Floating Time Transition System: More Efficient Analysis of Timed Actors

Ehsan Khamespanah[1,2], Marjan Sirjani[2], Mahesh Viswanathan[3], and Ramtin Khosravi[1]

[1] School of Electrical and Computer Engineering, University of Tehran - Iran
[2] School of Computer Science, Reykjavik University - Iceland
[3] Department of Computer Science University of Illinois at Urbana-Champaign - USA

Abstract. The actor model is a concurrent object-based language in which event-driven and asynchronously communicating actors are units of concurrency. Actors are widely used in modeling real-time and distributed systems. Floating-Time Transition System (FTTS) is proposed as an alternative semantics for timed actors, and schedulability and deadlock-freedom analysis techniques are developed for FTTS. The absence of shared variables and blocking send or receive, and the presence of single threaded actors along with non-preemptive execution of each message server, ensures that the execution of message servers do not interfering with each other. The Floating-Time Transition System semantics exploits this by executing message servers in isolation, and relaxing the synchronization of progress of time among actors, and thereby has fewer states in the transition system. Considering an actor-based language, we prove the weak bisimulation relation between FTTS and Timed Transition System which is generally the standard semantic framework for discrete timed systems. Thus, the FTTS semantics preserves event-based branching-time properties. Our experimental results show a significant reduction in the state space for most of the examples we have studied.

Keywords: Actor model, Timed Rebeca, Verification, State Space Reduction, Floating Time Transition System, Timed Transition System

1 Introduction

The semantics of real-time systems is often defined assuming an ambient global time that ticks uniformly for all participants in a distributed system. Even when individual local clocks are assumed to have skews, these skews are modelled relative to this ambient global time. For systems where the time domain is taken to be discrete (i.e., the set of natural numbers), this results in the semantics being described using a *Timed Transition System (TTS)*. In a timed transition system, transitions are partitioned into two classes: instantaneous transitions (in which time does not progress), and time ticks when the global clock is incremented. These time ticks happen when all participants "agree" for time elapse. Such TTS-based semantics is standard and has been defined for a variety of formalisms [17,

5, 9, 13]. Note that, using TTS is not limited to discrete time systems. It also have been used to give semantics for timed languages and formalisms that assume continuous or dense time domains.

The timed transition system semantics, unfortunately, suffers from the usual state space explosion problem (in addition to being infinite in many cases). The transition system contains arbitrary interleavings of independent actions of the various components of a distributed system, resulting in a large state space. In the presence of a global clock and timing information this may become even more acute.

A very different semantics, called Floating Time Transition System (FTTS), was proposed in [16] for a timed actor-based language called Timed Rebeca [23]. Timed Rebeca has been used in a number of applications. Examples of such case studies include analysis of routing algorithms and scheduling policies in NoC (Network on Chip) designs [25, 24]; schedulability analysis of distributed real-time sensor network applications [20], more specifically a real-time continuous sensing application for structural health monitoring in [18]; evaluation of different dispatching policies in clouds with priorities and deadlines in Mapreduce clusters, based on the work in [11].

Floating Time Transition Systems (FTTS) define a semantics where actors in a distributed system proceed at their own rates with local clocks widely apart, instead of moving in a lock step fashion with the global time as in TTS. Recall that in the Actor model [3] of computation, actors encapsulate the concept of concurrent behavior. Each actor provides services that 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. In FTTS semantics, each transition is the complete execution of a message server of an actor (which contains both timed and untimed statements), without any interleaving with the steps of other actors. Since actors execute a message to completion in this semantics, actors may have different local times in states of FTTS, as their local times are increased by timed statements of message servers. Relaxing the synchronization of progress of time among actors in FTTS can significantly reduce the size of the state space as it avoids many of the interleavings present in the TTS semantics.

The main contribution of this paper is the establishment of the bisimularity of the TTS and FTTS semantics for Timed Rebeca. Moreover, since the starting time of the execution of actions is also preserved, we can prove the preservation of any timed property of actions that is bisimulation invariant. Examples of such properties include μ-calculus with weak modalities. Such a logic preservation result is stronger than previous results about this and other reduction techniques, which only establish the preservation of "reachability"-type properties. In [16], we showed that FTTS preserves assertion-based properties like schedulability and deadlock avoidance. Similarly, many other works on reduction techniques for asynchronous systems papers like [8, 12, 19] consider assertion-based properties.

For timed systems, the norm is to show that there is a timed weak bisimulation relation between two timed transition systems to prove that they preserve the same set of timed branching-time properties (e.g. TCTL). Proving the existence of such a relation is impossible when one of the transition systems does not have progress-of-time transitions which is the case of relation between TTS and FTTS. In this paper, we proved that the actions and the execution time of the actions are preserved in FTTS using an innovative approach for defining relation between the states of a TTS and its corresponding FTTS.

Our bisimulation proof relies on observing that the FTTS semantics exploits key features of the actor model of computation. In such a model there is no shared memory, and sends and receives are non-blocking. Moreover, actors are single-threaded, with message servers being executed non-preemptively. This means that message servers can be executed in an isolated fashion, as is carried out in FTTS, without compromising the semantics of the model. Since our correctness proof of FTTS relies only on certain features of the actor model (rather than something specific to timed Rebeca), it suggests that FTTSs can be used in the analysis of other actor models and languages, and more generally, in other asynchronous event-based models.

We present experimental results that demonstrate the savings obtained from using FTTS. We have developed a toolset for generating the state space of a given Timed Rebeca model based on both the TTS and FTTS semantics that is accessible through the Rebeca homepage [1]. We show that using the FTTS semantics results in a smaller state space, fewer transitions, and less model checking time when compared with the TTS semantics (Section 4). In some case studies, using FTTS results in a state space which is 10 times smaller than its observational equivalent state space in TTS semantics.

2 Background

2.1 Timed Rebeca

Timed Rebeca is an extension of Rebeca [26] with time-related features for modeling and verification of time-critical systems. We describe Timed Rebeca language constructs using a simple ticket service example (see Figure 1). The abstract syntax of the language is given in Appendix A.

Each 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). In this example (Figure 1), we have three reactive classes `TicketService`, `Agent`, and `Customer`. Each reactive class declares a set of *state variables* which define the local state of the rebecs of that class (like `issueDelay` of `TicketService` which defines the time needed to issue a ticket). Following the actor model, the communication in the model takes place by rebecs sending asynchronous messages to each other. Each rebec has a set of *known rebecs* to which it can send messages. For example, a rebec of type `TicketService` knows a rebec of type `Agent` (line 2), to which it can send messages (line 12). Reactive classes declare

the messages to which they can respond. The way a rebec responds to a message is specified in a *message server*. A rebec can change its state variables through assignment statements (line 13), make decisions through conditional statements (not appearing in our example), and communicate with other rebecs by sending messages (line 12). Iterative behavior is modeled by rebecs sending messages to themselves (line 38). Since the communication is asynchronous, each rebec has a *message queue* from which it takes the next incoming message. A rebec takes the first message from its bag, executes the corresponding messages server atomically, and then takes the next message (or waits for the next message to come) and so on.

```
1  reactiveclass TicketService {
2    knownrebecs {Agent a;}
3    statevars {
4      int issueDelay, nextId;
5    }
6    msgsrv initial(int myDelay) {
7      issueDelay = myDelay;
8      nextId = 0;
9    }
10   msgsrv requestTicket() {
11     delay(issueDelay);
12     a.ticketIssued(nextId);
13     nextId = nextId + 1;
14   }
15 }
16 reactiveclass Agent {
17   knownrebecs {
18     TicketService ts;
19     Customer c;
20   }
21   msgsrv requestTicket() {
22     ts.requestTicket()
23         deadline(5);
24   }
25   msgsrv ticketIssued(byte id) {
26     c.ticketIssued(id);
27   }
28 }
29 reactiveclass Customer {
30   knownrebecs {Agent a;}
31   msgsrv initial() {
32     self.try();
33   }
34   msgsrv try() {
35     a.requestTicket();
36   }
37   msgsrv ticketIssued(byte id) {
38     self.try() after(30);
39   }
40 }
41
42 main {
43   Agent a(ts, c):();
44   TicketService ts(a):(3);
45   Customer c(a):();
46 }
```

Fig. 1. The Timed Rebeca model of ticket service system.

Timed Rebeca allows nondeterministic assignment to model nondeterministic behavior of message servers. In this paper we consider the fragment of language *without* such nondeterministic assignment. Thus, message servers in this paper specify deterministic behavior. Note, however, that even the Timed Rebeca language considered in this paper exhibits nondeterminism that results

from the interleaving of the executions of different rebecs due to concurrency; more details follow in the section defining the semantics.

Finally, the `main` block is used to instantiate the rebecs in the system. In our example (lines 43-45), three rebecs are created receiving their known rebecs and the arguments to their `inital` message servers upon instantiation.

In a Timed Rebeca model, although there is a notion of global time, each rebec has its own local clock. The local clocks can be considered as synchronized distributed clocks. Though methods (message servers) are executed atomically, passing of time while executing a method can still be modeled. In addition, instead of a queue for messages, there is a bag of messages for each rebec, ordering its messages based on their arrival time.

Timed Rebeca adds three primitives to Rebeca to address timing issues: *delay*, *deadline* and *after*. A *delay* statement models the passing of time for a rebec during execution of a message server (line 11). 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. For example, the periodic task of requesting a new ticket is modeled in line 38 by the customer sending a `try` message to itself and letting the receiver (itself) take it from its queue only after 30 units of time. The term `deadline` n shows that if the message is not taken in n units of time, it will be purged from the receiver's queue automatically. For example, line 23 indicates that a `requestTicket` message to the ticket service must be started to execute before five units from sending the message. Note that, the deadline is counted from the time of the sending of the message.

2.2 Semantics of Timed Rebeca

Prior to the detailed definition of semantics of Timed Rebeca, we formalize the definition of a rebec and a model in Timed Rebeca. A rebec r_i with the unique identifier i is defined as the tuple $(\mathcal{V}_i, \mathcal{M}_i, \mathcal{K}_i)$ where $\mathcal{V}_i$ is the set of its state variables, $\mathcal{M}_i$ is the set of its message servers, and $\mathcal{K}_i$ is the set of its known rebecs. The set of all the values of the state variables of r_i is denoted by $Vals_i$. For a Timed Rebeca model $\mathcal{M}$, there is a universal set $\mathcal{I}$ which contains identifiers of all the rebecs of $\mathcal{M}$.

A (timed) message is defined as $tmsg = ((sid, rid, mid), ar, dl)$, where rebec r_{sid} sends the message $m_{mid} \in \mathcal{M}_{rid}$ to rebec r_{rid}. This message is delivered to the rebec r_{rid} at $ar \in \mathbb{N}_0$ as its arrival time and the message should be served before $dl \in \mathbb{N}_0$ as its deadline. For the sake of simplicity, we ignore the parameters of the messages here. Each rebec r_i has a message bag $\mathcal{B}_i$ which can be defined as a multiset of timed messages. $\mathcal{B}_i$ stores the timed messages which are sent to r_i. The set of possible states of $\mathcal{B}_i$ is denoted by $Bags_i$.

In the following sections, two different semantics for Timed Rebeca models are defined, called *timed transition system* and *floating time transition system*. FTTS is defined in [16] as the natural semantics of Timed Rebeca but the relation between TTS and FTTS for Timed Rebeca has not been investigated before. Timed

transition system is generally the standard semantic framework for timed systems, and we define the formal semantics of Timed Rebeca in TTS in Section 2.3. Floating time transition system exploits key features of actor models to generate smaller transition systems compared to TTS. The absence of shared variables, and blocking send or receive, and the presence of single threaded actors along with non-preemptive execution of each message server, ensures that the execution of a message server does not interfere with the execution of another message server of a different rebec. The floating time transition system semantics exploits this by executing message servers in isolation, and thereby having fewer states in the transition system.

2.3 Semantics of Timed Rebeca in Timed Transition System

Timed Transition System of the Timed Rebeca model $\mathcal{M}$ is a tuple of $TTS = (S, s_0, Act, \rightarrow)$ 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.

States. A state $s \in S$ consists of the local states of the rebecs, together with the current time of the state. The local state of rebec r_i in state s is defined as the tuple $(V_{s,i}, B_{s,i}, pc_{s,i}, res_{s,i})$, where

- $V_{s,i} \in Vals_i$ is the values of the state variables of r_i
- $B_{s,i} \in Bags_i$ is the message bag of r_i
- $pc_{s,i} \in \{null\} \cup (\mathcal{M}_i \times \mathbb{N})$ is the program counter, tracking the execution of the current message server (*null* if r_i is idle in s)
- $res_{s,i} \in \mathbb{N}_0$ is the resuming time, if r_i is executing a `delay` in s

So, state $s \in S$ can be defined as $(\prod_{i \in \mathcal{I}} (V_{s,i}, B_{s,i}, pc_{s,i}, res_{s,i}), now_s)$ where $now_s \in \mathbb{N}$ is the 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 rebecs are set to their initial values (according to their types), the `initial` message is put in the bag of all rebecs having such a message server, the program counters of all rebecs are set to *null*, and the time of the state is set to zero.

Actions. There are three possible types of actions: sending a message *tmsg* (as defined in Section 2.2 there is $tmsg = ((sid, rid, mid), ar, dl)$), executing a statement by an actor (which we consider as an internal transition τ), 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 rebec r_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 r_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 r_i is called *enabled rebec*.
2. **(Internal action)** In state s, there exist r_i such that $pc_{s,i} \neq null$ and $res_{s,i} = now_s$. The statement of message server of r_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 rebec r_i or send a message to another rebec. 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) 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 $\nexists r_i \cdot ((pc_{s,i} = null \wedge E_{s,i} \neq \emptyset) \vee (pc_{s,i} \neq null \ \wedge \ res_{s,i} = now_s))$, 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 rebec r_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.

Later, for each state of a TTS we need to find messages which are sent by a given rebec. Therefore, we define the following function which returns a bag of messages which are sent by a rebec.

Definition 1 (Sent Messages in TTS). *For a given state $s \in S$ and rebec r_i, function $sent(s, r_i)$ returns bag of messages which are sent by r_i in state s. In other words, $tmsg \in sent(s, r_i)$ if and only if for message $tmsg = ((sid, rid, mid), ar, dl)$ there is $\exists\, r_j \cdot tmsg \in B_{s,j} \wedge sid = r_i$.* □

2.4 Semantics of Timed Rebeca in Floating Timed Transition System

The notion of floating time transition system (FTTS) as a semantics for Timed Rebeca has been introduced in [16]. States in floating time transition system

contain the local times of each rebec, in addition to values of their state variables and the bag of their received messages. However, the local times of rebecs in a state can be different, and there is no unique value for time in each state. Such a semantics is reasonable when one is only interested in the order of visible events. FTTS may not be appropriate for analyses that require reasoning about all synchronized global states of a Timed Rebeca model. The key features of Rebeca actors that make FTTS a reasonable semantics are having no shared variables, no blocking send or receive, single-threaded actors, and atomic (non-preemptive) execution of each message server which give us an isolated message server execution. This means that the execution of a message server of a rebec will not interfere with execution of a message server of another rebec. Therefore, we can execute all the statements of a given message server (even delay statements) during a single transition. This makes the transition system significantly smaller, because there will be only one kind of action, which is taking a message and executing the corresponding message server entirely.

The operational semantics of a Timed Rebeca model $\mathcal{M}$ is defined as a floating time transition system $FTTS = (S', s'_0, Act' , \hookrightarrow)$ and is as described below. In this paper, we use the primed version for letters and notations related to FTTS except for transitions which are shown by $\hookrightarrow$ (for TTS we use the unprimed letters).

States. Similar to TTS, a state $s \in S'$ consists of the local states of the rebecs. However, the current time is kept separately for each rebec, denoted by $now_{s',i}$. We will see shortly, the message servers are executed entirely in one transition; therefore, there is no need to keep track of the program counter and the resuming time. 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. s'_0 is the initial state of the Timed Rebeca model $\mathcal{M}$ where the state variables of the rebecs are set to their initial values (according to their types), the `initial` message is put in the bag of rebecs, and the current times of all the rebecs are set to zero.

Actions. As mentioned before, there is only one kind of action, which is taking a message and executing the corresponding message server entirely. Therefore, $Act' = \bigcup_{i \in \mathcal{I}} ((\mathcal{I} \times i \times \mathcal{M}_i) \times \mathbb{N} \times \mathbb{N})$ is defined as the set of all the possible timed messages.

Transition Relations. We first define the notion of *release time* of a message. A rebec r_i in a state $s' \in S'$ has a number of timed messages in its bag. The release time of $tmsg = ((sid, rid, mid), ar, dl) \in B_{s',i}$ 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 r_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 rebec r_i in state s' is $E_{s',i} = \{tmsg \in B_{s',i} | \forall tmsg' \in B_{s',i} \cdot rt_{tmsg} \leq rt_{tmsg'}\}$ which are the messages with the smallest 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 r_i and all the statements in the message server corresponding to *tmsg* are executed. Here, r_i is called *enabled rebec*. The effect of executing non-delay statements is changing the state variables and sending messages to other rebecs. The effect of executing a delay statement with parameter $d \in \mathbb{N}$ is the increase in the local time of the running rebec by d units of time.

We define bag of sent messages in FTTS the same as what we defined in TTS.

Definition 2 (Sent Messages in FTTS). *For a given state $s' \in S'$ and rebec r_i, function $sent(s', r_i)$ returns bag of messages which are sent by r_i in state s'. In the other words, $tmsg \in sent(s', r_i)$ if and only if for message $tmsg = ((sid, rid, mid), ar, dl)$ there is $\exists\, r_j \cdot tmsg \in B_{s',j} \wedge sid = r_i$.* □

As There is no explicit reset operator for the time in Timed Rebeca, progress of time results in an infinite number of states in the transition systems of both FTTS and TTS. However, Timed Rebeca models are models of reactive systems which generally show periodic or recurrent behaviors. Hence, if we ignore the absolute time of the states, usually finite number of untimed traces are generated for Timed Rebeca models. Based on this fact, in [16] we presented a new notion for equivalence relation between two states to make the transition systems finite, called *shift equivalence relation*. In 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 (value of *now*, arrival times of messages, and 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 [16], shift equivalence relation makes transition systems of the majority of Timed Rebeca models finite.

3 An Action-Based Weak Bisimulation Between TTS and FTTS

As described in Section 2.4, in FTTS representation of a Timed Rebeca model, all the statements of a message server are executed at once during a single transition. In contrast, the TTS semantics executes one statement at a time, and interleaves the execution of different message servers. We demonstrate despite these differences, these semantics are equivalent in some sense. To this end, we define an action-based weak bisimulation (observational equivalence) relation between $TTS = (S, s_0, Act, \rightarrow)$ and $FTTS = (S', s'_0, Act', \hookrightarrow)$ for a given Timed Rebeca model $\mathcal{M}$. Note that in the following text we denote the states of FTTS as the primed version of the states in TTS.

This definition is valid for Zeno-free Timed Rebeca models. As the model of time in Timed Rebeca is discrete, the execution of infinite number of message servers in zero time is the only scenario resulting Zeno behavior. In other words, execution of infinite number of message servers which make progress in time

goes to the infinity, as the smallest time elapse in Timed Rebeca is one unit. Therefore, the Zeno behavior happens if and only if there is a cycle of message servers invocations among different actors without progress of time. This can be detected by performing a depth-first-search (DFS) in both TTS and FTTS [15].

Prior to the formal definition of the relation between the states of FTTS and TTS the following definitions and proposition are required to make the relation easy to understand.

We begin by defining the observable and τ actions in both transitions systems. All actions in FTTS are observable. In the TTS, only *taking-event transitions* are observable. Therefore, *time transitions* and *internal transitions* in TTS are assumed to be τ transitions. In other words, only taking-event actions are observable in TTS and FTTS. This definition conforms the definition of *events* and *observer primitives* in the actor model which is introduced by Agha et. al. in [2] as a reference actor framework. Next, we define the notion of a *completing trace* for a rebec r_i in TTS state s as an execution which results in completing the execution of the message server of r_i that has already commenced in state s. Note that during a completing trace for r_i the other rebecs, may complete their servers (or not), and may start the execution of new message servers. We begin by first defining an execution.

Definition 3 (Execution Trace). *Execution trace from state s in TTS is a sequence of transitions from state s to one of its reachable states u, shown by $s \xrightarrow{act_1} s_1 \xrightarrow{act_2} \cdots \xrightarrow{act_n} u$.* □

Definition 4 (Completing Trace for a Rebec). *A given execution trace from state s to state u in TTS is a completing trace for rebec r_i if and only if r_i does not execute any taking-event transition from s to u and in u we have $pc_{u,i} = null$. Here, we also define $CT_{s,i}$ as one of the completing traces from s for rebec r_i (no matter which one in the case there are more than one completing traces from s for rebec r_i).* □

Note that, as there is no preemption in the message server execution and there is no infinite message server body in Timed Rebeca, there is a completing trace for all the rebecs from all the states.

We define three functions on the completing traces. The first one returns the value of the state variables of the specific rebec at the last state of the trace (the rebec that the completing trace is defined for). The second one returns the time of the last state of the trace. The third one returns the bag of messages that are sent by the specific rebec during this trace.

Definition 5 (Three Functions on completing traces). *Function $state_i(CT_{s,i})$ returns the values of state variables of r_i in the target state of trace $CT_{s,i}$. Function $now_i(CT_{s,i})$ returns the time of the target state of trace $CT_{s,i}$. Function $sent_i(CT_{s,i})$ returns a bag of messages where $tmsg = ((sid, rid, mid), ar, dl) \in sent_i(CT_{s,i})$ if and only if $tmsg$ is sent during the execution of completing trace $CT_{s,i}$ and $sid = r_i$.* □

Based on the isolated execution of rebecs (no shared variables and no preemption of a message server) we can easily conclude that in case of more than one completing trace for a rebec, any of the completing traces ends in the same values for state variables, the same state time, and the same bag of sent messages.

Proposition 1 (Completing Traces end in the same final condition). *Assume that there are two different completing traces $CT^1_{s,i}$ and $CT^2_{s,i}$ from a given state $s \in S$ and rebec r_i. We have $sent_i(CT^1_{s,i}) = sent_i(CT^2_{s,i})$, $now_i(CT^1_{s,i}) = now_i(CT^2_{s,i})$, and $state_i(CT^1_{s,i}) = state_i(CT^2_{s,i})$.*

Proof. It is presented in Appendix B.

Next, we define a projection function for states of TTS and FTTS. Projection functions extract values of state variables and the collection of messages which are sent by one rebec from a given TTS or FTTS state. Using these projection functions, we get uniform views from states of TTS and FTTS which is necessary for the definition of the action-based weak bisimulation relation. To this aim, as the execution of a message in TTS is completed in multi steps, the projection function in TTS is defined based on completing traces to be able to have access to the valuation of state variables and bags of sent messages after completing the execution of currently executing messages.

Definition 6 (Projection Function in TTS). *For a given TTS state $s \in S$ and rebec r_i, projection function $Proj(s, i)$ returns a collection of $state_i(CT_{s,i})$, $now_i(CT_{s,i})$, and $sent(s, i) \cup sent_i(CT_{s,i})$.Here, $CT_{s,i}$ is one of completing traces of rebec r_i in state s.* □

Definition 7 (Projection Function in FTTS). *For a given FTTS state $s' \in S'$ and rebec r_i, projection function $Proj(s', i)$ returns a collection of the values of state variables of r_i in s', $now(s', i)$, and $sent(s', i)$.* □

Using the above definitions, we define the action-based weak bisimulation relation among states of TTS and FTTS. Two states in TTS and FTTS are in the relation if and only if the projection of states to each rebec is the same. This way, we will prove that two states have the same future behavior in Theorem 1. Figure 2 shows how states in TTS are mapped to their corresponding states in FTTS. As the observational behavior of s_1 and s'_1 are the same (only the observable action a is enabled), s_1 is mapped to s'_1 and as the observational behavior of s_2, s_3, and s_4 are the same as the observational behavior of s'_2 (the observable actions b and c are enabled), they are mapped to s'_2.

Definition 8 (Relation among states of TTS and FTTS). *Two states $s \in S$ and $s' \in S'$ are in relation $\mathcal{R} \subseteq S \times S'$ if and only if for every rebec r_i there is $Proj(s, i) = Proj(s', i)$.* □

Directly from the definition of relation $\mathcal{R}$ it is concluded that the bag of enabled taking-event messages in s and s' are the same.

Proposition 2 (Relation $\mathcal{R}$ preserves enabled messages). *States which are in relation $\mathcal{R}$ has the same bag of enabled messages and the enabled messages have the same enabling time.*

Proof. It is presented in Appendix C.

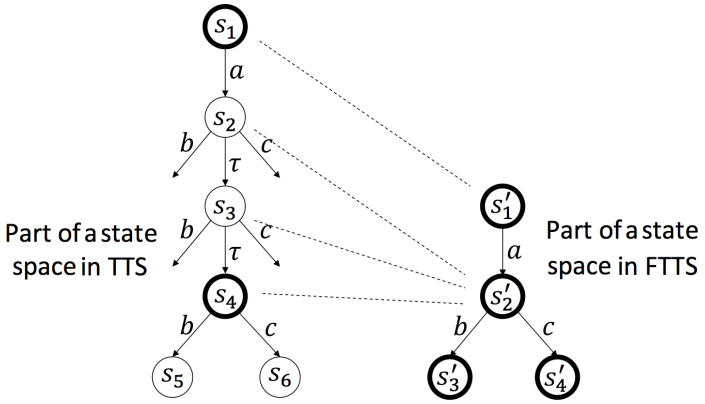

Fig. 2. How states in TTS are mapped to states of FTTS with the same future behaviors.

Having the same enabled messages (messages with the same signature and the same execution time) in two given states $s \in S$ and $s' \in S'$ where $s \mathcal{R} s'$, we are able to prove that s and s' have the same future behavior. To this aim, we have to prove that $\mathcal{R}$ is an action-based weak bisimulation relation.

Definition 9 (Action-based weak bisimulation relation). *A relation $\mathcal{P}$ over two transition systems $TS_1 = (S_1, s_{1_0}, Act_1, \rightarrow_1)$ and $TS_2 = (S_2, s_{2_0}, Act_2, \rightarrow_2)$ where TS_2 is τ-free transition system, is an action-based weak bisimulation relation if the following conditions hold for states of TS_1 and TS_2.*

1. *$\forall s_1, t_1 \in S_1$ and $s_2 \in S_2$ where $s_1 \mathcal{P} s_2$, in case of $s_1 \xrightarrow{\alpha}_1 t_1$ where $\alpha \in Act_1$ then $\exists\, t_2 \in S_2$ such that $s_2 \xrightarrow{\alpha}_2 t_2$ and $t_1 \mathcal{P} t_2$ and in case of $s_1 \xrightarrow{\tau}_1 t_1$ there is $t_1 \mathcal{P} s_2$.*
2. *$\forall s_2, t_2 \in S_2$ and $s_1 \in S_1$ where $s_1 \mathcal{P} s_2$, for a message $\alpha \in Act_2$ such that $s_2 \xrightarrow{\alpha}_2 t_2$ then $\exists\, s', s'', \ldots, s^{(k)}, t_1 \in S_1$ (for $k \geq 0$) such that $s_1 \xrightarrow{\tau}_1 s' \xrightarrow{\tau}_1 s'' \xrightarrow{\tau}_1 \cdots \xrightarrow{\alpha}_1 t_1$ and $t_1 \mathcal{P} t_2$.*

□

Theorem 1. *The relation $\mathcal{R}$ is an action-based weak bisimulation relation between states of TTS and FTTS.*

Proof. It is presented in Appendix D.

We discussed in Section 2.4 that in actor systems we are interested in relation among actions of systems and the time where they are triggered (messages are taken from bags). So, we have to find the most expressive action-based logic which is preserved in action-based weak bisimulation relation. As mentioned in [27], weak bisimulation relation preserves properties in form of modal μ-calculus with weak modalities. Weak-bisimulation relation does not preserve complete modal μ-calculus. Weak modal μ-calculus has the same syntax as modal μ-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 μ-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) φ. The semantics of box is defined dually.

Corollary 1. *Transition systems of Timed Rebeca models in TTS and FTTS are equivalent respect to all formulas that can be expressed in modal μ-calculus with weak modalities where the actions are taking messages from bags.* □

4 Experimental Results

We provide four case studies of different sizes to illustrate the reduction in state space size, number of transitions, and time consumption of model checking using FTTS in comparison with TTS. The host computer of model checking toolset was a desktop computer with 1 CPU (2 cores) and 6GB of RAM storage, running Microsoft Windows 7 as the operating system. The selected case studies are the models of a *Wireless Sensor and Actuator Networks (WSAN)*, the simplified version of *Scheduler of Hadoop*, a *Ticket Service* system, and simplified version of *802.11 Wireless Protocol*. We developed a toolset for generating transition systems of Timed Rebeca models based on the semantics of both FTTS and TTS, as a part of the Afra project [4]. The Timed Rebeca code of the case studies and the model checking toolset are accessible from Rebeca homepage [1].

The details of the *Ticket Service* case study is explained in Section 2.1. Catching the deadline of issuing the ticket is the main property of this model. We created different sizes of ticket service model by varying the number of customers, which results in four to ten rebecs in the model. In the case of the simplified version of *802.11 Wireless Protocol*, we modeled three wireless nodes which are communicating via a medium. The medium sets random back-off time when more than one node starts to send data, to resolve data collision in the medium. Deadlock avoidance is the main property of this model. In the third case study, a WSAN is modeled as a collection of actors for sensing, radio communication, data processing, and actuation. Schedulability of the model is verified as the main property of this model. Finally, we modeled a simplified version of the behavior of MapReduce of Hadoop system, called YARN. We modeled one client which submits jobs to YARN resource manager. The resource manager distributes the submitted job among application masters and application masters split the job into some tasks and distribute tasks among some nodes. This model has 32 rebecs and is model checked to meet deadline of jobs.

Using FTTS results in significant reduction in the size of the state space for the majority of timed actor models. As shown in Table 1, in *Yarn* model we have about 90% of reduction. the reason is many delay statements in the message

[4] The latest version of the toolset (version 2.5.0) is accessible from `http://www.rebeca-lang.org/wiki/pmwiki.php/Tools/RMC`

Problem	Size	Using FTTS			Using TTS			Reduction	
		states	trans	time	states	trans	time	states	trans
Yarn	-	1.30K	5.71K	< 1 sec	11.03K	61.08K	6 secs	88%	91%
WSAN	**33,6,4,2**	977	1.5K	< 1 sec	1.92K	2.52K	< 1 sec	49%	41%
	25,5,4,10	1.85K	2.54K	< 1 sec	3.72K	4.55K	< 1 sec	50%	44%
	30,6,4,2	4.75K	5.78K	< 1 sec	9.35K	10.46K	2 secs	50%	45%
	25,6,4,2	17.02K	20K	5 secs	34.5K	37.85K	24 secs	51%	47%
	20,6,4,2	28.19K	32.19K	16 secs	57.62K	62.21K	64 secs	51%	48%
Ticket Service	**1**	5	6	< 1 sec	8	9	< 1 sec	38%	33%
	2	51	77	< 1 sec	77	107	< 1 sec	34%	28%
	3	252	418	< 1 sec	360	550	< 1 sec	30%	24%
	4	1.29K	2.21K	< 1 sec	1.82K	2.89K	< 1 sec	30%	24%
	5	7.53K	12.8K	< 1 sec	10.7K	16.9K	< 1 sec	30%	24%
	6	51.6K	84.7K	2 secs	73.5K	114K	2 secs	30%	26%
	7	408K	650K	18 secs	582K	884K	24 secs	30%	26%
802.11 Prot.	**2**	1.12K	2.09K	2 secs	1.92K	2.62K	2 secs	10%	4%
	3	59K	196K	122 secs	61K	198K	153 secs	3%	1%

Table 1. Number of states and transitions, time consumption, and reduction ratio in model checking based on floating time transition system and timed transition system.

servers of *Yarn* model which results in splitting the execution of message servers in TTS. Interleaving of the execution of these parts results in larger state spaces in TTS. The same argument is valid to support results of *Ticket Service* and *WSAN*. In case of *WSAN*, in each row, the configuration (the numbers which are separated by dash) is a combination of the sampling rate, the number of nodes, the packet size, and the sensor task delay of the model, respectively. As the complexity of these examples are less than *Yarn* model, the reduction is about 50%. There are some exceptional models in which the state space size and the number of transitions in TTS and FTTS are close to each other. The model of *802.11 prot.* is one of them. As there is no delay statement in the body of the message servers of *802.11 prot.*, the execution of the message servers is without progress of time. Therefore, atomic execution of message servers in FTTS and the rather fine-grain execution of message servers in TTS results in state spaces with comparable sizes. The effectiveness of FTTS is reduced in this kind of models.

Table 1 also shows that using FTTS reduces the model checking time consumption (even in case of *802.11 prot.*). It is because of the simplicity of the generated state space in FTTS, using atomic execution of message servers.

5 Related Work

Here, we give an overview of the approaches which are used for dealing with time in some widely used real-time system modeling and verification languages.

Real-Time Maude. Real-Time Maude [21, 22] is a high level declarative programming language supporting specification of real-time and hybrid systems in timed rewriting logic. Real-Time Maude supports both discrete and continuous time models. A set of tools are developed for time-bounded analysis of real-time Maude. *Timed rewrite* and *Timed search* build traces of the model from its initial state and checks whether a specific state is reachable or not. *Timed model checking* verifies models against time-bounded TLTL formulas. Recently, Real-Time Maude is equipped with a model checker for TCTL properties [17]. Comparing to FTTS, all the mentioned tools are working on lock step fashion which results in generating timed transition systems of the models. To the best of our knowledge, no reduction technique is implemented for real-time Maude models to relax lock step fashion. In addition, timed transition systems of real-time Maude models are generated to the defined time-bound. In contrary, using shift equivalence relation in FTTS, there is no need to define time-bound to achieve finite transition system.

Timed Automata. Timed automata [4] 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. The values of all clocks are increased in the same rate or can be reset to zero while moving from one state to other states. Constraints over clocks can be added as enabling conditions on both states and transitions. Timed automata support parallel composition as a convenient approach for modeling complex systems. As described in [7], parallel composition of timed automata is based on the handshaking actions. Timed automata support both continuous and discrete timed models [10, 14]. UPPAAL [9] generates region transition system of timed automata (symbolic representation of timed transition system of the timed automata) and apply verification techniques on it. Modeling of real-time distributed systems with asynchronous message passing between components using synchronous communication of automata increases the number of states dramatically (because of many synchronizations among automta for model asynchronous behavior, as shown in [16] in detail). In contrast, using FTTS requires fewer synchronizations, because messages are executed atomically.

Erlang. Erlang is a dynamically-typed general-purpose programming language which was developed in 1986 [6]. The concurrency model of Erlang is based on the actor model. Fredlund et. al. in [13] proposed a timed extension of McErlang as a model checker of timed Erlang programs. In comparison with FTTS, McErlang provides fine-grain model checker for Erlang systems which results in generating timed transition system; however, states in FTTS are coarse-grain and more abstract than that of McErlang. Experimental results in [16] show very well the efficiency of FTTS in comparison with the results of the approach of McErlang.

Partial Order Reduction. The reduction from TTS to FTTS has aspects that are similar to partial order reduction (POR). In fact the relationship between POR and FTTS is subtle. FTTS is unaware of any independence relation, persistence/ample sets for timed actor systems that will result in POR techniques

producing FTTS as the reduced transition system. Moreover, not only is the formal relationship between FTTS and POR nontrivial, POR techniques for timed systems were empirically compared against the FTTS semantics and found that the FTTS results in smaller transition systems in [16].

6 Conclusion

In this paper we proved that there is a weak bisimulation relation between timed transitions system (TTS) – as a standard semantics of discrete time systems – and floating time transitions system (FTTS) – as a natural semantics for time actor systems. FTTS was previously introduced in [16] along with an algorithm for schedulability and deadlock freedom analysis. Proving the weak bisimilarity of TTS and FTTS, enables one to use FTTS for verification of branching-time properties in addition to previously proposed analyses.

Experimental evidence supports our theoretical observation that FTTS of Timed Rebeca models are smaller than TTS in general. In case of models with many concurrently executing actors, FTTS is up to 90% smaller than TTS. Therefore, we can efficiently model check more complicated models. In addition, our technique and the proofs are based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. So, they are generalized enough to be applied to computation models which have message-driven communication and autonomous objects as units of concurrency such as agent-based systems.

Acknowledgements

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

References

1. Rebeca Home Page. http://www.rebeca-lang.org
2. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. J. Funct. Program. 7(1), 1–72 (1997), http://journals.cambridge.org/action/displayAbstract?aid=44065
3. Agha, G.A.: ACTORS - A Model of Concurrent Computation in Distributed Systems. MIT Press series in artificial intelligence, MIT Press (1990)
4. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
5. Archer, M., Lim, H., Lynch, N.A., Mitra, S., Umeno, S.: Specifying and proving properties of timed I/O automata in the TIOA toolkit. In: 4th ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2006), 27-29 July 2006, Embassy Suites, Napa, California, USA. pp. 129–138. IEEE Computer Society (2006), http://dx.doi.org/10.1109/MEMCOD.2006.1695916

6. Armstrong, J.: A History of Erlang. In: Ryder, B.G., Hailpern, B. (eds.) HOPL. pp. 1–26. ACM (2007)
7. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
8. Basu, S., Bultan, T., Ouederni, M.: Synchronizability for verification of asynchronously communicating systems. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7148, pp. 56–71. Springer (2012), http://dx.doi.org/10.1007/978-3-642-27940-9_5
9. 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. Lecture Notes in Computer Science, vol. 1066, pp. 232–243. Springer (1995)
10. Bozga, M., Maler, O., Tripakis, S.: Efficient verification of timed automata using dense and discrete time semantics. In: Pierre, L., Kropf, T. (eds.) CHARME. Lecture Notes in Computer Science, vol. 1703, pp. 125–141. Springer (1999)
11. Cho, B., Rahman, M., Chajed, T., Gupta, I., Abad, C., Roberts, N., Lin, P.: Natjam: design and evaluation of eviction policies for supporting priorities and deadlines in mapreduce clusters. In: Lohman, G.M. (ed.) SoCC. p. 6. ACM (2013)
12. Desai, A., Garg, P., Madhusudan, P.: Natural proofs for asynchronous programs using almost-synchronous reductions. In: Black, A.P., Millstein, T.D. (eds.) Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014. pp. 709–725. ACM (2014), http://doi.acm.org/10.1145/2660193.2660211
13. Earle, C.B., Fredlund, L.Å.: Verification of Timed Erlang Programs Using McErlang. In: Giese, H., Rosu, G. (eds.) FMOODS/FORTE. Lecture Notes in Computer Science, vol. 7273, pp. 251–267. Springer (2012)
14. Ibarra, O.H., Su, J.: Generalizing the discrete timed automaton. In: Yu, S., Paun, A. (eds.) CIAA. Lecture Notes in Computer Science, vol. 2088, pp. 157–169. Springer (2000)
15. 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. pp. 55–66. ACM (2014), http://doi.acm.org/10.1145/2687357.2687366
16. 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, 184–204 (2015), http://dx.doi.org/10.1016/j.scico.2014.07.005
17. Lepri, D., Ábrahám, E., Ölveczky, P.C.: Timed CTL Model Checking in Real-Time Maude. In: Durán, F. (ed.) WRLA. Lecture Notes in Computer Science, vol. 7571, pp. 182–200. Springer (2012)
18. Linderman, L.E., Mechitov, K., Spencer, B.F.: TinyOS-based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control. Structural Control and Health Monitoring 20(6), 10071020 (June 2013)
19. Manohar, R., Martin, A.J.: Slack elasticity in concurrent computing. In: Jeuring, J. (ed.) Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1422, pp. 272–285. Springer (1998), http://link.springer.de/link/service/series/0558/bibs/1422/14220272.htm

20. 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 (2013)
21. Ölveczky, P.C., Meseguer, J.: Specification and Analysis of Real-Time Systems Using Real-Time Maude. In: Wermelinger, M., Margaria, T. (eds.) FASE. Lecture Notes in Computer Science, vol. 2984, pp. 354–358. Springer (2004)
22. Ölveczky, P.C., Meseguer, J.: Real-Time Maude 2.1. Electr. Notes Theor. Comput. Sci. 117, 285–314 (2005)
23. 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, 41–68 (2014)
24. Sharifi, Z., Mohammadi, S., Sirjani, M.: Comparison of NoC Routing Algorithms Using Formal Methods. To be published in proceedings of PDPTA'13 (2013)
25. 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)
26. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and Verification of Reactive Systems using Rebeca. Fundam. Inform. 63(4), 385–410 (2004)
27. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings. Lecture Notes in Computer Science, vol. 1384, pp. 167–183. Springer (1998), http://dx.doi.org/10.1007/BFb0054171

A Abstract Syntax of Timed Rebeca

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

Fig. 3. Abstract syntax of Timed Rebeca (a slightly revised version of the syntax presented in [23]). Angle 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.

B Proof of Proposition 1

As mentioned in the semantics of Timed Rebeca, execution of a message server is not interfered with the execution of other rebecs because in Timed Rebeca there is no shared variable or any kind of preemption of execution of a message server while its executing. In addition, we assumed that there is no non-deterministic expression in messages servers of rebecs. Therefore, in all the completing traces from state s, execution of τ transitions which are related to r_i ends in the same values for state variables and bag of sent messages. On the other hand, as *delay* statements which are related to the execution of r_i are the same in two different competing traces, the time at the target states of $CT^1_{s,i}$ and $CT^2_{s,i}$ are the same.

□

C Proof of Proposition 2

Assume that for given states $s \in S$ and $s' \in S'$ there is $s\ \mathcal{R}\ s'$. Then, $\forall\ i \in \mathcal{I} \cdot Proj(s,i) = Proj(s',i)$ which results in $\forall\ i \in \mathcal{I} \cdot sent(s,i) \cup sent_i(CT_{s,i}) =$

$sent(s', i)$. As a result, there is $\bigcup_{i \in \mathcal{I}} (sent(s, i) \cup sent_i(CT_{s,i})) = \bigcup_{i \in \mathcal{I}} (sent(s', i))$ which implies that $\bigcup_{i \in \mathcal{I}} (B_{s,i}) \cup \bigcup_{i \in \mathcal{I}} sent_i(CT_{s,i}) = \bigcup_{i \in \mathcal{I}} (B_{s',i})$. As the messages in $\bigcup_{i \in \mathcal{I}} sent_i(CT_{s,i})$ will be send in the future, none of the enabled messages in s are in $\bigcup_{i \in \mathcal{I}} sent_i(CT_{s,i})$. Therefore, enabled messages in $\bigcup_{i \in \mathcal{I}} (B_{s',i})$ are in $\bigcup_{i \in \mathcal{I}} (B_{s,i})$.

On the other hand, based on the definition of enabled messages in TTS, enabled rebecs are not busy with the execution of messages in s. So, their completing trace are empty trace. Assume that r_i is one of the enabled rebecs of s. Having $CT_{s,i} = \emptyset$ results in $now_i(CT_{s,i}) = now(s)$. Therefore, as $Proj(s, i) = Proj(s', i)$ there is $now(s', i) = now_i(CT_{s,i}) = now(s)$. So, for enabled rebecs in s, their local times in s' is the same as the time of state s.

Finally, as in s and s' there are the same messages in the bag of enabled rebecs and their times are the same, based on the definition of enabled rebecs in Section 2.2, s and s' have the same bag of enabled rebecs. This property holds for both conditions one and two.

□

D Proof of Theorem 1

To prove that the first condition of action-based weak bismulation holds for $\mathcal{R}$, based on the type of $tmsg$ the following two cases are possible.

- $\mathbf{s} \overset{\mathbf{tmsg}}{\rightarrow} \mathbf{t}$: Based on the definition of relation $\mathcal{R}$, in this case projection function for all the rebecs in s and t return the same value except for the sender and receiver of $tmsg$. For the sender rebec (assume that it is r_i) the difference is in the bag of sent messages, results in $sent_{t,i} = sent_{s,i} - tmsg$. On the other hand, projection function in s' and t' have the same value for all the rebecs except the sender and receiver of $tmsg$. For the sender rebec (assume that it is r_i) the difference is in the bag of sent messages, results in $sent_{t',i} = sent_{s',i} - tmsg$.
  For the receiver rebec (assume that it is r_j), there is a completing trace $CT_{t,j}$ such that $Proj(t, j)$ returns valuation of state variables of r_j from the target state of $CT_{t,j}$ and messages which are sent by r_j in t in union with messages which are sent during $CT_{t,j}$. In FTTS state t', projection function returns valuation of state variables and the sent messages of r_j after the execution of all the statements of $tmsg$ (i.e. doing transition $tmsg$ in FTTS) which is the same as what projection function returns in t. Therefore, there is $t \mathcal{R} t'$ as the results of projection function in t and t' are the same for all the rebecs.
- $\mathbf{s} \overset{\tau}{\rightarrow} \mathbf{t}$: As transition from s to t is not observable, we have to show that there is relation $\mathcal{R}$ between t and s'. This way, doing a τ transition from s results in stuttering in s' as one of the properties of action-based weak bisimulation relations.
  Assume that τ transition belongs to rebec r_i. Doing τ transition by r_i makes projection function return the same result in s and t for all the rebecs except r_i. It is because of the fact that only r_i has progress which may result in changing the valuation of its state variables or sending a message to other rebec. For r_i in state s one of the completing traces is a trace which contains

τ transition from s to t as its first transition. Therefore, completing traces of r_i which are started from s and t are ended in the same target state, results in $Proj(s, i) = Proj(t, i)$. Therefore, result of projection function for all the rebecs in TTS and FTTS are the same and t is in relation $\mathcal{R}$ with s'.

To prove the second condition, as all the transitions in FTTS are taking-event transitions, *tmsg* must be taking-event transition. On the other hand, transition *tmsg* is enabled in s as we discussed in Proposition 2. Now we can prove that t and t' are in relation $\mathcal{R}$ with the argument the same as what we did in case $s \xrightarrow{tmsg} t$ of condition one. □