Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca

Luca Aceto[a], Matteo Cimini[a], Anna Ingolfsdottir[a], Ali Jafari[a], Arni Hermann Reynisson[a], Steinar Hugi Sigurdarson[a], Marjan Sirjani[a,b]

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

Abstract

In this paper we propose Timed Rebeca as an extension of the Rebeca language that can be used to model distributed and asynchronous systems with timing constraints. Timed Rebeca restricts the modeller to a pure asynchronous actor-based paradigm, where the structure of the model represents the service oriented architecture, while the computational model matches the network infrastructure. The modeller can specify both computational and network delay, and assign deadlines for serving a request. We provide the formal semantics of the language using Structural Operational Semantics, and show its expressiveness by means of examples. We developed a tool for 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. The results of the simulations can then be employed to select the most appropriate values for the parameters in the model. Simulation is shown to be an effective analysis support, specially where model checking faces almost immediate state explosion in an asynchronous setting.

1. Introduction

This paper presents an extension of the actor-based Rebeca language [1, 2] 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 [3] as an agent-based language, and is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation [4]. 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 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 some of the dominating applications, such as multi-core programming and web services [5].

Reactive Objects Language, *Rebeca* [1], 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 [6, 7, 1]. With its simple, message-driven and object-based computational model, Java-like syntax, and accompanying set of verification tools, Rebeca is an interesting and easy-to-learn model for practitioners.

Motivation and Contribution. 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. In this work we present

- Timed Rebeca by extending Rebeca with time constraints,

- the formal semantics of Timed Rebeca using Structural Operational Semantics (SOS) [8],
- a tool for mapping Timed Rebeca models to Erlang,
- examples of applications of Timed Rebeca to different small and medium sized case studies, and
- experimental results from the simulation of the resulting Timed Rebeca models using McErlang [9].

The main contribution of this work is offering a pure asynchronous actor-based modelling language with timing primitives and analysis support. 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.

This paper is an extended version of [10]. The main extensions are presenting a formal mapping as a syntax-directed translation from Timed Rebeca to Erlang, and a medium-sized case study where we model and simulate the BitTorrent [11] protocol using Timed Rebeca and McErlang [9].

Comparison with other timed models. Comparing with the well-established timed models, like timed automata [12], TCCS [13], and real-time Maude [14], Timed Rebeca offers an actor-based syntax and a built-in actor-based computational model, which restricts the style of modelling to an event-based concurrent object-based paradigm. Modelling time-related features in computational models has been studied for a long time [15, 12]; while we have no claims of improving the expressiveness of timed models, we believe that our model is highly usable due to its actor-based nature and Java-like syntax. The usability is due to the one to one correspondence between the entities of the real world and the objects in the model, and the events and actions of the real world and the computational model. Moreover, the syntax of the language is familiar for software engineers and practitioners.

Comparison with other timed actor models. We know of a few other timed actor-based modelling languages [16, 17, 18] that we will discuss in more detail in Section 6, where we discuss further related work. In [16] a central synchronizer acts like a coordinator and enforces the real-time and synchronization constraints (called interaction constraints). A language for the coordinated actors is briefly proposed in [17]; however, the main focus is having reusable real-time actors without hardwired interaction constraints. The constraints declared within the central synchronizer in this line of work can be seen as the required global properties of a Timed Rebeca model. We capture the architecture and configuration of a system via a Timed Rebeca model and then we can check whether the global constraints are satisfied. The language primitives that we use to extend Rebeca are consistent with the proposal in [17]. The primitives proposed in [18] are different from ours; that paper introduced an *await* primitive whereas we keep the asynchronous nature of the model.

Analysis support. In order to analyze Timed Rebeca models, we developed a tool to facilitate their simulation. In a parallel project [19], a mapping from Timed Rebeca to timed automata is developed and UPPAAL [20] is used for model checking. The asynchronous nature of Rebeca models causes state explosion while model checking even for small models. One solution is using a modular approach like in [21]. Here, we selected an alternative solution as a complementary tool for analysis. Using our tool we can translate a Timed Rebeca model to Erlang [22], set the parameters which represent the environment and component variables, and run McErlang [9] to simulate the model. The tool allows us to change the settings of different timing parameters and rerun the simulation in order to investigate different scenarios, find potential bugs and problems, and optimize the model by manipulating the settings. 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 our experiments we could find timing problems that caused missing a deadline, or an unstable state in the system.

The formal semantics presented in this paper is the basis for the correct mapping from Timed Rebeca to Erlang. The tool together with some examples can be found at [23].

Our choice to use the actor-based programming language Erlang is also based on the idea of covering the whole life cycle of the system in the future, and of providing a refinement step for implementing the code from our Timed Rebeca model.

Overview of the paper. In Section 2 we explain Timed Rebeca, the timing features that we have considered in designing the language, and its formal semantics. In Section 3 we present the mapping from Timed Rebeca to Erlang, and in Section 4 we show our case studies and simulation results. Section 5 includes our medium-size example, the BitTorrent protocol. We discuss the related work and specially timed actor-based models in Section 6. Finally, we conclude and sketch our future work in Section 7.

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 [1].

Timing features in an asynchronous and distributed setting. To decide on the timing primitives to be added to the Rebeca syntax, we first considered the different timing features that a modeller might need to address in a message-based, asynchronous and distributed setting. These features (like the computation time, or periodic events) can be common in any setting.

1. **Computation time**: the time needed for a computation to take place.
2. **Message delivery time**: the time needed for a message to travel between two objects, which depends on the network delay (and possibly other parameters).

3. **Message expiration**: the time within which a message is still valid. The message can be a request or a reply to a request (that is, a request being served).
4. **Periods of occurrences of events**: the time periods for periodic events.

We introduce an extension of Rebeca with real-time primitives to be able to address the above-mentioned timing features. In a Timed Rebeca model, each rebec has its own local clock, which can be considered as synchronized distributed clocks[1]. Methods are still executed non-preemptively, and we 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. 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 (computation time), and *now* returns the local time of the rebec. The keywords *after* and *deadline* can only be used in conjunction with a method call. Each rebec knows about its local time and can put *deadline* on the messages that are sent declaring that the message will not be valid after the deadline (modelling the message expiration). The *after* primitive, attached to a message, can be used to declare a constraint on the earliest time at which the message 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 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 set to the current local time of the sender rebec (value of the *now* of the sender rebec) when the message is sent. If the message is augmented with an *after* then the value of the argument of the *after* will be added to the time tag. The intuition is that a message cannot be taken (served) before the time that the time tag determines.

[1]In this paper we do not address the problem of distributed clock synchronization; several options and protocols for establishing clock synchronization in a distributed system are discussed in the literature, including [24].

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 non-preemptively, 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 has to be adjusted. If the time tag of the received message is greater than the value of the current local time (*now*) of the receiver then *now* will be increased to the time tag of the received message. The value of *now* of the rebec 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, 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 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 no longer valid and is purged from the bag. Deadlines are used to model message expirations (timeouts).

- **After**: *r.m() after(t)*, where *r* denotes a rebec name, *m* denotes a method name of *r* and *t* is a natural number, means that the message *m* is sent to the rebec *r* and is put in the message bag. The message cannot be

$$
\begin{aligned}
\mathit{Model} &::= \mathit{EnvVar}^{*}\ \mathit{Class}^{*}\ \mathit{Main}\\
\mathit{EnvVar} &::= \mathbf{env}\ T\ \langle v\rangle^{+};\\
\mathit{Main} &::= \mathbf{main}\ \{\ \mathit{InstanceDcl}^{*}\ \}\\
\mathit{InstanceDcl} &::= C\ r(\langle r\rangle^{*}) : (\langle c\rangle^{*});\\
\mathit{Class} &::= \mathbf{reactiveclass}\ C\ \{\ \mathit{KnownRebecs}\ \mathit{Vars}\ \mathit{MsgSrv}^{*}\ \}\\
\mathit{KnownRebecs} &::= \mathbf{knownrebecs}\ \{\ \mathit{VarDcl}^{*}\ \}\\
\mathit{Vars} &::= \mathbf{statevars}\ \{\ \mathit{VarDcl}^{*}\ \}\\
\mathit{VarDcl} &::= T\ \langle v\rangle^{+};\\
\mathit{MsgSrv} &::= \mathbf{msgsrv}\ M(\langle T\ v\rangle^{*})\ \{\ \mathit{Stmt}^{*}\ \}\\
\mathit{Stmt} &::= v = e;\ |\ r = \mathit{new}\ \mathrm{C}(\langle e\rangle^{*});\ |\ \mathit{Call};\ |\ \mathit{if}\ (e)\ \{\ \mathit{Stmt}^{*}\ \}\ [\mathit{else}\ \{\ \mathit{Stmt}^{*}\ \}]\ |\\
&\quad \mathbf{delay}(t);\ |\ \mathbf{now}();\\
\mathit{Call} &::= r.M(\langle e\rangle^{*})\ [\mathbf{after}(t)]\ [\mathbf{deadline}(t)]
\end{aligned}
$$

Figure 1: Abstract syntax of Timed Rebeca. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition at least once, superscript $*$ for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. Identifiers *C*, *T*, *M*, *v*, *c*, and *r* denote class, type, method, variable, constant, and rebec names, respectively; and *e* denotes an (arithmetic, boolean or nondetermistic choice) expression.

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, and also periodic events.

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 for that purpose. 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.

Note that a set of environment variables are defined on the first line of the model. Timed Rebeca models can be made parametric by defining a set of environment variables at the top of the model code file. The modeller can set values for the variables when starting a simulation and inspect different behaviours of the model with regards to different time parameters.

```
env int requestDeadline, checkIssuedPeriod, retryRequestPeriod, newRequestPeriod,
    serviceTime1, serviceTime2;

reactiveclass Agent {
  knownrebecs { TicketService ts1; TicketService ts2; }
  statevars { int attemptCount; boolean ticketIssued; int token; }
  msgsrv initial() { self.findTicket(ts1); } // initialize system, check 1st ticket service
  msgsrv findTicket(TicketService ts) {
    attemptCount += 1; token += 1;
    ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
    self.checkTicket() after(checkIssuedPeriod); // check if a reply to the request has been
        received
  }
  msgsrv ticketIssued(int tok) { if (token == tok) { ticketIssued = true; } }
  msgsrv checkTicket() {
    if (!ticketIssued && attemptCount == 1) {        // no ticket from 1st service,
      self.findTicket(ts2);                          // try the second TicketService
    } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
      self.retry() after(retryRequestPeriod);        // restart from the first TicketService
    } else if (ticketIssued) {                       // the second TicketService replied,
      ticketIssued = false;
      self.retry() after(newRequestPeriod);          // new request by a customer
    }
  }
  msgsrv retry() {
    attemptCount = 0; self.findTicket(ts1);      // restart from the first TicketService
  }
}

reactiveclass TicketService {
  knownrebecs { Agent a; }
  msgsrv initial() { }
  msgsrv requestTicket(int token) {
    int wait = ?(serviceTime1,serviceTime2);    // the ticket service sends the reply
    delay(wait);                                // after a non-determinstic delay of
    a.ticketIssued(token);                      // either serviceTime1 or serviceTime2
  }
}

main {
  Agent a(ts1, ts2):();              // instantiate agent, with two known rebecs
  TicketService ts1(a):();           // instantiate 1st and 2nd ticket services, with
  TicketService ts2(a):();           // the agent as their known rebecs
}
```

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 [8]. The behaviour of Timed Rebeca programs is given by the transition relation $\rightarrow$, which describes the stepwise evolution of a system. The definition of the relation $\rightarrow$ relies in turn on the transition relation $\xrightarrow{\tau}$, which expresses the effect of the atomic execution of methods.

Form of the Transition Relations. Steps in the transition relation $\rightarrow$ take the form

$$(Env, B) \rightarrow (Env', B').$$

The states of the system are pairs of the form (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 *self*, the name of the rebec, *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 of A to its body.

The bag B 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 general form of the steps in the transition relation $\xrightarrow{\tau}$ is

$$(S, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B').$$

A single step of $\xrightarrow{\tau}$ consumes all the body S of the executed method and provides the value resulting from its execution starting from its environment σ in the context of the system state (Env, B). As the rules defining $\xrightarrow{\tau}$ will make clear, having the bag B as a component of configurations 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 notion of configuration used above, 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'.

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

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

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

Definition of $\xrightarrow{\tau}$. The transition relation $\xrightarrow{\tau}$ describes the execution of methods in the style of natural semantics [25]. 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.

Figure 2 shows the SOS rules defining $\xrightarrow{\tau}$. 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. Moreover, $eval(\overline{v}, \sigma)$ indicates the tuple of actual parameters that results by evaluating the tuple of expressions $\overline{v}$ in the environment σ.

- 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 (we omit the rules that accomplish this). A message is put in the bag in order to execute the *initial* method of the newly created rebec. This message can be executed immediately and never expires.

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

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

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

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

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

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

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

Figure 2: The Method-Execution Transitions Rules. In rule *create*, the rebec name A should be fresh, that is, it should not appear in *Env*. The function *eval* evaluates expressions in a given environment in the expected way. In each rule, we assume that σ is not contained in *Env*.

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 [22]. The motivation for translating Timed Rebeca models to Erlang code is to be able to use McErlang [9] 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.

In what follows, we recapitulate briefly on the Erlang programming language and we describe the translation. We reserve Section 3.1 for a formal account of the encoding. The reader can find an implementation of it at [23].

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 and timing 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.
- `after` is used in conjunction with a `receive` and is followed by a timeout block as shown in Listing 2, after the specified time (deadline for receiving the required pattern) the process executes the timeout block.

```
receive
  Pattern1 when Guard1 -> Expr1;
  Pattern2 when Guard2 -> Expr2;
  ...
after
  Time -> Expr
end
```

Listing 2: Syntax of a receive with timeout.

- `erlang:now()` returns the current time of the process.

When a process reaches a **`receive`** expression it looks in the queue and takes a 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.

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.

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.

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

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

Figure 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. Note that {} and [] are parts of the syntax of Erlang representing tuples and lists, respectively.

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 in the timeout block of a receive with an empty pattern, the timeout block is always executed, sending the message after the specified time
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 Erlang

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

```
ticketService() ->
  receive
    % wait for a message with a set of known rebecs
    {Agent} ->
      % proceed to the next behaviour
      ticketService(#ticketService_knownrebecs{agent=Agent})
  end.
ticketService(KnownRebecs) ->
  receive
    % wait for the 'initial' message
    initial ->
      % process message 'initial' and proceed to the next behaviour
      ticketService(KnownRebecs, #ticketService_statevars{})
  end.
ticketService(KnownRebecs, StateVars) ->
  receive
    % wait for each message server
    requestTicket ->
      % process message 'requestTicket' and loop
      ticketService(KnownRebecs, StateVars)
  end.
```

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

```
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.

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 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.

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.

3.1. The mapping, formally

In Figure 4, we provide the abstract syntax on which our encoding is based. The encoding takes the form of a syntax-directed translation from Timed Rebeca to Erlang. The abstract syntax that we give in Figure 4 rephrases the one defined in Figure 1 of Section 2 in order to better support a recursive translation of the language[2].

[2]The abstract syntax in Figure 1 makes use of features of Extended BNF grammars such as the superscript $*$ to indicate the repetition of zero or more times of a syntactic entity. Although this allows for a compact presentation of the language, that formulation is less suited for a syntax-directed translation.

$$
\begin{aligned}
\textit{Model} &::= r_1 \ldots r_i\ m \\
\textit{Main} &::= \texttt{main}\{d_1 \ldots d_i\} \\
\textit{InstanceDcl} &::= t_r v_r\ \texttt{(}t_1 v_1 \ldots t_i v_i\texttt{)}\texttt{:}\texttt{(}k_1 \ldots k_j\texttt{)} \\
\textit{Class} &::= \texttt{reactiveclass}\ n\ \{k\ s\ in\ m_1\ m_2 \ldots m_i\} \\
\textit{KnownRebecs} &::= \texttt{knownrebecs}\ \{t_1 v_1 \ldots t_i v_i\} \\
\textit{Vars} &::= \texttt{statevars}\ \{t_1 v_1 \ldots t_i v_i\} \\
\textit{MsgSrv} &::= \texttt{msgsrv initial(}t_1 v_1 \ldots t_i v_i\texttt{)}\ \{s_1 \ldots s_j\}\ | \\
&\quad \texttt{msgsrv}\ m\texttt{(}t_1 v_1 \ldots t_i v_i\texttt{)}\ \{s_1 \ldots s_j\} \\
\textit{Stmt} &::= v = e;\ |\ r.m(e_1 \ldots e_i);\ | \\
&\quad \texttt{delay(}e\texttt{);}\ |\ r.m(e_1 \ldots e_i)\ \texttt{after(}e_a\texttt{)}\ \texttt{deadline(}e_d\texttt{)};
\end{aligned}
$$

Figure 4: Unfolded version of the abstract syntax in Figure 1. The non-terminals on the righthand side are replaced with variables to make the explanation of the mapping easier.

Moreover, only relevant syntactic categories are taken into account in this section; for instance, the encoding of *if* statements and of many expressions are not shown since their implementation is trivial or subsumed by the encoding of other constructs. The meaning of the meta-variables that occur in the abstract syntax below is made clear later when a description of the various mappings involved in the translation is given.

We provide the encoding by means of a number of mappings, one for each syntactic category of Timed Rebeca. Some of the encodings are parametrized by the information contained in a structure we call *conf*. A structure *conf* is associated with each rebec and contains three fields: *knownrebecs*, *statevar* and *localvars*. Intuitively, the field *knownrebecs* contains the set of known rebecs, and *statevar* and *localvars* contains the set of variable names for both state and local variables. In our encodings we use the standard dot notation in order to access the structure *conf*, e.g. *conf.statevar* is used to access the field *statevar* of the structure *conf*. Before showing the formal encoding, a brief description of the mappings involved in the encoding is in order.

$$
\begin{array}{lll}
\mathcal{MO}[\![r_1 \ldots r_i\ m]\!] = & \mathcal{R}[\![r_1]\!] & conf_1 \\
& \vdots & \\
& \mathcal{R}[\![r_i]\!] & conf_i \\
& \mathcal{M}[\![m]\!] &
\end{array}
$$

Figure 5: Encoding of models.

$\mathcal{MO}[\![r_1 \ldots r_i\ m]\!]$: **Encoding for** *Model***.** Here $r_1 \ldots r_i$ are rebecs and m is the code in *main*. This function initializes the structure *conf* for each rebec as a preprocessing step and encodes each rebec passing this structure as parameter. Then, it encodes the code in *main*. Figure 5 presents the encoding of models. The reader should bear in mind that the mapping $\mathcal{MO}$ calculates the structure $conf_i$ for each rebec r_i in a preprocessing step.

$\mathcal{R}[\![$**`reactiveclass`** $n\ \{k\ s\ in\ m_1\ m_2 \ldots m_i\}]\!]$**: Encoding for** *Class***.** Here n is the name of the reactive class, k is a sequence of knownrebecs, s is a sequence of state variables, in is the initial method and $m_1 \ldots m_i$ are methods. This function encodes the reactive class in three Erlang functions with same name. These functions accept different formal parameters and they have different signatures, i.e. they are considered as different functions by Erlang.

1. The first Erlang function accepts the known rebecs and calls the second function.
2. The second function accepts the initial message and, once arrived, it runs the corresponding code obtained by the mapping $\mathcal{B}$. The mapping $\mathcal{B}$ returns a new set of state variables; indeed variables might have been changed during the execution of an *inital* method. Since structures are immutable in Erlang, they cannot be modified directly in our encoding. To overcome this difficulty we follow a standard solution. We create a new structure and return it as value. After the execution of the code for *initial*, this function calls the third function, passing to it this new set of state variables as well as the set of known rebecs.
3. The third function waits for incoming messages, which correspond to method calls. Once a message has arrived, the function runs the corresponding code obtained again with the mapping

```
$\mathcal{R}[\![$reactiveclass $n$ $\{k\ s\ in\ m_1\ m_2 \ldots m_i\}]\!]\ conf$ =
  $n$ () ->
    receive
      {$\mathcal{B}[\![k]\!]\ conf$} ->
        $n$ (#$conf.knownrebecs${$\mathcal{B}[\![k]\!]\ conf$})
    end.
  $n$ (KnownRebecs) ->
    StateVars=#$conf.statevars${},
    LocalVars=#$conf.localvars${},
    {NewStateVars, _} = receive
      $\mathcal{B}[\![in]\!]\ conf$
    end.
    $n$(KnownRebecs, NewStateVars)
  $n$ (KnownRebecs,StateVars) ->
    LocalVars=#$conf.localvars${},
    {NewStateVars, _} = receive
      $\mathcal{B}[\![m_1]\!]\ conf$
      ⋮
      $\mathcal{B}[\![m_i]\!]\ conf$
    end.
    $n$(KnownRebecs, NewStateVars)
```
Figure 6: Encoding of rebecs.

$\mathcal{B}$. After this, it is ready to accept again messages by function calling itself, but with the modified set of state variables. Indeed variables might have been changed during the execution of a method.

The reader can find the encoding of rebecs in Figure 6.

$\mathcal{B}[\![$**msgsrv** $m(t_1v_1 \ldots t_iv_i)\{s_1 \ldots s_j\}]\!]$: **Encoding for** *MsgSrv*. Here m is the name of the method, $t_1 \ldots t_i$ are type names, $v_1 \ldots v_i$ are identifiers, and $s_1 \ldots s_j$ are the statements of the body of the method. This function makes use of pattern maching in order to wait for a message of the form

$$\{Sender, TT, DL, m, w_1 \ldots w_i\}.$$

```
𝓑⟦msgsrv m(t₁v₁ ... tᵢvᵢ){s₁ ... sⱼ}⟧ conf =
  {{Sender, TT, DL, m, w₁ ... wᵢ} ->
    TimeNow = tr_now(),
    if
      (DL == ok orelse TimeNow < DL)) ->
        {NewStateVars, _} = AP(𝒮⟦s₁⟧ conf ... 𝒮⟦sⱼ⟧ conf),
        n (KnownRebecs,NewStateVars);
    true ->
      % dropping message
    end
```

Figure 7: Encoding of message servers.

These messages are basically the messages that are exchanged within the actual Timed Rebeca. When such a message has arrived, we check if the time tag of the message is not higher than the current time, and also if the deadline of the message has not expired. If the message has not expired, we execute the statements of the method, otherwise a null action is executed. A few peculiarities of this encoding deserve some explanation.

- Performing a null action corresponds to what in Timed Rebeca is the discarding of the message. Indeed, in the Erlang system, the message has been delivered and it will be not processed again.
- The execution of statements is quite involved. Indeed, structures are immutable in Erlang, but in Timed Rebeca the execution of some statements might change variable values. This means that subsequent statements should be executed knowing the new values for variables. Our solution is to execute every statement as a function that receives the set of variables as argument, and returns a new set of variable. The auxiliary function AP takes care of correctly composing these functions.
- The reader might wonder why we discard also messages whose time tag is higher than the current time. We indeed have seen that the processing of this type of messages must be simply postponed. To understand this, the reader should consider that in order to encode a Timed Rebeca **after** construct, we rely on the Erlang **after** construct, which would actually send the message

only at the right time. The Erlang system takes care of this aspect for us.

- The function *tr_now*() recovers the current time from the Erlang primitive *erlang_now*().

We present the encoding of message servers in Figure 7.

$\mathcal{S}$**: Encoding of** *Stmt***.** This function encodes statements from Timed Rebeca into anonymous functions in Erlang. Functions receive as input the set of values and return a new set of values. The two relevant cases to discuss are the method invocation when it involves **`after`** and `deadline` constructs, and the *delay* statement. In what follows we consider in detail only these two cases.

- $\mathcal{S}[\![$`r.m(`$e_1 \ldots e_i$`) after(`a`) deadline(`d`);`$]\!]$: This function creates a new process using the Erlang primitive *spawn*. This new process makes use of a receive statement with an empty body and of the Erlang **`after`** to send the message. As said previously, the Erlang system takes care of the timing aspect for us and the message will be sent only at the proper time. Thanks to the *spawn* primitive the sender process does not stop its execution by the effect of an **`after`** statement, the new process will wait instead. The reader should also notice that the message sent by this new process contains the father process as sender, not itself.
- $\mathcal{S}[\![$`delay(`e`);`$]\!]$: This function simply performs a receive statement with an empty body and an **`after`** call in order to let the specified amount of time pass. A null action is performed afterwards.

In Figure 8 we show the encoding of some selected statements.

other mappings: The mappings $\mathcal{I}$ and $\mathcal{T}$ translate the names of identifiers and types from Timed Rebeca to Erlang. The mapping $\mathcal{E}$ encodes expressions and it is implemented as expected. The mapping $\mathcal{K}$ translates constants from Timed Rebeca to Erlang, while the mapping $\mathcal{M}$ encodes the *main* code using $\mathcal{PC}$ and $\mathcal{L}$ in order to encode instance declarations. Figure 9 shows some of these mappings, namely it presents the encoding of the main functions and instance declarations.

```
$\mathcal{S}[\![v{=}e;]\!]\ conf =$  fun(StateVars, LocalVars) ->
                   $v \in conf.statevars \longrightarrow$
                    {StateVars $conf.statevars2$ = $\mathcal{E}[\![e]\!]conf$, LocalVars}
                   $v \in conf.localvars \longrightarrow$
                    {StateVars, LocalVars $conf.localvars2$ = $\mathcal{E}[\![e]\!]conf$}
                   otherwise $\longrightarrow$ error
                 end
$\mathcal{S}[\![r.m(e_1 \ldots e_i);]\!]\ conf =$  fun(StateVars, LocalVars) ->
                       tr_send($\mathcal{I}[\![r]\!]$,$\mathcal{I}[\![m]\!]$,{$\mathcal{E}[\![e_1]\!] \ldots \mathcal{E}[\![e_i]\!]$}),
                       {StateVars, LocalVars}
                     end
$\mathcal{S}[\![r.m(e_1 \ldots e_i)$ after($a$) deadline($d$);$]\!]\ conf =$
      fun(StateVars, LocalVars) ->
        tr_sendafter($\mathcal{E}[\![e_a]\!]$,$\mathcal{I}[\![r]\!]$,$\mathcal{I}[\![m]\!]$,{$\mathcal{E}[\![e_1]\!]conf \ldots \mathcal{E}[\![e_i]\!]conf$},
                     $\mathcal{E}[\![e_d]\!]$),{StateVars, LocalVars}
      end
$\mathcal{S}[\![$delay($e$);$]\!]\ conf =$  fun(StateVars, LocalVars) ->
                         tr_delay($\mathcal{E}[\![e]\!]conf$),
                         {StateVars, LocalVars}
                       end
```

Figure 8: Encoding of some selected statements.

```
$\mathcal{M}[\![$main{$d_1 \ldots d_i$}$]\!]$ = main() ->
      $\mathcal{PC}[\![d_1]\!]$,
      $\vdots$
      $\mathcal{PC}[\![d_i]\!]$,
      $\mathcal{L}[\![d_1]\!]$,
      $\vdots$
      $\mathcal{L}[\![d_i]\!]$.
$\mathcal{PC}[\![t_r v_r\ (t_1 v_1 \ldots t_i v_i):(k_1 \ldots k_j)]\!]$ = $\mathcal{I}[\![v_r]\!]$ = spawn(fun() -> $\mathcal{I}[\![t_r]\!]$() end)
$\mathcal{L}[\![t_r v_r\ (t_1 v_1 \ldots t_i v_i):(k_1 \ldots k_j)]\!]$ =
      $\mathcal{I}[\![v_r]\!]$ ! {$v_1 \ldots v_i$} end),
      tr_send($\mathcal{I}[\![v_r]\!]$, initial, {$\mathcal{K}[\![k_1]\!] \ldots \mathcal{K}[\![k_i]\!]$})
```

Figure 9: Encoding of the main function and instance declarations.

Moreover, to be able to simulate programs with McErlang, the translated Erlang code is instrumented with calls to the *probe_state* McErlang function at the end of each message server. That way McErlang is notified of state changes for each execution of a message server and its runtime monitors can abort the simulation if it has run into an assertion violation.

The reader should bear in mind that for the sake of the presentation we do not show the whole encoding. However, an implementation of it in Erlang can be found in [23].

4. Simulation of Timed Rebeca Models Using McErlang

In this section, we investigate three case studies. The first case study is a simple communication protocol in which the basic ideas underlying the modelling are presented. The second one is the ticket service model displayed in Listing 1 and the third one is a model of a sensor network. In the two last case studies, we run a simulation for ten times and for each case for 30 minutes. In the case that a runtime monitor fails we stop the simulation. When a runtime monitor fails it means that an erroneous state has been reached. The simulations are run in a setting in which a time unit is 1000 ms. The experimental platform is Macbook 2.0GHz Intel Core 2 Duo - Aluminum 4GB memory, Mac OS X, 10.6.6, and Erlang R13B04.

Runtime Monitors. One of the reasons for using McErlang is to be able to write code that monitors the state of the simulation and either let the simulation continue running or stop the simulation due to an erroneous state or unexpected behaviour in the program. Essentially, McErlang monitors give rise to the ability to write assertions over states. An assertion is typically a check on the value of a variable, and the simulation stops and reports any violation. For example, in one of our case studies an assertion checks for the health status of a scientist. If the assertion becomes false, we can stop the simulation and report a failure of the system.

Event Graph. Before presenting the case studies, we need to explain *event graphs* in which a highly abstracted view of scheduling events can be depicted. Event graphs have a single type of node and two types of edges. The nodes represent events in a system. Edges correspond to the scheduling of other events [26]. Jagged incoming edges denote an initial event. Edges can optionally be associated with a boolean condition for scheduling an event and/or a time delay which means that an event will

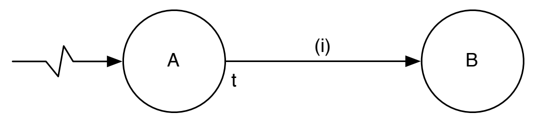


Figure 10: Example of an event graph.

be scheduled after the delay. Figure 10 shows an example of an event graph where event *B* is scheduled by *A* after *t* a delay of t time units and if condition *(i)* is true.

Event graphs are widely used in simulation and analysis of complex systems within the engineering community. More specifically, they are used to represent graphically discrete-event simulation models. We use event graphs in this paper only to give a highly abstracted view of how events are scheduled in our case studies. An event is defined as receiving a message. We adopt an alternative notation where conditional edges are thicker, even if the conditions are not specified [27]. Additionally, we add a label below each node that shows in which reactive class the event occurs. We decided to draw the event graphs based on reactive classes, and not on rebecs, to have a simpler view of the dependencies among events in the system.

Communication Protocol. The simple communication protocol is an example from [28] that consists of a *sender* agent and a *receiver* agent. The sender agent sends a message to the receiver and waits for an acknowledgement. If an acknowledgement is not received within 8 time units the sender resends the message. The receiver agent receives the message and replies with an acknowledgement. A successful outcome is that the sender agent receives an ack message before 8 time units have passed. Communications from the sender agent to the receiver agent takes 3 ± 1 time units and may fail, while a message from the receiver agent to the sender agent takes 2 ± 1 time units and may also fail. This is a simple model where the execution terminates after the sender agent receives the acknowledgement.

Figure 11 shows the event graph of the simple protocol. The graph shows how the system is initialized from the sender agent. Moreover, there is a loop in the graph (start → check ack → start) which tells us that if we select certain time constraints in the model, the model might result in an imbalance situation and an infinite computation. Listing 5 shows the Timed Rebeca code for the model.

```

reactiveclass SenderAgent(3) {
  knownrebecs { ReceiverAgent receiverAgent; }

  statevars { boolean receivedAck; }

  msgsrv initial() { self.start(); }

  msgsrv start() {
    time sendDelay = ?(-1,2,3,4); // -1=fail -- 2,3,4=delays
    if (sendDelay != -1) {
      receiverAgent.send() after(sendDelay);
    }
    self.checkAck() after(8);
  }

  msgsrv ack() { receivedAck = true; }

  msgsrv checkAck() {
    if (!receivedAck) self.start();
  }
}

reactiveclass ReceiverAgent(3) {
  knownrebecs { SenderAgent senderAgent; }

  statevars {}

  msgsrv initial() {}

  msgsrv send() {
    time sendDelay = ?(-1,1,2,3); // -1=fail -- 1,2,3=delays
    if (sendDelay != -1) {
      senderAgent.ack() after(sendDelay);
    }
  }
}

main {
  ReceiverAgent receiverAgent(senderAgent):();
  SenderAgent senderAgent(receiverAgent):();
}
```

Listing 5: A Timed Rebeca model of the simple communication protocol example.

Repeated simulations of the model show us that the model behaves as expected. They reveal that the system sometimes drops messages due to the nondeterministic choice of the network delay and this will execute the loop (start → check ack → start). The loop was not repeated indefinitely, which is not surprising since nondeterministic choices are implemented as random selections in the Erlang mapping.

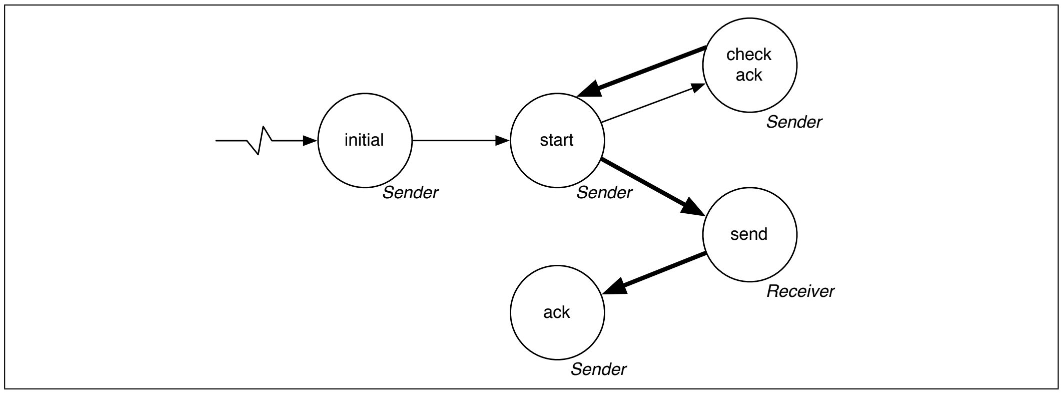


Figure 11: Event graph of the simple communication protocol model.

Ticket Service. The ticket service model is described in Section 2. Figure 12 shows the event graph of the ticket service model. The graph shows how the system is initialized in the Agent class by scheduling a *find ticket* event. The *find ticket* event will always schedule two simultaneous events, *request ticket* and *check ticket*. *Request ticket* event will schedule a *ticket issued* event after a delay. However, there is a network delay on the scheduling of *check ticket*, which means that it may be scheduled later than the *ticket issued* event. In that case, either a *find ticket* event or *retry* event may be scheduled. Notice that the model is reactive, and will continue to schedule the *find ticket* event to start the cycle all over again.

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 2 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 6 in Appendix A 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

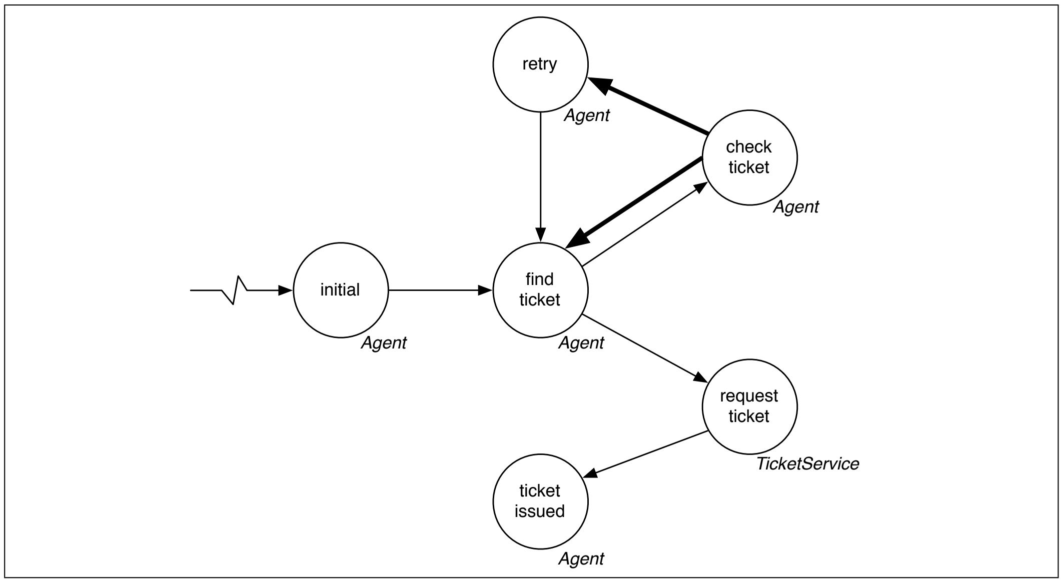


Figure 12: Event graph of the ticket service model.

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	Not issued
2	2	1	1	4	7	Not issued
2	2	1	1	3	7	Ticket issued

Table 2: Experimental simulation results for ticket service.

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.

Figure 13 shows the event graph of the sensor network model. The graph shows how the system is initialized in the Sensor and Admin classes. The sensor class schedules *do report* events and continues to do so throughout the life cycle of the system. The admin class schedules an event that checks the sensor values repeatedly. These events make the model reactive. The *check sensors* event may set off a routine that checks if a scientist has acknowledged the presence of dangerous gas levels. Additionally, it may

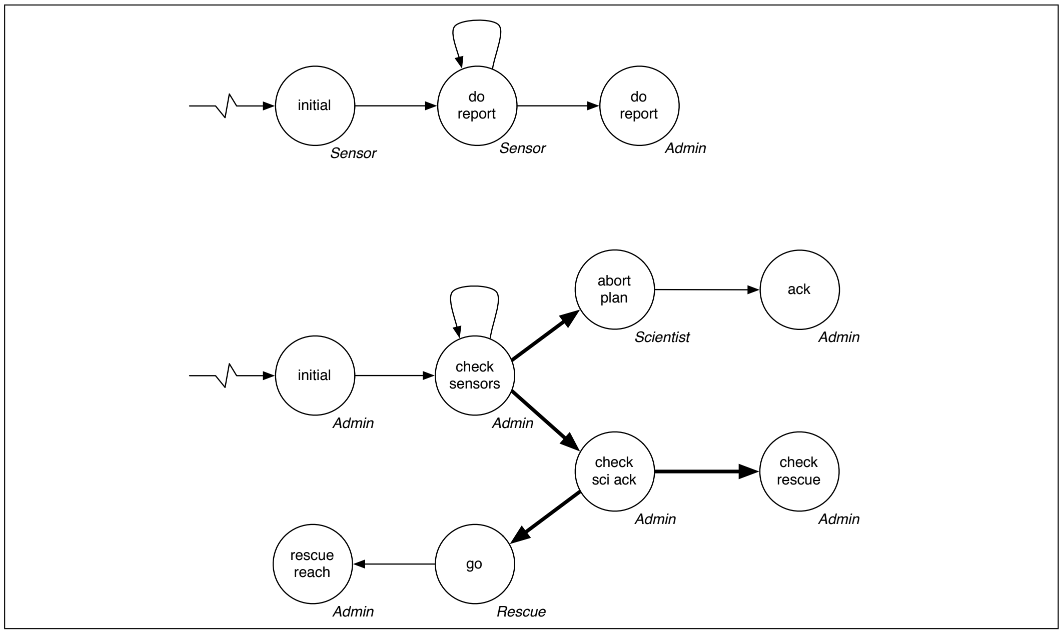


Figure 13: Event graph of the sensor network model. Notice there are two events that start the system.

schedule the scientist to abort whatever is being performed. If the scientist is instructed to abort, he will send an acknowledgement. However, the *ack* event may be scheduled after the *check ack* event. In that case, *go* is scheduled, which will schedule a *rescue reach* event. Alas, the *rescue reach* event may be scheduled after *check rescue*, in which case the scientist would be dead.

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

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-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 3. 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. Implementing a Simplified BitTorrent Protocol in Timed Rebeca

In this section, we first give a brief explanation of the BitTorrent protocol, and then we describe implementing a simplified BitTorrent protocol in Timed Rebeca. Finally, we show the effect of different aspects of the protocol such as file size, network size, network delay, and download to upload ratio on download time by running the model for different scenarios.

BitTorrent Protocol. BitTorrent(BT) is a P2P application that aims at enabling fast and efficient distribution and downloading of large files by using the upload bandwidth of the downloading peers [11, 29]. In BitTorrent, a file is divided into many equal sized shares (typically 256KB) for download, each share is called *chunk*. While downloading the pieces of file, each peer uploads the pieces that it has already acquired to other peers. In a BT network we can distinguish between two classes of nodes: *seeds* and *leechers*. Seeds are peers who have a complete copy of the resource, whereas leechers are nodes who are currently downloading.

The only centralized component of BT is an entity called *tracker*. The tracker is responsible for helping peers find each other and for keeping the download/upload statistics of each peer. In order to download a file, peers download a *.torrent* file from a web server to access the tracker and join the system. A peer contacts the tracker to obtain a list containing a random subset of the peers currently in the system (both seeds and leechers). This peer will establish connection directly to peers in the peer set, which become its neighbours. Each peer looks for opportunities to download pieces from, and upload pieces to, its neighbours in its peer set.

BT employs a *tit-for-tat* rate-based download strategy, which encourages cooperation among peers and guards against free-riding, i.e. consuming resources without contributing to the system. Under this strategy, a peer must upload as well as download at the same time when interacting with another peer. Each peer preferentially uploads to peers from which it has the highest downloading rates. Another vital mechanism employed in BT is the piece selection mechanism. Peers in BitTorrent use a *local rarest first (LRF)* technique to select which share to request. Peers always select to download the rarest pieces within their peer set. It tries to download the pieces that is least replicated among its neighbours. Actually, the LRF strategy is combined with a random choice in the practical deployment.

BitTorrent Protocol in Timed Rebeca. We implement a simplified model of BitTorrent in Timed Rebeca. (See the pseudo code in Listing 7 in Appendix B.) In our model, we abstract the dynamic creation of peers which means all peers constructing a network should be determined at the beginning of the simulation. So, new peers cannot join the network during simulation. Also, joined peers cannot leave the system, except some peers after becoming seeds with a low probability.

Figure 14 depicts the event graph of connecting a peer to the tracker.

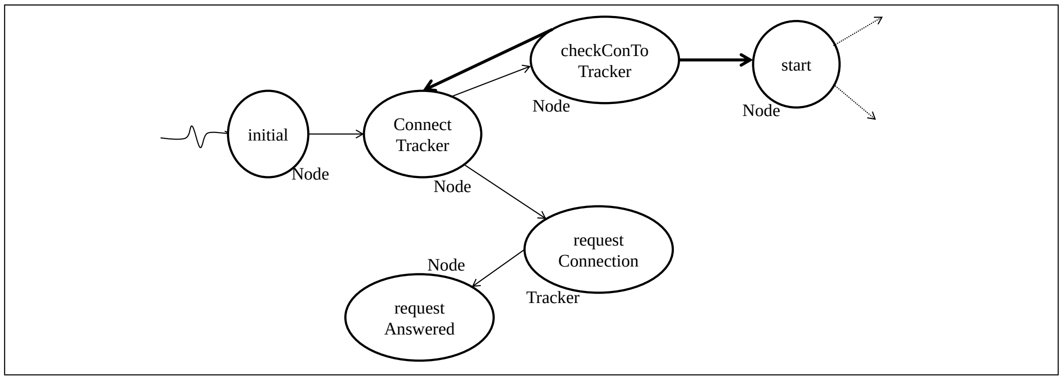


Figure 14: Event graph of connecting a peer to the tracker.

In BitTorrent, when a peer wants to download a file, it first contacts a centralized entity called tracker to get a random list of peers in the system. Our model has a rebec *t1* instantiated from the reactive class *tracker* that shows the tracker in the protocol and *n* rebecs instantiated from the reactive class *peer*. The value of *n* determines the number of peers in the system that are interested in downloading a file. We consider peer *p1* to show how we implement different parts of the BitTorrent protocol in Timed Rebeca. Peer *p*1 is initialized by setting values to some *statevars*, described in Table 4, and sending message *connectTracker* to itself. Then, it sends a connection request to rebec *t1* by sending a message *requestConnection* to it. The deadline for the message to be served is *requestDeadlineForConTracker* time units. Tracker *t1* will respond to this request by sending a message *requestAnswered* to peer *p1* after *networkDelay* time units. This delay represents the time needed for establishing a connection in the network. After *checkConTracker* time units, peer *p1* will check the connection request status by sending *checkContoTracker* to itself. If it has not received any reply from the tracker *t1*, it resends the connection request after *retryConTracker* time units.

Event graph of connecting a peer to other peers in its list is represented in Figure 15. After connecting to the tracker and getting the random list of peers that exist in the system, *p1* should establish a connection to these peers by sending a *start* message to itself. Indeed, in our model peers do not obtain the random list of peers from tracker *t1*; instead we determine it as the *knownrebecs* for each peer in the main part. Also, each peer has a connection degree which is specified in the main part. this illustrates the number of peers in the list to which the connections

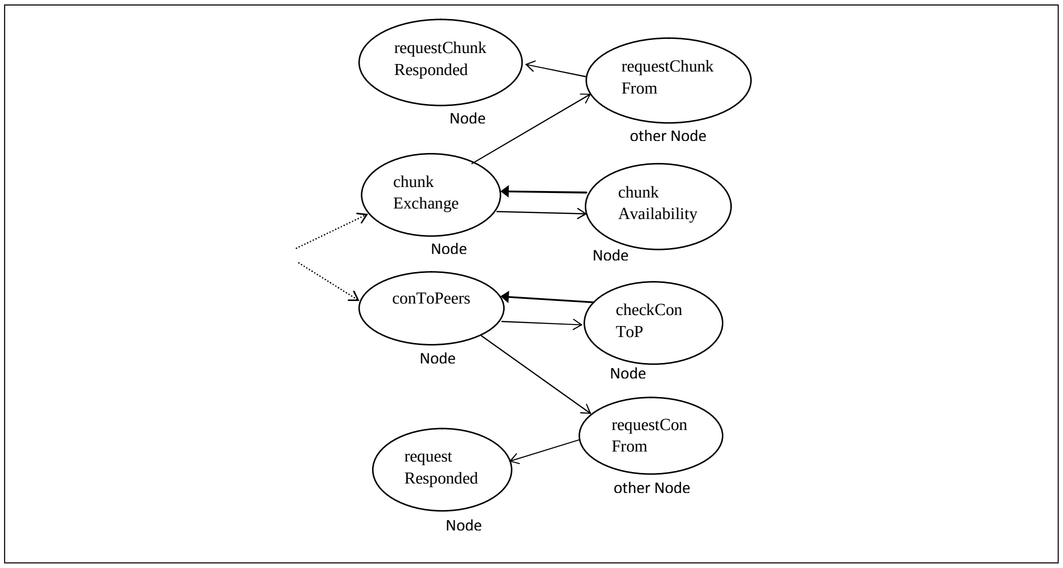


Figure 15: Event graph of connecting to the peers and downloading chunks from them.

should be established. Therefore, *p1* requests connection to each of its peers by sending a message *requestconFrom*. The deadline for the message to be served is *requestDeadlineForConPeer* time units. After *checkConPeer* time units, a peer will check its request status by sending a *checkConToP* message to itself. Each peer has an array *conToP* which is used to save the status of its connections to the other peers. If it has not received any reply from the peer, it will send again its request after *retryConPeer* time units. The other peer responds to *p1* by sending a message *requestResponded* to it. Receiving this message represents the establishment of a connection.

Figure 15 shows the event of requesting chunks from connected peers. After establishing connections, *p1* starts downloading all chunks which it does not have from the connected peers by sending a message *chunkExchange* to itself. After receiving this message, *p1* selects the connected peers non-deterministically and sends downloading requests to them by sending a message *requestChunkFrom* for the chunks that it does not have. When a peer receives a downloading chunk request, if it can reply to it, it will send a message *requestChunkResponded* to peer *p1* after (*chunkSize* × *networkDelay*) time units. This amount of time should elapse as downloading time of a chunk in the network. Peer *p1* investigates the availability of its chunks after *checkChunkPeriod* time units by sending a message *checkChunkAvail-*

chunk	Each item of the array shows chunk availability
contoTracker	The status of connectivity to the tracker
conToP	Each item of the array shows the status of connectivity to neighbours
peerDegree	Number of peers to which a connection should be established
downloadBandwidth	Download bandwidth in terms of chunk
uploadBandwidth	Upload bandwidth in terms of chunk
isFreeRider	When it is set to true, the peer acts as a free-rider
leaveSysAfterBecomeSeed	When it is set to true, the peer leaves the system after becoming seed

Table 4: State variables of each peer in the model.

ability to itself. Then, it requests downloading chunks again for the ones that it could not download before by sending a message *chunkExchange* to itself. This procedure means a downloading request should be responded after *checkChunkPeriod* time units, otherwise peer *p1* will request chunks from other peers.

In Table 4, we list the variables of each peer which are used to implement different aspects of the protocol in our model. In the following, we explain the download and upload bandwidth variables in more details.

When a peer requests downloading a chunk from its neighbours in *chunkExchange* message server, the value of the *downloadBandwidth* variable is decreased by 1. It is increased by 1 when a downloading request is completed by receiving a *requestChunkResponded* message. In the *requestChunkFrom* message server, if the peer is a free-rider or has left the system because of becoming a seed, a download request cannot be responded by this peer and the download bandwidth of the requesting peer should be released by sending a *freedwBandwidth* message to it. In the *freedwBandwidth* message server, variable *downloadBandwidth* is increased by 1.

When a downloading request is received by getting *requestChunkFrom*, the uploading bandwidth is examined and if there is enough bandwidth available, the value of the variable *uploadBandwidth* is decreased by 1 and the request is answered. Also, when downloading a chunk is completed by getting *requestChunkResponded* message, the upload bandwidth of the replier should be released by sending a *freeupBandwidth* message to it. In the *freeupBandwidth* message server, the value of the variable *uploadBandwidth* is increased by 1.

Experimental Results. We describe the results of the simulations carried out to show the effect of different parameters on our model. The experiment

networkDelay	2
chunkSize	1
requestDeadlineForConTracker	4
checkConTracker	6
retryConTracker	10
requestDeadlineForConPeer	8
checkConPeer	6
retryConPeer	10
checkChunkPeriod	8

Table 5: The values of timing parameters for all scenarios.

platform is 1.2 GHz Intel Core 2 with 2GB of memory. Running operating system is windows 7, and we use Erlang R13B04. As we described before, our BitTorrent model is a simplified model in which peers cannot join or leave the system during simulation, except that some peers can leave the system after becoming seeds. Also, we do not model the tit-for-tat mechanism. So, the achieved results cannot be compared with real measurements or performance analyses in BitTorrent networks.

Each run will be executed until all peers in the network complete downloading a file. It is obvious that changing timing parameters such as deadlines, and delays, depicted in Listing 7 as environment variables, can affect the overall download time. But, our objective is to investigate the effect of file size, download to upload bandwidth ratio, number of peers, and network delay on download time since an important performance measure from the user's point of view is download time. To show it, we organize 4 experiments which are explained in more detail in the following. So, we do not change deadlines and delays during experiments. Table 5 illustrates the values of timing parameters which are used for all experiments (time unit is in second).

For each scenario, we determine the neighbours of each peer to which a connection should be established. To do this, we use a randomly generated undirected graph in which each peer has a degree between 4 and 6. A peer knows its neighbours as known rebecs in the main part. Also, each peer can have some chunks of the file before connecting to the network, which are randomly specified in the main part. For the sake of clarity, a

Download/upload ratio	(2/1)	(1/1)	(1/2)
Download time for scenario 1	897.5	810.3	1076
Download time for scenario 2	990.7	963.5	868.1
Download time for scenario 3	787.6	755	844.2
Download time for scenario 4	973.1	738.2	770.9
Download time for scenario 5	1069	811	1300
Average download time (in sec.)	943.6	815.6	971.9

Table 6: Average download time of five different scenarios (topology and chunk availability) for different download/upload ratios. We consider a network consisting of 10 peers. Time is given in seconds.

specific topology and chunk availability is called a scenario. For all runs, a peer is chosen as a free-rider with the probability of 0.2. Also, a peer leaves the system after completing the download with the probability of 0.2. The probabilistic behaviour can be modelled using nondeterministic assignment in Rebeca which is choosing a value from a given set nondeterministically, and is implemented using the random function of Erlang.

In the first experiment, we run five different scenarios, each of them with three different download to upload ratios. In these scenarios, the network consists of 10 peers and the corresponding file is split up into eight chunks. Peer bandwidth is based on the number of chunks that can be downloaded or uploaded, which is supposed to be set to six chunks for all peers. We use the same value for peer bandwidth for all experiments. Table 6 demonstrates how download to upload bandwidth ratio can affect download time. To have an accurate result, we average the obtained download times for each ratio. We can find that when download and upload bandwidth have the same value, download time has the minimum value. Since the number of connected peers to each peer, the number of chunks to be downloaded, and peer bandwidth are supposed to be set to a small value, this bandwidth adjustment seems to be convenient.

The second experiment has the same values for the parameters as the first one, but for a network consisting of 20 peers. Table 7 exhibits the results obtained from averaging download times of three different scenarios, each of them executed for three different ratios. These scenarios differ from those we use in the first experiment due to the different number of peers in the network. Table 6 and 7 together provide a good insights about the

Download/upload ratio	(2/1)	(1/1)	(1/2)
Download time for scenario 6	1235.5	1117.9	1238.7
Download time for scenario 7	1083.3	1432.8	1486
Download time for scenario 8	1528	1189.8	1351.7
Average download time (in sec.)	1282.3	1246.8	1358.8

Table 7: Average download time of three different scenarios (topology and chunk availability) for different download/upload ratios. We consider a network consisting of 20 peers. Time is given in seconds.

File size	4 chunks	8 chunks	12 chunks
Downloading time for scenario 1	730.5	810.3	1157.1
Downloading time for scenario 2	530.1	963.5	884.01
Downloading time for scenario 3	465.4	755.01	1006.5
Downloading time for scenario 4	602.1	738.2	1260.8
Downloading time for scenario 5	634.1	811.03	1027.8
Average download time (in sec.)	592.4	815.6	1067.3

Table 8: Average download time of five different scenarios (topology and chunk availability) with 10 peers and different file sizes. Time is given in seconds.

effect of network size on download time for different download to upload ratios. We define the overall download time of the system as the sum of download times of all peers existing in the system. Therefore, download time should increase proportionally with network size. Comparing results of the same ratio from the two tables reveals this consequence. Also, for the same reason as in the previous experiment, when download and upload bandwidths have equal values, the minimum download time can be achieved.

The third experiment also has the same values for bandwidth, and network size as in the first one. According to the results, the minimum download time can be obtained when download to upload bandwidth ratio is one. This ratio is set to one for all the other scenarios. We run five scenarios, each of them with three different file sizes. Both topology and chunk availability patterns are the same as the scenarios in the first experiment. Then, we average the achieved download times of five scenarios

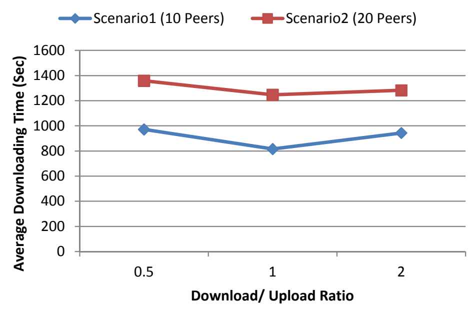
Network delay	2	3	4
Download time for scenario 1	810.34	965	1104.7
Download time for scenario 2	963.5	1079.8	1074.2
Download time for scenario 3	755.01	1112.4	1142.7
Download time for scenario 4	738.2	1199.7	1023.3
Download time for scenario 5	811.03	1129.9	1202.9
Average download time (in sec.)	815.6	1089.2	1109.6

Table 9: Average download time of five different scenarios (topology and chunk availability) with 10 peers and different network delays. Time is given in seconds.

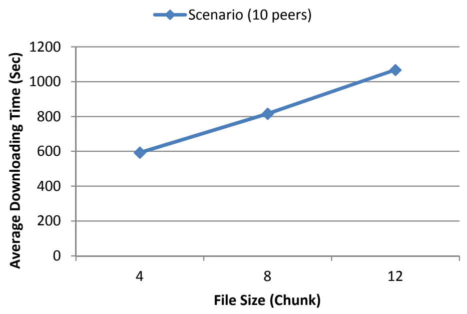
related to each file size. We find that download time increases with file size since each peer should download more chunks from connected peers in the system. Table 8 illustrates the achieved results.

In the last experiment, we investigate the effect of network delay on download time. To perform it, we utilize the same scenarios as in the first experiment i.e. the same topology and chunk availability patterns. The number of peers, file size, and peer bandwidth are set to 10, 8, and 6 respectively. Also, download to upload ratio is supposed to be assigned to 1. We run our five scenarios for three times, each of them with a different network delay. Network delay specifies how long it takes for a bit to be disseminated across the network from one node to another. So, changing it should affect the needed time in which a chunk is downloaded in the network and, therefore, the overall download time of the system. The achieved results, depicted in Table 9, show the same outcome.

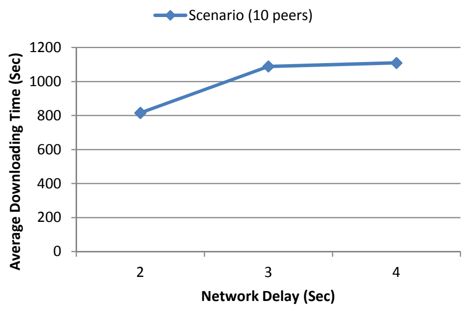
In the Tables 6, 7, 8, and 9 although the last row representing the average download time shows the results as expected, some of the rows seem to show some discrepancies. The reason for such results is that the performance of the model depends on a few parameters that can change the expected outcome. Some of these parameters are determined in run-time. In each scenario, the following parameters are variable and can affect the outcome: topology of the model, chunk availability, free-riding and leaving the network after becoming a seed. Considering these parameters, the average downloading time increases when a node is waiting for chunks which are unavailable on its neighbors, and/or most of its neighbors are free-riders, and/or leave the system immediately after becoming a seed (after completing downloading chunks). Also, if a chunk becomes avail-



(a) Average downloading time for different network sizes and different download/upload ratios.



(b) Average downloading time for different file sizes.



(c) Average downloading time for different network delays.

Figure 16: Average downloading time according to Tables 6, 7, 8, and 9.

able on a non-free-rider neighbor only after a long time then the average downloading time will increase.

In summary, in the previous experiments we investigated the effect of download to upload ratio, network size, file size, and network delay on the overall download time of the system. Figure 16 summarizes our results. Figure 16(a) shows how downloading time changes based on network size. Also, Figure 16(a) represents the changes in downloading time according to download/upload ratio. Figure 16(b) shows how file size can effect the average downloading time. Figure 16(c) represents the effect of delay in the network on the downloading time. The results show that our model can capture the features of the protocol correctly. In order to obtain more accurate results, we need to add more details to the model. Our plan is to implement tit-for-tat as a peer selection mechanism and different piece selection algorithms, and add dynamic behavior such as dynamic creation of peers to the model.

6. Related Work

Different approaches are used in designing formal modelling languages for real-time systems.

Timed automata. The model of timed automata, introduced by Alur and Dill [12], 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.

UPPAAL is a toolbox based on timed automata. The language describes systems as networks of timed automata with extension of data variables [30]. The simulator allows the modeller to examine the state space during the early stages of development. The model checking tool provides verification by means of exhaustive checking of the state space generated by the model. The model checking tool is supported by a specification language to check for reachability properties. UPPAAL allows us to model synchronous time varying behaviours while Timed Rebeca focuses on distributed and asynchronous agents. There has been some work on verification of Timed Rebeca models by means of translation to UPPAAL but simple models run into state explosion problem [19].

Real-Time Maude. Maude is a high level declarative programming language. It supports executable specification and programming in rewriting logic. Moreover, it supports equational logic and algebraic specification. It can deal with nondeterministic concurrent computations and has support for concurrent object oriented computation models [31]. Real-Time Maude is an extension to Maude. It supports both discrete and dense time domains. As with Maude, zero-time transitions are defined with rewrite rules while time elapse is defined by *tick* rewrite rules. The Real-Time Maude offers timed rewriting for simulations, timed search for reachability analysis and time bounded LTL model checking [32].

Timed Rebeca and Real-Time Maude are different in the computational paradigms that they naturally support. Real-Time Maude is a low level and powerful language. It allows modellers to control what computational model they base their model on, as long as it can be expressed in rewriting logic. Timed Rebeca is based on actor based model of computation. Timed

Rebeca benefits from its similarity with other commonly used programming languages and is more susceptible to get used by modellers without intimate knowledge of the theory behind modelling.

In the following we discuss related work on real-time actor-based modelling languages.

RT-synchronizer. A real-time actor model, RT-synchronizer, is proposed in [16], 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 [17]. 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 relative to the time that the method executing the send is invoked. In Timed Rebeca, we have the constructs *after* and *deadline*, which are representing the same concepts, respectively, except that they are relative to the time that the message (itself) is sent. So, it more directly reflects the computation architecture including the network delays. In our language, it is also possible to consider a time *delay* in the execution of a computation where in [17] it is possible to specify an upper bound on the execution time of a method. 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.

Schedulability for Rebeca Models. There is also some work on schedulability analysis of actors [33], 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 [34]. 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. 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 [35], 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 [34] 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 [18] 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.

7. Future Work

In this paper we have presented essentially two semantics for Timed Rebeca. One is the operational semantics given in Section 2.1 that associates a transition system with each Timed Rebeca program. The other is implicitly given by the translation from Timed Rebeca to Erlang given in Section 3. The two semantics serve different purposes, the SOS semantics gives a conceptual understanding of the behaviour of Timed Rebeca programs and this is the cornerstone for all the subsequent work on Timed Rebeca—see, e.g.,[36, 37, 38]. On the other hand, the semantics via translation to Erlang provided the first implementation of Timed Rebeca and has paved the way to the simulation-based analysis of Timed Rebeca programs using McErlang reported in Sections 4–5.

This begs the question of whether the Erlang encoding of Timed Rebeca into Erlang we proposed in this paper is "correct" with respect to our reference SOS semantics. It would be desirable to establish an operational correspondence result or a bisimulation relating the two semantics. This is, however, a major research challenge that we leave for future work. In particular, in order to prove such a correspondence result we would need a yardstick formal semantics for Erlang (such as the one presented in [39]).

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. We have been in contact with the team working on McErlang [40] and they developed a prototype to support time. This work is still in its preliminary stages.

The formal semantics for Timed Rebeca presented in this paper is also used in another parallel line of work [19]. The aim of that study is to map Timed Rebeca to timed automata [12] in order to use UPPAAL [20] 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), "Metatheory of Algebraic Process Theories" (nr. 100014021) and "Timed Asynchronous Reactive Objects in Distributed Systems: TARO" (nr. 110020021) of the Icelandic Research Fund.

References

[1] 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.

[2] M. Sirjani, M. M. Jaghoori, Ten years of analyzing actors: Rebeca experience, in: Formal Modeling: Actors, Open Systems, Biological Systems, 2011, pp. 20–56.

[3] 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).

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

[5] 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, Springer, 2007, pp. 293–307. doi:10.1007/978-3-540-74459-7-19.

[6] 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. doi:10.1007/s00236-009-0111-x.

[7] 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.

[8] G. D. Plotkin, A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (Sep. 1981).

[9] L. Fredlund, H. Svensson, McErlang: a model checker for a distributed functional programming language, SIGPLAN Not. 42 (9) (2007) 125–136. doi:http://doi.acm.org/10.1145/1291220.1291171.

[10] L. Aceto, M. Cimini, A. Ingólfsdóttir, A. H. Reynisson, S. H. Sigurdarson, M. Sirjani, Modelling and simulation of asynchronous real-time systems using Timed Rebeca, in: FOCLASA, 2011, pp. 1–19.

[11] B. Cohen, Incentives build robustness in BitTorrent, Tech. rep., bittorrent.org, http://www.ittc.ku.edu/~niehaus/classes/750-s06/documents/BT-description.pdf (2003).

[12] R. Alur, D. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183–235. doi:10.1016/0304-3975(94)90010-8.

[13] W. Yi, CCS + time = an interleaved model for real time systems, in: Proceedings of ICALP91, Vol. 510 of Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 217–228.

[14] P. C. Ölveczky, J. Meseguer, Specification of real-time and hybrid systems in rewriting logic, Theor. Comput. Sci. 285 (2) (2002) 359–405. doi:10.1016/S0304-3975(01)00363-2.

[15] H. G. Baker, Actor systems for real-time computation, Tech. rep., MIT (1978).

[16] 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.

[17] B. Nielsen, 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.

[18] J. Bjørk, E. B. Johnsen, O. Owe, R. Schlatte, Lightweight time modeling in Timed Creol, in: RTRTS, 2010, pp. 67–81.

[19] M. J. Izadi, An actor-based model for modeling and verification of real-time systems - Master Thesis, University of Tehran, Iran (2010).

[20] UPPAAL, UPPAAL Homepage, http://uppaal.com.

[21] M. M. Jaghoori, F. de Boer, T. Chothia, M. Sirjani, Task scheduling in Rebeca, in: Proc. Nordic Workshop on Programming Theory (NWPT'07), 2007, extended abstract.

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

[23] ICEROSE, ICEROSE Homepage, http://en.ru.is/icerose/applying-formal-methods/projects/.

[24] A. S. Tanenbaum, M. van Steen, Distributed systems - principles and paradigms (2. ed.), Pearson Education, 2007.

[25] G. Kahn, Natural semantics, in: F.-J. Brandenburg, G. Vidal-Naquet, M. Wirsing (Eds.), STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings, Vol. 247 of Lecture Notes in Computer Science, Springer-Verlag, 1987, pp. 22–39. doi:10.1007/BFb0039592.

[26] A. H. Buss, Modeling with event graphs, in: Proceedings of the 28th conference on Winter simulation, WSC '96, IEEE Computer Society, Washington, DC, USA, 1996, pp. 153–160, http://dx.doi.org/10.1145/256562.256597.

[27] A. Law, Simulation modeling and analysis, McGraw-Hill, 1991.

[28] I. Satoh, M. Tokoro, Time and asynchrony in interactions among distributed real-time objects, in: in Proceedings of 9th European Conference on Object-Oriented Programming, Springer-Verlag, 1995, pp. 331–350.

[29] A. Jafari, M. S. Talebi, A. Khonsari, G. Sepidnam, Maximizing download bandwidth for file sharing in BitTorrent-like peer-to-peer networks, in: ICPADS, 2008, pp. 344–350.

[30] K. G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, Int. Journal on Software Tools for Technology Transfer 1 (1997) 134–152.

[31] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. MartÌ-Oliet, J. Meseguer, J. F. Quesada, Maude: specification and programming in rewriting logic, Theoretical Computer Science 285 (2) (2002) 187 – 243. doi:10.1016/S0304-3975(01)00359-0.

[32] P. C. Ölveczky, J. Meseguer, Specification and Analysis of Real-Time Systems Using Real-Time Maude, in: M. Wermelinger, T. Margaria (Eds.), FASE, Vol. 2984 of Lecture Notes in Computer Science, Springer, 2004, pp. 354–358.

[33] L. Nigro, F. Pupo, Schedulability analysis of real time actor systems using coloured Petri Nets, in: Proc. Concurrent Object-Oriented Programming and Petri Nets, 2001, pp. 493–513. doi:10.1007/3-540-45397-0-21.

[34] M. M. Jaghoori, F. de Boer, T. Chothia, M. Sirjani, Schedulability of asynchronous real-time concurrent objects, Logic and Algebraic Programming 78 (5) (2009) 402–416.

[35] F. S. de Boer, T. Chothia, M. M. Jaghoori, Modular schedulability analysis of concurrent objects in Creol, in: FSEN, 2009, pp. 212–227.

[36] H. Kristinsson, Event-based analysis of real-time actor models - Reykjavik University, Iceland, Master's thesis, http://rebeca.cs.ru.is/files/MasterThesisHaukurKristinsson2012.pdf (2012).

[37] E. Khamespanah, Z. Sabahi Kaviani, R. Khosravi, M. Sirjani, M.-J. Izadi, Timed-Rebeca schedulability and deadlock-freedom analysis using floating-time transition system, in: Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, AGERE! '12, ACM, New York, NY, USA, 2012, pp. 23–34. doi:10.1145/2414639.2414645.

[38] B. Magnusson, Simulation-based analysis of Timed Rebeca using TeProp and SQL - Reykjavik University, Iceland, Master's thesis, http://rebeca.cs.ru.is/files/MasterThesisBrynjarMagnusson2012.pdf (2012).

[39] H. Svensson, L.-A. Fredlund, C. Benac Earle, A unified semantics for future Erlang, in: Proceedings of the 9th ACM SIGPLAN workshop on Erlang, Erlang '10, ACM, New York, NY, USA, 2010, pp. 23–32. doi:10.1145/1863509.1863514.

[40] McErlang, McErlang Homepage, https://babel.ls.fi.upm.es/trac/McErlang.

Appendix A. Rebeca Model for the Sensor Network

```
env int netDelay;
env int adminCheckDelay;
env int sensor0period;
env int sensor1period;
env int scientistDeadline;
env int rescueDeadline;

reactiveclass Sensor {
    knownrebecs {
        Admin admin;
    }

    statevars {
        int period;
    }

    msgsrv initial(int myPeriod) {
        period = myPeriod;
        self.doReport();
    }

    msgsrv doReport() {
        int value;
        value = ?(2, 4); // 2=safe gas levels, 4=danger gas levels
        admin.report(value) after(netDelay);
        self.doReport() after(period);
    }
}

reactiveclass Scientist {
    knownrebecs {
        Admin admin;
    }

    msgsrv initial() {}

    msgsrv abortPlan() {
        admin.ack() after(netDelay);
    }
}

reactiveclass Rescue {
    knownrebecs {
        Admin admin;
    }

    msgsrv initial() {}

    msgsrv go() {
        int msgDeadline = now() + (rescueDeadline-netDelay);
        int excessiveDelay = ?(0, 1); // unexpected obstacle might occur during rescue
        delay(excessiveDelay);
        admin.rescueReach() after(netDelay) deadline(msgDeadline);
    }
```

```
}

reactiveclass Admin {
    knownrebecs {
        Sensor sensor0;
        Sensor sensor1;
        Scientist scientist;
        Rescue rescue;
    }

    statevars {
        boolean reported0;
        boolean reported1;
        int sensorValue0;
        int sensorValue1;
        boolean sensorFailure;
        boolean scientistAck;
        boolean scientistReached;
        boolean scientistDead;
    }

    msgsrv initial() {
        self.checkSensors();
    }

    msgsrv report(int value) {
        if (sender == sensor0) {
            reported0 = true;
            sensorValue0 = value;
        } else {
            reported1 = true;
            sensorValue1 = value;
        }
    }

    msgsrv rescueReach() {
        scientistReached = true;
    }

    msgsrv checkSensors() {
        if (reported0) reported0 = false;
        else sensorFailure = true;

        if (reported1) reported1 = false;
        else sensorFailure = true;

        boolean danger = false;
        if (sensorValue0 > 3) danger = true;
        if (sensorValue1 > 3) danger = true;

        if (danger) {
            scientist.abortPlan() after(netDelay);
            self.checkScientistAck() after(scientistDeadline); // deadline for the scientist
                to answer
        }

        self.checkSensors() after(adminCheckDelay);
    }

    msgsrv checkRescue() {
        if (!scientistReached) {
            scientistDead = true; // scientist is dead
        } else {
            scientistReached = false;
        }
    }

    msgsrv ack() {
        scientistAck = true;
    }

    msgsrv checkScientistAck() {
        if (!scientistAck) {
            rescue.go() after(netDelay);
            self.checkRescue() after(rescueDeadline);
        }
        scientistAck = false;
    }
}

main {
    Sensor sensor0(admin):(sensor0period);
    Sensor sensor1(admin):(sensor1period);
    Scientist scientist(admin):();
    Rescue rescue(admin):();
    Admin admin(sensor0, sensor1, scientist, rescue):();
}
```
Listing 6: A Timed Rebeca model of the sensor network example

Appendix B. Timed Rebeca Model for BitTorrent Protocol

```
env int networkDelay;
env int chunkSize;
env int requestDeadlineForConTracker;
env int checkConTracker;
env int retryConTracker;
env int requestDeadlineForConPeer;
env int checkConPeer;
env int retryConPeer;
env int checkChunkPeriod;

reactiveclass Peer(3)
{
	knownrebecs
	{
			Tracker t1;					// a centralized entity that gives the list of peers to be connected
			Peer peer[6];					// peers to be connected
	}
	statevars
	{
			boolean chunk[8];					//shows the availability of chunks for peer
			int token; int peerToken[6];
			boolean contoTracker;
			boolean conToP[6];
			int peerDegree;					// number of connected peers
			int downloadBandwidth;
			int uploadBandwidth;
			boolean isFreeRider;
			boolean leaveSysAfterBecomeSeed;
	}
	msgsrv initial(int degree, boolean cnk[])
	{
			peerDegree = degree;
			for(int i=0; i < 8; i++)
				chunk[i] = cnk[i];				// we can set the availability of chunks for peer
			downloadBandwidth = 4;
			uploadBandwidth = 2;
			int fRiding = ?(1:5);
			if (fRiding == 2)
			{
				isFreeRider = true;
				downloadBandwidth =downloadBandwidth + uploadBandwidth;
			}
			self.connectTracker();
	}
	msgsrv connectTracker()
	{
			token+=1;
			t1.requestConnection(token) deadline(now()+requestDeadlineForConTracker);
```

```
                    self.checkContoTracker()   after (checkConTracker);
    }
    msgsrv requestAnswered(int tok)
    {
            if (token==tok)
                contoTracker=true;
    }
    msgsrv checkContoTracker()
    {
            if(!contoTracker)
                self.connectTracker() after(retryConTracker);
            else if(contoTracker)
                    self.start();
    }
    msgsrv start()
    {
            for(int i= 0; i < 4; i++)
                self.consToPeers(i);
            if(peerDegree ==5)
                self.consToPeers(4);
            else if(peerDegree ==6)
            {
                self.consToPeers(4);
                self.consToPeers(5);
            }
            self.chunkExchange();
    }
    msgsrv checkChunkAvailability()
    {
            if(downloadBandwidth == 0)
                downloadBandwidth = downloadBandwidth+1;
            if(!chunk[0] || !chunk[1] || !chunk[2] || !chunk[3] || !chunk[4] ||
               !chunk[5] || !chunk[6] || !chunk[7])
                self.chunkExchange();
    }
    msgsrv chunkExchange()
    {
            for (int i =0; i <8; i++)
            {
                if(peerDegree ==4)
                    int choosePeerForCh = ?(0:3);
                else if(peerDegree ==5)
                        int choosePeerForCh = ?(0:4);
                    else if(peerDegree ==6)
                            int choosePeerForCh = ?(0:5);
                if (!chunk[i] && downloadBandwidth > 0)
                {
                        if (conToP[choosePeerForCh])
                        {
                            peer[choosePeerForCh].requestChunkFrom(i);
                            downloadBandwidth = downloadBandwidth-1;
                        }
                }
            }
            self.checkChunkAvailability() after(checkChunkPeriod);
    }
    msgsrv freeDwBandwidth()
```

```
	{
			downloadBandwidth = downloadBandwidth+1;
	}
	msgsrv requestChunkFrom(int chunkn)
	{
				if (!isFreeRider && !leaveSysAfterBecomeSeed)
				{
						if(uploadBandwidth > 0 && chunk[chunkn])
							sender.requestChunkResponded(chunkn)
							    after(chunkSize*networkDelay);
				}
				else if(isFreeRider || leaveSysAfterBecomeSeed)
					sender.freedwBandwidth();
	}
	msgsrv freeUpBandwidth()
	{
			uploadBandwidth = uploadBandwidth+1;
	}
	msgsrv requestChunkResponded(int chunkn)
	{
			if(!chunk[chunkn])
			{
				chunk[chunkn]=true;
				downloadBandwidth=downloadBandwidth+1;
				sender.freeUpBandwidth();
			}
			if(chunk[0] && chunk[1] && chunk[2] && chunk[3] && chunk[4] && chunk[5]
			    && chunk[6] && chunk[7])
			{
				int leavingProbability = ?(1:5);
				if(leavingProbability == 3)
					leaveSysAfterBecomeSeed = true;
				else
						uploadBandwidth= uploadBandwidth +
						    downloadBandwidth;
			}
	}
	msgsrv consToPeers(int peerNumber)
	{
			peerToken[peerNumber]+=1;
			peer[peerNumber].requestconFrom(peerToken[peerNumber])
			    deadline(now()+requestDeadlineForConPeer);
			self.checkConToP(peerNumber)  after(checkConPeer);
	}
	msgsrv checkConToP(int peerNumber )
	{
			if(!conToP[peerNumber])
			{
				self.consToPeers(peerNumber) after(retryConPeer);
			}
	}
	msgsrv requestconFrom(int tok)
	{
			sender.requestResponded(tok)  after(networkDelay);
	}
	msgsrv requestResponded(int tok)
	{
```

```
                    for(int i=0; i<peerDegree; i++)
                    {
                          if(sender == peer[i])
                          {
                                if(peerToken[i]==tok)
                                      conToP[i]=true;
                          }
                    }
      }
}
//-------------------------------------------------------------- tracker
reactiveclass Tracker (3)
{
      knownrebecs{
                    Peer p[10];
      }
      statevars {}
      msgsrv initial(){}
      msgsrv requestConnection(int token)
      {
                    sender.requestAnswered(token)  after(networkDelay) ;

      }
}

main
{
              Peer p1(t1,[p2,p3,p4,p9,p7,p8]):(4,[true, false, false, false, false, false,
                   false, false]);
              Peer p2(t1,[p1,p5,p6,p7,p10,p2]):(5,[false,false, false, false, true, true,
                   false, false]);
              Peer p3(t1,[p1,p4,p6,p7,p8,p3]):(5,[false, false, true, true, true, false,
                   false, true]);
              Peer p4(t1,[p1,p3,p8,p9,p4,p5]):(4,[true, false, false, false,true, false,
                   false, true]);
              Peer p5(t1,[p2,p6,p7,p10,p8,p9]):(4,[false,true, true, true, false, true,
                   true, true]);
              Peer p6(t1,[p2,p3,p5,p7,p8,p]):(5,[true, false, true, true, false, false,
                   false, false]);
              Peer p7(t1,[p2,p3,p5,p6,p9,p10]):(6,[false,true, true, true, false, false,
                   false, true]);
              Peer p8(t1,[p3,p4,p6,p9,p10,p5]):(5,[false,true, false, true, false, true,
                   false, false]);
              Peer p9(t1,[p1,p4,p7,p8,p3,p2]):(4,[false, false, true, true, false, true,
                   true, false]);
              Peer p10(t1,[p2,p5,p7,p8,p3,p4]):(4,[false,true, false, true, false, false,
                   false, true]);
              Tracker t1([p1,p2,p3,p4,p5,p6,p7,p8,p9,p10]):();
}
```

Listing 7: A Timed Rebeca model of BitTorrent protocol presented in pseudo code.