Statistical Model Checking of Timed Rebeca Models

Ali Jafari[a,*], Haukur Kristinsson[a], Ehsan Khamespanah[a,b], Marjan Sirjani[a], Brynjar Magnusson[a]

[a] *Reykjavik University, School of Computer Science and CRESS*
[b] *University of Tehran, School of ECE*

Abstract

The actor-based language, Timed Rebeca, was introduced to model distributed and asynchronous systems with timing constraints and message passing communication. A toolset was developed for automated translation of Timed Rebeca models to Erlang. The translated code can be executed using a timed extension of McErlang for model checking and simulation. In this work, we added a new toolset that provides statistical model checking of Timed Rebeca models. Using statistical model checking, we are now able to verify larger models against safety properties comparing to McErlang model checking. We examine the typical case studies of elevators and ticket service to show the efficiency of statistical model checking and applicability of our toolset.

Keywords:
Statistical Model Checking, McErlang, Timed Rebeca, Performance Analysis, Real-time systems

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 [1] and then elaborated by Agha [2, 3] and Talcott [4]. 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 [5] and Timed Rebeca [6] were proposed.

Background. The *Reactive Objects Language*, *Rebeca* [7], 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 [8, 9]. Timed Rebeca [6] is proposed as an extension of the Rebeca language with time constraints and analysis support. The formal semantics of Timed Rebeca was offered using Structural Operational Semantics (SOS) rules [10].

In the first implementation of Timed Rebeca, a toolset was developed to translate Timed Rebeca models to Erlang programs [11] automatically, and McErlang [12] was used to simulate the translated Erlang

*Tel: +354 776 6603, +98 936 723 6840
Email address: ali11@ru.is (Ali Jafari)

program [6]. 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 larger 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.

In [13], 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 [14]. To improve the usability of Timed Rebeca, the language is extended to support a *list* data structure and the 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 [6], and we adjusted the implementation of the tool accordingly. As stated in [14], 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 [15]. 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.

Contributions. This paper is an extended version of the work in [13]. In the conference paper [13], we used checkpoint (user-defined) monitors and predefined monitors of McErlang for verification of safety properties. As state space explosion is an inevitable problem in model checking, for large Timed Rebeca models we face state explosion using this approach.

Statistical model checking. In this work, we provide statistical model checking of Timed Rebeca models, as an alternative approach to avoid an exhaustive exploration of the state space of the model. So, we are able to verify larger Timed Rebeca models. To this end, a new toolset is developed which is used together with the existing one for verification of Timed Rebeca models. In this approach, we run multiple simulations by McErlang, and then the mean value of correctness of the model is calculated for a given safety property. This tool is different from the simulation tool developed in the previous work. The statistical model checking approach is explained in Section 5.

Performance evaluation. In the conference paper, we used the simulation capability of McErlang for performance evaluation of Timed Rebeca models. The statistical methods are applied to the obtained data from different simulation runs in order to compute performance measures of the model, such as the mean response time for a request to be served. In this paper, we extended our approach in [13], and now we also calculate the confidence interval. This way we can indicate the accuracy of simulation results while in [13] the number of simulation runs were chosen by the modeler randomly and the confidence interval was not considered. This method is explained in Section 6.3.

To show the efficiency of our approaches in this work, we examine the elevator case study by applying statistical model checking, and computing confidence interval for simulation results. In the statistical model checking, we increase the number of floors to get a very large model, for which the model checking of McErlang is not applicable because of the state space explosion problem. Also, a new case study, ticket

service system, is analyzed in Section 7.1. The efficiency and applicability of the statistical model checking approach depends only on the size of our models. The only parameter showing the size of a model is the number of rebecs (actors) and the message passing between them. So, if we increase the number of rebecs (actors) greatly, a simple case study like ticket service can imitate a complicated system.

Applications. Since its introduction, Timed Rebeca has been used in different areas. One example is in analyzing different routing algorithms and scheduling policies in NoC (Network on Chip) designs, specifically the GALS (Globally Asynchronous Locally Synchronous) NoC [16, 17]. 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 [18], 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 [19]. 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, Timed Rebeca is an imperative actor-based modeling language. So, by using Timed 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 Timed Rebeca you are using a model-driven development approach. You can start with 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.

The authors in [20] present an approach to verify safety properties of Erlang-like, higher-order concurrent programs automatically. Following the Core Erlang [21], λActor is introduced as a prototypical functional language which is augmented with asynchronous message-passing concurrency and dynamic process creation. The authors formalize an abstract model of λActor programs, called Actor Communicating System (ACS). A tool is developed to generate an ACS from an annotated Erlang module, for which safety properties like unreachability of error program locations and mutual exclusion can be defined. This approach starts from an implemented code, while using Timed Rebeca we start from a model. The same discussion holds here as the one comparing Erlang and Timed Rebeca.

Two of the mostly used timed modeling languages are UPPAAL [22] and real-time Maude [23]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata [24], 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 differ greatly, while Timed 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 the message queue can cause state explosion in UPPAAL very quickly. The verification tools are different in Timed Rebeca and UPPAAL. Timed properties can be checked in UPPAAL while in this work we focus on checking Timed Rebeca safety properties, which is explained in Section 5.

Real-time Maude is a language accompanied with a tool for 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.

In [25], authors introduce UPPAAL SMC in which systems are represented via networks of automata. In UPPAAL SMC, each component of the system is modeled with an automaton whose clocks can evolve with various rates. To provide efficient analysis of probabilistic properties, statistical model checking is used as a technique for fully stochastic models. The work supports modeling and performance analysis of systems with continuous time behaviors and dynamical features. The modeling languages used in Timed Rebeca

and UPPAAL SMC are different, while Timed Rebeca has a Java-like syntax, UPPAAL uses automata. In UPPAAL SMC time is continuous, but in Timed Rebeca time is discrete. In this work, timed performance and functional properties are supported, but in UPPAAL SMC probabilistic performance properties are validated.

There are some works on safety critical real-time Java programs [26, 27] and WCET analysis of Java Bytecode-based programs [28, 29]. A new approach is presented in [26] for schedulability analysis of Safety Critical Hard Real-time Java programs. The approach is based on a translation of programs, written in the Safety Critical Java (SCJ) [30], to timed automata models which are verified by the UPPAAL model checker. In this approach, worst case execution time (WCET) calculation and schedulability analysis are performed to verify that deadline misses never occur. The authors in [28] present a tool for statically determining the WCET of Java Bytecode-based programs. In this approach, the Java program, the JVM, and the hardware are modeled as Networks of Timed Automata (NTA) and given to the UPPAAL model checking tool. While the above works only support schedulability analysis of Java programs, verification of any safety property will be possible in Timed Rebeca if the property can be defined by a checkpoint function. Additionally, performance evaluation of Timed Rebeca models is also provided in this paper. Moreover, the modeling paradigm is different in Timed Rebeca and Real-time Java.

Regarding other analysis techniques and tools for Timed Rebeca, a new approach was proposed for schedulability and deadlock freedom analysis of Timed Rebeca models in [31]. The authors proposed the notion of Floating Time Transition System (FTTS) for which the formal definition is presented. The authors proved a bisimulation relation between FTTS and the transition system derived from the SOS rules of Timed Rebeca in [6]. They developed a verification tool based on FTTS and integrated it in the Afra toolset [32]. In this work, the verification of Timed Rebeca models is restricted to deadlock freedom and schedulability analysis, and the performance evaluation of Timed Rebeca models is not supported.

Another work on verification of Timed Rebeca models is presented in [33]. In this paper, authors defined an executable formal semantics for Timed Rebeca in Real-Time Maude. This enables a wide range of formal analysis methods for Timed Rebeca models, including simulation, reachability analysis, and both timed and untimed temporal logic model checking. The presented semantics executes all deterministic instantaneous statements in a message server in a single "atomic" step. This approach significantly reduces the number of interleavings and drastically improves the performance of model checking analyses. In addition, in this work, dynamic topology and dynamic creation in Timed Rebeca models is supported. Although the proposed approach covers analysis of an extended version of Timed Rebeca, there is no way for using high-level user defined functions in the models. These functions must be defined in the Maude language which requires expertise in rewriting logic. The direct model checking approach of TCTL properties for Timed Rebeca models in [34] suffers from the same limitations; however, it verifies majority of TCTL formulas in $O(n^2 \cdot |\Phi|)$ for a given formula Φ. This order is the most efficient algorithm for verification of TCTL formulas in discrete time systems which is the same as the order of the verification of CTL formulas.

Paper Organization. The rest of the paper is organized as follows. Section 2 gives a brief introduction to Timed Rebeca. Considering the Timed Rebeca language presented in [6], 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 the 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 explains statistical model checking of larger Timed Rebeca models against safety properties. Section 6 describes the simulation of Timed Rebeca models using McErlang. The result is a dataset including useful information about system behavior to which different analysis methods can be applied. To show the result's precision, we calculate the confidence interval for performance measures under study. In Section 7, we apply all methods proposed in the previous sections to the typical examples of elevator and ticket service. Finally, Section 8 concludes the paper.

2. Timed Rebeca

Timed Rebeca is proposed as an extension to Rebeca, for modeling and verification of real-time distributed systems [6]. Rebeca [7, 35] is an actor-based language for modeling and verifications of reactive systems with asynchronous communication among actors. Each actor has an unbounded buffer, called the message queue, for its arriving messages. Each actor takes a message, that can be considered as an event from the top of its message queue, and executes its corresponding message server (also called a method).

In Timed Rebeca, each actor (also called as rebec) has its own local clock, but there is also a notion of global time based on synchronized distributed clocks of all the rebecs. Instead of a message queue for each rebec, there exists a bag containing all the messages sent for each rebec. Messages that are sent to a rebec are put in its message bag together with their arrival time (called their time tag), and their deadline. Methods are executed atomically, but the 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 as follows.

- 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)*, 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)*, 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. Note that *After* primitive can also be used to model periodic events. If we send a message in a loop with *After(t)*, this will cause having the message in the message queue every t units of time. In Timed Rebeca, loops are modeled by sending a message to self.

The scheduler decides which message is to be executed next based on the time tags of the messages. The time tag of a message is the value of local clock of the sender rebec when the message was sent, added to the value of the argument of the *after* if the message is sent with an *after*. The scheduler takes a message from the message bag, executes the corresponding message server atomically, and then takes another message. Every time the scheduler takes a message for execution, it chooses a message with the least time tag. Before the execution of the corresponding method starts, the local time of the receiver rebec is set to the maximum value between its current time and the time tag of the message [6].

An example of a Timed Rebeca model is shown in Listing 1. This is a model of a ticket service system. In the main part, the rebecs are instantiated from the reactive classes. For each rebec, its known rebecs are specified as arguments, e.g. rebecs `ts1` and `ts2` are the known rebecs of rebec `agent` (Line 49). The initial values of the state variables can be specified as arguments in the rebec instantiation (empty parentheses in Line 49 can be used for this purpose, otherwise the default values are used). For example,"Agent agent(ts1, ts2):(10, false, 2)" creates an agent and the values of its state variables `attemptCount`, `ticketIssued` and `token` are initialized to 10, false and 2, respectively. A reactive class has an argument of type integer denoting a user-specified upper bound for its queue size (Agent(3) in Line 4). This is necessary to prevent state space explosion in model checking.

The model in Listing 1 consists of two reactive classes: `Agent` and `TicketService`. The agent `a` starts by sending a message to the first ticket service `ts1` and requesting a ticket (Line 13). The message has a deadline of `requestDeadline` time units. When the message is received by the ticket service `ts1`, it issues the ticket after `serviceTime1` or `serviceTime2` units of time (Line 42-44). The issuing process is performed by sending a message back to the agent `a`. After requesting a ticket to `ts1`, agent `a` sends a message to itself after `checkIssuedPeriod` time units (Line 14). This message checks whether the ticket has been issued or not. If the ticket is issued, the model continues to the next customer and request a new ticket after `newRequestPeriod` time units (Line 26). If the ticket was not issued by `ts1`, agent `a` immediately sends a message to the second ticket service `ts2` (Line 21). This scenario is repeated recurrently.

```
1  env int requestDeadline, checkIssuedPeriod,
       retryRequestPeriod;
2  env int newRequestPeriod, serviceTime1,
       serviceTime2;
3  env int maxIssued; // maximum number of requests
4  reactiveclass Agent(3) {
5    knownrebecs { TicketService ts1; TicketService
         ts2; }
6    statevars { int attemptCount; boolean
         ticketIssued; int token; }
7    msgsrv initial() { self.findTicket(ts1); }
8
9    msgsrv findTicket(TicketService ts) {
10     attemptCount = attemptCount + 1;
11     token = token + 1;
12   if(token <= maxIssued) {
13     ts.requestTicket(token)
           deadline(requestDeadline);
14     self.checkTicket() after(checkIssuedPeriod);}
15   }
16
17   msgsrv ticketIssued(int tok) { if (token == tok)
         ticketIssued = true; }
18
19   msgsrv checkTicket() {
20     if (!ticketIssued && attemptCount == 1 &&
           token < maxIssued+1) {
21       self.findTicket(ts2);
22     } else if (!ticketIssued && attemptCount == 2
           && token < maxIssued+1) {
23       self.retry() after(retryRequestPeriod);
24     } else if (ticketIssued && token <
           maxIssued+1) {
25       ticketIssued = false;
26       self.retry() after(newRequestPeriod);
27     }
28   }
29
30   msgsrv retry() {
31     attemptCount = 0;
32     self.findTicket(ts1);
33   }
34 }
35
36 reactiveclass TicketService(3) {
37   knownrebecs { Agent agent; }
38   statevars { }
39   msgsrv initial() { }
40
41   msgsrv requestTicket(int token) {
42     int wait = ?(serviceTime1,serviceTime2);
43     delay(wait);
44     agent.ticketIssued(token);
45   }
46 }
47
48 main {
49   Agent agent(ts1, ts2):();
50   TicketService ts1(agent):();
51   TicketService ts2(agent):();
52 }
```

Listing 1: Timed Rebeca model - Ticket service system.

3. Mapping Timed Rebeca Models to Erlang Programs

The McErlang is a model checking and simulation tool for Erlang programs. In [14], authors introduced a timed semantics of Erlang in McErlang with a close collaboration with Timed Rebeca team. The new timed semantics provides the model checking of Erlang programs with timing features. When the first version of Timed Rebeca was proposed in [6], McErlang did not provide the timed semantics for Erlang programs. In this section, we explain a new mapping algorithm for Timed Rebeca models to Erlang programs while conforming to the new timed features of McErlang. Since McErlang is used as the backend model checker and the simulation tool, this mapping is necessary. We also explain new features added to the 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 3.4.

3.1. Handling Time in Erlang

Here, we briefly explain timed Erlang semantics introduced in [14] 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. If a message matches any of the patterns, e.g. $Pattern_j$, and the corresponding guard, $Guard_j$, evaluates to true, the message is removed from the mailbox and evaluation continues with expression $Expr_j$.

The oldest message in the process mailbox is evaluated to be matched against the patterns according to the above procedure. If no pattern and guard match this message, the same procedure continues with the second oldest message, and so on. If no pattern is matched, the process waits for at least `TimeoutValue` milliseconds to receive a matching message. This is the *minimum* amount of time that a timer elapses until the timeout happens. If the timeout occurs, the expression `TimeoutExpression` is evaluated. A zero deadline means, if no matching message is in the mailbox, the timeout can happen immediately. The atom `infinity` may be used as a time deadline to show that the timeout never happens.

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

Listing 2: Erlang syntax of a receive with timeout.

3.2. Timed Semantics of Erlang in McErlang

The main changes made to McErlang to implement a timed semantics of Erlang are to record the current time in the state representation of a running program, and to modify the behaviour of the receive statement in the model checker so that the current time is considered when timeouts are handled [14].

In Listing 2, there is no guarantee for exactly when the timeout happens after a timer has elapsed `TimeoutValue` milliseconds. In the timed semantics of Erlang, it is possible to specify the urgency of a state with the function `mce_erl:urgent(MaximumWait)`. The parameter `MaximumWait` specifies the *maximum* number of milliseconds the process can remain in the current state, if it has transitions enabled. As an example consider the code in Listing 3, a process is spawned and waits between 1000 and 1500 milliseconds for a message to arrive before timing out. In this example, we force the timeout to happen before 1500 milliseconds if the process does not receive a message.

```
spawn (fun () ->
    mce_erl : urgent (1500),
    receive Msg -> ok
    after 1000 -> bad
    end
end)
```

Listing 3: An Erlang code with urgency construct implemented in McErlang.

In McErlang with timed Erlang 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.

3.3. Adapting Timed Rebeca with Timed Semantics of McErlang

The timed version of McErlang proposed in [14] makes the formal verification of timed programs written in Erlang programming language possible. In timed semantics, timed actions, i.e. actions with a timeout clause, are ordered based on the timeout value while untimed actions, i.e. actions without a timeout clause, are executed infinitely fast.

In the Timed Rebeca language, timed behaviors are defined by using timing primitives of *after*, *delay*, and *deadline*. The execution order of messages are specified based on the values of these primitives. In this section, we explain the new mapping of a Timed Rebeca model to an Erlang program according to the timed semantics of Erlang in McErlang. There are two main points to consider regarding the new mapping. Firstly, the mapping algorithm of timing features in Timed Rebeca to Erlang must be changed according to the new timed features of McErlang like timestamps and the urgency construct. Secondly, the 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 the system clock [6]. 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 [6], but with a few and very important differences in the implementation. We use clock references accessible from API `mce_erl_time` to map timed actions from Timed Rebeca to Erlang. The main difference is that in the new version we use the simulation/model time and not the real system time (like when a real execution of the program is in order).

An ordinary message send in Timed Rebeca, i.e. message send without `after` primitive, is translated to a regular message send in Erlang as shown in Listing 4. Instead of tagging the message with the local time of the sender, as we did in our previous mapping, 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. The default value for deadline is `inf` (standing for infinity) which denotes no deadline.

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 expired, this condition $deadline + was(ref) < nowRef()$ is satisfied.

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

Listing 4: Pseudo Erlang code for a message send in Timed Rebeca

In Timed Rebeca semantics, a message with the `after(Timeunits)` statement is put in the message bag of the receiver, and it can not be taken from the bag before the specified time, i.e. `Timeunits` milliseconds, has elapsed. In mapping to Erlang, a function is spawned and waits for `Timeunits` milliseconds 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 5.

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

Listing 5: Pseudo Erlang code for a message send with after primitive in Timed Rebeca

In Timed Rebeca, the `delay(Timeunits)` statement makes the local time of a rebec advance for the specified amount of time (`Timeunits` milliseconds). In Erlang, the delay is translated to the receive statement including just a timeout value as shown in Listing 6. Since there is no pattern in the receive statement, the timeout clause (after clause) will be executed after the specified time (`Timeunits` milliseconds) immitating the delay statement in Timed Rebeca. As stated in [14], 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.

```
timedelay(Timeunits) ->
  % McErlang Urgent Delay
  urgent(Timeunits),
  % Delay by Timeunits
  receive
    after (Timeunits) -> ok
  end.
```

Listing 6: Pseudo Erlang code for a delay statement in Timed Rebeca

Performing timed and untimed actions in the correct sequence. In Timed Rebeca, the execution order of messages is specified with respect to the values of timing primitives `delay` and `after`. In the previous paragraph, we explained how timing primitives in Timed Rebeca are translated to Erlang code. We also explained how a message deadline in Timed Rebeca can be handled using timestamps in McErlang. To execute messages in the correct order in Erlang according to the Timed Rebeca semantics, we should take into account more considerations in Erlang:

- actions without timeout clause (equivalent to messages without after in Timed Rebeca) should be executed infinitely fast (immediately).
- actions with timeout clause (equivalent to 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 the timed extension in McErlang, we can change the way in which timed (with timeout) and untimed (without timeout) actions are treated using the function `mce_erl:urgent (MaximumWait)`. 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.

3.4. 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 a list data structure, capability of calling custom functions from the Erlang language, and checkpoints. Table 1 shows the syntax of the extensions and their abstract mapping to Erlang.

Checkpoint functions can be used in both simulation and model checking. They are considered as markers in the code that indicate important events. Checkpoints are also used to expose the value of variables in a Timed Rebeca model to McErlang. For simulation, a checkpoint is translated to an Erlang function, and for model checking a checkpoint is translated to a probe in Erlang.

A checkpoint has two mandatory arguments: a *label* and at least one *term*. The label is an arbitrary name which is defined by the modeler and is used to refer to the checkpoint. Note that every piece of data of any type is called a term in Erlang. So, all variables in a Timed Rebeca model are translated to terms. The terms in a checkpoint are variables that are added to the checkpoint function as its arguments. The value of terms can be retrieved during simulation or model checking in McErlang.

Another extension in Timed Rebeca language is the ability of calling custom functions in Erlang. A modeler can define a function in Erlang and then call it 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 a Timed Rebeca model. Using this extension, the Timed Rebeca language has the same programming power as the Erlang language.

This way, the applications in which implementing buffers or queues is essential, like schedulers, can be modeled using the list data structure in Timed Rebeca language. The elements of a list are of type integer. They can be defined inside message servers as a local variable or as a state variable. In order to facilitate working with the list data structure, the following functions are defined: `remove(intValue)`, `size()`, `first()`, `last()`, `insert(intValue)`. Functions `remove(intValue)` removes the integer value of `intValue` from the list and function `insert(intValue)` inserts the value of `intValue` at the end of the list. Functions `first()` and `last()` return the first and the last elements of the list, respectively.

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

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

4. Model Checking of Timed Rebeca Models Using McErlang Monitors

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 the corresponding Erlang program of a Timed Rebeca model in order to verify safety properties of the Timed Rebeca model. 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

McErlang allows safety monitors to access both states of the 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 a Timed Rebeca model are defined based on the values of its variables. This is why we added the checkpoint construct to Timed Rebeca language. A checkpoint in a Timed Rebeca model can include the values of specific variables. As we discussed in Section 3.4, the value of intended variables are passed as arguments to checkpoints. Also, the occurrence of interesting events can be specified using checkpoints. While doing model checking, in the corresponding Erlang program, checkpoints are translated to probes, which are accessible by safety monitors in McErlang.

4.2. Defining Safety Monitors

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

Deadlock Monitor. Detecting deadlock in non-terminating systems is essential. The predefined monitor in Listing 7 can be used to investigate the deadlock of Timed Rebeca models. As lines 13 to 20 of Listing 7 show, deadlock is detected by checking the status of processes. If the status of all the processes is marked as *blocked*, deadlock is reported.

Maximum Queue Length Monitor. Although in theory message queues are unbounded in Timed Rebeca, 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 a queue overflow error. 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 8 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*.

```
monitorType() -> safety.

init(State) -> {ok,State}.

stateChange(State,MonState,_) ->
  case is_deadlocked(State) of
    true -> deadlock;
    false -> {ok, MonState}
  end.

is_deadlocked(State) ->
  State#state.ether =:= [] andalso
    case mce_erl:allProcesses(State) of
      [] -> false;
      Processes ->
       case mce_utils:find(fun (P) ->
        P#process.status =/= blocked end,
         Processes) of
           {ok, _} -> false;
            no -> true
         end
       end.
```

Listing 7: McErlang - Deadlock monitor

```
monitorType() -> safety.

init(MaxQueueSize) -> {ok,MaxQueueSize}.

stateChange(State, MaxQueueSize, _) ->
    case mce_utils:find
      (fun (P) -> length(P#process.queue) > MaxQueueSize end,
       mce_erl:allProcesses(State)) of
      {ok, P} -> {exceeds, P};
      _ -> {ok, MaxQueueSize}
    end.
```

Listing 8: McErlang - MaxQueue monitor

Checkpoint Monitor. The purpose of defining checkpoints in a Timed Rebeca model is the verification of safety properties using McErlang. Generally, a safety monitor is a function which is called after the creation of each state of the model. The monitor returns `satisfied` if the state satisfies the specified conditions, otherwise it returns `violation`. If a safety monitor is defined based on the information provided by checkpoints (which is available for McErlang from the translated Erlang program), the monitor is called *checkpoint monitor*. This type of monitor should be implemented by a modeler, while the previously mentioned monitors are available in McErlang.

Listing 9 shows a template for checkpoint monitors. Any user-defined function can be used in the template. For example, we define the function `checkLabelCheckPoint` and use it in the monitor (Line 13), in which actions (obtained from the function `actions`) and a checkpoint label is used as arguments. If a checkpoint with the label `CheckpointLabel` occurs in a state, the monitor halts with a `violation`. If the verification terminates without any violation, it is guaranteed that the checkpoint never happens in any paths of the state space.

We also developed some other functions to make it easier for a modeler to write the safety specifications in a monitor. The signature of each function and a brief explanation are listed below. The implementation of these functions are accessible from [32].

- Checking if a message server is dropped because the deadline is missed. In the following function, the term is equal to the message server name.
  - *checkDropMsgsrv(Actions, CheckpointTerm)*
- Checking if a checkpoint with the specified label occurs.
  - *checkLabelCheckPoint(Actions, CheckPointLabel)*

- Compare the checkpoint term with an integer or boolean. In the following functions, *MaxValue/MinValue* is the maximum/minimum value for the specified term. In the function *checkTermValue*, the value of the specified term is checked to be equal to *value*.
  - *checkTermMaxValue(Actions, CheckPointLabel, CheckpointTerm, MaxValue)*
  - *checkTermMinValue(Actions, CheckPointLabel, CheckpointTerm, MinValue)*
  - *checkTermValue(Actions, CheckPointLabel, CheckpointTerm, value)*

```
monitorType() -> safety.

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

stateChange(_,satisfied,Stack) ->
  % Monitor Setup
  % Usage: checkpoint(Label,Term);
  CheckpointLabel = checkpoint_label, % Not needed when using function checkDropMsgsrv.
  CheckpointTerm = checkoint_term,   % Not applicable when using function checkLabelCheckPoint.

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

Listing 9: A template (pseudo code) for checkpoint monitors which is used by McErlang

5. Statistical Model Checking of Timed Rebeca Models

In the previous section we showed that how safety monitors can be defined for the corresponding Erlang program of a Timed Rebeca model, using the checkpoints of the Timed Rebeca model. So, the McErlang can be used as a back-end model checker for the verification of safety properties of the Timed Rebeca model. The major limiting factor in applying model checking for verification of real world systems is the huge amount of space and time required to store and explore the state space. Alternatively, statistical model checking can be used and it does not have the problem of state explosion. Statistical model checking does not guarantee the correctness of systems, however, it provides an approximation of correctness. The main idea behind this approach is analyzing N different independent random executions of a given system $Z_1, Z_2, \cdots, Z_N$ (i.e. independent samples of random variable Z) to approximate the correctness of the system. If $Z_1, Z_2, \cdots, Z_N$ are identically distributed with mean μ_Z, there is a technique that approximates the value of μ_Z by $\tilde{\mu}_Z = (Z_1 + Z_2 + \cdots + Z_N)/N$. This way, $\tilde{\mu}_z$ is computed as (ϵ, δ)-approximation of μ_Z. We say $\tilde{\mu}_Z$ is an (ϵ, δ)-approximation of μ_Z if $Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$. Here, ϵ is the error value and δ is the confidence value of the approximated value of $\tilde{\mu}_Z$.

In case of statistical model checking, for a given ϵ and δ, we have to provide an upper bound N as the number of simulation traces which are required to compute (ϵ, δ)-approximation of the correctness of the system. Based on the *zero-one estimator theorem*, if the range of the values of random variable Z is in $[0, 1]$ by $N > 4\ln(2/\delta)/\mu_Z\epsilon^2$ number of samples, the value of μ_Z is approximated by $\tilde{\mu}_Z$ for error value ϵ and confidence interval $1 - \delta$ [36]. But, applying the zero-one estimator theorem encounters a difficulty which is the fact that N depends on $1/\mu_Z$, the inverse of the value that one intends to approximate. In addition, the factor of $1/\mu_Z\epsilon^2$ makes the value of N unnecessarily large. A more practical approach for this problem is proposed by Dagum et el. in [37] for computing N, called the *generalized zero-one estimator theorem*. Based on this work, Grosu et al. in [38] presented an optimal approximation algorithm to provide N, as shown below.

$$N = \frac{\Upsilon_2 \times \epsilon}{\tilde{\mu}_Z}$$

$$\Upsilon_2 = 2(1 + \sqrt{\epsilon})(1 + 2\sqrt{\epsilon})(1 + \frac{ln3/2}{ln2/\delta})\Upsilon$$

$$\Upsilon = \frac{4}{\epsilon^2}(e - 2)ln(2/\delta)$$

In this formula, the value of N depends on the value of $\tilde{\mu}_Z$ which is the raw estimate of μ_Z. Here $\tilde{\mu}_Z = (1+(1+\epsilon)\Upsilon)/N'$, where N' is the number of traces which are needed to be analyzed until at least $\lfloor 1+(1+\epsilon)\Upsilon \rfloor$ of them satisfies the given property. For the raw estimation of μ_Z, values of $\min\{1/2, \sqrt{\epsilon}\}$ and $\delta/3$ are used to compute the value of Υ.

Now, we have to specify the subset of formulas which can be model checked by statistical model checking approach of this paper. As shown in [39], formulas with unbounded until operators (and nested until operators) can be model checked using statistical model checking. As a result, the approach of this paper works for formulas with until operators which are both safety and monitor-based LTL properties. So, in a nutshell, Timed Rebeca models can be verified against *safety properties*, using predefined monitors like *Deadlock Monitor* and *Maximum Queue Length Monitor*, and checkpoint monitors.

As a final step of developing a statistical model checker, we have to implement the above algorithm to calculate an approximation of the mean value of correctness. As Figure 1 shows, the *statistical model checking (SMC)* component works with the present tool, which was developed in [13]. The *simulation wrapper* component is employed to generate needed simulation traces for the SMC component. Figure 1 demonstrates the analysis tool-set which includes the SMC component and the performance evaluation tool. In the following section, we describe the architecture of the performance evaluation tool.

6. Performance Evaluation of Timed Rebeca Models

In addition to its model checking facilities, McErlang provides facilities for 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 simulation run. In each simulation run, we choose the simulation time long enough to reach the steady state of the system. As we model reactive systems, which generally show recurrent behavior, having long simulation runs can guarantee reaching the steady state (if there is any). 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.

Since the resulting information of a 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 [40].

The non-determinism caused by concurrency is resolved by the scheduler of McErlang. McErlang scheduler selects the process that must be executed in the next step based on the uniform distribution. Obviously, resolving non-determinism using uniform distribution affects the performance analysis results. In this work, we follow the community that uses simulation-based and statistical model checking approaches for performance evaluation of concurrent systems.

6.1. Performance Evaluation Tool-set

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

- *translator*: for translating Timed Rebeca models to Erlang programs.
- *trace analyzer*: to apply statistical analysis methods to stored information. Different analysis techniques are implemented in this component.
- *simulation wrapper*: it 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 *simulation wrapper* component sends Timed Rebeca models to the *translator* component to be translated to an Erlang program. The translated Erlang program is sent to McErlang for

simulation. The generated data from the simulation is sent to the *simulation wrapper* component at runtime. The *simulation wrapper* component categorizes the simulation data of different simulation runs in a way to be used by *trace analyzer*.

We implement two different analysis techniques in the component *trace analyzer*, called *checkpoint analysis* and *paired-checkpoint analysis*, to provide performance evaluation of Timed Rebeca models. In the next section, we explain how information provided by checkpoints can be used in *trace analyzer* to achieve performance measures of interest.

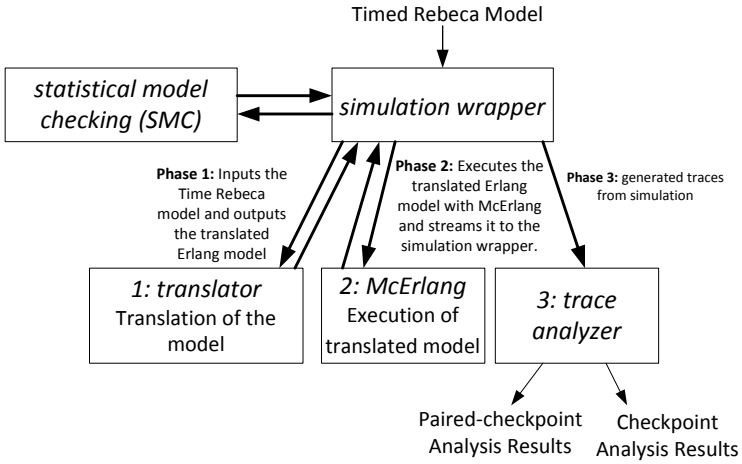

Figure 1: Architecture of analysis tool-set.

6.2. Checkpoints analysis in Simulation

As we discussed in Section 3.4, 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 the occurrence of events. We analyse models based on the 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), the label, the time of observing the checkpoint and the name of the rebec including the checkpoint are stored for performance evaluation purposes.

```
1  env int requestDeadline, checkIssuedPeriod,
       retryRequestPeriod;
2  env int newRequestPeriod, serviceTime1,
       serviceTime2;
3  reactiveclass Agent(3) {
4    ...
5    msgsrv findTicket(TicketService ts) {
6      attemptCount = attemptCount + 1;
7      token = token + 1;
8      checkpoint(requestStart,token);
9      ts.requestTicket(token)
           deadline(requestDeadline);
10     self.checkTicket() after(checkIssuedPeriod);
11   }
12
13   msgsrv ticketIssued(int tok) {
14     if (token == tok) { ticketIssued = true;
15       checkpoint(ticketIssued,tok);}
16     else { checkpoint(ticketNotIssued,tok); }
17     }
18
19     msgsrv checkTicket() { ... }
20     msgsrv retry() { ... }
21 }
22
23 reactiveclass TicketService(3) {
24   ...
25   msgsrv requestTicket(int token) { ... }
26 }
27 main { ... }
```

Listing 10: Timed Rebeca model - Ticket service system

To illustrate the role of checkpoints in the performance evaluation of Timed Rebeca models, we add some checkpoints to the ticket service model in Listing 1, as shown in Listing 10. For the sake of simplicity, we

keep the message servers to which checkpoints are added and delete other message servers. The performance of our model is influenced by the timing variables. Simulation of the Timed Rebeca model reveals the effect of these variables in the average response time.

Three different checkpoints are defined in order to collect the required data for performance evaluation of the model. These checkpoints store data about when the request is sent to the ticket service (line 8), when the ticket is received by the agent *a*, i.e. the ticket is issued (line 15), and whether the ticket is not issued (line 16). We are able to define as many checkpoints as needed depending on the safety properties and the performance measures we are interested in. In these checkpoints we should provide the value of variables which are needed for the intended analysis.

In the following subsections, we explain how the performance evaluation of Timed Rebeca models is performed.

6.2.1. Paired-checkpoint analysis

The paired-checkpoint method is implemented in the *trace analyzer* tool. In paired-checkpoint analysis, two checkpoints are grouped together. The modeler specifies paired checkpoints with the use of labels when running the tool. The elapsed time between observing two paired checkpoints is important and can show different performance measures. There is a command in our tool that enables the modeler to specify paired checkpoints. For example, the starting checkpoint in line 8 (labelled by *requestStart*) shows that the request is sent to the ticket service and the ending checkpoint in line 15 (labelled by *ticketIssued*) represents that the ticket was issued. Consequently, the passed time between the occurrence of these two checkpoints is considered as the response time of the issued ticket.

6.2.2. Checkpoint analysis

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 ticket service system, we are interested in knowing how many tickets are issued by ticket service *ts1* and how many of them are issued by ticket service *ts2*. This information is available in the simulation results by defining the checkpoint with label *ticketIssued* in the model. When a ticket is issued at run-time, the time of occurrence and the name of rebec including the checkpoint are stored in the simulation results.

6.3. Confidence Interval

While using statistical methods, there is an important question of how precise the results are. In our previous work [13], the number of simulation runs are selected by the user without considering any criteria for the measurement's precision. Here, we calculate the confidence interval for simulation results to indicate their accuracy.

The confidence interval shows how close our measurement is to the original value if the experiment is repeated. The margin of the error is calculated from the following formula.

$$Z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$$

where σ is the standard deviation of the intended phenomenon (like response time), n is the sample size, α is the confidence level and $Z_{\alpha/2}$ is the confidence coefficient. The most commonly used confidence levels are 90%, 95% and 99%. Suppose the confidence level is 95% ($\alpha = 0.95$), to find the value of $Z_{\alpha/2}$ the z table is checked for the value of $0.95/2 = 0.475$ [41]. In the z table, the intersection of row 1.9 and the column of 0.06 shows a cell with the value of 0.475 (or the closet value to 0.475), so $Z_{0.475}$ equals to 1.96.

The confidence interval is obtained from the following formula, where $\bar{x}$ is the mean value of the intended phenomenon (like response time).

$$\bar{x} \pm Z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$$

7. Case Studies and Experimental Results

In this section, we present two case studies to illustrate the applicability of the approaches of this work. For each case study, after intuitive description of the model using an event graph [42], the detailed description of the Timed Rebeca model is presented. We use an event graph to give a highly abstracted view of events and their causality relations. Event graphs are widely used for the explanation of event-based models. In this graph, the vertices represent events in a system and the edges represent the causality relation between events (vertices). Additionally, we add a label below each vertex that shows in which reactive class the event occurs. Edges can be conditional (thick edge), mandatory (thin edge) or marking an initial event (jagged edge). Model checking, statistical model checking and performance evaluation are applied for the case studies. In model checking using McErlang, we have limitations on the size of the models to avoid state space explosion. In statistical model checking, we are able to check larger models, and increment the size of the models greatly.

7.1. Ticket Service System

Our first case study is the ticket service system, which is shown in Listing 1. As we already described the details of this model in Section 2, here, we only demonstrate the event graph of Ticket Service model in Figure 2. As shown in Figure 2, initially the message server `initial` in the rebec `agent` sends a message to itself that triggers the event (the message server) `findTicket`. Execution of this event causes sending a message to the rebec `TicketService` which raises the event `requestTicket`. After a number of trials (which is modeled by causality relation among `findTicket`, `checkTicket`, and `retry`), the event `ticketIssued` is raised to inform that a ticket is issued.

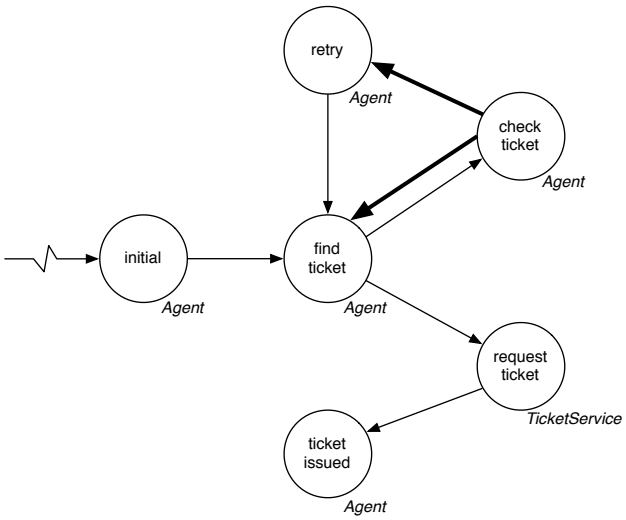

Figure 2: Event graph of the ticket service model.

7.1.1. Model Checking using McErlang Monitors

The model in Listing 1 is revised to be usable in monitor-based model checking. A variable is added to the model to restrict the number of ticket requests that are sent to `ts1` and the `ts2`. The maximum number of ticket requests is set to seven. This modification is necessary to avoid state space explosion.

We are interested in checking whether a ticket is issued in the system. So, we add a checkpoint with label `ticketIssued` to the model where a ticket is issued (refer to Listing 10). The checkpoint monitor shown in Listing 11 is used for safety verification. The property is satisfied if a ticket is issued. This property verification is performed by using the predefined function `checkLabelCheckPoint`, explained in Section 4.2.

```
monitorType() -> safety.

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

stateChange(_,satisfied,Stack) ->
  CheckpointLabel = ticketissued,
  Actions = actions(Stack),

  checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 11: The checkpoint monitor for checking whether a ticket is issued.

The results of model checking of the Ticket Service system using McErlang is shown in Table 2. We considered different settings for the model each of which has different values for variables. As shown in the table, there is no tickets issued in the first three settings.

Setting	Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2	Max Ticket Requests	Result
1	2	1	1	1	3	7	7	**violation (170737 states)**
2	2	1	1	1	4	7	7	**violation (199709 states)**
3	2	2	1	1	4	7	7	**violation (153377 states)**
4	2	2	1	1	3	7	7	**satisfied (6248 states)**
5	2	2	1	1	2	7	7	**satisfied (4398 states)**
6	2	3	1	1	2	7	7	**satisfied (4311 states)**
7	2	4	1	1	2	7	7	**satisfied (4311 states)**

Table 2: Verification results for ticket service. Property is satisfied if at least one ticket is issued.

7.1.2. Statistical Model Checking

We verify the ticket service model shown in Listing 10, with a huge number of ticket requests in the model. We aim at chcking the safety property of "at least one ticket is issued". For each setting in Table 2, we run the statistical model checking (SMC) component with different error values and confidence values. Table 3 shows the results for setting 4. The results for settings 5, 6, and 7 are the same for setting 4. Table 4 shows the verification results for setting 1. Settings 2 and 3 have the same results as setting 1, because no ticket is issued in these settings.

For a given safety property, we run as many simulations as needed to get $N_{ct} = \lfloor 1 + (1+\epsilon)\Upsilon \rfloor$ number of traces that satisfy the safety property (refer to Section 5 for Υ formula). In each simulation run, a random trace is explored to check the safety property. The mean value of correctness of the property is defined as $\tilde{\mu}_Z = N_{ct}/N'$, where N' is the total number of simulation runs (explored traces).

Considering the error value and the confidence value of the first experiment of Table 3, $N_{ct} = 289$. We run as many simulations as needed to get 289 traces that satisfy the defined property. The total simulation runs (traces) for this experiment is 289 ($N' = 289$), meaning all traces satisfied the property. So, in this experiment the mean value of correctness is one, $\tilde{\mu}_Z = 1$. More accurately, we obtain an (ϵ, δ)-approximation of the mean value of correctness where $Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$, μ_Z is the real mean value of correctness. For the first experiment of Table 3, $Pr[|\mu_Z - 1| < 0.05] \geq 0.95$.

As we described before, we run as many simulations as needed until N_{ct} traces satisfies the property. If the model never satisfies the property, the simulation should continue forever to find N_{ct} satisfied traces.

Experiment#	Number of traces to be satisfied	Total number of traces (N')	Error value (ϵ)	Confidence value (δ)	Mean value of correctness ($\tilde{\mu}_Z$)
1	289	289	0.05	0.05	1
2	203	203	0.1	0.01	1

Table 3: Statistical model checking results for ticket service model with parameters equal to setting 4. The mean value of correctness is calculated for safety property of "at least one ticket is issued".

To avoid this situation, in the implementation of SMC component we stop simulation (generating traces) if the first N_{ct} traces does not satisfy the property. This case happens for setting 1, so the mean value of correctness equals zero as presented in Table 4.

Experiment#	Number of traces to be satisfied	Total number of traces (N')	Error value (ϵ)	Confidence value (δ)	Mean value of correctness ($\tilde{\mu}_Z$)
1	289	289	0.05	0.05	0
2	203	203	0.1	0.01	0

Table 4: Statistical model checking results for ticket service model with parameters equal to setting 1. The mean value of correctness is calculated for safety property of "at least one ticket is issued".

We are also able to verify the model with more actors (rebecs) for which the model checking approach based on McErlang monitors explodes. For example, the number of agents and ticket services is increased to four and nine, respectively. We check the safety property of "at least one ticket is issued" for this model. The mean value of correctness equals one for the following parameters: $\epsilon = 0.05$, $\delta = 0.05$. We use a different setting which is not listed in Table 2. In this setting, the values of variables (from left to right in Table 2) equal 3, 3, 2, 2, 4, 7. So, large ticket service models can be verified against safety properties using statistical model checking.

7.1.3. Performance Evaluation

In the simulation, the limitation of the number of ticket requests is removed from the model. Considering the verification results, we know that some tickets are issued in settings 4, 5, 6, and 7. We use the methods introduced in Section 6 to evaluate the performance evaluation of different settings of the model. For each setting, the mean response time to ticket requests are calculated using the paired-checkpoint analysis. The simulation results are shown in Table 5. Each setting is simulated 5 times, each for 200 seconds. The error margin is calculated for different confidence levels of 99%, 95% and 90%. The simulation results show that setting 7 has the most issued tickets, around 50% of all requests. In settings 4, 5, and 6, the 0.1%, 10%, and 22% of all ticket requests are successfully served, respectively.

Setting	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Starting checkpoints	Checkpoint pairs
4	3.0	3.0	3.0	0	3.0	3.0	3.0	519350	614
5	2.1±0.00114	2.1±0.000864	2.1±0.00073	0.1	2	3.0	2.0	511709	51476
6	4.0	4.0	4.0	0	4.0	4.0	4.0	363891	81585
7	3.0	3.0	3.0	0	3.0	3.0	3.0	573551	286948

Table 5: Paired-checkpoint evaluation for Ticket Service. The specification of settings are available in Table 2 where all settings guarantee that some tickets are issued. SD, WCT, BCT denote standard deviation, worst case time and best case time, respectively.

Figures 3 and 4 show the distribution of issued tickets between ticket services for settings 4, 5, 6, and 7. The results are obtained by using checkpoint analysis method. The results show that almost 66% of tickets are issued by ticket service 1 in setting 7. The similar distribution trend exists for setting 6. Therefore, we are able to reason about the model behavior with different settings.

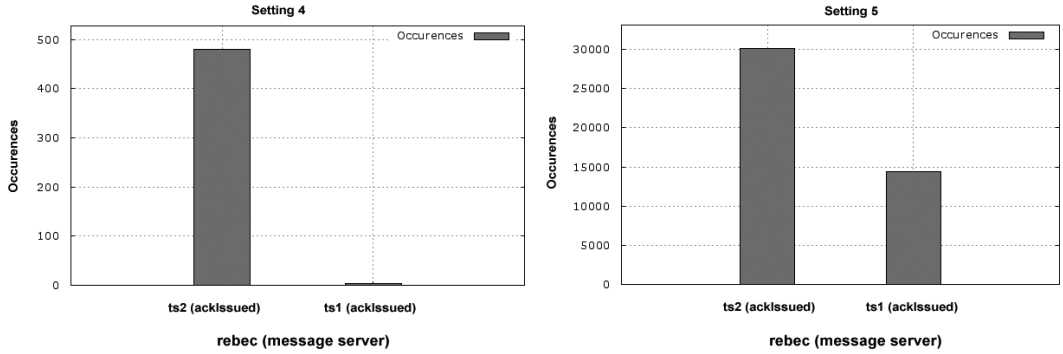

Figure 3: The distribution of issued tickets between ticket services for settings 4 and 5.

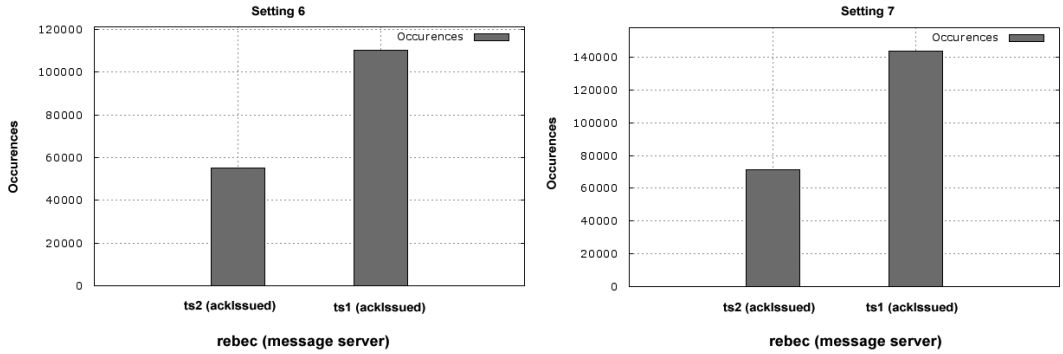

Figure 4: The distribution of issued tickets between ticket services for settings 6 and 7.

7.2. The Elevator System

Our second case study is an elevator system, where a centralized coordinator dispatches the coming requests among the elevators, and also decides on the direction of elevators movement. In this system, the approach of dispatching requests is called the *scheduling policy*, and the decision on the movement of elevators between the floors is called the *movement policy*. Figure 5 shows the event graph of the elevator model. As shown in the figure, a person requests to enter one of the elevators by raising event `callElevator` or he is already in an elevator and presses one button to ask it to stop on one of the floors by raising event `requestFloor`. Both of these events result in raising `handleRequest` event which is the event of the centralized coordinator. Based on the current locations of the elevators and received requests, the centralized coordinator schedules movement for elevators by raising `moveUp`, `moveDown`, and `stopOpen` events.

7.2.1. Timed Rebeca Model

The Timed Rebeca code of the elevator system is shown in Listing 12. The number of rebecs in the main part can be changed in order to make different variants of the elevator system with different sizes (e.g. we increase the number of floors from three to ten in Section 7.2.3). There are four reactive classes `Person`, `Floor`, `Elevator`, and `Coordinator` in this model. Rebecs `el1` and `el2` are instantiated from `Elevator` as the two elevators of the system (Lines 108 and 109). Also, rebecs `floor1` to `floor3`, rebec `pers`, and rebec `coord` are instantiated from reactive classes `Floor`, `Person`, and `Coordinator` repectively, to show that there are three floors, one person and one coordinator in the model (Lines 113-119).

The rebec `pers` starts the model. In the initialization phase, the message `go` is sent to the `pers` by itself (Line 90). The message server `go` models all behaviors of the `pers`. At the start point, the person is put in one of the floors non-deterministically (Line 93). If the person is in the first floor or in the second one, the

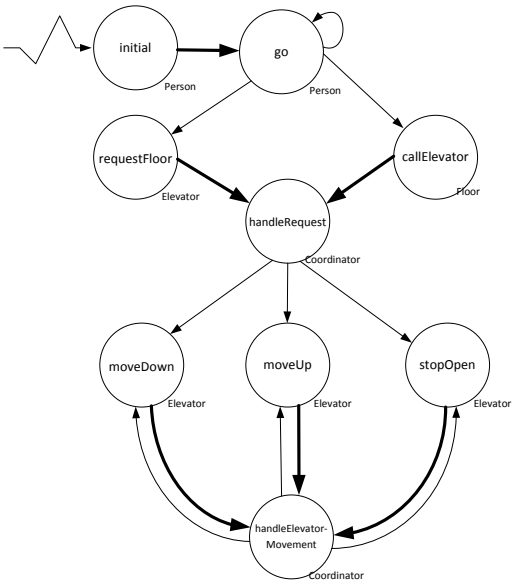

Figure 5: Event graph of the centralized elevator system.

`callElevator` message is sent to one of the floors non-deterministically (Lines 94 and 96-99). Sending this message shows that the person standing in the specified floor presses the button and asks for an elevator to come. This request has to be forwarded to the appropriate elevator later to be served (in the message server `handleRequest`). Without loss of generality, being in floor 3 is assumed as a special case in the message server `go`. Here, being in floor 3 (fc = 3 in Line 100) implies that the person is inside one of the two elevators and requests to go to one of the floors specified by `flr` (Lines 95 and 101-102). As in this case the person is inside the elevator, it sends its request directly to the elevator by sending the `requestFloor` message. All the requests are modeled through sending the messages `requestFloor` (Lines 101-102) and `callElevator` (Lines 97-99) by the person, and are forwarded (Lines 12 and 34) to the message server `handleRequest` in the coordinator.

Algorithms which are related to the scheduling and movement policies are implemented in the message servers `handleRequest` and `handleElevatorMovement` of the `Coordinator`. Different types of request are served in the `handleRequest` message server. For example, the conditional statement in line 60 contains the handling mechanism of requests which are sent from floors. Based on the implemented policy, if a floor requests an elevator and one of the elevators is on the requested floor, that elevator is assigned to the floor (Lines 62-65). Otherwise, one of the elevators is selected non-deterministically (Line 61) and the request is assigned to that elevator (Lines 66-70). There are more cases which are eliminated here and can be found in Appendix A.

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).

7.2.2. Model Checking using McErlang Monitors

The complete Timed Rebeca model for the elevator system can be found at [32] and [43]. To avoid state space explosion, we use the model with three floors for model checking (as shown in Listing 12). We use checkpoint monitors as discussed in Section 4, to verify the safety properties of the model.

```
env int simDelay, simItter, elevMoveDelay;
env int doorDelay1, ..., doorDelay4;

reactiveclass Floor(4) {
  knownrebecs { Coordinator coord; }
  statevars {
    int floorIdent;
    boolean isRestricted;
  }
  msgsrv initial(int floorID) { ... }
  msgsrv callElevator() {
    coord.handleRequest(floorIdent,true);
  }
}

reactiveclass Elevator(4) {
  knownrebecs { Coordinator coord; }
  statevars {
    int movementDelay;
    boolean isRestricted;
  }
  msgsrv initial(int mDelay) { ... }
  msgsrv moveUp(int floor) {
    delay(movementDelay);
    coord.handleElevatorMovement(1,1);
  }
  msgsrv moveDown(int floor) { ... }
  msgsrv stopOpen(int floor, int movementwas) {
    delay(?(doorDelay1, ..., doorDelay4));
    coord.handleElevatorMovement(0,movementwas);
  }
  msgsrv requestFloor(int floor){
  // Send request to handler
    coord.handleRequest(floor,false);}
  msgsrv stopRequest(int floor) { ... }
}

reactiveclass Coordinator(4) {
  knownrebecs {
    Floor flr1, flr2, flr3;
    Elevator el1, el2;
  }
  statevars {
    int el1loc, el1move, el2loc, el2move,
        scheDelay;
    list<int> el1Q, el2Q;
  }
  msgsrv initial() { ... }
  msgsrv handleElevatorMovement(int movement,int
      movementbefore) {
    if(sender == el1 && movement != 0) {
      el1move = movement;
      if(movement == -1) { el1loc -= 1; }
      else if(movement == 1) { el1loc += 1; }
      ...
    } else if(sender == el1) { ... }
    if(sender == el2 && movement != 0) { ... }
    else if(sender == el2) { ... }
  }
  msgsrv handleRequest(int floor, boolean isFloor){
  //Requests from floors.
    if(isFloor == true) {
      int choice = ?(1,2);
      if(erlang.contains(el1Q,floor) != 1 &&
          erlang.contains(el2Q,floor) != 1) {
        if(el1loc == floor || el2loc == floor) {
          if(el1loc==floor) el1Q.insert(floor);
          else if(el2loc==floor) el2Q.insert(floor);
        } else if(el1loc == el2loc) {
          if(choice == 1) el1Q.insert(floor);
          else if(choice == 2) el2Q.insert(floor);
        }
        ...
      } else { ... }
      if(el1move == 0 && el1Q.size() > 0) { ... }
      if(el2move == 0 && el2Q.size() > 0) { ... }
    }
  // Requests from elevator.
    else { ... }
}

reactiveclass Person(4) {
  knownrebecs {
    Floor flr1, flr2, flr3;
    Elevator el1, el2;
  }
  statevars {
    int delayinsec, itterations;
  }
  msgsrv initial(int d, int i) {
    delayinsec = d;
    itterations = i;
    self.go(delayinsec,0);
  }
  msgsrv go(int delays, int incritt) {
    int fc = ?(1, 2, 3);
    int flr = ?(1, 2, 3);
    int elv = ?(1,2);
    if(fc == 1 || fc == 2) {
      if(flr == 1) { flr1.callElevator(); }
      if(flr == 2) { flr2.callElevator();
      if(flr == 3) { flr3.callElevator(); }
    } else {
      if(elv == 1) { el1.requestFloor(flr); }
      else if(elv == 2) { el2.requestFloor(flr); }
    }
    delay(delays);
    if(incritt < itterations) {
      self.go(delayinsec,incritt+1);
    }

  }
}

main {
  Elevator el1(coord):(elevMoveDelay);
  Elevator el2(coord):(elevMoveDelay);
  Floor flr1(coord):(1);
  Floor flr2(coord):(2);
  Floor flr3(coord):(3);
  Coordinator coord(flr1, flr2, flr3, el1, el2):();
  Person pers(flr1, flr2, flr3, el1, el2):( ... );
}
```

Listing 12: The Timed Rebeca model of the elevator system.

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 13.

```
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 13: 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 6.

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 6: Safety verification results for the elevator system.

7.2.3. Statistical Model Checking

In the previous section we checked the elevator system moving between three floors. Here, we model check a larger elevator model by increasing the number of floors to ten, for which the monitor-based model checking is not applicable because the state space is very large. To have the elevator system with ten floors, some parts of the code in Listing 12 (e.g. the main part) must change. The checkpoint `elevatorLocation1` and `elevatorLocation2` is defined in the model to make the value of elevator locations (the floor numbers from which the elevator passes) available for model verification. We injected a bug in the model to provide a few situations in which the elevators can go to the floors which do not exist. We check whether elevators stop in the correct floors ranging from one to ten. We define two safety properties: the elevator location is greater than zero, and the elevator location is less than or equal to ten.

Here, we use the capability of McErlang to get needed simulate traces for statistical model checking. In each simulation run, a randomly chosen trace is investigated for the defined safety property. For each trace, we use the *simulation wrapper* component to execute one simulation, each with 15000 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. All these parameters are set by using environment variables in Listing 12 (Lines 1-2).

Table 7 shows the model checking results for the safety property of "the location of elevator1 is less than or equal to 10". The mean value of correctness of the property is calculated for different error values (ϵ) and confidence values (δ). To understand the way of computing the mean value of correctness and its meaning, we explain the first experiment of Table 7 in more detail.

Considering error value (0.01) and confidence value (0.1), $N_{ct} = 1189$ traces have to satisfy the defined property (refer to Section 7.1.2 for formulas). The total simulation runs (traces) to get this number of satisfied traces is $N' = 1248$. This means that some traces do not satisfy the property as we expected. So, in this experiment the mean value of correctness is $\tilde{\mu}_Z = 0.953$. More accurately, we obtain an (ϵ, δ)-approximation of the mean value of correctness where $Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$, μ_Z is the real mean value of correctness. Therefore, for the first experiment of Table 7, $Pr[|\mu_Z - 0.953| < 0.01] \geq 0.9$.

Tables 8, 9, and 10 show the model checking results for other safety properties. In this section, for each simulation run of each experiment, 150 floor requests are sent randomly to the elevators where requests are sent every 2 units of time. Also, movement between floors takes 2 units of time. The needed time for opening and closing of an elevator door is set non-deterministically to 1, 2, 4 or 6 time units. The scheduling policy is *shortest distance* and the movement policy is *up priority*. The detailed explanations on different policies can be found in Section 7.2.4.

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator1location ≤ 10
1	1189	1248	0.01	0.1	0.953
2	523	556	0.03	0.03	0.941
3	289	296	0.05	0.05	0.976
4	203	210	0.1	0.01	0.967

Table 7: Statistical model checking results for the elevator system. The mean value of correctness is calculated for safety property of "the elevator1 location is less than or equal to 10".

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator2location ≤ 10
1	1189	1256	0.01	0.1	0.947
2	523	548	0.03	0.03	0.954
3	289	306	0.05	0.05	0.945
4	203	214	0.1	0.01	0.949

Table 8: Statistical model checking results for the elevator system. The mean value of correctness is calculated for safety property of "the elevator2 location is less than or equal to 10".

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator1location > 0
1	1189	1189	0.01	0.1	1
2	523	523	0.03	0.03	1
3	289	289	0.05	0.05	1
4	203	203	0.1	0.01	1

Table 9: Statistical model checking results for the elevator system. The mean value of correctness is calculated for safety property of "the elevator1 location is greater than zero".

To show the applicability of our approach for larger models, we increase the number of floors to 15 and 20. For these two extended models, we check the safety property of the elevator1 location shouldn't exceed the number of floors. The obtained results are shown in Tables 11 and 12.

7.2.4. Performance Evaluation

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

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator2location > 0
1	1189	1189	0.01	0.1	1
2	523	523	0.03	0.03	1
3	289	289	0.05	0.05	1
4	203	203	0.1	0.01	1

Table 10: Statistical model checking results for the elevator system. The mean value of correctness is calculated for safety property of "the elevator2 location is greater than zero.".

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator1location ≤ 15
1	289	292	0.05	0.05	0.99
2	203	208	0.1	0.01	0.976

Table 11: Statistical model checking results for the elevator system with 15 floors. The mean value of correctness is calculated for safety property of "the elevator1 location is less than or equal 15.".

Experiment#	Number of satisfied traces	Total number of traces	Error value (ϵ)	Confidence value (δ)	Mean value of correctness for Elevator1location ≤ 20
1	289	295	0.05	0.05	0.98
2	203	209	0.1	0.01	0.971

Table 12: Statistical model checking results for the elevator system with 20 floors. The mean value of correctness is calculated for safety property of "the elevator1 location is less than or equal 20.".

revealed by comparing the mean response time of the scenarios. The simulation of the scenarios take place with the same settings to be able to compare the simulation results.

Scheduling Policy. *Shortest distance*, *shortest distance with movement priority*, and *shortest distance with load balancing* are three different scheduling policies which are studied in the experiments. Listing 14 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 15 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 `el1` is not moving towards the requested floor and `el2` is moving towards it, although the new request is closer to `el1`, it is enqueued in the queue of `el2`.

The third scheduling policy is implemented as shown in Listing 16 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 `el2`, and the queue size of `el1` is less than the queue size of `el2`, then the requested floor is enqueued in the queue of `el1`.

Movement policy. We implemented two movement policies which are *up priority* and *maintain movement*. Listing 17 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 18 in Appendix A represents the pseudo code of *maintain movement* policy. In this 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.

Simulation Results. 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 the *simulation wrapper* component to execute 10 simulations, each with 15000 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 Tables 13, 14, 15 and 16. Each row of the tables represents the mean response time to requests of a specific floor. The margin error is calculated for three different confidence levels of 99%, 95%, and 90%.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	58.5± 2.83	58.5± 2.16	58.5± 1.81	76.2	29.0	683	1	4772
2	44.4± 2.1	44.4± 1.6	44.4± 1.34	61.0	18.0	564	1	5591
3	33.1± 1.46	33.1± 1.12	33.1± 0.93	46.1	14.0	467	1	6568
4	24.5± 0.92	24.5± 0.7	24.5± 0.58	30.6	12.0	317	1	7361
5	20.6± 0.63	20.6± 0.48	20.6± 0.4	21.6	13.0	196	1	7880
6	17.5± 0.4	17.5± 0.31	17.5± 0.26	14.6	13.0	131	1	8182
7	14.6± 0.3	14.6± 0.23	14.6± 0.19	10.9	12.0	85	1	8615
8	13.4± 0.29	13.4± 0.22	13.4± 0.18	10.6	11.0	82	1	8966
9	14.7± 0.34	14.7± 0.26	14.7± 0.22	12.3	11.0	89	1	8777
10	18.0± 0.37	18.0± 0.28	18.0± 0.24	13.3	15.0	99	1	8442

Table 13: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement Policy: **Up priority**. SD, WCT, BCT stands for standard deviation, worst case time and best case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	21.6± 0.42	21.6± 0.32	21.6± 0.27	15.5	18.0	95	1	9004
2	17.3± 0.37	17.3± 0.28	17.3± 0.24	14.1	12.0	87	1	9508
3	14.8± 0.30	14.8± 0.23	14.8± 0.19	11.7	11.0	68	1	9926
4	14.6± 0.27	14.6± 0.21	14.6± 0.17	10.5	12.0	72	1	9915
5	14.7± 0.247	14.7± 0.188	14.7± 0.158	9.5	12.0	65	1	9762
6	14.6± 0.25	14.6± 0.191	14.6± 0.16	9.7	12.0	62	1	9915
7	14.3± 0.27	14.3± 0.205	14.3± 0.17	10.4	11.0	77	1	9919
8	14.8± 0.3	14.8± 0.23	14.8± 0.19	11.8	11.0	80	1	9930
9	17.1± 0.36	17.1± 0.28	17.1± 0.23	13.9	12.0	81	1	9555
10	21.7± 0.42	21.7± 0.32	21.7± 0.27	15.5	17.0	86	1	9021

Table 14: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst case time and best case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	28.3± 0.622	28.3± 0.474	28.3± 0.397	19.9	24.0	99	1	6767
2	22.4± 0.534	22.4± 0.407	22.4± 0.341	17.9	17.0	92	1	7420
3	18.7± 0.426	18.7± 0.325	18.7± 0.272	15.0	14.0	90	1	8168
4	16.7± 0.349	16.7± 0.267	16.7± 0.223	12.5	14.0	78	1	8444
5	16.3± 0.307	16.3± 0.234	16.3± 0.196	11.0	14.0	67	1	8457
6	16.2± 0.3	16.2± 0.229	16.2± 0.192	10.9	14.0	63	1	8688
7	16.8± 0.344	16.8± 0.262	16.8± 0.219	12.3	14.0	73	1	8449
8	18.6± 0.427	18.6± 0.326	18.6± 0.273	15.0	14.0	79	1	8142
9	21.6± 0.516	21.6± 0.393	21.6± 0.329	17.6	17.0	92	1	7691
10	28.1± 0.618	28.1± 0.471	28.1± 0.394	19.9	24.0	103	1	6843

Table 15: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with movement priority**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst case time and best case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	28.1± 0.5	28.1± 0.381	28.1± 0.319	16.4	28.0	79	1	7096
2	22.9± 0.452	22.9± 0.345	22.9± 0.289	15.3	21.0	76	1	7554
3	18.9± 0.36	18.9± 0.282	18.9± 0.236	13.0	16.0	67	1	8161
4	16.8± 0.306	16.8± 0.234	16.8± 0.195	10.9	14.0	64	1	8354
5	15.5± 0.255	15.5± 0.194	15.5± 0.163	9.2	14.0	53	1	8600
6	15.6± 0.262	15.6± 0.2	15.6± 0.167	9.5	14.0	52	1	8695
7	16.5± 0.305	16.5± 0.232	16.5± 0.194	10.9	14.0	63	1	8457
8	19.2± 0.378	19.2± 0.288	19.2± 0.241	13.2	16.0	66	1	8071
9	22.7± 0.447	22.7± 0.341	22.7± 0.285	15.2	21.0	68	1	7627
10	28.4± 0.508	28.4± 0.387	28.4± 0.324	16.7	28.0	85	1	7140

Table 16: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with load balancing**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst case time and best case time, respectively.

Table 17 shows the mean response time to all floor requests of each configuration. 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.

Configuration	**Mean 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 17: Simulation results for different configurations of the elevators system. Each row contains the results related to all floor requests of each configuration.

8. Conclusion

In the work presented in this paper and the conference paper [12], we developed techniques and extensions for making modeling and analysis of Timed Rebeca models easier. From modeling point of view, we proposed an extension to the Timed Rebeca language (introduced in [6]) which provides the ability of calling Erlang functions. This way, the modeler may define functions and modules using all the programming features of Erlang which makes modeling easier than before. We also added the list data structure to Timed Rebeca, which is useful in modeling queues and buffers.

From analysis point of view, the most significant extension is adding *checkpoint* functions to Timed Rebeca models. Our extensions in the language as well as timed extensions in McErlang provide us with model checking and performance evaluation of timed models. We developed a toolset to translate the Timed Rebeca models to Erlang. 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 addition to these analysis facilities, we developed statistical model checking tool for Timed Rebeca models. Using statistical model checking, we are able to verify safety properties of larger models for which the McErlang model checking suffers from the state space explosion problem.

McErlang is used to generate simulation traces of Timed Rebeca models. The traces are used for performance evaluation and statistical model checking of Timed Rebeca models. In simulation, the statistical methods are applied to simulation traces to reveal the system performance. In this work, two kinds of performance analysis are provided, which are paired-checkpoint analysis and checkpoint analysis. In checkpoint analysis, our focus is on the evolution of a particular parameter during time. In paired-checkpoint analysis, we study the difference between two values, like the duration of waiting, or service. This way, we provide the performance analysis of the system.

We evaluated the developed toolset and the proposed approaches using two case studies. In the elevator example, for different configurations we measure the response time of the requests arriving from each floor. Each configuration includes different scheduling algorithm and movement policy, which are responsible for assigning the requests to the elevators and determining how the elevators move between the floors, respectively. We also checked safety properties using both McErlang as a back-end model checker and the statistical model checking approach. In the ticket service example, for different settings the mean response time to ticket requests are calculated. Also, the safety property of "at least one ticket is issued" is checked using safety monitors and the statistical model checking method.

Acknowledgment

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

Appendix 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 14: 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 15: Timed Rebeca pseudo code for scheduling policy *shortest distance with movement priority. [...]* denotes the deleted code which has been already shown in Listing 14. 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 16: Timed Rebeca pseudo code for scheduling policy *shortest distance with load balancing. [...]* denotes deleted code which has been already shown in Listing 14. 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
  ....
}
```

Listing 17: 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 18: Timed Rebeca pseudo code for movement policy *Maintain movement*. The pseudo code shows the algorithm for elevator 1.

References

[1] 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).

[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, G. Rozenberg (Eds.), Foundations of Object-Oriented Languages, Springer-Verlag, Berlin, Germany, 1990, pp. 1–59.

[4] I. A. Mason, C. L. Talcott, Actor Languages: Their Syntax, Semantics, Translation, and Equivalence, Theoretical Computer Science 220 (2) (1999) 409–467.

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

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

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

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

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

[10] G. D. Plotkin, A Structural Approach to Operational Semantics, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (Sep. 1981).

[11] Erlang, Erlang Programming Language Homepage, http://www.erlang.org.

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

[13] H. Kristinsson, A. Jafari, E. Khamespanah, B. Magnusson, M. Sirjani, Analysing Timed Rebeca Using McErlang, in: Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE! 2013, ACM, New York, NY, USA, 2013, pp. 25–36.

[14] C. B. Earle, 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, Springer-Verlag, Berlin, Heidelberg, 2012, pp. 251–267.

[15] 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, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 162–175.

[16] Z. Sharifi, M. Mosaffa, S. Mohammadi, 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.

[17] Z. Sharifi, S. Mohammadi, M. Sirjani, Comparison of NoC Routing Algorithms Using Formal Methods, To be published in proceedings of PDPTA'13, 2013.
[18] L. Linderman, K. Mechitov, B. F. Spencer, TinyOS-based Real-Time Wireless Data Acquisition Framework for Structural Health Monitoring and Control, Structural Control and Health Monitoring 20 (6) (June 2013) 1007–1020.
[19] B. Cho, M. Rahman, T. Chajed, I. Gupta, C. Abad, N. Roberts, P. Lin, Natjam: Design and Evaluation of Eviction Policies for Supporting Priorities and Deadlines in Mapreduce Clusters, in: Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, ACM, New York, NY, USA, 2013, pp. 6:1–6:17.
[20] E. D'Osualdo, J. Kochems, C.-H. Ong, Automatic Verification of Erlang-Style Concurrency, in: F. Logozzo, M. Fhndrich (Eds.), Static Analysis, Vol. 7935 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 454–476.
[21] R. Carlsson, An introduction to Core Erlang, in: In Proceedings of the PLI'01 Erlang Workshop, 2001.
[22] UPPAAL, UPPAAL Homepage, http://www.uppaal.com.
[23] P. C. Ölveczky, J. Meseguer, Semantics and Pragmatics of Real-Time Maude, Higher Order Symbol. Comput. 20 (1-2) (2007) 161–196.
[24] R. Alur, D. Dill, A Theory of Timed Automata, Theoretical Computer Science 126 (1994) 183–235. doi:10.1016/0304-3975(94)90010-8.
[25] A. David, K. Larsen, A. Legay, M. Mikučionis, D. Poulsen, Uppaal SMC tutorial, International Journal on Software Tools for Technology Transfer 17 (4) (2015) 397–415.
[26] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, K. G. Larsen, Model-based Schedulability Analysis of Safety Critical Hard Real-time Java Programs, in: Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '08, ACM, New York, NY, USA, 2008, pp. 106–114.
[27] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, J. Vitek, Java for Safety-Critical Applications, in: 2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009), 2009.
[28] C. Frost, C. S. Jensen, K. S. Luckow, B. Thomsen, WCET Analysis of Java Bytecode Featuring Common Execution Environments, in: Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems, JTRES '11, ACM, New York, NY, USA, 2011, pp. 30–39.
[29] M. Schoeberl, W. Puffitsch, R. U. Pedersen, B. Huber, Worst-case Execution Time Analysis for a Java Processor, Softw. Pract. Exper. 40 (6) (2010) 507–542.
[30] M. Schoeberl, H. Søndergaard, B. Thomsen, A. P. Ravn, A Profile for Safety Critical Java, in: Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece, 2007, pp. 94–101.
[31] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, M. Izadi, Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system, Sci. Comput. Program. 98 (2015) 184–204. doi:10.1016/j.scico.2014.07.005.
URL http://dx.doi.org/10.1016/j.scico.2014.07.005
[32] Rebeca, Rebeca homepage, http://www.rebeca-lang.org.
[33] Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, P. C. Ölveczky, E. Khamespanah, Formal semantics and analysis of timed rebeca in real-time maude, in: FTSCS, 2013, pp. 178–194.
[34] E. Khamespanah, R. Khosravi, M. Sirjani, Efficient TCTL model checking algorithm for timed actors, in: E. G. Boix, P. Haller, A. Ricci, C. Varela (Eds.), Proceedings of the 4th International Workshop on Programming based on Actors Agents & Decentralized Control, AGERE! 2014, Portland, OR, USA, October 20, 2014, ACM, 2014, pp. 55–66. doi:10.1145/2687357.2687366.
URL http://doi.acm.org/10.1145/2687357.2687366
[35] M. Sirjani, M. M. Jaghoori, Formal Modeling, Springer-Verlag, Berlin, Heidelberg, 2011, Ch. Ten Years of Analyzing Actors: Rebeca Experience, pp. 20–56.
[36] R. M. Karp, M. Luby, N. Madras, Monte-carlo approximation algorithms for enumeration problems, J. Algorithms 10 (3) (1989) 429–448.
[37] P. Dagum, R. M. Karp, M. Luby, S. M. Ross, An optimal algorithm for monte carlo estimation, SIAM J. Comput. 29 (5) (2000) 1484–1496.
[38] R. Grosu, S. A. Smolka, Quantitative model checking, in: T. Margaria, B. Steffen, A. Philippou, M. Reitenspieß (Eds.), International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004, October 30 - November 2, 2004, Paphos, Cyprus. Preliminary proceedings, Vol. TR-2004-6 of Technical Report, Department of Computer Science, University of Cyprus, 2004, pp. 165–174.
[39] A. Legay, B. Delahaye, S. Bensalem, Statistical model checking: An overview, in: Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings, 2010, pp. 122–135.
[40] J. S. Simonoff, Smoothing Methods in Statistics, Springer, 1998.
[41] Z Table Site, http://www.statisticshowto.com/tables/z-table/.
[42] A. H. Buss, Modeling with Event Graphs, In Proceedings of the 28th conference on winter simulation, 1996, pp. 153–160.
[43] H. Kristinsson, Event-based Analysis of Real-Time Actor Models - Master Thesis, Reykjavik University, Iceland (2012).

LaTeX Source Files

Click here to download LaTeX Source Files: CLSS-revised version.zip