. As timed automata and UPPAAL were our first choice for model checking Timed Rebeca [19] we noticed this deficiency immediately. This point is also mentioned in [28] on modeling distributed systems using timed automata.

TLA+. TLA+ is a formal specification language that extends Temporal Logic of Actions (TLA) by set theory and first-order logic [27]. A TLA specification is a temporal formula, often named *Spec*, showing the initial state predicate, the actions, and the specification variables. Applying rules of actions on the initial state results in changing the values of specification variables and generation of the next states of the system. TLA+ includes modules and ways of combining them to form larger specifications [29].

TLC [45] is an on-the-fly model checker of TLA+ which uses explicit state representation. Although there are many constructs in TLA+, TLC can handle a subclass of TLA+ specifications that seems to include the ones that are needed in describing actual systems [29]. TLC verifies both safety and liveness properties. To verify safety properties of a model, TLC explores all reachable states of the given model to find a state in which an invariant is not satisfied or deadlock occurs. Liveness properties are model checked using tableau method of Manna et al. in [33].

TLA+ is a high level specification language which needs knowledge of set theory and logic. The main similarity between our work and Lamport's work in analyzing our timed models is the similarity between shift equivalent relation and symmetry relation.

Erlang. Erlang is a dynamically-typed general-purpose programming language which was developed in 1986 [9]. Erlang was mostly used for telephony applications such as switches. Erlang is designed for the implementation of distributed, real-time and fault-tolerant applications. Its concurrency model is based on the actor model.

McErlang is a model checker for Erlang. McErlang [17] supports full Erlang features. McErlang offers ability of expressing correctness properties in the form of monitors (safety or Büchi), abstraction algorithms to reduce state-space, and exploration algorithms to verify or simulate Erlang programs [17]. Fredlund et. al. in [16] proposed timed extension of McErlang as a model checker of timed Erlang programs. In this extension a new API is introduced to provide the definition and manipulation of time-stamps.

Erlang as a programming language and Timed Rebeca as a modeling language are both targeting actor systems. However, their model checkers follow different approaches. McErlang provides fine-grain model checker for Erlang systems which results in state space explosion quickly. In contrast, states in bounded floating-time transition system of Timed Rebeca are coarse-grain and more abstract than that of McErlang. Our experimental results show very well the efficiency of our approach.

Analysis of Timed Rebeca. Before introducing bounded floating-time transition system, two other approaches have been developed for verification of Timed Rebeca models. These approaches translate Timed Rebeca model to timed automata and Erlang to use their back-end model checkers as verification engine [19, 5]. An ongoing project is using real-time Maude as the back-end analysis engine. As mentioned above, because of difficulties of modeling asynchronous message passing in timed automata, and fine-grain model checking of McErlang and real-time Maude, all these three approaches result in early state space explosion. The state space generation approach that we have used in BFTTS is an innovative and novel technique that gives us a significant reduction and enables us to model check larger systems.

8. Conclusion and Future Work

In this paper we introduced the floating-time transition system for schedulability and deadlock freedom analysis of Timed Rebeca models. Floating-Time transition system exploits the key features of Timed Rebeca. In summary, having no shared variables, no blocking send or receive, single-threaded actors, and non-preemptive execution of each message server give us an isolated message server execution, meaning that execution of a message server of a rebec will not interfere with execution of a message server of another rebec. Moreover, for checking schedulability and deadlock freedom we can focus only on events. In FTTS each transition shows releasing an event, or in other words execution of a message server of a rebec. Hence, in each state in FTTS rebecs may have different local times, but the transitions still gives us a correct order of release times of events of a specific rebec. The floating-time transition system of Timed Rebeca models is derived from the SOS semantics presented in [5]. Our proposed approach is implemented as a part of Afra toolset [2]. Experimental evidence supports that direct model-checking of Timed Rebeca models using floating-time transition system decreases both model-checking state space size and time consumption in comparison with translating to secondary models such as timed automata and Erlang. Therefore, we can efficiently model-check more complex models.

In addition, our technique is based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. So, the proposed transition system and analysis techniques are general enough to be applied to similar computation models where they have message-driven communication and autonomous objects as units of concurrency such as agent-based systems.

In an ongoing work at University of Illinois at Urbana-Champaign, Timed Rebeca and our model checking tool are being used for modeling and analysis of distributed real-time sensor network applications. As a case study a real-time continuous sensing application for structural health monitoring [1] is considered which is built largely from component middleware services of the Illinois SHM Services Toolsuite [31]. This is an example where efficient resource utilization is critical, since it directly determines the scalability (number of nodes) and fidelity (sampling frequency) of the data acquisition process. Timed Rebeca is being used to find a configuration and scheduling that improves resource utilization.

We are now improving our tool to support the event-based property language proposed in [32]. This will give us the ability of checking a subset of MTL-like properties where the propositions are defined on events instead of states.

Acknowledgements

The work on this paper has been partially supported by the project Timed Asynchronous Reactive Objects in Distributed Systems: TARO (nr. 110020021) of the Icelandic Research Fund.

References

[1] *Illinois SHM Services Toolsuite*. http://shm.cs.illinois.edu/software.html.

[2] *Rebeca Home Page*. http://www.rebeca-lang.org.

[3] Norman Abramson. THE ALOHA SYSTEM: Another Alternative for Computer Communications. In *AFIPS '70 (Fall): Proceedings of the November 17-19, 1970, fall joint computer conference*, pages 281–285, New York, NY, USA, 1970. ACM.

[4] Luca Aceto, Matteo Cimini, Anna Inglfsdttir, Ali Jafari, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and simulation of asynchronous real-time systems using timed rebeca. *Journal version, sent for publication*, 2013. http://rebeca.cs.ru.is/files/Papers/2013/Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca.pdf.

[5] Luca Aceto, Matteo Cimini, Anna Ingólfsdóttir, Arni Hermann Reynisson, Steinar Hugi Sigurdarson, and Marjan Sirjani. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In Mohammad Reza Mousavi and António Ravara, editors, *FOCLASA*, volume 58 of *EPTCS*, pages 1–19, 2011.

[6] Gul A. Agha. *ACTORS - A Model of Concurrent Computation in Distributed Systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[7] Rajeev Alur, Costas Courcoubetis, and David L. Dill. Model-Checking for Real-Time Systems. In *LICS*, pages 414–425, 1990.

[8] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[9] Joe Armstrong. A History of Erlang. In Barbara G. Ryder and Brent Hailpern, editors, *HOPL*, pages 1–26. ACM, 2007.

[10] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[11] Gerd Behrmann, Alexandre David, and Kim Guldstrand Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.

[12] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems. In Rajeev Alur, Thomas A. Henzinger, and Eduardo D. Sontag, editors, *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer, 1995.

[13] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In Alan J. Hu and Moshe Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer, 1998.

[14] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[15] Conrado Daws and Sergio Yovine. Two Examples of Verification of Multirate Timed Automata with Kronos. In *RTSS*, pages 66–75. IEEE Computer Society, 1995.

[16] Clara Benac Earle and Lars-Åke Fredlund. Verification of Timed Erlang Programs Using McErlang. In Holger Giese and Grigore Rosu, editors, *FMOODS/FORTE*, volume 7273 of *Lecture Notes in Computer Science*, pages 251–267. Springer, 2012.

[17] Lars-Åke Fredlund and Hans Svensson. McErlang: A Model Checker for a Distributed Functional Programming Language. In Ralf Hinze and Norman Ramsey, editors, *ICFP*, pages 125–136. ACM, 2007.

[18] C. Hewitt. 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.

[19] Mohammad-Javad Izadi. 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.

[20] Mohammad Mahdi Jaghoori, Frank S. de Boer, Tom Chothia, and Marjan Sirjani. Schedulability of Asynchronous Real-Time Concurrent Objects. *J. Log. Algebr. Program.*, 78(5):402–416, 2009.

[21] Mohammad Mahdi Jaghoori, Frank S. de Boer, and Marjan Sirjani. Task Scheduling in Rebeca. In *The 19th Nordic Workshop on Programming Theory, Oslo, Norway, October 10-12*, pages 16–18, 2007.

[22] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. Modere: The Model-Checking Engine of Rebeca. In Hisham Haddad, editor, *SAC*, pages 1810–1815. ACM, 2006.

[23] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca. *Acta Inf.*, 47(1):33–66, 2010.

[24] Ehsan Khamespanah, Zeynab Sabahi Kaviani, Ramtin Khosravi, Marjan Sirjani, and Mohammad-Javad Izadi. Timed-Rebeca Schedulability and Deadlock-Freedom Analysis Using Floating-Time Transition System. In *SPLASH*. ACM, 2012.

[25] Ron Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[26] Haukur Kristinsson. Event-Based Analysis of Real-Time Actor Models. Master's thesis, Reykjavk University, School of Computer Science, Iceland, 2012. http://rebeca.cs.ru.is/files/Thesis/EVENT-BASED ANALYSIS OF REAL-TIME ACTOR MODELS - Haukur Kristinsson.pdf.

[27] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[28] Leslie Lamport. Real-Time Model Checking Is Really Simple. In Dominique Borrione and Wolfgang J. Paul, editors, *CHARME*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.

[29] Leslie Lamport, John Matthews, Mark R. Tuttle, and Yuan Yu. Specifying and Verifying Systems with TLA+. In Gilles Muller and Eric Jul, editors, *ACM SIGOPS European Workshop*, pages 45–48. ACM, 2002.

[30] Daniela Lepri, Erika Ábrahám, and Peter Csaba Ölveczky. Timed CTL Model Checking in Real-Time Maude. In Francisco Durán, editor, *WRLA*, volume 7571 of *Lecture Notes in Computer Science*, pages 182–200. Springer, 2012.

[31] Lauren Linderman, Kirill Mechitov, and Billie F. Spencer. TinyOS-Based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control. *Structural Control and Health Monitoring*, 2012.

[32] Brynjar Magnusson. Simulation-Based Analysis of Timed Rebeca Using TeProp and SQL. Master's thesis, Reykjavk University, School of Computer Science, Iceland, 2012. http://rebeca.cs.ru.is/files/Thesis/SIMULATION-BASED ANALYSIS OF TIMED REBECA USING TEPROP AND SQL - Brynjar Magnusson.pdf.

[33] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification*. Springer, 1992.

[34] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude: A Tool for Simulating and Analyzing Real-Time and Hybrid Systems. *Electr. Notes Theor. Comput. Sci.*, 36:361–382, 2000.

[35] Peter Csaba Ölveczky and José Meseguer. Specification of Real-Time and Hybrid Systems in Rewriting Logic. *Theor. Comput. Sci.*, 285(2):359–405, 2002.

[36] Peter Csaba Ölveczky and José Meseguer. Specification and Analysis of Real-Time Systems Using Real-Time Maude. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 354–358. Springer, 2004.

[37] Peter Csaba Ölveczky and José Meseguer. Real-Time Maude 2.1. *Electr. Notes Theor. Comput. Sci.*, 117:285–314, 2005.

[38] Peter Csaba Ölveczky and José Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20(1-2):161–196, 2007.

[39] Shangping Ren and Gul Agha. RTsynchronizer: Language Support for Real-Time Specifications in Distributed Systems. In Richard Gerber and Thomas J. Marlowe, editors, *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 50–59. ACM, 1995.

[40] Hamideh Sabouri and Marjan Sirjani. Slicing-Based Reductions for Rebeca. In *Proceedings of FACS 2008*. ENTCS, 2008.

[41] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. Modular Verification of a Component-Based Actor Language. *J. UCS*, 11(10):1695–1717, 2005.

[42] Marjan Sirjani and Mohammad Mahdi Jaghoori. Ten Years of Analyzing Actors: Rebeca Experience. In Gul Agha, Olivier Danvy, and José Meseguer, editors, *Formal Modeling: Actors, Open Systems, Biological Systems*, volume 7000 of *Lecture Notes in Computer Science*, pages 20–56. Springer, 2011.

[43] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Inform.*, 63(4):385–410, 2004.

[44] Wang Yi. CCS + Time = An Interleaving Model for Real Time Systems. In Javier Leach Albert, Burkhard Monien, and Mario Rodríguez-Artalejo, editors, *ICALP*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer, 1991.

[45] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA^{+} Specifications. In Laurence Pierre and Thomas Kropf, editors, *CHARME*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.