Modelling and Simulation of Real-Time Systems using Timed Rebeca

Luca Aceto[1], Matteo Cimini[1], Anna Ingolfsdottir[1], Arni Hermann Reynisson[1], Steinar Hugi Sigurdarson[1], and Marjan Sirjani[1,2]

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

Abstract. In this paper we propose an extension of the Rebeca language that can be used to model distributed and asynchronous systems with timing constraints. We provide the formal semantics of the language using Structural Operational Semantics, and show its expressiveness by means of examples. We provide an automated translation from Timed Rebeca to the Erlang language, which provides a first implementation of Timed Rebeca. We can use the tool to set the parameters of Timed Rebeca models, which represent the environment and component variables, and use McErlang to run multiple simulations for different settings.

1 Introduction

This paper presents an extension of the actor-based Rebeca language [20] that can be used to model distributed and asynchronous systems with timing constraints. This extension of Rebeca is motivated by the ubiquitous presence of real-time computing systems, whose behaviour depends crucially on timing as well as functional requirements.

A well-established paradigm for modelling the functional behaviour of distributed and asynchronous systems is the actor model. This model was originally introduced by Hewitt [7] as an agent-based language, and is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation [1]. In response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message it receives. Actors have encapsulated states and behaviour, and are capable of creating new actors, and redirecting communication links through the exchange of actor identities. Different interpretations, dialects and extensions of actor models have been proposed in several domains and are claimed to be the most suitable model of computation for the most dominating applications, such as multi-core programming and web services [8].

Reactive Objects Language, *Rebeca* [20], is an operational interpretation of the actor model with formal semantics and model-checking tools. Rebeca is designed to bridge the gap between formal methods and software engineers.

The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and is embodied in verification tools [12, 19, 20]. With its simple, message-driven and object-based computational model, Java-like syntax, and set of verification tools, Rebeca is an interesting and easy-to-learn model for practitioners.

Although actors are attracting more and more attention both in academia and industry, little has been done on timed actors and even less on analyzing timed actor-based models. In this paper, we will show how to extend Rebeca with time constraints, and introduce Timed Rebeca as a language for designing and analyzing timed systems in distributed and asynchronous settings.

Timed Rebeca 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. The structure of the model can very well represent service oriented architectures, while the computational model matches the network infrastructure. Hence the model captures faithfully the behaviour of the system in a distributed and asynchronous world.

Timed Rebeca is supported by a tool by means of which we can translate a timed Rebeca model to Erlang [5], set the parameters which represent the environment and component variables, and run McErlang [6] to simulate the model. The tool together with some examples can be found at [9]. Using this tool, we can change the settings of different parameters and rerun the simulation in order to investigate different scenarios, find potential bugs and problems, and optimize the model by manipulating the settings of its variables. The parameters can be timing constraints on the local computations (e.g., deadlines for accomplishing a requested service), computation time for providing a service, and frequency of a periodic event. Parameters can also represent network configurations and delays.

In this study, we also present the formal semantics of Timed Rebeca using Structural Operational Semantics (SOS) [17]. The formal semantics is the basis for the mapping from Rebeca to Erlang.

Related Work Different approaches are used in designing formal modelling languages for real-time systems. The model of timed automata, introduced by Alur and Dill [2], has established itself as a classic formalism for modelling real-time systems. The theory of Timed Automata is a timed extension of automata theory, using clock constraints on both locations and transitions. In many other cases the proposed modelling languages for real-time systems are extensions of existing languages with real-time concepts—see, for example, TCCS [22] and Real-time Maude [16].

A real-time actor model, RT-synchronizer, is proposed in [18], where a centralized synchronizer is responsible for enforcing real-time relations between events. Actors are extended with timing assumptions, and the functional behaviours of actors and the timing constraints on patterns of actor invocation are

separated. The semantics for the timed actor-based language is given in [14]. Two positive real-valued constants, called *release time* and *deadline*, are added to the *send* statement and are considered as the earliest and latest time when the message can be invoked. In Timed Rebeca, we have the constructs *after* and *deadline*, which are representing the same concepts, respectively. In our language, it is also possible to consider a time *delay* in the execution of a computation. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, our model allows us to work at a lower level of abstraction. Using Timed Rebeca, a modeller can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various points of view.

There is also some work on schedulability analysis of actors [15], but this is not applied on a real-time actor language. Time constraints are considered separately. Recently, there have been some studies on schedulability analysis for Rebeca models [11]. This work is based on mapping Rebeca models to Timed Automata and using UPPAAL to check the schedulability of the resulting models. Deadlines are defined for accomplishing a service and each task spends a certain amount of time for execution. In the above-mentioned papers, modelling of time is not incorporated in the Rebeca language.

Creol is a concurrent object-oriented language with an operational semantics written in an actor-based style, and supported by a language interpreter in the Maude system. In [4], Creol is extended by adding best-case and worst-case execution time for each statement, and a deadline for each method call. In addition, an object is assigned a scheduling strategy to resolve the nondeterminism in selecting from the enabled processes. This work is along the same lines as the one presented in [11] and the focus is on schedulability analysis, which is carried out in a modular way in two steps: first one models an individual object and its behavioural interface as Timed Automata, and then one uses UPPAAL to check the schedulability considering the specified execution times and the deadlines. In this work, network delays are not considered, and the execution time is weaved together with the statements in a fine-grained way.

In [3] a timed version of Creol is presented in which the only additional syntax is read-only access to the global clock, plus adding a data-type *Time* together with its accompanying operators to the language. Timed behaviour is modelled by manipulating the *Time* variables and via the *await* statement in the language.

2 Timed Rebeca

A Rebeca model consists of a set of *reactive classes* and the *main* program in which we declare reactive objects, or rebecs, as instances of *reactive classes*. A reactive class has an argument of type integer, which denotes the length of its message queue. The body of the reactive class includes the declaration for its *known rebecs*, variables, and methods (also called message servers). Each method body

consists of the declaration of local variables and a sequence of statements, which can be assignments, *if* statements, rebec creation (using the keyword *new*), and method calls. Method calls are sending asynchronous messages to other rebecs (or to self) to invoke the corresponding message server (method). Message passing is fair, and messages addressed to a rebec are stored in its message queue. The computation takes place by taking the message from the front of the message queue and executing the corresponding message server [20].

In this paper we introduce an extension of Rebeca with real-time features. We consider a global clock for our timed Rebeca models. Methods are still executed atomically, but we can model passing of time while executing a method. Instead of a message queue for each rebec, we have a bag containing the messages that are sent. Each rebec knows about its local time and can put deadlines on the service requests (messages) that are sent declaring that the request will not be valid after the deadline (modelling the timeout for a request). When a message is sent there can also be a constraint on the earliest time at which it can be served (taken from the message bag by the receiver rebec). The modeller may use these constraints for various purposes, such as modelling the network delay or modelling a periodic event.

The timing primitives that are added to the syntax of Rebeca are *delay*, *now*, *deadline* and *after*. Figure 1 shows the grammar for Timed Rebeca. The *delay* statement models the passing of time for a rebec during execution of a method, and *now* returns the local time of the rebec. The keywords *after* and *deadline* can only be used in conjunction with a method call. The messages that are sent are put in the message bag together with their time tag and *deadline* tag. 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 *now* when the message was sent, with the value of the argument of the *after* added to it when the message is augmented with an *after*. The intuition is that a message cannot be taken (served) before the time that the time tag determines.

The progress of time is modeled locally by the delay statement. Each delay statement within a method body increases the value of the local time (variable *now*) of the respective rebec by the amount of its argument. When we reach a *call* statement (sending a message), we put that message in the message bag augmented with a time tag. The local time of a rebec can also be increased when we take a message from the bag to execute the corresponding method.

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 (*now*) of the receiver rebec is set to the maximum value between its current time and the time tag of the message. The current local time of each rebec is the value of *now*. This value is frozen when the method execution ends until the next method of the same rebec is taken for execution.

The arguments of *after* and *delay* are relative values, but when the corresponding messages are put in the message bag their tags are absolute values,

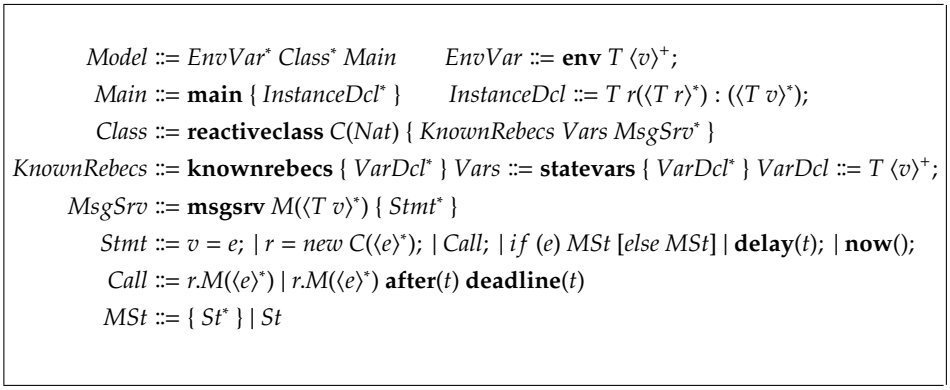

Fig. 1: Abstract syntax of Timed Rebeca. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition more than once, superscript ∗ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Identifiers *C*, *T*, *M*, *v* and *r* denote class, type, method, variable and rebec names, respectively; Nat denotes a natural number; and *e* denotes an (arithmetic, boolean or nondetermistic choice) expression.

which are computed by adding the relative values of the arguments to the value of the variable *now* of the sender rebec (where the messages are sent). To summarize, Timed Rebeca extends Rebeca with the following four constructs.

- **Delay**: *delay(t)*, where *t* is a positive natural number, will increase the value of the local clock of the respective rebec by the amount *t*.
- **Now**: *now()* returns the time of the local clock of the rebec from which it is called.
- **Deadline**: *r.m() deadline(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a positive natural number, means that the message *m* is sent to the rebec *r* and 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 timeouts.
- **After**: *r.m() after(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a positive natural number, means that the message *m* is sent to the rebec *r* and is put in the message bag. The message cannot be taken from the bag before *t* time units have passed. After statements can be used to model network delays in delivering a message to the destination.

Ticket Service Example We use a ticket service as a running example throughout the article. Listing 1 shows this example written in Timed Rebeca. The ticket service model consists of two reactive classes: *Agent* and *TicketService*. Two rebecs, *ts*1 and *ts*2, are instantiated from the reactive class *TicketService*, and one rebec *a* is instantiated from the reactive class *Agent*. The agent *a* is initialized by sending a message $findTicket$ to itself in which a message *requestTicket* is sent to the ticket service *ts*1 or *ts*2 based on the parameter passed to $findTicket$. The deadline for

the message *requestTicket* to be served is *requestDeadline* time units. Then, after *checkIssuedPeriod* time units the agent will check if it has received a reply to its request by sending a *checkTicket* message to itself, modelling a periodic event. There is no receive statement in Rebeca, and all the computation is modeled via asynchronous message passing, so, we need a periodic check. The *attemptCount* variable helps the agent to keep track of the ticket service rebec that the request is sent to. The *token* variable allows the agent to keep track of which incoming *ticketIssued* message is a reply to a valid request. When any of the ticket service rebecs receives the *requestTicket* message, it will issue the ticket after *serviceTime1* or *serviceTime2* time units, which is modelled by sending *ticketIssued* to the agent with the *token* as parameter. The expression ?(*serviceTime*1, *serviceTime*2) denotes a nondeterministic choice between *serviceTime*1 and *serviceTime*2 in the *assignment* statement. Depending on the chosen value, the ticket service may or may not be on time for its reply.

```
1  env int requestDeadline, checkIssuedPeriod, retryRequestPeriod, newRequestPeriod,
       serviceTime1, serviceTime2;
2
3  reactiveclass Agent {
4    knownrebecs { TicketService ts1; TicketService ts2; }
5    statevars { int attemptCount; boolean ticketIssued; int token; }
6    msgsrv initial() { self.findTicket(ts1); } // initialize system, check 1st ticket service
7    msgsrv findTicket(TicketService ts) {
8      attemptCount += 1; token += 1;
9      ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
10     self.checkTicket() after(checkIssuedPeriod); // check if the request is replied
11   }
12   msgsrv ticketIssued(int tok) { if (token == tok) { ticketIssued = true; } }
13   msgsrv checkTicket() {
14     if (!ticketIssued && attemptCount == 1) {        // no ticket from 1st service,
15       self.findTicket(ts2);                          // try the second TicketService
16     } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
17       self.retry() after(retryRequestPeriod);        // restart from the first TicketService
18     } else {                                         // the second TicketService replied,
19       self.retry() after(newRequestPeriod);          // new request by a customer
20     }
21   }
22   msgsrv retry() {
23     attemptCount = 0; self.findTicket(ts1);     // restart from the first TicketService
24   }
25 }
26
27 reactiveclass TicketService {
28   knownrebecs { Agent a; }
29   msgsrv initial() { }
30   msgsrv requestTicket(int token) {
31     int wait = ?(serviceTime1,serviceTime2);    // the ticket service sends the reply
32     delay(wait);                                // after a non-determinstic delay of
33     a.ticketIssued(token);                      // either serviceTime1 or serviceTime2
34   }
35 }
36
37 main {
38   Agent a(ts1, ts2):();               // instantiate agent, with two known rebecs
39   TicketService ts1(a):();            // instantiate 1st and 2nd ticket services, with
40   TicketService ts2(a):();            // the agent as their known rebecs
41 }
```

Listing 1. A Timed Rebeca model of the ticket service example

2.1 Structural Operational Semantics for Timed Rebeca

In this section we provide an SOS semantics for Timed Rebeca in the style of Plotkin [17]. The behaviour of Timed Rebeca programs is described by means of the transition relation $\rightarrow$ that describes the evolution of the system.

The states of the system are pairs (Env, B), where Env is a finite set of environments and B is a bag of messages. For each rebec A of the program there is an environment σ_A contained in Env, that is a function that maps variables to their values. The environment σ_A represents the private store of the rebec A. Besides the user-defined variables, environments also contain the value for the special variables *now*, the current time, and *sender*, which keeps track of the rebec that invoked the method that is currently being executed. The environment σ_A also maps every method name to its body.

The bag contains an unordered collection of messages. Each message is a tuple of the form $(A_i, m(\overline{v}), A_j, TT, DL)$. Intuitively, such a tuple says that at time TT the sender A_j sent the message to the rebec A_i asking it to execute its method m with actual parameters $\overline{v}$. Moreover this message expires at time DL.

The system transition relation $\rightarrow$ is defined by the rule *scheduler*:

$$(scheduler)\ \frac{(\sigma_{A_i}(m), \sigma_{A_i}[\text{now} = \max(TT, \sigma_{A_i}(now)),\ \overline{[arg = v]},\ sender = A_j],\ Env,\ B) \xrightarrow{\tau} (\sigma'_{A_i},\ Env',\ B')}{(\{\sigma_{A_i}\} \cup Env, \{(A_i, m(\overline{v}), A_j, TT, DL)\} \cup B) \rightarrow (\{\sigma'_{A_i}\} \cup Env',\ B')}$$

where $\sigma_{A_i}(now) \leq DL$ and $\min(TT, B, A_i)$. The *scheduler* rule allows the system to progress by picking up messages from the bag and executing the corresponding methods. The first side condition of the rule, namely $\sigma_{A_i}(now) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The second side condition is the predicate $min(TT, B, A_i)$, which is satisfied whenever the time tag TT is the smallest time tag for the messages for the rebec A_i in the bag B. The premise executes the method m, as described by the transition relation $\xrightarrow{\tau}$, which will be defined below. The method body is looked up in the environment of A_i and is executed in the environment of A_i modified as follows: (1) The variable *sender* is set to the sender of the message. (2) In executing the method m, the formal parameters $\overline{arg}$ are set to the values of the actual parameters $\overline{v}$. Methods of arity n are supposed to have $arg_1, arg_2, \ldots, arg_n$ as formal parameters. This is without loss of generality since such a change of variable names can be performed in a pre-processing step for any program. (3) The variable *now* is set to the maximum between the current time of the rebec and the time tag of the selected message.

The execution of the methods of rebec A_i may change the private store of the rebec A_i, the bag B by adding messages to it and the list of environments by creating new rebecs through *new* statements. Once a method is executed to completion, the resulting bag and list of environments are used to continue the progress of the whole system.

The transition relation $\xrightarrow{\tau}$ describes the execution of methods in the style of natural semantics [13]. (See Figure 2 for selected rules. The full set of rules may be

found in Appendix A.) Since in this kind of semantics the whole computation of a method is performed in a single step, this choice perfectly reflects the atomic execution of methods underlying the semantics of the Rebeca language. The general form of this type of transition is $(S, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')$. A single step of $\xrightarrow{\tau}$ consumes all the code S and provides the value resulting from its execution. Carrying the bag B is important because new messages may be added to it during the execution of a statement S. Also Env is required because *new* statements create new rebecs and may therefore add new environments to it. In the semantics, the local environment σ is separated from the environment list Env for the sake of clarity. The result of the execution of the method thus amounts to the modified private store σ', the new list of environments Env' and the new bag B'.

The rules for assignment, conditional statement and sequential composition are standard. The rules for the timing primitives deserve some explanation.

- Rule *msg* describes the effect of method invocation statements. For the sake of brevity, we limit ourselves to presenting the rule for method invocation statements that involve both the *after* and *deadline* keywords. The semantics of instances of that statement without those keywords can be handled as special cases of that rule by setting the argument of *after* to zero and that of *deadline* to $+\infty$, meaning that the message never expires. Method invocation statements put a new message in the bag, taking care of properly setting its fields. In particular the time tag for the message is the current local time, which is the value of the variable *now*, plus the number d that is the parameter of the *after* keyword.
- Delay statements change the private variable *now* for the considered rebec.

Finally, the creation of new rebecs is handled by the rule *create*. A fresh name A is used to identify the newly created rebec and is assigned to *varname*. A new environment σ_A is added to the list of environments. At creation time, σ_A is set to have its method names associated to their code. A message is put in the bag in order to execute the *initial* method of the newly created rebec.

$$(msg)\ (varname.m(\overline{v})\ after(d)\ deadline(DL), \sigma, Env, B) \xrightarrow{\tau} (\sigma, Env, \{(\sigma(varname), m(\overline{eval(v, \sigma)}), \sigma(self), \sigma(now) + d, \sigma(now) + DL)\} \cup B)$$

$$(delay)\ (delay(d), \sigma, Env, B) \xrightarrow{\tau} (\sigma[now = \sigma(now) + d], Env, B)$$

$$(create)\ (varname = new\ O(\overline{v}), \sigma, Env, B) \xrightarrow{\tau} (\sigma[varname = A], \{\sigma_A[now = \sigma(now)]\} \cup Env, \{(A, initial(\overline{eval(v, \sigma)}), \sigma(self)), \sigma(now), +\infty)\} \cup B)$$

Fig. 2: Selected Method-Execution Transition Rules. In rule *create*, the rebec name A should not appear in the range of the environment σ. The function *eval* evaluates expressions in a given environment in the expected way.

3 Mapping from Timed Rebeca to Erlang

In this section, we present a translation from the fragment of Timed Rebeca without rebec creation to Erlang (for a more extended explanation see [9]). The motivation for translating Timed Rebeca models to Erlang code is to be able to use McErlang [6] to run experiments on the models. This translation also yields a first implementation of Timed Rebeca.

McErlang is a model-checking tool written in Erlang to verify distributed programs written in Erlang. It supports Erlang datatypes, process communication, fault detection and fault tolerance and the Open Telecom Platform (OTP) library, which is used by most Erlang programs. The verification methods range from complete state-based exploration to simulation, with specifications written as LTL formulae or hand-coded runtime monitors. This paper focuses on simulation since model checking with real-time semantics is not yet offered by McErlang. Note, however, that our translation opens the possibility of model checking (untimed) Rebeca models using McErlang, which is not the subject of this paper.

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

Erlang has few concurrency primitives:

- `Pid = spawn(Fun)` creates a new process that evaluates the given function `Fun` in parallel with the process that invoked spawn.
- `Pid !Msg` sends the given message `Msg` to the process with the identifier `Pid`.
- **`receive ... end`** receives a message that has been sent to a process; message discrimination is based on pattern matching.

Erlang also offers the **`after`** construct that allows a process to receive a message with a timeout, as shown in Listing 2. When a process reaches a **`receive`** expression it looks in the queue and takes the message that matches the pattern if the corresponding guard is true. A guard is a boolean expression, which can include the variables of the same process. The process looks in the queue each time a message arrives until the timeout occurs.

```
1 receive
2   Pattern1 when Guard1 -> Expr1;
3   Pattern2 when Guard2 -> Expr2;
4   ...
5 after
6   Time -> Expr
7 end
```

Listing 2. Syntax of a receive with timeout.

Mapping The abstract syntax for a fragment of Erlang that is required to present the translation is shown in Figure 3. Table 1 offers an overview of how a construct in one language relates to one in the other. We discuss the general principles behind our translation in more detail below.

$$
\begin{aligned}
&\mathit{Program} ::= \mathit{Function}^* \quad \mathit{Function} ::= v(\mathit{Pattern}^*) \rightarrow e \\
&\quad \mathit{Expr} ::= e_1\ op_e\ e_2 \mid e(\langle e \rangle^*) \mid \textbf{case}\ e\ \textbf{of}\ \mathit{Match}\ \textbf{end} \mid \textbf{receive}\ \mathit{Match}\ [\textbf{after}\ \mathit{Time}\ \rightarrow\ e]\ \textbf{end} \\
&\qquad \mid \textbf{if}\ \langle \mathit{Match} \rangle^*\ \textbf{end} \mid e_1\ !\ e_2 \mid e_1\ ,\ e_2 \mid \mathit{Pattern} = e \mid \mathit{BasicValue} \mid v \mid \{\langle e \rangle^*\} \mid [\langle e \rangle^*] \\
&\quad \mathit{Match} ::= \mathit{Pattern}\ \textbf{when}\ \mathit{Guard} \rightarrow e \\
&\quad \mathit{Pattern} ::= v \mid \mathit{BasicValue} \mid \{\langle \mathit{Pattern} \rangle^*\} \mid [\langle \mathit{Pattern} \rangle^*] \quad \mathit{Time} ::= \text{int} \\
&\quad \mathit{Value} ::= \mathit{BasicValue} \mid \{\langle \mathit{Value} \rangle^*\} \mid [\langle \mathit{Value} \rangle^*] \quad \mathit{BasicValue} ::= \text{atom} \mid \text{number} \mid \text{pid} \mid \text{fid} \\
&\quad \mathit{Guard} ::= g_1\ op_g\ g_2 \mid \mathit{BasicValue} \mid v \mid g(\langle g \rangle^*) \mid \{\langle g \rangle^*\} \mid [\langle g \rangle^*]
\end{aligned}
$$

Fig. 3: Abstract syntax of a relevant subset of Erlang. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Identifiers v, p and g denote variable names, patterns and guards, respectively, and e denotes an expression.

Reactive classes are translated into three functions, each representing a possible behaviour of an Erlang process: 1) the process waits to get references to known rebecs, 2) the process reads the initial message from the queue and executes it, 3) the process reads messages from the queue and executes them. Once processes reach the last function they enter a loop. Erlang pseudocode for the reactive class *TicketService* in the Rebeca model in Listing 1 is shown in Listing 3.

```
1  ticketService() ->
2    receive
3      % wait for a message with a set of known rebecs
4      {Agent} ->
5        % proceed to the next behaviour
6        ticketService(#ticketService_knownrebecs{agent=Agent})
7    end.
8  ticketService(KnownRebecs) ->
9    receive
10     % wait for the 'initial' message
11     initial ->
12       % process message 'initial' and proceed to the next behaviour
13       ticketService(KnownRebecs, #ticketService_statevars{})
14   end.
15 ticketService(KnownRebecs, StateVars) ->
16   receive
```

Timed Rebeca	Erlang
Model →	A set of processes
Reactive classes →	A process whose behaviour consists of three functions
Known rebecs →	Record of variables
State variables →	Record of variables
Message server →	A match in a receive expression
Local variables →	Record of variables
Message send →	Message send expression
Message send w/after →	Message send expression inside a receive with a timeout
Message send w/deadline →	Message send expression with the deadline as parameter
Delay statement →	Empty receive with a timeout
Now expression →	System time
Assignment →	Record update
If statement →	If expression
Nondeterministic selection →	Random selection in the simulation tool

Table 1. Structure of the mapping from Timed Rebeca to Erlang.

```
17    % wait for each message servers
18    requestTicket ->
19      % process message 'requestTicket' and loop
20      ticketService(KnownRebecs, StateVars)
21  end.
```

Listing 3. Pseudo Erlang code capturing the behaviour of the ticketService process.

A message server is translated into a match expression (see Figure 3), which is used inside **`receive ... end`**. In Listing 3, `requestTicket` is the pattern that is matched on, and the body of the message server is mapped to the corresponding expression.

Message send is implemented depending on whether **`after`** is used. If there is no **`after`**, the message is sent like a regular message using the `!` operator, as shown on line 4 in Listing 4. However, if the keyword **`after`** is present a new process is spawned which sleeps for the specified amount of time before sending the message as described before. Setting a deadline for the delivery of a message is possible by changing the value `inf`, which denotes no deadline (as shown on line 3 in Listing 4), to an absolute point in time. Messages are tagged with the time at which they were sent. For the simulation we use the system clock to find out the current time by calling the Erlang function **`now()`**.

Moreover, since message servers can reply to the sender of the message, we need to take care of setting the sender as part of the message as seen on line 4 in Listing 4.

```
1  Sender = self(),
2  spawn(fun() ->
3    receive after 15 ->
```

```
4       TicketService ! {{Sender, now(), inf}, requestTicket}
5     end
6   end)
```

Listing 4. Example of a message send after 15 time units in Erlang.

As there is no pattern to match with, the *delay* statement is implemented as a receive consisting of just a timeout that makes the process wait for a certain amount of time. For example, *delay*(10) is translated to `receive after 10 ->ok end`.

The *deadline* of each message is checked right before the body of the message server is executed. The current time is compared with the deadline of the message to see if the deadline has expired and, if so, the message is purged.

4 Simulation of Timed Rebeca Using McErlang

In this section, we present experimental results for two case studies. The first case study is the ticket service model displayed in Listing 1 and the second is a model of a sensor network. In each case we run a simulation for 30 minutes or until a runtime monitor fails, which means that an erroneous state has been reached. The simulations are run in a setting in which a time unit is 1000 ms.

Ticket Service The ticket service model is described in Section 2. For each simulation, we change one of the following parameters: the amount of time that is allowed to pass before a request is processed, the time that passes before agent checks if he has been issued a ticket, the amount of time that passes before agent tries the next ticket service if he did not receive a ticket, the amount of time that passes before agent restarts the ticket requests in case neither ticket service issued a ticket and two different service times, which are non-deterministically chosen as delay time in a ticket service and model the processing time for a request. Table 4 shows different settings of those parameters for which the ticket services never issue a ticket to the agent because of tight deadlines, as well as settings for which a ticket is issued during a simulation of the model.

Sensor Network We model a simple sensor network using Timed Rebeca. (See Listing 5 in Appendix B for the complete description of the model.) A distributed sensor network is set up to monitor levels of toxic gasses. The sensor rebecs (`sensor0` and `sensor1`), announce the measured value to the admin node (`admin` rebec) in the network. If the admin node receives reports of dangerous gas levels, it immediately notifies the scientist (`scientist` rebec) on the scene about it. If the scientist does not acknowledge the notification within a given time frame, the admin node sends a request to the rescue team (`rescue` rebec) to look for the scientist. The rescue team has a limited amount of time units to reach the scientist and save him.

The rebecs `sensor0` and `sensor1` will periodically read the gas-level measurement, modelled as a non-deterministic selection between `GAS_LOW` and

Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2	Result
2	1	1	1	3,4	7	No ticket issued
2	2	1	1	4	7	No ticket issued
2	2	1	1	3	7	Ticket issued

Table 2. Experimental simulation results for ticket service.

Network delay	Admin period	Sensor 0 period	Sensor 1 period	Scientist deadline	Rescue deadline	Result
1	4	2	3	2	3	Mission failed
1	4	2	3	2	4	Mission success
2	1	1	1	4	5,6,7	Mission failed
2	4	1	1	4	7	Mission success

Table 3. Experimental simulation results for sensor network.

`GAS_HIGH`, and send their values to `admin`. The `admin` continually checks, and acts upon, the sensor values it has received. When the `admin` node receives a report of a reading that is life threatening for the `scientist` (`GAS_HIGH`), it notifies him and waits for a limited amount of time units for an acknowledgement. The `rescue` rebec represents a rescue team that is sent off, should the `scientist` not acknowledge the message from the `admin` in time. We model the response speed of the rescue team with a non-deterministic delay of 0 or 1 time units. The `admin` keeps track of the deadlines for the `scientist` and the `rescue` team as follows:

- the `scientist` must acknowledge that he is aware of a dangerous gas-level reading before *scientistDeadline* time units have passed;
- the `rescue` team must have reached the `scientist` within *rescueDeadline* time units.

Otherwise we consider the mission failed.

The model can be parameterized over the values of network delay, `admin` sensor-read period, `sensor0` read period, `sensor1` read period, scientist reply deadline and rescue-team reply deadline, as shown in Table 4. In that table, we can see two different cases in which we go from mission failure to mission success between simulations. In the first scenario, we go from mission failure to success as we increase the rescue deadline, as expected. In the second scenario, we changed the parameters to model a faster sensor update and we observed mission failure. In this scenario, increasing the rescue deadline further (from 5 to 7) is insufficient. Upon closer inspection, we observe that our model fails to cope with the rapid sensor updates and admin responses because it enters an unstable state. The admin node initiates a new rescue mission while another is still ongoing, eventually resulting in mission failure. This reflects a design flaw in the model for frequent updates that can be solved by keeping track of

an ongoing rescue mission in the model. Alternatively, increasing the value of `admin` sensor-read period above half the rescue deadline eliminates the flaw and the simulation is successful again.

5 Future Work

The work reported in this paper paves the way to several interesting avenues for future work. In particular, we have already started modelling larger real-world case studies and analyzing them using our tool. We plan to explore different approaches for model checking Timed Rebeca models. It is worth noting that the translation from Timed Rebeca to Erlang immediately opens the possibility of model checking untimed Rebeca models using McErlang. This adds yet another component to the verification toolbox for Rebeca, whose applicability needs to be analyzed via a series of benchmark examples. As mentioned in the paper, McErlang supports the notion of time only for simulation and not in model checking, and therefore cannot be used as is for model checking Timed Rebeca models. We plan to explore different ways in which McErlang can be used for model checking Timed Rebeca. One possible solution is to store the local time of each process and write a custom-made scheduler in McErlang that simulates the way the Timed Rebeca scheduler operates. The formal semantics for Timed Rebeca presented in this paper is also used in another parallel line of work [10]. The aim of that study is to map Timed Rebeca to timed automata [2] in order to use UPPAAL [21] for model checking Timed Rebeca models. The translation from Timed Rebeca to timed automata will be integrated in our tool suite. We are also working on a translation of Timed Rebeca into (Real-time) Maude. This alternative translation would allow designers to use the analysis tools supported by Maude in the verification and validation of Timed Rebeca models. Our long-term goal is to have a tool suite for modelling, executing, simulating, and model checking asynchronous object-based systems using Timed Rebeca.

Acknowledgements The work on this paper has been partially supported by the projects "New Developments in Operational Semantics" (nr. 080039021), "Meta-theory of Algebraic Process Theories" (nr. 100014021) and "Timed Asynchronous Reactive Objects in Distributed Systems: TARO" (nr. 110020021) of the Icelandic Research Fund.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1990.
2. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte. Lightweight time modeling in Timed Creol. In *RTRTS*, pages 67–81, 2010.
4. F. S. de Boer, T. Chothia, and M. M. Jaghoori. Modular schedulability analysis of concurrent objects in Creol. In *FSEN*, pages 212–227, 2009.

5. Erlang. Erlang Programming Language Homepage. http://www.erlang.org.
6. L.-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, 42(9):125–136, 2007.
7. 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.
8. C. Hewitt. What is commitment? Physical, organizational, and social (revised). In *Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II*, Lecture Notes in Computer Science, pages 293–307. Springer, 2007.
9. ICEROSE. ICEROSE Homepage. http://en.ru.is/icerose/applying-formal-methods/projects/TARO.
10. M. J. Izadi. An actor-based model for modeling and verification of real-time systems - Master Thesis, University of Tehran, Iran, 2010.
11. M. M. Jaghoori, F. de Boer, T. Chothia, and M. Sirjani. Schedulability of asynchronous real-time concurrent objects. *Logic and Algebraic Programming*, 78(5):402–416, 2009. A preliminary version appeared in NWPT/FLACOS 2007 as an extended abstract.
12. M. M. Jaghoori, M. Sirjani, M. R. Mousavi, E. Khamespanah, and A. Movaghar. Symmetry and partial order reduction techniques in model checking Rebeca. *Acta Informaticae*, 47(1):33–66, 2009.
13. G. Kahn. Natural semantics. In F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer-Verlag, 1987.
14. B. Nielsen and G. Agha. Semantics for an actor-based real-time language. In *Proceedings of The Fourth International Workshop on Parallel and Distributed Real-Time Systems (WPDRS'96)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, Apr.
15. L. Nigro and F. Pupo. Schedulability analysis of real time actor systems using coloured petri nets. In *Proc. Concurrent Object-Oriented Programming and Petri Nets*, pages 493–513, 2001.
16. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theor. Comput. Sci.*, 285(2):359–405, 2002.
17. G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, Sept. 1981.
18. S. Ren and G. Agha. RTsynchronizer: Language support for real-time specifications in distributed systems. In *Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 50–59, 1995.
19. M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Model checking, automated abstraction, and compositional verification of Rebeca models. *Journal of Universal Computer Science*, 11(6):1054–1082, 2005.
20. M. Sirjani, A. Movaghar, A. Shali, and F. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informatica*, 63(4):385–410, Dec. 2004.
21. UPPAAL. UPPAAL Homepage. http://uppaal.com.
22. W. Yi. CCS + time = an interleaved model for real time systems. In *Proceedings of ICALP91*, volume 510 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1991.

A Method-Execution Transition Rules

$$(msg)\ (varname.m(\overline{v})\ after(d)\ deadline(DL), \sigma, Env, B) \xrightarrow{\tau} (\sigma, Env, \{(\sigma(varname), m(\overline{eval(v,\sigma)}), \sigma(self), \sigma(now)+d, \sigma(now)+DL)\} \cup B)$$

$$(delay)\ (delay(d), \sigma, Env, B) \xrightarrow{\tau} (\sigma[now = \sigma(now)+d], Env, B)$$

$$(assign)\ (x = e, \sigma, Env, B) \xrightarrow{\tau} (\sigma + [x = eval(e,\sigma)], Env, B)$$

$$(create)\ (varname = new\ O(\overline{v}), \sigma, Env, B) \xrightarrow{\tau} (\sigma[varname = A], \{\sigma_A[now = \sigma(now)]\} \cup Env, \{(A, initial(\overline{eval(v,\sigma)}, \sigma(self)), \sigma(now), +\infty)\} \cup B)$$

$$(cond_1)\ \frac{eval(e,\sigma) = true \quad (S_1, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

$$(cond_2)\ \frac{eval(e,\sigma) = false \quad (S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2,\ \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

$$(seq)\ \frac{(S_1, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B'),\ (S_2, \sigma', Env', B') \xrightarrow{\tau} (\sigma'', Env'', B'')}{(S_1; S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma'', Env'', B'')}$$

Fig. 4: The Method-Execution Transitions Rules. In rule *create*, the rebec name A should not appear in the range of the environment σ. The function *eval* evaluates expressions in a given environment in the expected way.

B Rebeca Model for the Sensor Network

```
1  env int netDelay;
2  env int adminCheckDelay;
3  env int sensor0period;
4  env int sensor1period;
5  env int scientistDeadline;
6  env int rescueDeadline;
7
8  reactiveclass Sensor(3) {
9     knownrebecs {
10       Admin admin;
11    }
12
13    statevars {
```

```
14          int period;
15      }
16
17      msgsrv initial(int myPeriod) {
18          period = myPeriod;
19          self.doReport();
20      }
21
22      msgsrv doReport() {
23          int value;
24          value = ?(2, 4); // 2=safe gas levels, 4=danger gas levels
25          admin.report(value) after(netDelay);
26          self.doReport() after(period);
27      }
28  }
29
30  reactiveclass Scientist(3) {
31      knownrebecs {
32          Admin admin;
33      }
34
35      msgsrv initial() {}
36
37      msgsrv abortPlan() {
38          admin.ack() after(netDelay);
39      }
40  }
41
42  reactiveclass Rescue(3) {
43      knownrebecs {
44          Admin admin;
45      }
46
47      msgsrv initial() {}
48
49      msgsrv go() {
50          int msgDeadline = now() + (rescueDeadline-netDelay);
51          int excessiveDelay = ?(0, 1); // unexpected obstacle might occur during rescue
52          delay(excessiveDelay);
53          admin.rescueReach() after(netDelay) deadline(msgDeadline);
54      }
55  }
56
57  reactiveclass Admin(3) {
58      knownrebecs {
59          Sensor sensor0;
60          Sensor sensor1;
61          Scientist scientist;
62          Rescue rescue;
63      }
64
65      statevars {
66          boolean reported0;
67          boolean reported1;
68          int sensorValue0;
69          int sensorValue1;
70          boolean sensorFailure;
71          boolean scientistAck;
72          boolean scientistReached;
73          boolean scientistDead;
74      }
75
76      msgsrv initial() {
77          self.checkSensors();
78      }
79
80      msgsrv report(int value) {
81          if (sender == sensor0) {
```

```
82              reported0 = true;
83              sensorValue0 = value;
84          } else {
85              reported1 = true;
86              sensorValue1 = value;
87          }
88      }
89
90      msgsrv rescueReach() {
91          scientistReached = true;
92      }
93
94      msgsrv checkSensors() {
95          if (reported0) reported0 = false;
96          else sensorFailure = true;
97
98          if (reported1) reported1 = false;
99          else sensorFailure = true;
100
101         boolean danger = false;
102         if (sensorValue0 > 3) danger = true;
103         if (sensorValue1 > 3) danger = true;
104
105         if (danger) {
106             scientist.abortPlan() after(netDelay);
107             self.checkScientistAck() after(scientistDeadline); // deadline for the scientist to
                    answer
108         }
109
110         self.checkSensors() after(adminCheckDelay);
111     }
112
113     msgsrv checkRescue() {
114         if (!scientistReached) {
115             scientistDead = true; // scientist is dead
116         } else {
117             scientistReached = false;
118         }
119     }
120
121     msgsrv ack() {
122         scientistAck = true;
123     }
124
125     msgsrv checkScientistAck() {
126         if (!scientistAck) {
127             rescue.go() after(netDelay);
128             self.checkRescue() after(rescueDeadline);
129         }
130         scientistAck = false;
131     }
132 }
133
134 main {
135     Sensor sensor0(admin):(sensor0period);
136     Sensor sensor1(admin):(sensor1period);
137     Scientist scientist(admin):();
138     Rescue rescue(admin):();
139     Admin admin(sensor0, sensor1, scientist, rescue):();
140 }
```

Listing 5. A Timed Rebeca model of the sensor network example