Analysing Timed Rebeca Using McErlang

Haukur Kristinsson
Reykjavik University
haukurk@ru.is

Ali Jafari
Reykjavik University
ali11@ru.is

Ehsan Khamespanah
University of Tehran, Reykjavik University
e.khamespanah@ut.ac.ir, ehsan13@ru.is

Brynjar Magnusson
Reykjavik University
brynjar07@ru.is

Marjan Sirjani
Reykjavik University
marjan@ru.is

Abstract

Although timed actor-based models attracted more attention in the recent years, there is not much done on analyzing and model checking of such systems. The actor-based language, Timed Rebeca, was introduced to model distributed and asynchronous systems with timing constraints, and a supporting tool was developed for automated translation of Timed Rebeca models to Erlang. The translated code can be executed using McErlang. In this paper, we propose extensions for Timed Rebeca language to improve the usability of the language. Designers can now develop models expressing more complex behaviors by calling Erlang custom functions and using lists. Moreover, to be able to use the new version of McErlang which supports timing constructs of Timed Rebeca we changed the mapping and the tool accordingly. This gives us the possibility of model checking and simulation of Timed Rebeca models for the first time. We can use the safety monitors in McErlang to verify safety properties in the model. Also, statistical methods are applied to the simulation results to reason about the system behavior. We examine the typical case study of elevators to show the applicability of our technique and tool.

Keywords Model Checking, Simulation, Actors, Erlang, Rebeca, Real-time Systems

SPLASH '13, October 27-28, 2013 - Indianapolis, Indiana, USA..

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

1. Introduction

In analyzing real-time systems, performance evaluation is a complementary issue to functional verification. Therefore, analysis techniques should consider both correctness and performance to guarantee quality of systems. Different formal timed models have been proposed for modeling and verification of real-time systems. On the other hand, different approaches have been suggested for performance evaluation of real-time systems. Numerical analysis and simulation techniques that are based on statistical methods are two widely used approaches for performance evaluation. In this work, we provide a unified analysis technique and toolset for both verification of correctness and performance evaluation of real-time distributed systems with asynchronous message passing.

A well-established paradigm for modeling the functional behavior of distributed systems with asynchronous message passing is the actor model. This model was originally introduced by Hewitt [9] and then elaborated by Agha [2, 3] and Talcott [15]. Although actors are attracting more and more attention both in academia and industry, little work has been done on timed actors and even less on analyzing timed actor-based models. To address the specification and verification of real-time systems, a few timed actor-based modeling languages such as RT-synchronizer [19] and Timed Rebeca [1] were proposed.

Background. The *Reactive Objects Language*, *Rebeca* [25], is an actor based modeling language which can be used in a model-driven methodology, in which the designer builds an abstract model where each component is a reactive object communicating through non-blocking asynchronous messages. Rebeca is an operational interpretation of the actor model with formal semantics and model-checking tools [10, 24]. Timed Rebeca [1] is proposed as an extension of Rebeca language with time constraints and analysis support.

The formal semantics of Timed Rebeca was offered using Structural Operational Semantics (SOS) rules [17].

In the first implementation of Timed Rebeca, a tool was developed to translate Timed Rebeca models to Erlang programs automatically, and McErlang [7] was used to simulate the translated Erlang program [1]. At that time, McErlang, a model checking and simulation tool for Erlang, did not support model checking of Erlang program with timing features. In the untimed version of McErlang, simulation takes place by simply executing the Erlang program, and the reason for using McErlang is the monitors provided by this tool. By using monitors one can stop the execution by observing an erroneous state or unexpected behavior in the program, it is also possible to collect the necessary data during the execution. This tool can be used to run multiple simulations for different settings of parameters in a Timed Rebeca model, and then the results of the executions can be employed to select the most appropriate values for the parameters. This version of McErlang is not efficient for timed models since the progress of time is modeled by the system time, a model with an average size takes a long time to be executed.

Contribution. In this work, we extended the previous version of Timed Rebeca to improve its usability, and also to be able to use the timed version of McErlang which is recently developed [6]. To improve the usability of Timed Rebeca, the language is extended to support *list* data structure and capability of calling custom functions from Erlang, this way the effort for modeling more complicated systems using Timed Rebeca is decreased. Moreover a function named *checkpoint* is added to the language to be able to provide more data to McErlang and hence get more valuable data in the analysis.

Based on the timed version of McErlang, we changed the mapping of timing primitives of Timed Rebeca models to Erlang presented in [1], and we adjusted the implementation of the tool accordingly. As stated in [6], during the development of McErlang with timed semantics there has been a close collaboration between the two teams. So, the timed semantics of McErlang supports the timing features of Timed Rebeca very well. Now, using the *checkpoint* functions we are able to model check and simulate Timed Rebeca models by McErlang.

The approach employed in the timed version of McErlang is inspired by Lamport's approach to real-time model checking [13]. The McErlang team used the idea of maximum-time-elapse for progress of time. The timer is increased based on the time of the occurrence of the next event, so, we have a jump to the next value for the timer instead of having a tick function to increase the timer by one. Finding the next event is not difficult in Erlang, as all the real-time computations are encountered within *receive* statements where timeouts are defined (in an optional *after* clause). Hence, simulation of Timed Rebeca models is much more efficient comparing to the previous work where McErlang basically executed the Erlang programs.

Applications. Since its introduction, Timed Rebeca is used in different areas. One example is in analyzing different routing algorithms and scheduling policies in NoC (Network on Chip) designs, specially the GALS (Globally Asynchronous Locally Synchronous) NoC [20, 21]. Another example is schedulability analysis of distributed real-time sensor network applications, more specifically a real-time continuous sensing application for structural health monitoring in [14], which is an ongoing project. Another ongoing project is on evaluating different dispatching policies in clouds where we have priorities and deadlines in MapReduce clusters, based on the work in [8]. The extensions provided by the work presented in this paper can help in modeling more complicated designs, and also collect more useful data during simulation runs.

Comparing to others. Comparing to Erlang which is a functional actor-based programming language, Rebeca is an imperative actor-based modeling language. So, by using Rebeca while respecting the actor programming style you can write your code in an imperative style which is more familiar to most of the programmers nowadays. Moreover, by using Rebeca you are using a model-driven development approach. You can start by small models and use model checking and simulation to find possible correctness problems in your core algorithms, and also find how to improve the performance by changing some parameters while the code is still small, understandable, and easily manageable.

Two of the mostly used timed modeling languages are UPPAAL [26] and real-time Maude [16]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata [4], extended with data types (bounded integers, arrays etc.). The tool is currently the most well-known model checker for real-time systems. The modeling languages used by Timed Rebeca and UPPAAL differs greatly, while Rebeca has a programming-like syntax UPPAAL uses automata. UPPAAL is more convenient for modeling systems with synchronous agents while Timed Rebeca focuses on distributed and asynchronous agents. Modeling message queue can cause state explosion in UPPAAL very quickly. The verification tools are different in Timed Rebeca and UPPALL. The type of properties supported by UPPALL and Timed Rebeca is different. Timed properties can be checked in UPPAAL while in Timed Rebeca timed properties are not supported. In Timed Rebeca safety properties are checked which are explained in more details in Section 4.

Real-time Maude is a language accompanying with a tool supporting the formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, and emphasizes generality and ease of specification, and is suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques,

including timed rewriting for simulation purposes, and time-bounded linear temporal logic model checking. Timed Rebeca and Real-Time Maude are different in the computational paradigms that they naturally support. Timed Rebeca is based on actor based model of computation while you are free in your modeling style using real-time Maude. Timed Rebeca benefits from its similarity with other commonly used programming languages and is more susceptible to get used by modelers without intimate knowledge of the formal methods.

Regarding other analysis techniques and tools for Rebeca, a new approach was proposed for schedulability and deadlock freedom analysis of Timed Rebeca models in [11]. Authors proposed notion of *floating time transition system* for which the formal definition is presented. Authors defined a bisimulation relation between this transition system and the transition system derived from SOS rules of Timed Rebeca [1]. They developed a verification tool based on floating time transition system and integrated it in Afra tool-set [18]. Afra is an integrated environment for modeling and verification of Rebeca models. Although the proposed approach covers two important safety properties of distributed real-time systems, the performance evaluation of Timed Rebeca models is not supported. In addition, the verification of Timed Rebeca models is restricted to schedulability and deadlock freedom properties.

The rest of the paper is organized as follows. Section 2 gives a brief introduction to Timed Rebeca. Considering Timed Rebeca language presented in [1], Section 3 defines a new mapping for timing primitives of Timed Rebeca to Erlang while adapting to timed extensions of McErlang. It also includes new features added to Timed Rebeca language to increase its usability. Section 4 explains how safety monitors in McErlang can be used to verify safety properties of Timed Rebeca models. Section 5 describes how Timed Rebeca models can be simulated by McErlang. The result is a dataset including useful information about system behavior to which different analysis methods can be applied. Section 6 explains a typical example of elevator. We model check and simulate the elevator model to investigate the efficiency of different scheduling algorithms for it. Finally, Section 7 concludes the paper.

2. Timed Rebeca

Timed Rebeca is proposed as an extension to Rebeca, for modeling and verification of real-time distributed systems [1]. Rebeca [23, 25] is an actor-based language for modeling and verifications of reactive systems with asynchronous communication among actors, called rebecs. Each rebec has an unbounded buffer, called message queue, for its arriving messages. Each rebec takes a message, that can be considered as an event from the top of its message queue, and execute its corresponding message server (also called a method).

In Timed Rebeca model, each rebec has its own local clock, but there is also a notion of global time based on synchronized distributed clocks of the rebecs. Instead of a message queue for each rebec, there exists a bag containing all the arrival messages of each rebec. Messages that are sent to a rebec are put in its message bag together with their arrival time, called time tag, and deadline. Methods are executed atomically, but passing of time during the execution of methods can be modeled. In addition, communication delay and deadline for execution of messages can be defined in the model. The timing primitives that are added to the Rebeca syntax to support these features are *delay*, *deadline*, and *after*. The descriptions of these constructs are in the following.

- Delay: *delay(t)*, where *t* is a positive natural number, increases the value of the local clock of the respective rebec by the amount *t*.
- Deadline: *r.m() deadline(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a natural number, means that the message *m* is sent to the rebec *r* and it is put in the message bag. After *t* units of time the message is not valid any more and is purged from the bag. Deadlines are used to model message expirations (timeouts).
- After: *r.m() after(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a natural number, means that the message *m* is sent to the rebec *r* and it is put in the message bag. The message cannot be taken from the bag before *t* time units is passed. *After* primitive is used to model network delays in delivering a message to the destination. It also can be used to model periodic events.

An example of a Timed Rebeca model is shown in Listing 1. This is a model of a single server queueing system. The model consists of two reactive classes: *ArrivalProcess* and *Server*. As shown in Line 7, The instance of *ArrivalProcess* sends a message to the instance of *Server* to be put in the queue of the server. As specified by *after* primitive in the send statement, it takes one time unit to deliver the message to the server, and as specified by *deadline* primitive it should be served in two time units to prevent the timeout. The statement *delay(2)* in line 8 shows that some process is taking place that takes two units of time. After the delay the rebec sends a message to itself to model its periodic behavior. *Server* has a queue, defined as a variable of type *list* in line 15, which stores the requests that are sent to the rebec. *List* is one of the extensions of Timed Rebeca language which is explained in Section 3.3. Requests are served in lines 24 and 25 of *serverinit* message server. Sending message to itself in line 27 models periodic behavior of the server.

```
1 reactiveclass ArrivalProccess(2) {
2  knownrebecs { Server srv; }
3  statevars { int token; }
4
5  msgsrv initial() { token = 1; self.sendRequest(); }
```

```
 msgsrv sendRequest() {
   srv.queue(token) after(1) deadline(2);//Send token to Server
   delay(2);                             //Process delay
   token = token + 1;                    //Increment Token
   self.sendRequest() after (5);  //Periodic task definition
 }
}
reactiveclass Server(2) {
 knownrebecs { ArrivalProcess proc; }
 statevars { list<int> qserver; }

 msgsrv initial() { self.serveinit(); }
 msgsrv queue(int tok) {
   qserver.insert(tok);     //Insert token to Server Queue
   delay(1);                //Process delay
 }
 msgsrv serverinit() {
    if(qserver.size() > 0) {
       int reqToken = qserver.first(); //Take the first request
       delay(1);                       //Processing time
    }
    self.serverinit() after(1);     //Periodic task
 }
}
main {
  ArrivalProccess proc(srv):();
  Server srv(proc):();
}
```

Listing 1. Timed Rebeca model - Single server queueing system. An extended version of this Timed Rebeca model is shown in Section 5.

3. Mapping for Timed Rebeca Models

In this section, we explain the mapping algorithm for Timed Rebeca models to Erlang while conforming new timed features of McErlang. When the first version of Timed Rebeca was proposed in [1], McErlang did not provide timed semantics. We also explain new features added to Timed Rebeca language to make it more convenient to use. New features include checkpoint, calling custom functions, and list data structure which are explained in more details in Section 3.3.

3.1 Timed Semantics of McErlang

Here, we briefly explain timed semantics of McErlang introduced in [6] which will be used in the new mapping of Timed Rebeca models to Erlang. Erlang handles time with the use of *after* as a timeout clause in a *receive* statement as Listing 2 shows. When a process reaches a receive expression, it looks for the oldest message in the mailbox of the process to match it with any of the patterns, for example with $Pattern_j$. Also, the corresponding guard, i.e. $Guard_j$ should be satisfied. If no pattern is matched and *TimeoutValue* is reached then the expression *TimeoutExpression* is evaluated. The timing features of Timed Rebeca, *delay*, *deadline*, and *after* can all be modeled using the *after* and *timeout* in Erlang.

```
receive
  Pattern1 when Guard1 -> Expr1;
  ...
  PatternN when GuardN -> ExprN;
after
  TimeoutValue -> TimeoutExpression
end
```

Listing 2. Erlang syntax of a receive with timeout.

Handling non-timed and timed actions. In Timed Rebeca, during model checking or simulation we have to respect the order of execution of rebecs based on the order implied by the timing specified in *delay* and *after* primitives. In McErlang no order of execution is respected based on delays. A naive mapping to Erlang may overlook this difference in the semantics. Urgent constructs in McErlang are defined to give us the behavior that is consistent with the semantics of Timed Rebeca. In Erlang it is possible to assign urgency, or a maximum waiting time of zero to a receive statement with no timeout clause. This makes all non-timed message passing (actions) in the translated code to only happen instantaneously, or *infinitely fast* as stated in McErlang references. Furthermore, the semantics of McErlang state that the time cannot advance if there is a transition (message) enabled with maximum waiting time of zero. In a similar way, urgency can be assigned to non-zero timeouts which forces respecting of the order.

Timestamps. In McErlang with timed semantics a new API *mce_erl_time* is introduced to provide the definition and manipulation of timestamps. This new API has the following functions.

- now(): returns the current time.
- nowRef(): stores the current time in a clock reference.
- was(Ref): returns the time stored in a clock reference.
- forget(Ref): stops a stored clock reference.

Some points should be considered in using this API. The absolute values returned from calls to *now()* can not be used by the program. They can only be compared with the previously recorded clocks, i.e., relative comparisons are permitted that shows how much time has elapsed since an event happened.

3.2 Adapting Timed Rebeca with Timed Semantics of McErlang

The timed version of McErlang proposed in [6] makes the formal verification of timed programs written in Erlang programming language possible. In timed semantics, timed actions, i.e. actions with timeout clause, are ordered based on the timeout value while untimed actions, i.e. actions without timeout clause, are executed infinitely fast.

Timing primitives of *delay* and *after* in Timed Rebeca are mapped to receive statement with timeout clause in Erlang. There are two main point to consider regarding the new timed semantics of McErlang. Firstly, the mapping algorithm of timing features in Timed Rebeca to Erlang should be changed according to new timed features of McErlang like timestamps. Secondly, new mapping algorithm for Timed Rebeca models should make the correct order of execution of actions possible. In the following paragraphs we explain these two points in more details.

Mapping timing primitives of Timed Rebeca to Erlang In the previous Timed Rebeca mapping to Erlang, function *now()* was used to obtain the current time by using system clock [1]. Timed behaviors like sending messages with *deadline*, *after*, and *delay* statements were implemented in terms of the system clock. In our new mapping, we use the same concepts as described in [1] but with a few differences in implementation. We use clock references accessible from API *mce_erl_time* to map timed actions from Timed Rebeca to Erlang.

Message send in Timed Rebeca is translated to a regular message send in Erlang. Instead of tagging the message with the local time of the sender, we utilize a clock reference which is sent as a parameter to the receiver. The clock reference is obtained from calling *nowRef()* and stored in the variable *TT*. The clock can be remembered later for relative comparisons by calling *was(Ref)*. Message send also consists of some other information for the receiver such as deadline, message name, and parameters as illustrated in Listing 3. The default value for deadline is *inf* (standing for infinity) which denotes no deadline.

```
1 messagesend(Sender, Rebec, Msg, Params, Deadline) ->
2   % Start a clock reference and save it to TT
3   TT = nowRef(),
4   spawn(fun () ->
5         Rebec ! {{Sender, TT, Deadline}, Msg, Params}
6   end).
```

Listing 3. Pseudo McErlang code for message send

After receiving a message, its deadline should be checked by the receiver before processing it. The timestamp of the message is the local time of the sender when sending the message and can be remembered using function *was(Ref)*. The local time of the receiver when receiving the message can be obtained by function *nowRef()*. So, if the message has not been expired, this condition $deadline + was(ref) < nowRef()$ should be satisfied.

In Timed Rebeca semantics, a message with the *after* statement should be put in the message bag of the receiver, and it can not be taken from the bag before the specified time has elapsed. In mapping to Erlang, a function is spawned and waits for the specified amount of time before sending the message. The function is an empty receive statement with a timeout clause, and sending the message is placed in the timeout clause as demonstrated in Listing 4.

```
1 messagesend(Sender, After, Rebec, Msg, Params, Deadline) ->
2   TT = nowRef(),
3   spawn(fun () ->
4     % Delay Process By "After".
5    recieve
6    after(Timeunits) ->
7         Rebec ! {{Sender, TT, Deadline}, Msg, Params}
8   end).
```

Listing 4. Pseudo McErlang code for message send with After

The *delay* statement makes the local time of a rebec advances with the specified amount of time. In Erlang, the delay is translated to the receive statement including just a timeout value as shown in Listing 5. Since there is no pattern in the receive statement, the timeout clause (after clause) will be executed after the specified time. As stated in [6], the function *mce_erl:urgent(MaximumWait)* can be used to determine the urgency of a state, i.e., how much the process can stay in this state. So, we use the urgent function in the McErlang code to make the delayed process run immediately after the timeout expires.

```
1 timedelay(Timeunits) ->
2   % McErlang Urgent Delay
3   urgent(Timeunits),
4   % Delay by Timeunits
5   receive
6     after (Timeunits) -> ok
7   end.
```

Listing 5. Pseduo McErlang code for a message with Delay

Performing timed and untimed actions in the correct sequence In Timed Rebeca, the execution order of messages are specified with respect to the values of primitives *delay* and *after*. Messages with no accompanying timing primitives are called untimed messages. We explained the corresponding Erlang code of Timed Rebeca statements including: delay statement, sending untimed messages, and sending messages with an after primitive. To execute messages in the correct order based on Timed Rebeca semantics, we should take into account more considerations in corresponding Erlang codes as follows.

- messages without timeout clause in Erlang (messages without after in Timed Rebeca) should be executed infinitely fast (immediately).
- messages with timeout clause in Erlang (delays or messages with after in Timed Rebeca) should be executed immediately after the timeout expires. The messages are ordered based on their timeout.

Using timed extensions in McErlang, we can change the way in which timed and untimed actions are treated. We can use the function *mce_erl:urgent(MaximumWait)* to specify the urgency of actions. To execute the untimed actions infinitely fast, the *MaximumWait* parameter is set to zero. To execute the timed actions immediately after their timeout expires, the *MaximumWait* parameter is set to the value of timeout. Since the urgency of actions are defined, actions can be ordered by using function *timeRestrict*.

3.3 New Extensions of Timed Rebeca Language

We added some capabilities to Timed Rebeca in order to increase the modeling power of the language. These additions include list data structure, capability of calling custom functions from Erlang language, and checkpoints. Table 1 shows the abstract mapping of Timed Rebeca extensions to Erlang.

Checkpoint functions can be used in both simulation and model checking. They are considered as a marker in the code that indicates important events. Checkpoints are used to pass the necessary and useful information of a state to the tool. A checkpoint has two mandatory arguments: *label* and *term*. *Label* is an arbitrary name which is defined by the modeler and is used to refer to a checkpoint. *Term* are variables that are added to the checkpoint function as its arguments. Terms are used to pass necessary information to the checkpoint function such that it can be retrieved during simulation or model checking.

Timed Rebeca Syntax		**Erlang / McErlang**
$list < int > N$;	$\rightarrow$	Erlang list data type as a variable with name N.
$erlang.func(V_1,...,V_n)$;	$\rightarrow$	Call to function $func$ with parameters V_1,...,V_n.
$checkpoint(L,T_1(,T_2,...,T_n))$;	$\rightarrow$	Erlang Output Function for simulation. L and T_i are the arguments.
$checkpoint(L,T_1(,T_2,...,T_n))$;	$\rightarrow$	McErlang probe for model checking. L and T_i are its label and term respectively.

Table 1. Abstract mapping of Timed Rebeca extensions to Erlang and McErlang, where $func$ is the name of a function, L is a label for a checkpoint, and T_i is the term of a checkpoint (a state or a local variable name). When doing model checking T_i is used to define a term of the generated McErlang probe.

Another notable extension in Timed Rebeca language is calling custom functions from models. The benefit of this approach is that a modeler can define functions in Erlang language and then call them from the Timed Rebeca model. For example, in Timed Rebeca there is no function for searching a list. So, this function can be defined in Erlang and be called in Timed Rebeca model. Using this extension, Timed Rebeca language has the same programming power as Erlang language.

Applications in which implementing buffers or queues are essential, like for schedulers, can be modeled by using list data structure added to Timed Rebeca language. The elements of a list are of the type of integer. They can be defined inside message servers as a local variable or as a state variable. Some useful functions are defined in order to be able to manipulate elements of a list.

4. Model Checking Timed Rebeca Models

McErlang provides two types of model checking facilities for verification of *safety* properties and *Linear Temporal Logic (LTL)* formulas, using *safety monitors* and *büchi monitors* respectively. In this work safety monitors are used for corresponding Erlang code of Timed Rebeca models in order to verify safety properties of Timed Rebeca models. For a given Erlang program, a safety monitor is defined as a function which is called after creation of each state of the model. If the content of the state is invalid, the safety monitor reports the state as an erroneous state.

4.1 Checking Safety Properties

To define safety properties, McErlang allows safety monitors to access both states of program and the sequence of actions, as labels of transitions among states, but the values of program variables are not allowed to be accessed. However, the safety properties of Timed Rebeca models are defined based on the value of variables. We should translate the variables of Timed Rebeca model in a way to be accessible in safety monitors. To this end, when model checking, checkpoints are translated to McErlang probes, which are accessible by monitors. As we discussed in Section 3.3, the value of intended variables are passed as arguments to checkpoints. Also, the occurrence of interesting events can be specified using checkpoints.

4.2 Defining Safety Monitors

In this subsection, we explain two predefined safety monitors for Timed Rebeca models and present a framework for defining monitors in McErlang, using checkpoints.

Deadlock monitor Detecting deadlock in non-terminating systems is essential. The predefined monitor in Listing 6 can be used to investigate the deadlock of Timed Rebeca models. As lines 13 to 20 of Listing 6 show, deadlock is detected by checking the status of processes. If status of all the processes is marked as *blocked*, deadlock is reported.

```
1  monitorType() -> safety.
2
3  init(State) -> {ok,State}.
4
5  stateChange(State,MonState,_) ->
6    case is_deadlocked(State) of
7      true -> deadlock;
8      false -> {ok, MonState}
9    end.
10
11 is_deadlocked(State) ->
12   State#state.ether =:= [] andalso
13     case mce_erl:allProcesses(State) of
14       [] -> false;
15       Processes ->
16        case mce_utils:find(fun (P) ->
17         P#process.status =/= blocked end,
18          Processes) of
19            {ok, _} -> false;
20             no -> true
21          end
22        end.
```

Listing 6. McErlang - Deadlock monitor

Maximum Queue Length Monitor Although in theory message queues are unbounded in Timed Rebeca, but in model checking and simulation we need a maximum length for each queue to keep the state space bounded. Trying to put messages beyond the queue size of a rebec results in queue over flow error in Timed Rebeca models. The predefined

maximum queue-size monitor in McErlang can be used to monitor the size of a rebec's queue. As lines 7 to 10 of Listing 7 show, if a queue of any process exceeds its maximum size, a violation is reported by the monitor. The maximum queue size is specified by parameter *MaxQueueSize*.

```
1  monitorType() -> safety.
2
3  init(MaxQueueSize) -> {ok,MaxQueueSize}.
4
5  stateChange(State, MaxQueueSize, _) ->
6      case mce_utils:find
7        (fun (P) -> length(P#process.queue) > MaxQueueSize end,
8         mce_erl:allProcesses(State)) of
9        {ok, P} -> {exceeds, P};
10       _ -> {ok, MaxQueueSize}
11     end.
```

Listing 7. McErlang - MaxQueue monitor

User-defined Monitor for Checkpoint The purpose of defining checkpoints in Timed Rebeca model is the verification of safety properties using McErlang. User-defined monitors return *satisfied* if a state which the monitor examines satisfies the required conditions. Otherwise it returns *violation*. Listing 8 shows a template for user defined monitors in which checkpoints are used to verify a property. The monitor returns *satisfied* if checkpoint with the specified label *CheckpointLabel* cannot be detected, otherwise a *violation* will be reported. Looking for a checkpoint in actions is performed by function *checkLabelCheckPoint* as shown in line 14. We also developed some other functions to make definition of monitors easier. The signature of each function signature and a brief explanation are listed in the following.

- Checking if a dropped event happens for a message server, because of the deadline missed.
  - checkDropMsgsrv(Actions *stateActions*, Atom *msgSrvName*)
- Checking if a checkpoint occurs.
  - checkLabelCheckPoint(Actions *stateActions*, Atom CheckPointLabel)
- Compare the checkpoint *term* with an integer or boolean.
  - checkTermMaxValue(Actions *stateActions*, Atom *CheckpointTerm*, Int *MaxValue*)
  - checkTermMinValue(Actions *stateActions*, Atom *CheckpointTerm*, Int *MinValue*)
  - checkTermValue(Actions *stateActions*, Atom *CheckpointTerm*, Int *SatisfyInt*)
  - checkTermValue(Actions *stateActions*, Atom *CheckpointTerm*, Boolean *SatisfyBool*)

```
monitorType() -> safety.

init(_) -> {ok, satisfied}.

stateChange(_,satisfied,Stack) ->
  % Monitor Setup
  % Usage: checkpoint(Label,Term);
  % Note: Dropped message have "drop" label so its not needed.
  CheckpointLabel = checkpoint_label, % Not needed for checking
       expired message probes.
  CheckpointTerm =
       not_applicable_when_using_checkLabelCheckPoint,
  % EOF

  Actions = actions(Stack),
  checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 8. McErlang - A template for user defined monitors in which Timed Rebeca checkpointus are used

5. Simulation

In addition to the model checking facilities of McErlang, it provides simulation of Erlang programs. In the simulation mode, the next state of an Erlang program is determined randomly, by choosing one of the available transitions from the current state. Therefore, a randomly chosen path of execution is explored in each run of simulation.

After translating Timed Rebeca model to Erlang program, McErlang is configured to be used in simulation mode. To have an accurate understanding of the model's behavior, data is gathered from different simulation runs, each of them including a different trace. For performance evaluation, statistical methods are applied to the collected data and the results are used to reason about the behavior of the model.

5.1 Performance Evaluation Toolset

We implement a toolset to provide performance evaluation of Timed Rebeca models using McErlang. As shown in Figure 1, the toolset contains three components as the following.

- *timedreb2erl*: for translating Timed Rebeca models to Erlang programs.
- *timedrebanalysis*: to apply statistical analysis methods to stored information. Different analysis techniques are implemented in this component.
- *timedrebsim*: as a simulation wrapper component which sends required data to other components and stores data of simulation runs. Modeler can define the number of simulations as well as the duration of each simulation run.

Figure 1 shows that *timedrebsim* component sends Timed Rebeca models to *timedreb2erl* component to be translated to Erlang program. Translated Erlang program is sent to McErlang for simulation. The generated data from the simulation is sent to *timedrebsim* component at run-time. The *timedrebsim* component categorizes the simulation data of different simulation runs in a way to be used by *timedrebanalysis*.

We implement two different analysis techniques in component *timedrebanalysis*, called *checkpoint analysis* and *paired-checkpoint analysis*, to provide performance evaluation of Timed Rebeca models. In the next section, we ex-

plain how information provided by checkpoints can be used in *timedrebanalysis* to achieve performance measures of interest.

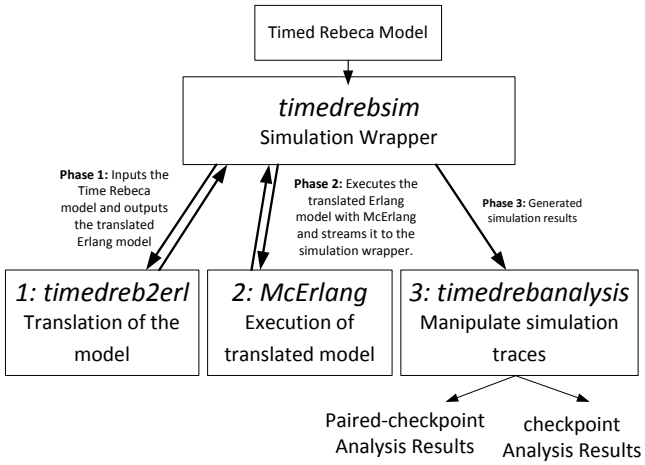

Figure 1. Architecture of analysis tool-set.

5.2 Evaluation of Checkpoints

As we discussed in Section 3.3, checkpoints were added to Timed Rebeca language to provide needed information for model checking and simulation. Each checkpoint is translated to a function such that McErlang can access the value of variables and be notified of occurrence of events. We analyse models based on information provided by checkpoints.

During the simulation, every time a checkpoint is executed the value of terms (variables or any value of available data types), label, the time of observing the checkpoint and the name of the rebec including checkpoint are stored for performance evaluation purposes.

To illustrate the role of checkpoints in performance evaluation of Timed Rebeca models, we extend the model of single server queuing system of Listing 1 to a more detailed one as shown in Listing 9. The performance of such models is influenced by the policy of serving the requests of the queues, called queuing policy. Simulation of the Timed Rebeca model of the queuing system allows us to find the system's performance measures including: throughput of the system, average waiting time in the queue, and server performance.

Five different checkpoints are defined in lines 8, 25, 37, 40, and 41 to collect required data for performance evaluation of the model. These checkpoints stores data about when the request is sent to the server, when it is received by the server, when the server puts the received message in its queue, when the received message leaves the message queue, and when the server finishes execution of a message, respectively. Throughput of the queuing system is measured using the starting checkpoint in line 8 (label *requestStart*) and the ending checkpoint in line 40 (label *requestFinished*). Waiting time of a request in queue is measured using the starting checkpoint in line 25 (label *requestInQueue*) and the ending checkpoint in line 37 (label *serverBegins*). Server performance is measured using the starting checkpoint in line 37 (label *serverBegins*) and the ending checkpoint in line 40 (label *requestFinished*).

In the following subsections, we use some of the aforementioned checkpoints to explain how performance evaluation of Timed Rebeca models is achieved by using implemented analysis techniques in component *timedrebanalysis*.

```
1  reactiveclass ArrivalProccess(2) {
2   knownrebecs { Server srv; }
3   statevars { int token; }
4
5   msgsrv initial() { token = 0; self.send(); }
6
7   msgsrv sendRequest() {
8       checkpoint(requestStart,token); // Start of Request
9       int DelayArrivalSend = ?(1,2,9,10); // Non-Deterministic
            Delay
10      delay(DelayArrivalSend); // Delay for Arrival Requests
11      srv.queue(token); // Send token to Queue of the Server
12      token = token + 1;
13      self.sendRequest(); // Iterate
14  }
15 }
16
17 reactiveclass Server(2) {
18  knownrebecs { ArrivalProcess proc; }
19  statevars { boolean serveractive; list<int> qserver; }
20
21  msgsrv initial() { serveractive = false; }
22
23  msgsrv queue(int tok)
24  {
25       checkpoint(requestInQueue,tok); // Mark Queue input
26       qserver.insert(tok);
27       if(serveractive == false) { // Initiate Server if not
             running
28         self.serverinit();
29       }
30  }
31  msgsrv serverinit() {
32       serveractive = true;
33
34       if(qserver.size() > 0) {
35         int reqToken = qserver.first(); // FIFO, take first
               request out
36         qserver.remove(reqToken);
37         checkpoint(serverBegins,reqToken); // Mark request
               processing
38         int processtime = ?(1,2,3,4,5);
39         delay(processtime); // Non-Deterministic processing time
40         checkpoint(requestFinished,reqToken); // Mark finished
               request
41         checkpoint(processQueueSize,qserver.size()); // Mark
               size of the queue
42       }
43
44       if(qserver.size() > 0) { // Continue processing if queue
             not empty
45          self.serverinit();
46       } else {
47          serveractive = false;
48       }
49  }
50 }
51
52 main {
53   ArrivalProccess proc(srv):();
54   Server srv(proc):();
55 }
```

Listing 9. Timed Rebeca model - Single server queueing system

5.2.1 Paired-checkpoint analysis

The paired-checkpoint method is implemented in the *timedrebanalysis* tool. In paired-checkpoint analysis, two checkpoints are grouped together. The modeler specifies paired checkpoints with the use of labels before running the tool. The elapsed time between observing two paired checkpoints is important and can show different performance measures. For example, in Listing 9, checkpoint with label *requestInQueue* shows that the request is enqueued and the checkpoint with label *serverBegins* represents that the request is taken from the queue to be served. Consequently, the passed time between the occurrence of these two checkpoints is considered as the waiting time in the queue.

Two performance measures, the average response time and the average waiting time, are important and can be measured by paired-checkpoint analysis. The checkpoints defined in the model can be paired in the *timedrebanalysis* tool to extract the interesting information from the data collected by tool *timedrebsim*. The corresponding checkpoints in the model to capture these two performance measures are listed as follows:

- Response time: the amount of time passed between requesting a service and completing the service by the server. checkpoints with labels *requestStart* and *requestFinished* represents the beginning and the end of request respectively.
- Waiting time: the amount of time that a request waits in the queue to be served. The checkpoint with label *requestInQueue* shows the request arrival to the queue and the checkpoint with label *serverBegins* represents the request departure from the queue.

5.2.2 Checkpoint analysis

Checkpoint analysis is another possible way to provide performance evaluation. In checkpoint analysis, instead of pairing checkpoints, a certain checkpoint is provided to expose the changes of a particular variable over time. For example, in the single server queueing system, the number of requests in the queue, i.e. queue size, is important and can be available in the simulation results by defining the checkpoint with label *processQueueSize* in the model. When a request is enqueued at run-time, the queue size as well as the present time is stored in the simulation results.

The *timedrebanalysis* tool is used to extract useful information corresponding to the queue size from simulation results collected by *timedrebsim* tool.

Since the resulted information of performance measurement may be very large, we use average moving method to reduce the dataset for visualization. This well-known method smooths out short-term fluctuations and highlights long-term trends of the data [22].

6. Case Study and Experimental Results

sham bokhor

In this section, we show a Timed Rebeca model for an elevator system in which a centralized coordinator manages how the requests are dispatched among the elevators, and also decides on the movements of the elevators. The approach for dispatching of requests is called *scheduling policy*, and the decision on the movement of elevators between the floors is called *movement policy*.

We implemented three different scheduling policies, namely *shortest distance*, *shortest distance with movement priority*, and *shortest distance with load balancing*, and two different movement policies, namely *up priority*, and *maintain movement*. We define four different configurations for the elevator system, each of them including one of the aforementioned scheduling and movement policies (all the combinations are not considered). The complete Rebeca model for the elevator system and more detailed explanations can be found at [18] and [12].

To ensure the correctness of the behavior of the model in each configuration, we first model check the model, then using simulation, we compare the performance of configurations in terms of expected response time. The proposed techniques introduced in Sections 4 and 5 are applied for model checking and simulation of the configurations, respectively.

6.1 Model of the Elevator System

Figure 2 shows the event graph of the elevator model. We use event graph [5] to give a highly abstracted view of events and their causality relations. Event graphs are widely used for the explanation of event-based models. The nodes represent events in a system and the edges represent the causality relation between events (nodes). Additionally, we add a label below each node that shows in which reactive class the event occurs. There are four reactive classes *Person*, *Floor*, *Elevator*, and *Coordinator* in the Timed Rebeca model of the elevator system. Rebecs *el1* and *el2* are instantiated from *Elevator* as the elevators of the system. Also, rebecs *floor1* to *floor3*, rebec *pers*, and rebec *coord* are instantiated from reactive classes *Floor*, *Person*, and *Coordinator* repectively, to show two elevators, three floors and one person in the model. Algorithms which are related to the scheduling and movement policies are implemented in the message servers *handlerequest* and *handleElevatorMovement* of Coordinator of the model.

The rebec *pers* sends the requests randomly to one of the floors and one of the elevators by sending messages *callElevator* and *requestFloor*, as shown in Figure 2. Sending a message to a floor, i.e. calling *callElevator*, shows that a person standing in a specified floor presses the button asking for an elevator to come. This request has to be forwarded to the

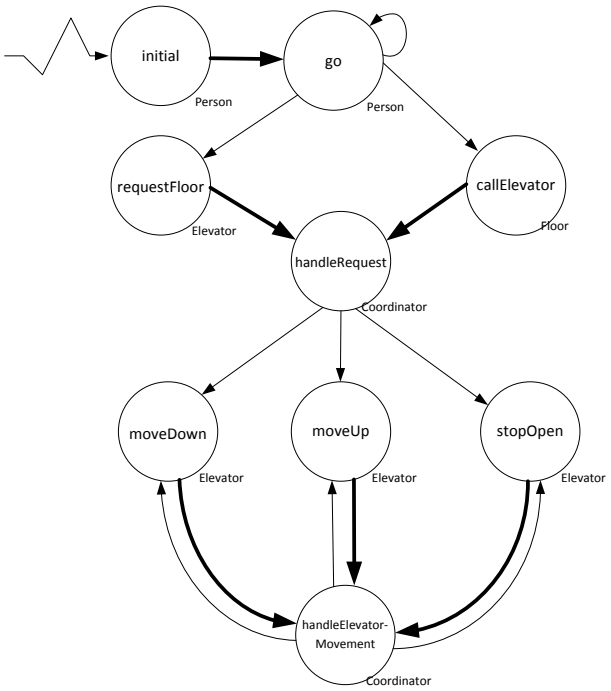

Figure 2. Event graph of the centralized elevator system.

appropriate elevator to be served. Sending a request to an elevator, i.e. calling *requestFloor*, implies that a person inside an elevator desires to go to the requested floor.

6.2 Safety Verification

We use checkpoint monitors as discussed in Section 4, to verify the safety properties of the model. The first safety property which is verified to ensure the correctness of the model is the value of the elevator location. This value must be within the valid range which is one to three. In Timed Rebeca model, the checkpoint *elevatorLocation* is defined to make the value of elevator locations available for model checking. To check the maximum and minimum value of checkpoint *elevatorLocation*, we use the predefined functions *checkTermMaxValue* and *checkTermMinValue* respectively, as shown in Listing 10.

```
monitorType() -> safety.

init(_) -> {ok, satisfied}.

stateChange(_,satisfied,Stack) ->
  Actions = actions(Stack),
  checkTermMinValue(Actions,elevatorLocation,0),
  checkTermMaxValue(Actions,elevatorLocation,3),
  checkTermValue(Actions,elevator1StopReqInList,-1),
  checkTermValue(Actions,elevator2StopReqInList,-1),
```

Listing 10. Checkpoint monitor for the elevator system with three floors.

We are also interested in checking whether the elevators stop on the floors which are not requested. The predefined function *CheckTermValue* is used to check whether the value of checkpoints *elevator1StopReqInList* and *elevator2StopReqInList* equals to -1, which means the elevator stops on the incorrect floors. The results of model checking of Elevator model, using the mentioned properties, are shown in Table 2.

Parameter	Condition	Result
Elevator location	Location $0 >$	Satisfied (40929 states) 112.4 seconds
Elevator location	Location < 3	Satisfied (40929 states) 111.6 seconds
Stop Queue 1	$\neq -1$	Satisfied (40929 states) 110.5 seconds
Stop Queue 2	$\neq -1$	Satisfied (40929 states) 109.5 seconds

Table 2. Safety verification results for the elevator system.

6.3 Simulation

In this section, we explain different scheduling and movement policies which are implemented in message servers *handleRequest* and *handleElevatorMovement*, respectively. We consider four different scenarios each of them with different scheduling and movement policies. The efficiency of the proposed scenarios is revealed by comparing the expected response time of scenarios. The simulation of the scenarios take place with the same settings to be able to compare the simulation results.

6.3.1 Scheduling Policy

Shortest distance, *shortest distance with movement priority*, and *shortest distance with load balancing* are three different scheduling policies which are studies in the experiments. Listing 11 in Appendix A shows the message server *handleRequest* in which two different requests are handled. First, the requests sent to a floor are enqueued in the nearest elevator to the floor based on the *shortest distance* scheduling policy. Second, the requests sent to an elevator are enqueued in it.

In the second algorithm which is shown in Listing 12 in Appendix A, both moving direction of the elevator and shortest distance are taken into account to enqueue the requests in the elevators. In this approach, for assigning a request to an elevator the moving direction of the elevators has precedence to the distance of the elevators to the floor from which the request is sent. For example, in the case that *elevator1* is not moving towards the requested floor and *elevator2* is moving towards it, although the new request is closer to *elevator1*, it is enqueued in the queue of *elevator2*.

The third scheduling policy is implemented as shown in Listing 13 in Appendix A. Here the main goal is to balance the number of the requests assigned to the elevators, called

load balancing policy. We also consider the *shortest distance* approach. The queue size of elevators has preference to the distance of request from the elevators. For example, we suppose that the requested floor is closer to *elevator2*, and the queue size of *elevator1* is less than the queue size of *elevator2*, then the requested floor is enqueued in the queue of*elevator1*.

6.3.2 Movement policy

We implemented two movement policies which are *up priority* and *maintain movement*. Listing 14 in Appendix A shows the message server *handleElevatorMovement*, in which *up priority* movement policy is implemented. The policy implies that the elevator attempts to go up first and serve the requests in the higher floors. This message server updates the elevator location and simulates its movement between different floors.

Listing 15 in Appendix A represents the pseudo code of *maintain movement* policy. In maintain movement policy, if the elevator is moving upward (downward) and there are requests from higher (lower) floors, the elevator will continue the moving direction and serve the requests, otherwise it changes its moving direction. In other words, the elevator responds to all requests on its way.

6.3.3 Simulation Results

As mentioned before, we consider four different configurations in which scheduling and movement policies are different:

- configuration 1: scheduling policy: *shortest distance*, movement policy: *up priority*
- configuration 2: scheduling policy: *shortest distance*, movement policy: *maintain movement*
- configuration 3: scheduling policy: *shortest distance with movement priority*, movement policy: *maintain movement*.
- configuration 4: scheduling policy: *shortest distance with load balancing*, movement policy: *maintain movement*.

For each configuration, we used *timedrebsim* to execute 10 simulations, each with 1500 random floor requests with delay of 2 time units. Delay of the elevator movement is 2 time units and the delay of an elevator door opening, and closing is set to a non-deterministic choice of 1, 2, 4 or 6 time units.

The results of analysis of four configurations are shown in Table 3. It shows that configuration of shortest distance policy as scheduling policy and maintain movement policy as movement policy results in the optimum solution among the suggested configurations. Although shortest distance with movement priority policy may seem to have better performance, experimental results show otherwise.

7. Conclusion

In this paper we presented an extension of Timed Rebeca language [1] and its supporting tool. The most significant extension is the definition of checkpoint functions in Timed Rebeca models. Our extensions in the language as well as timed extensions in McErlang provide us with model checking and simulation of timed models. A tool has been developed to translate the Timed Rebeca models to Erlang language. The mapping rules of translation from Timed Rebeca to Erlang, is modified to support timed extensions in McErlang. While model checking, safety monitors in McErlang can be defined to verify the correctness of models with respect to safety properties.

In the developed tool, McErlang is used to simulate Timed Rebeca models in which, the next program state is chosen randomly. Therefore, a random execution path is searched in each simulation run. To have more accurate performance analysis, we can run different simulations to gather data from different traces. The statistical methods are applied to the collected data to reveal the system behavior. Two kinds of performance analysis are provided using statistical methods, which are paired-checkpoint analysis and checkpoint analysis. In checkpoint analysis, our focus is on the evolution of a particular parameter during time. For example, in a single queueing system the number of requests in the queue, i.e. queue size, can be investigated. In paired-checkpoint analysis, we study the difference between two values, like the duration of waiting, or service.

We evaluate the developed tool and proposed methods on a typical case study including centralized elevators. We measure the response time for the requests arriving from each floor in different scenarios. Each scenario includes different scheduling algorithms and movement policies, which are responsible for assigning the requests to the elevators and determining how the elevators move between the floors, respectively.

The focus of our future work is on improving the model checking capabilities. In this work the modeler defines the monitors in order to verify the safety properties in the model. Our goal includes the automation of generating monitors based on a property language. Moreover, we shall try to cooperate with McErlang team in order to develop a more efficient algorithm for state space generation, possibly using coarse-grain transitions.

References

[1] L. Aceto, M. Cimini, A. Inglfsdttir, A. H. Reynisson, S. H. Sigurdarson, and M. Sirjani. Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In *FOCLASA'11*, pages 1–19, 2011.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1990.

[3] G. Agha. The Structure and Semantics of Actor Languages. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, edi-

Configuration	Expected response time (Average)	Median response time (Average)	Max response time (Average)	Total finished requests
1	25.93	14.8	271.3	75154
2	16.55	12.8	77.3	96455
3	20.37	16.6	83.6	79069
4	20.46	18.6	67.3	79755

Table 3. Experimental results summary for the different configurations of the elevators system.

tors, *Foundations of Object-Oriented Languages*, pages 1–59. Springer-Verlag, Berlin, Germany, 1990.

[4] R. Alur and D. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994. .

[5] A. H. Buss. Modeling with Event Graphs. In Proceedings of the 28th conference on winter simulation, pages 153–160, 1996.

[6] C. B. Earle and L. Fredlund. Verification of Timed Erlang Programs Using McErlang. In *Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 251–267, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30792-8. . URL `http://dx.doi.org/10.1007/978-3-642-30793-5_16`.

[7] L.-Å. Fredlund and H. Svensson. McErlang: A Model Checker For a Distributed Functional Programming Language. *SIGPLAN Not.*, 42(9):125–136, 2007. ISSN 0362-1340. .

[8] I. Gupta, B. Cho, M. R. Rahman, T. Chajed, C. L. Abad, N. Roberts, and P. Lin. Natjam: Eviction Policies For Supporting Priorities and Deadlines in Mapreduce Clusters. 2009. URL `https://www.ideals.illinois.edu/handle/2142/44871`.

[9] 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, Apr. 1972.

[10] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar. Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca. *Acta Informaticae*, 47(1):33–66, 2009. .

[11] E. Khamespanah, Z. Sabahi Kaviani, R. Khosravi, M. Sirjani, and M.-J. Izadi. Timed-rebeca Schedulability and Deadlock-freedom Analysis Using Floating-time Transition System. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, AGERE! '12, pages 23–34, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1630-9. . URL `http://doi.acm.org/10.1145/2414639.2414645`.

[12] H. Kristinsson. Event-based Analysis of Real-Time Actor Models - Master Thesis, Reykjavik University, Iceland, 2012.

[13] L. Lamport. Real-Time Model Checking is Really Simple. In *Proceedings of the 13 IFIP WG 10.5 international conference on Correct Hardware Design and Verification Methods*, CHARME'05, pages 162–175, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-29105-9, 978-3-540-29105-3. . URL `http://dx.doi.org/10.1007/11560548_14`.

[14] L. Linderman, K. Mechitov, and B. F. Spencer. TinyOS-based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control. *Structural Control and Health Monitoring*, 20(6):10071020, June 2013. .

[15] I. A. Mason and C. L. Talcott. Actor Languages: Their Syntax, Semantics, Translation, and Equivalence. *Theoretical Computer Science*, 220(2):409–467, June 1999. ISSN 0304-3975. URL `http://www.elsevier.com/cas/tree/store/tcs/sub/1999/220/2/3170.pdf`.

[16] P. C. Ölveczky and J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher Order Symbol. Comput.*, 20(1-2):161–196, June 2007. ISSN 1388-3690. . URL `http://dx.doi.org/10.1007/s10990-007-9001-5`.

[17] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.

[18] Rebeca. Rebeca homepage. http://www.rebeca-lang.org.

[19] S. Ren and G. Agha. RT-synchronizer: Language Support for Real-Time Specifications in Distributed Systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 50–59, 1995.

[20] Z. Sharifi, S. Mohammadi, and M. Sirjani. Comparison of NoC Routing Algorithms Using Formal Methods. To be published in proceedings of PDPTA'13, 2013.

[21] Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani. Functional and Performance Analysis of Network-on-Chips Using Actor-based Modeling and Formal Verification. To be published in proceedings of AVOCS'13, 2013.

[22] J. S. Simonoff. *Smoothing Methods in Statistics*. Springer, 1998. ISBN 978-0-387-94716-7.

[23] M. Sirjani and M. M. Jaghoori. Formal Modeling. chapter Ten Years of Analyzing Actors: Rebeca Experience, pages 20–56. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-24932-7. URL `http://dl.acm.org/citation.cfm?id=2074591.2074596`.

[24] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Model Checking, Automated Abstraction, and Compositional Verification of Rebeca Models. *Journal of Universal Computer Science*, 11(6):1054–1082, 2005.

[25] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Modeling and Verification of Reactive Systems using Rebeca. *Fundamenta Informatica*, 63(4):385–410, Dec. 2004.

[26] UPPAAL. UPPAAL Homepage. http://www.uppaal.com.

A. Pseudocode of Policies

```
msgsrv handleRequst(Floor f)
{
  if (sender is instance of Floor) {
    if Contains(ElvQueue1,f) || Contains(ElvQueue2,f)
      donothing;
    else {
      if (ElvLoc1 == f)
        Add(ElvQueue1,f);
      else if (ElvLoc2 == f)
          Add(ElvQueue2,f);
        else if (ElvLoc1 == ElvLoc2){
            RandQueue = chooseRand(ElvLoc1,ElvQueue2);
            Add(RandQueue,f);
          }
          else if (abs(f-ElvLoc1) > abs(f-ElvLoc2))
              Add(ElvQueue2,f);
            else
              Add(ElvQueue1,f);
    }
  }
  else if (sender is instance of Elevator){
    if (sender == Elevator1 && ElvLoc1 != f &&
          !Contains(ElvQueue1,f))
      Add(ElvQueue1,f);
    else if (sender == Elevator2 && ElvLoc2 != f &&
          !Contains(ElvQueue2,f))
      Add(ElvQueue2,f);
    else if (ElvLoc1 == f || ElvLoc2 == f)
      SendMessage(sender,StopAndOpen);
  }
  // any idle elevators should be started
  ...
}
```

Listing 11. Pseudo code of message server *HandleRequest* where the scheduling policy is shortest distance policy.

```
/* Scheduling policy: Shortest distance policy with movement
      priority. */
...
/* Check if any elevators are already located on the requested
      floor */
...
else if (abs(floor-Elv1Loc) > abs(floor-Elv2Loc)){
  if (floor > Elv2Location && Elv2Movment==1)
    Add(Elv2Queue,floor);
  else if (floor < Elv2Location && Elv2Movment==-1)
    Add(Elv2Queue,floor);
  else if (floor > Elv1Location && Elv1Movement==1)
    Add(ElvQueue1,floor);
  else if (floor < Elv1Location && Elv1Movement==-1)
    Add(ElvQueue1,floor);
  else
    Add(ElvQueue2,floor);
}
else{
  if (floor > Elv1Location && Elv1Movment==1)
    Add(Elv1Queue,floor);
  else if (floor < Elv1Location && Elv1Movment==-1)
    Add(Elv1Queue,floor);
  else if (floor > Elv2Location && Elv2Movement==1)
    Add(ElvQueue2,floor);
  else if (floor < Elv2Location && Elv2Movement==-1)
    Add(ElvQueue2,floor);
  else
    Add(ElvQueue1,floor);
}
...
```

Listing 12. Timed Rebeca pseudo code for scheduling policy *shortest distance with movement priority*. *[...]* denotes the deleted code which has been already shown in Listing 11. The variable *floor* is the requested floor number sent by the rebec *pers*.

```
/* Scheduling policy: Shortest distance policy with load
      balancing. */
...
/* Check if any elevators are already located on the requested
      floor */
...
else if (abs(floor-Elv1Loc) > abs(floor-Elv2Loc)){
  if (Size(Elv2Queue) < Size(Elv1Queue) || Size(Elv2Queue) =
        Size(Elv1Queue))
    Add(Elv2Queue,floor);
  else
    Add(Elv1Queue,floor);
}
else {
  if (Size(Elv1Queue) < Size(Elv2Queue) || Size(Elv1Queue) =
        Size(Elv2Queue))
    Add(Elv1Queue,floor);
  else
    Add(Elv2Queue,floor);
}
...
```

Listing 13. Timed Rebeca pseudo code for scheduling policy *shortest distance with load balancing*. *[...]* denotes deleted code which has been already shown in Listing 11. The variable *floor* is the requested floor number sent by the *pers* rebec.

```
msgsrv handleElevatorMovement(int movement)
{
  // movement=0 means elevator stopped,
  // movement=1 means elevator is going up
  // movement=-1 means elevator is going down
  if (sender == Elevator1 && movement != 0){ //moving elevator
    Elv1Movement = movement;
    if (movement == -1)
      Elv1Location-=1;
    else if (movement == 1)
        Elv1Location+=1;
    if (Size(Elv1Queue) > 0){
      if (Contains(Elv1Queue, Elv1Location)){
        Elv1Queue.Remove(Elv1Location);
        SendMessage(Elevator1,StopOpen);
      }
      else{
        if (Next(Elv1Queue,Elv1Location,1) != -1){
          Elv1Movement = 1;
          SendMessage(Elevator1,MoveUp);
        }
        else{
          Elv1Movement = -1;
          SendMessage(Elevator1,MoveDown);
        }
      }
    }
  }
  else if (sender == Elevator1){ // Stopped Elevator
    Elv1Movement = movement;
      if (Elv1Movement == 0 && Size(ElvQueue1) > 0){
      if (Next(Elv1Queue,Elv1Location,1) != -1){
        Elv1Movement = 1;
        SendMessage(Elevator1,MoveUp);
      }
      else{
        Elv1Movement = -1;
        SendMessage(Elevator1,MoveDown);

      }
    }
  }
  // movement for elevator 2
  ....
```

```
45 }
```

Listing 14. Timed Rebeca Pseudo code for message server *handleElevatorMovement* where the movement policy is *up priority* policy. Contains and Next are custom functions. Pseudo code presented is for Elevator 1 in the model.

```
1    /* Movement policy: Maintain movement Policy. */
2  ...
3  /* Check if elevators are on the requested floor before moving */
4  ...
5  /* If elevator queue is not empty: */
6  /* If movement is UP and there is a request higher than the
       current floor then go up. Otherwise go down. */
7  if (Movement==1){
8    if (Next(Elv1Queue,Elv1Location,1) != -1){
9      Elv1Movement=1;
10     SendMessage(Elevator1,MoveUp);
11   }
12   else{
13     Elv1Movement=-1;
14     SendMessage(Elevator1,MoveDown);
15   }
16 /* ElseIf movement is DOWN and there is a request lower than the
       current floor then go down. Otherwise go up. */
17 else{
18   if (Next(Elv1Queue,Elv1Location,-1) != -1){
19     Elv1Movement=-1;
20     SendMessage(Elevator1,MoveDown);
21   }
22   else{
23     Elv1Movement=1;
24     SendMessage(Elevator1,MoveUp);
25   }
26   ...
```

Listing 15. Timed Rebeca pseudo code for movement policy *Maintain movement*. The pseudo code shows the algorithm for elevator 1.