EVENT-BASED ANALYSIS OF REAL-TIME ACTOR MODELS

Haukur Kristinsson
Master of Science
Software Engineering
May 2012
School of Computer Science
Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

Event-based Analysis of Real-Time Actor Models

by

Haukur Kristinsson

Research thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment of
the requirements for the degree of
Master of Science in **Software Engineering**

May 2012

Research Thesis Committee:

Dr. Marjan Sirjani, Supervisor
Associate Professor, Reykjavík University, Iceland

Dr. Anna Ingólfsdóttir
Professor, Reykjavík University, Iceland

Dr. Yngvi Björnsson
Associate Professor, Reykjavík University, Iceland

Event-based Analysis of Real-Time Actor Models

Haukur Kristinsson

May 2012

Abstract

Although timed actor-based models have attracted considerable attention in the recent years, little work exists on analyzing and model checking such systems. The actor-based language, Timed Rebeca, was introduced to model distributed and asynchronous systems with timing constraints, and a supporting tool was developed for automated translation of its models to Erlang (Aceto et al., 2011). The translated code can be executed using McErlang. In this thesis, we propose extensions for Timed Rebeca to improve the usability of the language. These extensions besides the timed extensions provided for the model checker McErlang give us the possibility of model checking and doing event-based simulation of models for the first time. In addition, we apply trace-based statistical analysis and visualization methods to simulation results for reasoning about behavior of models. Examples of writing safety properties to verify correctness of models and for analyzing behaviors are provided. The examples and case studies presented in the thesis show the applicability of the tools and methods introduced.

Greining á rauntíma Actor líkönum

Haukur Kristinsson

Maí 2012

Útdráttur

Þrátt fyrir vinsældir og athygli Actor líkana síðustu ára, þá hefur lítið verið gert til að greina dreifð kerfi sem byggja á eiginleikum þeirra. Tungumálið, Timed Rebeca, er byggt á Actor líkaninu og var kynnt sem mál til að hanna líkön sem sýna hegðun dreifðra og samstilltra kerfa með tíma í fyrirrúmi. Tól sem breytti hönnuninni yfir í Erlang kóða var kynnt sem að svo var keyrt með því að nota McErlang líkana sannreynara (Aceto et al., 2011). Í þessari ritgerð er farið yfir nýjar tillögur á breytingum fyrir Timed Rebeca til þess að endurbæta notagildi málsins. Endurbæturnar gera okkur kleift að sannreyna og hermunargreina þessi líkön í fyrsta sinn. Auk þess sýnum við tölfræðilegar aðferðir sem nota ummerki í líkaninu sem er undir skoðun hverju sinni. Aðferðirnar gefa okkur innsýn í breytilega og óútreiknanlega hegðun líkananna. Farið verður yfir dæmi sem sýna notagildi tólanna og aðferðanna sem kynntar eru í ritgerðinni.

To my girlfriend Anna Marín Skúladóttir

Acknowledgements

Dr. Marjan Sirjani, for guiding me and introducing me to research in software engineering. Her insights and vision for the field made this project possible. It has been a privilege and a real experience for me.

Dr. Anna Ingólfsdóttir and Dr. Yngvi Björnsson for being in my committee and for their useful comments.

My fellow students and friends for our discussions and laughs together. I want to especially thank Brynjar Magnússon for his immeasurable assistance and collaboration while working on this project.

My family and girlfriend for all their support during this time.

Lars-Ake Fredlund for our discussion about explicit-time approach (Lamport, 2005) to model checking and for showing interest in our work by implementing real-time semantics for McErlang with the ability to support Timed Rebeca semantics.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Computer systems today are getting more complex and are evolving extremely fast. Today's businesses do not typically succeed without the use of computer system technologies, which are becoming an essential foundation of every business activity. Networked communications are connecting millions of devices together using distributed web services. A failure in these systems can result in great losses of revenues, unsatisfied system users, or even cause devastating events that affect human lives.

This results in an increasing need for tools and techniques that help in understanding and verifying the behavior of these systems. Such techniques provide answers to the questions of cost, performance and correctness that arise throughout the life of such computer systems.

The application of model-based engineering technologies to real-time distributed systems seems to be somewhat untended, yet a useful field that proposes guidelines for how to do analysis on the behavior of such systems. Model-based engineering guidelines show how to extract the abstract behavior of the system using a modeling language to build a model. Following the model creation we can apply analysis methods that give us insights into behaviors of the model.

Such analysis of software models with simulation has been increasingly used to address a variety of issues from strategic management of software development to supporting improvements in software designs. In this project our focus is on network applications and more specifically on distributed and asynchronous patterns, and the use of simulation and model checking to analyze them.

Formal methods are mathematically based techniques for developing, specifying, and verifying software or hardware systems. This is to help software engineers to develop correct

and reliable systems with precise mathematical models (Holloway, 1997). Civil engineers for example build mathematical design for buildings before constructing one. This is done to avoid a collapse of the building. Same should apply when designing software systems. Many incidences have been reported like the airport baggage-handling system in Denver which was to be the most advanced baggage handling system in the world. The system proved to be far more complex than some had originally believed. The problems building the system resulted in the newly complete airport sitting idle for 16 months while engineers worked on getting the baggage system to work. Estimated losses over 560$M USD (Calleam, 2008).

Verification of a model is to ensure that a specification is complete and that mistakes have not been made in implementation of the model. But even though no mistakes have been made we also need to validate the model. Validation ensures that the model meets its intended requirements in terms of the methods employed and the results obtained. Its goal is to make the model useful in the sense that the model addresses the right problem and provides accurate information about the system being modeled.

In this project we use Timed Rebeca as our modeling language which is an actor-based language. It provides a simple natural concurrency model, object-based computation, and timing primitives. The actor-model has been used both as a framework for a theoretical understanding of concurrency, and as the theoretical basis for several practical implementations of concurrent and distributed systems (Hewitt, 2010). An Actor is a computational entity that, in response to a message it receives, can concurrently:

- send messages to other Actors;
- create new Actors;
- designate how to handle the next message it receives.

Not much has been done on analyzing actor-based models with timing constraints. Actors are being used more in practice as programming languages like Erlang and libraries like Akka (Typesafe, 2012) are getting more popular.

For verifying Timed Rebeca models we propose the use of McErlang model checker for Erlang which has in parallel to this project been extended with real-time semantics. McErlang is used because we have an existing mapping from Timed Rebeca to Erlang (Aceto et al., 2011).

For validating models we propose using performance evaluation techniques by simulating models with McErlang and analyzing traces generated by the simulation. Simulation is preferred because formal verification methods typically suffer from state-explosion prob-

lem, unless the model is changed to be more abstract and to produce feasible state-space for model checking. In addition, when carrying out validation we do not always know what properties to look for in the system. Validation is done by using three performance analysis methods:

- Paired-checkpoint Analysis;
- Periodic Events Analysis;
- and Checkpoint Analysis.

These methods are based on statistical measurements and visualization of the simulated data of the model. With statistical methods and simulation we get more insights into the dynamic behavior of the model and how it evolves over time.

Throughout the thesis we show the applicability of our methods using examples and case studies.

1.1 Contribution

Proposed methods in this thesis are to provide effective and easy analysis methods and approaches for Timed Rebeca models.

Thesis contributions are:

- Extending Timed Rebeca by adding list data structure and ability of calling custom functions to overcome the issue of modeling certain behaviors in systems that was not possible before.
- Extending the mapping to Erlang to support McErlang timed semantics.
  - Providing verification of safety properties for Timed Rebeca models.
  - Providing us with simulation that follows the semantics of Timed Rebeca.
- Implementing a tool-set that provides analysis methods for Timed Rebeca models.
- Providing experimental results and examples.

1.2 Overview of the Thesis

The structure of the thesis is as follows: Chapter 2 introduces the concept of model-driven engineering which contain guidelines that this project follows. In addition we explain Timed Rebeca and its computational model: the Actor model. Chapter 3 introduces a mapping to McErlang with timed semantics. This mapping builds a basis for being able to apply methods used in the following chapters. In addition we propose new extensions to Timed Rebeca that gives us the ability to model more behaviors in a system. Chapter 4 presents how we use our new mapping to verify Timed Rebeca models with assertion-based model checking. It explains how we use monitors in McErlang to verify safety properties as assertions. In Chapter 5 we propose a way to validate models with simulation. The methods proposed uses performance evaluation to gain more insight into the behavior of models using a simulation and analysis tool-set. Chapter 6 then shows case studies and experimental results that are gained using methods proposed in Chapters 4 and 5. Related work discussions are in Chapter 7. Finally we present conclusions and future work in Chapter 8.

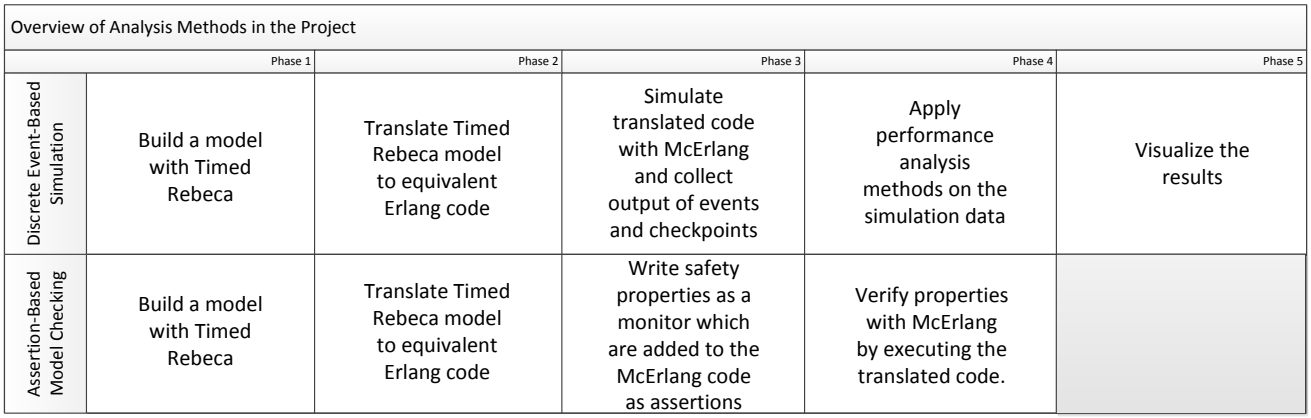

Figure 1.1: Overview of the project.

Methods applied in this project are called assertion-based model checking and discrete event-based simulation. The methods can be applied to models written with the modeling language Timed Rebeca. Assertion-based model checking is used to verify models with safety properties. The properties are written in Erlang and are called monitors. Discrete event-based simulation is used for validating models and to gain insights into their behaviors. This is done with performance analysis and visualization methods.

The overview in Figure 1.1 shows how the verification and the simulation process is carried out. Both methods begin by building a Timed Rebeca model that is translated to

an equivalent Erlang code. This means that every state in a Timed Rebeca model is assumed (showed by example in this thesis) to have equivalent state in Erlang. When doing assertion-based model checking we first need to provide a property to check. This property should be satisfied in every state of the Erlang program.The property is written in Erlang and is used by McErlang as a monitor. McErlang is then executed to run the generated Erlang code along with the monitor, checking if every state of the Erlang code satisfies the assertion. If any of the states does not satisfy the assertion then the monitor will return a violation.

Discrete event-based simulation is not based on property checking. To simulate the generated Erlang code we use McErlang. McErlang generates the output traces. We then provide three kinds of performance analysis methods to produce statistical information for the modeler. Besides statistical information, results can be visualized for better understanding. This gives us valuable insights into how our models are behaving.

Chapter 2

Background

2.1 Model-Driven engineering

The field of model-driven engineering focuses on exploiting the abstract behavior of a system in a model rather than the computational concept (Schmidt, 2006). In today's development cycle, bugs tend to cost more depending on the time of the development (Baier & Katoen, 2008), so having the correct and most efficient model of the system early in the development cycle is important. In this project, we wanted to offer ways that help software engineers to design reliable systems. We do this by providing them with a modeling language that has tools to verify specifications and have the capability to do performance analysis (with simulation), both that can detect and prevent design errors early in the life cycle for network applications with distributed and asynchronous patterns.

The process that we follow in this project is presented on Figure 2.1. It is based on the model-driven methodology and emphasizes on using an abstraction model of a system. The model is then used to verify that the design is able to handle our objectives that are specified in the requirements. The objectives that distributed systems have are often related to performance, speed, and the correctness of the system. The process has the emphasis that the model is verified before analysis as the simulation analysis relies on the model correctness.

Initially we need to have the requirements in place for us to construct a model. The model needs to catch all the required behaviors which can be a difficult job for the modeler as he needs to find the right abstraction for the model.

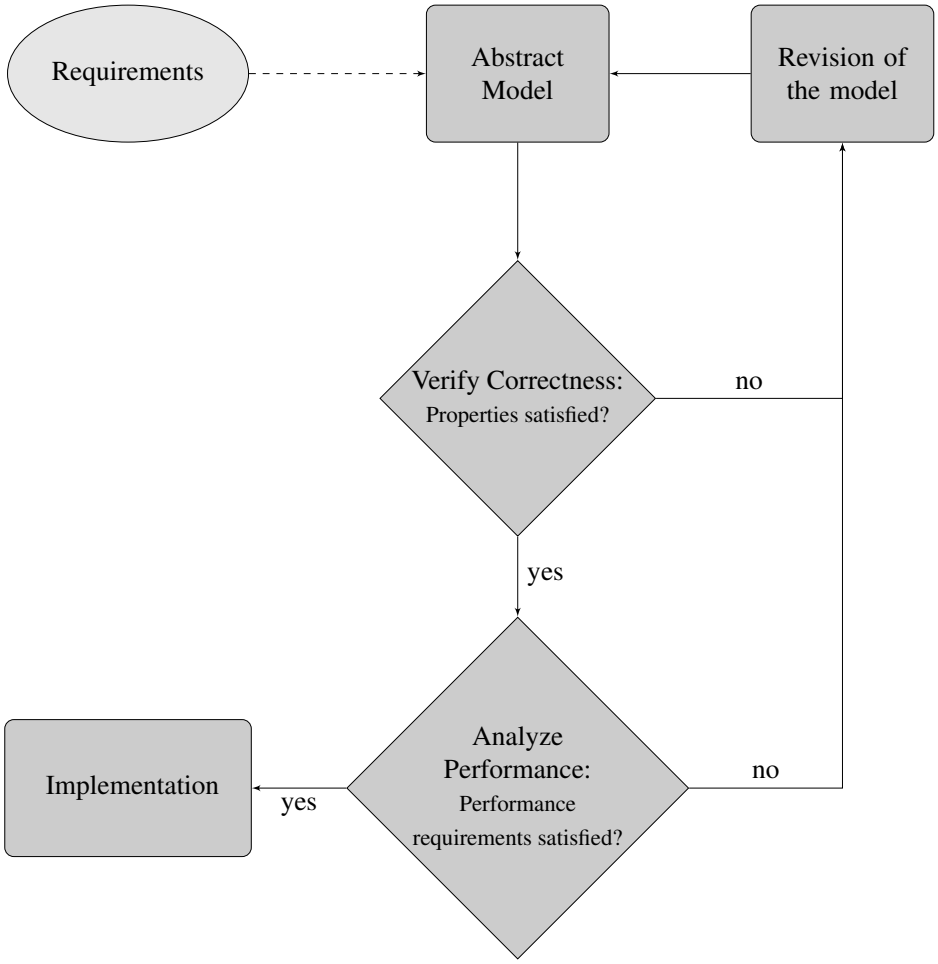

Figure 2.1: Evaluation Model

Verification of the model can be exhausting due to the complexity and state-explosion problem. This makes it often not possible to verify a complex event-based system without some major abstraction methods that reduces the state-space. The verification examines if the expected correctness properties are met by the model. In general correctness properties are classified in the broad categories: safety, reachability, and liveness properties.

When the model has been verified for correctness, the performance evaluation of the system design can be carried out. This for example can involve response time analysis methods, which gives the modeler insights into the performance of the system. We attain response time analysis by using paired-checkpoint analysis which measures time between checkpoints in the model. Goals or baselines need to be set before the analysis and behaviors need to be identified that needs to be evaluated. If the result is not as promised the modeler can improve the design and evaluate the new design until satisfactory results are found.

When the modeler is satisfied with the design of the system the development of the system can be carried out.

2.2 Actor Model

The Actor model is a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation: 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 received.

Actors have encapsulated states and behavior; and are capable of creating new actors, and redirecting communication links through the exchange of actor identities (Sirjani & Jaghoori, 2011).

The actor model was originally introduced by Hewitt (Hewitt, 1972) as an agent-based language back in 1973 for programming secure distributed systems with multiple users and multiple security domains. It was later developed into a concurrent object-based language by Agha in 1986 (Agha, 1986).

In an actor-based system all *actors* run concurrently and use asynchronous message passing for communication.

The main positive thing about the actor model is that it is simple, understandable, and easy to reason about. Erroneous behaviors such as deadlocks are more easily avoided than in traditional shared state threading. Actors are a higher level of abstraction than threads and locks. It can be applied to a significant proportion of problems.

2.3 Timed Rebeca

Reactive Objects Language, Rebeca (Sirjani, Movaghar, Shali, & de Boer, 2004), is an operational interpretation of the actor model with formal semantics and model checking tools. Rebeca is designed to bridge the gap between practical software engineering domains and formal methods. It does so by providing a simple, message-driven and

object-based computational model, a language with traditional Java-like syntax, and a set of verification tools.

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 (Sirjani et al., 2004; Aceto et al., 2011).

An extension to Rebeca with real-time features was introduced in (Aceto et al., 2011) with the name Timed Rebeca. In Timed Rebeca we consider a global clock (or more precisely speaking, synchronized local clocks) for our timed Rebeca models. Methods are still executed atomically, but we can model passing of time while executing a method. Instead of a message queue for each rebec, we have a bag containing the messages that are sent. Each rebec knows about its local time and can put deadlines on the service requests (messages) that are sent declaring that the request will not be valid after the deadline (modelling the timeout for a request). When a message is sent there can also be a constraint on the earliest time at which it can be served (taken from the message bag by the receiver rebec). The modeller may use these constraints for various purposes, such as modelling the network delay or modelling a periodic event. Timing primitives added to the syntax are *delay*, *now*, *deadline* and *after*. Figure 2.2 shows the grammar for Timed Rebeca (Aceto et al., 2011).

The following explains the four constructs (Aceto et al., 2011).

- **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 not valid any

more and is purged from the bag. Deadlines are used to model message expirations (timeouts).

- **After**: *r.m() after(t)*, where r denotes a rebec name, m denotes a method name of r and t is a natural number, means that the message m is sent to the rebec r and is put in the message bag. The message cannot be taken from the bag before t time units have passed. After statements can be used to model network delays in delivering a message to the destination, and also periodic events.

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

Figure 2.2: Abstract syntax of Timed Rebeca (Aceto et al., 2011). Angle brackets $\langle ... \rangle$ are used as meta parenthesis, superscript $+$ for repetition more than once, superscript $*$ for repetition zero or more times, whereas using $\langle ... \rangle$ with repetition denotes a comma separated list. Brackets [...] indicates that the text within the brackets is optional. Identifiers 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.

2.4 Erlang and McErlang

Erlang is a functional programming language which was developed by Joe Armstrong in 1986 (Armstrong, 2007). It was originally a proprietary language within Ericsson, but was released as open source in 1998. It is designed to be a general-purpose concurrent programming language for programming real-time distributed systems that are fault-tolerant and non-stop. Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Erlang processes are lightweight and belong to the language, not the operating system. The concurrency model of Erlang is built on the Actor model.

McErlang is a model checker that replaces the native Erlang runtime engine with a new one. The model checker has full Erlang data type support, support for general process communication, node semantics (inter-process communication behaves in a subtly different way from intra-process communication), fault detection, and fault tolerance. Because of the custom runtime engine it is able to capture all program states, taking into account the distribution, communication and message boxes. This way it can construct a complete state-space of a Erlang program. McErlang offers ability of expressing correctness properties in the form of monitors (safety or Büchi), abstraction algorithms to reduce state-space, and exploration algorithms to verify or simulate Erlang programs (Fredlund & Svensson, 2007).

Chapter 3

Event-Based Modeling with Timed Rebeca

The analysis of event-based real-time systems has not been widely studied in a formal way. These systems have though been used widely in the recent years.

Timed Rebeca design goal is to be a simple and easy-to-use language for modeling concurrent and distributed systems with timing constraints. Its ease of use is to bridge the gap between the formal community and practitioners. These goals are achieved by using a simple message-driven and object-based computational model, syntax that is similar to Java, and a set of tools that provide analysis and verification.

The analysis tools previously offered (Aceto et al., 2011) were the use of McErlang monitors to be able to halt an execution run when something unexpected happens (like a dropped message). We needed our language and its related tools to provide us with more variety of analysis techniques.

To provide an effective analysis capability we extended Timed Rebeca and the translation tool. We extended the language in a way to facilitate analyzing larger set of behaviors by adding lists as an additional data structure, and also adding the possibility of calling *custom functions* from native Erlang code. In addition we added tracing capability to Timed Rebeca models. We obtain traceability by adding the *checkpoint* construct to Timed Rebeca and translate it to

- an output function when doing simulation, or
- a McErlang probe when doing verification.

Before we translate a Timed Rebeca model we need to configure the tool and make it clear whether we will do simulation or verification. The use of probes is useful when doing verification in McErlang as they are easily obtainable from a monitor that runs in parallel with the model. With probes we are able to expose the value of Timed Rebeca variables to McErlang while there is no well-defined procedure to get the value of program variables of Erlang programs in McErlang. In simulation we use checkpoints to keep track of the execution of the model and to define our points of interest for the model.

In the previous work (Aceto et al., 2011), a mapping for Timed Rebeca to Erlang was introduced. The mapping was also extended for using McErlang tool to do simulations in an execution-manner. At the time of that work McErlang did not support time in model checking and all experiments were carried out by using simulation via McErlang. The simulation was done as an execution via Erlang itself but with support of using safety assertions as monitors via McErlang. If we choose to do experiments with execution alternative we have to accept some time slides which is caused by execution of each statement on a real machine. If a translated Timed Rebeca program runs for long enough these small time-slides accumulate and cause deviation from the expected interpretation of the corresponding model based on the formal semantics of Timed Rebeca.

By cooperation with McErlang team (mainly Lars-Ake Fredlund (Fredlund, 2012)), we extended our Timed Rebeca mapping to be able to use the capabilities of a new implementation of McErlang which supports model checking of timed systems. The implementation of timed semantics in the McErlang model checker (Earle & Fredlund, 2012), was inspired by the semantics in Lamport TLA+ specification language and the TLC model checker (Lamport, 2005, 2002), but adapted to Erlang.

This approach is referred as the *explicit-time approach* and is usually representing time as a value of a variable *now* and the passage of time is then modeled by a *Tick* action that increments *now*, based on *tick* rules. These rules define the minimum time between time values in the model ($t_1 = 0 * tick$, $t_2 = 1 * tick$, ... , $t_n = n * tick$). The timing constraints are then expressed with timer variables.

In Erlang, all real-time computations are done with *receive* statements that can include an *after* clause. The *after* clause defines the timeout of the *receive* as depicted in Listing 1.

The implementation of timed semantics in McErlang had no need for any *Tick* process or a timer process to increment the time. Instead the minimum delay is defined in each *after* clause inside a *receive*. In a sense the tick rule is then defined by these *after* clauses.

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

Listing 1: Erlang syntax of a receive with timeout. The process that includes the receive statement waits for a message that matches $Pattern_1$, ..., $Pattern_N$ and $Guard_1$, ... , $Guard_N$. If no pattern is matched with a received message it will evaluate *TimeoutExpression* after *TimeoutValue* seconds.

3.1 Extending Timed Rebeca

Extending the language without opposing the design goals of Rebeca is a challenge. We wanted to respect the design goals of Rebeca, and at the same time be able to model real-time systems with rather complicated behavior that needed to have buffers or queues. We decided to add lists, a simple yet valuable data type. These lists are unbounded and have the ability to have integers deleted, added, or removed from them. We also added the ability to call custom functions that are defined as native Erlang code. Custom functions give us the freedom of performing operations in a Timed Rebeca model, which could not be done using the constructs supported by Timed Rebeca itself.

Examples of how to use lists and custom functions in Timed Rebeca can be found in Listings 34 and 35, located in Appendices A. Both examples show us how to utilize these new extensions for Timed Rebeca with simple explanations.

In addition to the examples, a modified Timed Rebeca language description is presented in Appendix D.

For improving traceability we added the *checkpoint* function to be able to keep track of traces in our models and to expose variables within a Timed Rebeca model to McErlang. Checkpoints are twofold, depending on if you are using simulation or model checking:

- For simulation it outputs a given *term*[1] of a checkpoint and the timing of when it occurs.
- For model checking it generates probes that are written in Erlang and utilized by McErlang to expose variables within a Timed Rebeca model to verify the model.

Table 3.1 summarizes the extensions and the mapping. In the following we will explain the extensions and in Section 3.2 we will explain the mapping.

Extension	**Timed Rebeca Syntax**		**Erlang / McErlang**
Lists	**list**<int> N;	$\rightarrow$	Erlang list data type as a variable with name N.
Custom Function	**erlang**.$func$(V_1,...,V_n);	$\rightarrow$	Call to function $func$ with parameters V_1,...,V_n.
Checkpoints	**checkpoint**(L,T(,V_1,...,V_n));	$\rightarrow$	Erlang Output Function with L as the label of the checkpoint, T as the term and V_1,...,V_n as a optional tuple of parameter data which is used when doing simulation analysis.
	***checkpoint**(L,T(,T_2, ..., T_n));	$\rightarrow$	McErlang probe with L as the label of the checkpoint and T as the term. Additional parameters are extending the term to a tuple of data.

Table 3.1: Abstract mapping structure of extensions for Erlang and McErlang. Where $func$ is the name of a function, L a label for a checkpoint V as an state or a local variable name and T as the term of a checkpoint and is of the type integer or boolean expression. When doing model checking T is used to define a term of the generated McErlang probe. Function marked with (*) applies when doing model checking translation.

[1] In Erlang every piece of data of any data type is called a term and all Timed Rebeca variables are translated to terms.

3.1.1 Traceability with Checkpoints and Events

A major factor in Timed Rebeca is the ability to analyze systems and for that we need traceability. Models that have many non-deterministic behaviors and are often referred to as complex systems (Lu & högskola, 2010), have behaviors that are not easy to foresee beforehand. To be capable of tracing or verifying any behaviors, we first want to be able to trace it with marked events that are "*interesting*" points in the model. When simulating the model these points are then obtainable for use in analysis. Also, when translating for model checking purposes we want the ability of adding labels into appropriate states within the state-space that McErlang generates. This gives us an easy way to expose variables within a Timed Rebeca model as there is no well-defined mechanism for retrieving program variables in McErlang. To do this we added the syntax *checkpoint* as a marker which can be placed between any statements in a Timed Rebeca model. Checkpoints have two mandatory parameters, the *label* and the *term*. In addition to the explicit *checkpoint* constructs, we can trace the *events* in our simulation. An event happens when a message is taken from the queue.

In Listing 2 a simple model that consists of one reactive class CheckpointExample is shown. It has one rebec *proc* that is instantiated from the reactive class CheckpointExample. The initial message server of *proc* sends a message to itself which triggers a message iteration of the message server *go*. The message server *go* assigns a random number between 1 and 100 to a variable *rand*. Note that the checkpoint with the label *Start* is in two places in the model: in the initialization of the rebec *proc*, and when the variable *rand* is set to 50 and after the checkpoint with the label *Triggered* occurs with the term (second parameter of the checkpoint) *rand*. This is to be able to measure the time between checkpoints with the labels *Start* and *Triggered*.

```
1  reactiveclass CheckpointExample(3) {
2     knownrebecs { } statevars { }
3     msgsrv initial() {
4       checkpoint(Start,0); /* Mark start */
5       self.go(); /* Initial call to go */
6     }
7     msgsrv go() {
8       int rand = ?[1:100]; /* Random pick of integer */
9       if(rand == 50) { /* If we get 50 we trigger a event */
10        checkpoint(Triggered,rand); /* Mark Triggered */
11        checkpoint(Start,0); /* Mark Start again */ }
12      delay(1); % Each random assignment take 1 time unit.
13      self.go(); /* Itterate */
14    }
15 }
16 main { CheckpointExample proc():(); }
```

Listing 2: Timed Rebeca Model - Traceability of Random Values.

The checkpoints in the model can give us some information. We can identify when the variable *rand* is set to 50. Also when using checkpoints as probes, we can verify whether all assignments are between 1 and 100 by using safety checking monitor in McErlang. To summarize, using checkpoints improves the analyzability of Timed Rebeca as:

- We can have more precise analysis on simulation runs since *checkpoints* can be distinguished from message servers calls and we can use checkpoints to mark interesting points between statements in a sequential computation that happens inside a message server.
- We can use checkpoints to generate probes, to verify safety properties with McErlang.

In the following chapters, we propose some novel techniques to help the modeler to gain insight into the dynamic behavior of a complex event-based simulation model based on trace analysis. Traces are generated by the simulation processes each time a checkpoint occurs or when a message is taken from a message bag of a rebec (taking a message from the bag is an event). The information included for events are: message sending time, message arrival time, if the message is expired, rebec name, message server name, parameters of the message, and the sender of the message. For checkpoints, the information included are; time of the checkpoint, rebec name that included the checkpoint, and the parameters that we pass to the checkpoint. The first parameter that is mandatory is the label of the checkpoint; and we call the rest of the parameters that are optional the terms. The terms can be any variable or expression in the model and help us to get more information for our analysis.

We will also explain how we are able to use *checkpoints* to implement safety checking of some behaviors in a model with McErlang monitors.

3.2 Mapping of Timed Rebeca to McErlang's Timed Semantics

Timed Rebeca execution relies on the mapping to Erlang, which is a general-purpose concurrent programming language for developing real-time, distributed and fault-tolerant systems. The concurrency in Erlang is built to follow the Actor model (Hewitt, 1972). Erlang uses concurrent processes to structure the program. These processes have no shared memory and communicate by asynchronous message passing. Each process is created using the built-in function *spawn*(*Fun*). It creates a process with an identity, that we call *Pid* and evaluates the function *Fun*. All communications between processes in Erlang are message-based communications and are sent by using *Pid* ! *Message*, where each *Message* can be any of Erlang expressions or a term[2] and *Pid* is the identity of any process within the program. Each message acts as a pattern that matches a *receive* clause within a process.

To summarize, the concurrency primitives in Erlang are:

- *Pid* = *spawn*(*Fun*): creates a process and evaluates the given function *Fun*
- *Pid* ! *Message*: sends the message "*Message*" to the process with the identity *Pid*
- *receive ... end*: receives a message that has been sent to the process that evaluates it.

Erlang handles time with the use of *after* as a timeout clause in a receive statement as Listing 1 shows. When a process reaches a receive expression it looks for the oldest message in the mailbox of the process and matches it with any of the patterns Pat_1, ... , Pat_N also checking the guards $Guard_1$, ... , $Guard_N$. If no pattern is matched and *TimeoutValue* is reached then the expression *TimeoutExpression* is evaluated. The process looks in the queue each time a message arrives until the timeout occurs.

Formal semantics and a first implementation, as an automatic translation mapping of Timed Rebeca to Erlang, were introduced in (Aceto et al., 2011). This mapping was also extended to be used with McErlang, which provided an execution of a given Timed Rebeca model with the support of a monitor that could halt the execution if the assertion included in the monitor is violated.

In this project we extended the current mapping to be usable with timed semantics for the model checker McErlang (Earle & Fredlund, 2012). The new timed semantics offered support for timing primitives in Erlang as it was not supported in the previous versions.

[2] In Erlang every piece of data of any data type is called a term.

This project is done in tight cooperation with the author of the new implementation of McErlang (Fredlund, 2012).

An abstract mapping is shown in Table 3.2, which shows how each entity of Timed Rebeca is mapped to Erlang.

Timed Rebeca		**Erlang**
Rebeca model	$\rightarrow$	A set of functions
Reactive class	$\rightarrow$	Three functions
Known rebecs	$\rightarrow$	Dictionary of variables
State variables	$\rightarrow$	Dictionary of variables
Message server definition	$\rightarrow$	A match in a receive expression
Local variables	$\rightarrow$	Dictionary of variables
Message send statement	$\rightarrow$	Message send expression
Message send w/after	$\rightarrow$	Message send expression inside a receive with a timeout
Message send w/deadline	$\rightarrow$	Message send expression with the deadline as a parameter
Delay statement	$\rightarrow$	Empty receive with a timeout
Assignment	$\rightarrow$	Dictionary update
If statement	$\rightarrow$	Case expression
Nondeterministic selection	$\rightarrow$	Random selection in the simulation tool
Checkpoints	$\rightarrow$	Function
Custom Function	$\rightarrow$	Function

Table 3.2: Structure of the mapping from Timed Rebeca to Erlang (Aceto et al., 2011). The timing primitive $now()$ was omitted from Timed Rebeca grammar as we do not use local time due to implementation of relative time. Newly added constructs are added, namely checkpoints and custom functions.

The mapping is essentially the same and is still only providing a translation of a subset of Timed Rebeca operational semantics (rebec creation is not supported). Changes were though made regarding the use of time, timeouts, and how we handle events, and checkpoints. Note that although McErlang now supports model checking of timed Erlang programs, it does not match the Timed Rebeca formal semantics in a straight forward way, as the computational model is different in many ways. In the following subsections we will go into details of what was changed and how we were able to tailor McErlang to our needs.

3.2.1 Mapping Extensions

The former proposed extensions were added to the structure of the mapping shown in Table 3.2. In Timed Rebeca, lists are additional data structures and custom functions are Erlang functions that return an expression.

Checkpoints can be used for two purposes and can be mapped to different functions. When doing simulations the mapping provides us with an output function that allows the ability to collect Timed Rebeca expressions or variables while simulating. As when doing verification, the mapping will generate McErlang probes[3] that can easily be accessed from formerly mentioned monitors in McErlang. Checkpoints have one mandatory parameter, *label*, and optional parameters as *terms*. In simulation the terms provide more information while generating checkpoint traces. In model checking all the terms are put together as a tuple.

3.2.2 Timing features

The foundation of using the timing features in Timed Rebeca mapping to Erlang was the usage of the system time. In Timed Rebeca semantics, the expression *now*() returns the value of the local clock of a rebec and is used for

- tagging a message when it is sent with the local time of the sender rebec,
- determining the global expiration time of a message before it is sent, and
- evaluating message expiration times based on the local time of the receiver.

The local time of a rebec is obtained with the built-in function *timestamp*() in Erlang which is the system time. This can cause noticeable problems in model checking as every time we use it we obtain a new unique local time from the *timestamp*() function. The system time is presented as a tuple $\{MegaSeconds, Seconds, MicroSeconds\}$ in Erlang, which is the elapsed time since 00:00 GMT, January 1, 1970.

When a message is being sent between rebecs, this system clock value is saved into the program state generated by McErlang, in the form of a message parameter. This parameter will always be unique since when time passes we get a new unique value from the function. This means for a non-terminating model, we will undoubtedly always have unique values inside all states that include sending messages. Therefore the number of

[3] Probes are Erlang functions that inject information into a generated program state, readable by McErlang.

program states generated by McErlang will be unbounded and not feasible for model checking.

To address this problem, we utilized newly presented *clock references* from McErlang in the corresponding Erlang code instead of using *timestamp*() function. The clock references can relate to clocks in theory of Timed Automata and are initiated and reset to generate finite state spaces for models (Alur & Dill, 1994). In Timed Rebeca code, when a message is sent, a clock reference is created in the translated Erlang code. Relative time is acquired by using the new API *mce_erl_time* in McErlang. It has the following functions:

- now(), returns the current time.
- nowRef(), stores current time in a global reference.
- was(*Ref*), returns the stored time reference *Ref*.
- forget(*Ref*), removes the stored time reference *Ref* (time will not be increased in future states).

When a clock reference is created, its initial value will be $\{0, 0, 0\}$ to make it compatible with normal time format in Erlang. In Timed Rebeca we use the time domain as the set of natural numbers ($\mathbb{N}$), which is then used in **after**, **delays** and **deadlines**. In the translated code we need to translate these values to be compatible with McErlang. We denote the translated values as $\mathbb{N}_{erlang}$. For example when using **after**(1) in Timed Rebeca, the translated value will be $\{0, 0, 1\}$.

In the following paragraphs, we go into the details how we used McErlang's new timed API *mce_erl_time* to exclude the usage of timestamps in Erlang.

In the modified mapping we considered remembering the time only to be crucial when sending messages with the deadline construct. This is to be able to evaluate the message and to know if the message is expired or not. It is mandatory to know the sending time to be able to evaluate it to local time of the receiver.

Each send statement in Timed Rebeca can have timing primitives *after* (message delay, before sending) and *deadline* (message expiration time) that are explained in (Aceto et al., 2011). Communications between rebecs in Timed Rebeca, are translated as a process communication in the translated Erlang code. Formerly in the translated code, the sender process used the local time of when the message was sent and included it in the message that was being sent. This was included in the message for the receiver process to be able to compare the message send time to it's local clock of when the message arrived, seeing if the message is expired or not. The change was made in the translation and we added

formerly discussed *clock references* instead of local time before sending the message. This reference is sent as a message parameter which is denoted as *TT* in Listing 3.

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

Listing 3: Pseudo McErlang Message Send

The clock reference is remembered and referenced in all future states of the model, until it is stopped with the *forget*() function. To have finite state graphs this clock reference needs to be stopped right after the message had been evaluated by the receiver process. This is presented in Listing 4.

Expiration of a message in the translated code is handled by comparing the sending time of the message to the local time of the current process. Instead of the local time, we check the clock reference. For doing that we use the *was* function in McErlang. When the destination process in Erlang receives a message, it will evaluate the message expiration time with the expression (*was*(*Ref*) + $Deadline_{erlang}$), where *Ref* is the sending time of the message in a form of a clock reference and $Deadline_{erlang}$ is the relative deadline value converted to Erlang compatible time-stamp.

```
receive
 {StateVars, _}
 = {{Sender, TT, DeadLine}, msgsrv, {Params}} ->
    % Evaluate Deadline = inf or Deadline + was(TT) < statetime()
     true -> % Not expired.
      % Forget TT clock reference.
      {StateVars, LocalVars}
     false -> % Expired.
      % Forget TT clock reference.
      {StateVars, LocalVars}
    end,
end.
```

Listing 4: Pesudo McErlang Evaluation of a Message

In Timed Rebeca, when a rebec takes a message from the message bag it compares the sending time of the message with the deadline. In the translated code this is done by using the timed McErlang API and by adding the message parameters $Deadline_{erlang}$ to the clock reference TT and comparing it to the current state time. *mce_erl*:*now*() returns the current state time of the state and returns {0,0,0} if no clock has been initiated. A *message server* mapping function is presented in Listing 4. If the comparison shows

that the deadline is not yet passed then it will automatically use the *true* clause of the case clause. Otherwise it will be dropped due to a message expiration using the false clause.

3.2.3 Support for Timed Semantics

Mapping of Timed Rebeca to McErlang had some adaptation and challenges. The McErlang model checker is designed to verify Erlang programs. Erlang programs are based on the same concurrency model as Timed Rebeca (the actor model). But it still does not have the same computational interpretation.

```
1  ProccessOne = spawn(process, function, [1]),
2  ProccessTwo = spawn(process, function, [2]).
```

Listing 5: Erlang spawning processes. Processes are spawned with the Pid *ProcessOne* and *ProcessTwo*. Parameter for the spawn function respectively account for a parent process, a function to evaluate for the process, and a parameter to give to the function.

In Listing 5, the two processes are concurrently executed. So, when running a program like this in Erlang there is nothing that prevents ProccessTwo to run faster than ProccessOne, so in order to check all possible paths of the computation we need to check for all executions:

- If *ProcessOne* is spawned before *ProcessTwo*,
- if *ProcessTwo* is spawned before *ProcessOne*, and
- if *ProcessOne* and *ProcessTwo* are spawned at the same time.

That would also be the case in untimed Rebeca or when two events are happening exactly at the same time.

In Timed Rebeca when we have a **delay** or **after** we want to respect the progress of time strictly. A naive mapping to Erlang could overlook this problem. In Erlang no order of execution is respected based on delays. So we had to use the *urgent* constructs of McErlang in our mapping.

In the following paragraphs we want to show with examples how our mapping works by generating a state-space with McErlang.

Implementation illustrated with an example. To demonstrate the problem above, a scenario is presented in the form of a Timed Rebeca model in Listing 6, which models two

competing processes that are trying to get acknowledged by a listener that only responds once. For simplification purposes for the reader, an abstracted Erlang translate was created and is presented in Listing 38 located in Appendices A. The state graph that McErlang generated for the model is presented in Figure 3.1 as a Labeled Transition System.

```
reactiveclass Process(2) {
 knownrebecs { Listener lsnr; }
 statevars { boolean acknowledged; }

 msgsrv initial(int delaybeforesend) {
   acknowledged = false;
   delay(delaybeforesend);
   lsnr.receive(delaybeforesend);
 }

 msgsrv ack(boolean processed) {
   acknowledged = true;
   trace(isProcessProcessed, processed);
 }
}

reactiveclass Listener(2) {
 knownrebecs { Process proc1; Process proc2; }
 statevars { boolean received; }

 msgsrv initial() {
   received = false;
 }

 msgsrv.receive(int delaylabel)
 {
   // Note: Only receives once
   if(received == false) {
      received = true;
      sender.ack(true);
   }
 }
}
main {
 Listener listener(process1,process2):(); % Listener that only processes one message
 Process process1(lsnr):(2); % Delays by 2 before sending
 Process process2(lsnr):(3); % Delays by 3 before sending
}
```

Listing 6: Timed Rebeca Model - Competing Processes

The model consists of two reactive classes, Process and Listener. It has 3 rebecs: *listener* that is instantiated from the reactive class Listener, *process1* and *process2* that are both instantiated from the reactive class Process. The rebec *listener* starts by waiting for a single reply from the rebecs *process1* or *process2*. The initialization of the rebec *process1* delays by 2 time units before sending a message to the rebec *listener* while the second rebec *process2* delays by 3 time units before sending a message.

From the Timed Rebeca semantics, the obvious computation would be that *process1* that delays by 2 time units would be the one sending the first message and therefore rebec *listener* would reply to him first. Following that the rebec *process2* that has a delay of 3 time units would send a message to the rebec *listener* and get no reply back.

A generated state graph (Labeled Transition System) by McErlang is shown in Figure 3.1 and presents each program state as a node with a number as the label. The state graph shows that a path exists for both processes to get a reply. This can be seen from the transitions (messages) $0 \rightarrow 7$ and $0 \rightarrow 6$. For the Timed Rebeca model the branching transition $0 \rightarrow 6$ must not occur when exploring states in the model. This is because the transition has a path which ends having the possible transitions (messages) $5 \rightarrow 4$ and $6 \rightarrow 1$ where in both cases, *process2* gets acknowledged. This means that some paths in the model end up as replying to the process that delays by 3 time units. We therefore needed special care in our mapping to McErlang to follow the formal semantics of Timed Rebeca. Furthermore the transitions (messages) $7 \rightarrow 3$ and $3 \rightarrow 4$ should not be possible as it means that *process2* can send a message before *process1* gets acknowledged. This should not be possible as no delay is assigned to the acknowledge message.

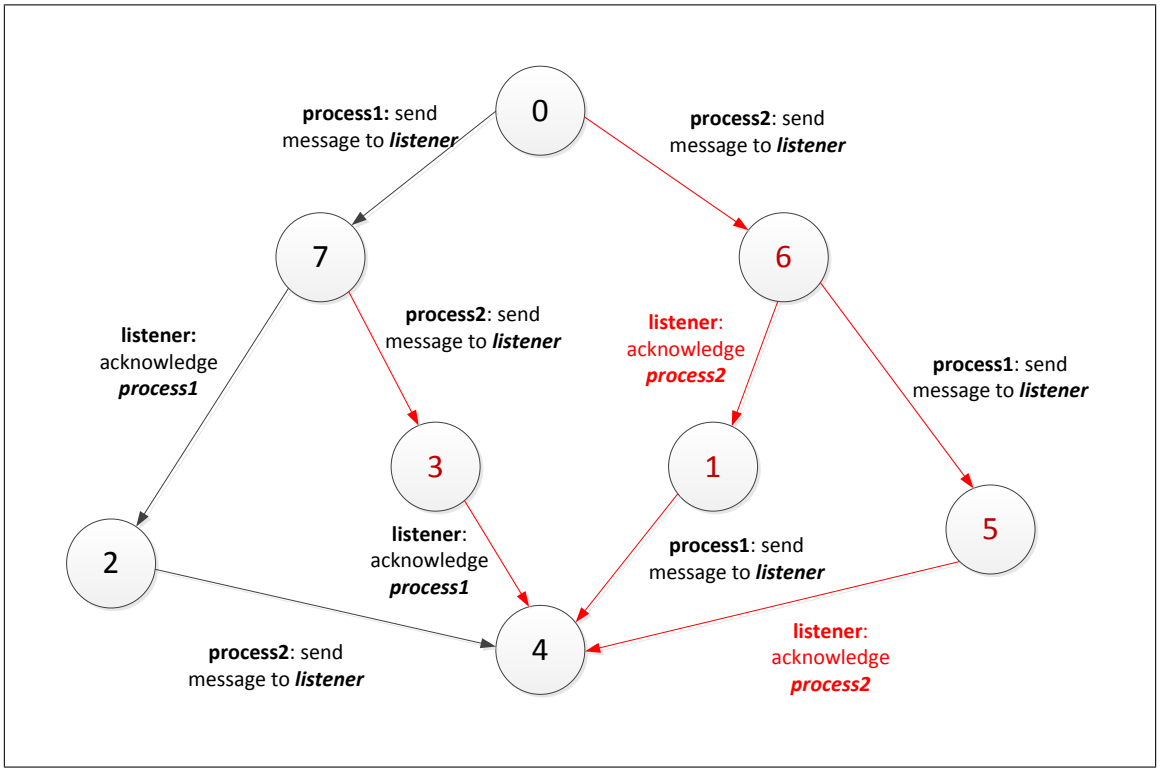

Figure 3.1: Labeled Transition System (state space) for the competing processes model in Listing 6. Each state in the transition system does **not** have an equivalent state in Timed Rebeca as it is not a correct implementation.

When looking at the state graph in Figure 3.1, we see two problems that need solving. The problems are because Timed Rebeca semantically demands,

- urgency to all messages without timeouts (message without after or delays in Timed Rebeca) to only have the possibility of happening instantaneously, and
- that messages with timeouts (message that have delays and afters in Timed Rebeca) need to only occur immediately after the timeout and need to be ordered based on their timeouts.

For the first problem we have to exploit the *urgency* feature of McErlang. From the work presented in (Earle & Fredlund, 2012), it is explained that by using *urgency* we are able to define a maximum waiting period until a timeout happens. Therefore we want to assign *urgency* with 0 as the maximum waiting time on all **receive** primitives that have no timeout clauses in Erlang. This makes all non-timed message passing (actions) in the translated code to only happen instantaneous, or "infinitely fast" as it is referred to in McErlang. Furthermore, the semantics of McErlang state that the time cannot advance if there is a transition (message) enabled from a program state with maximum waiting time of 0.

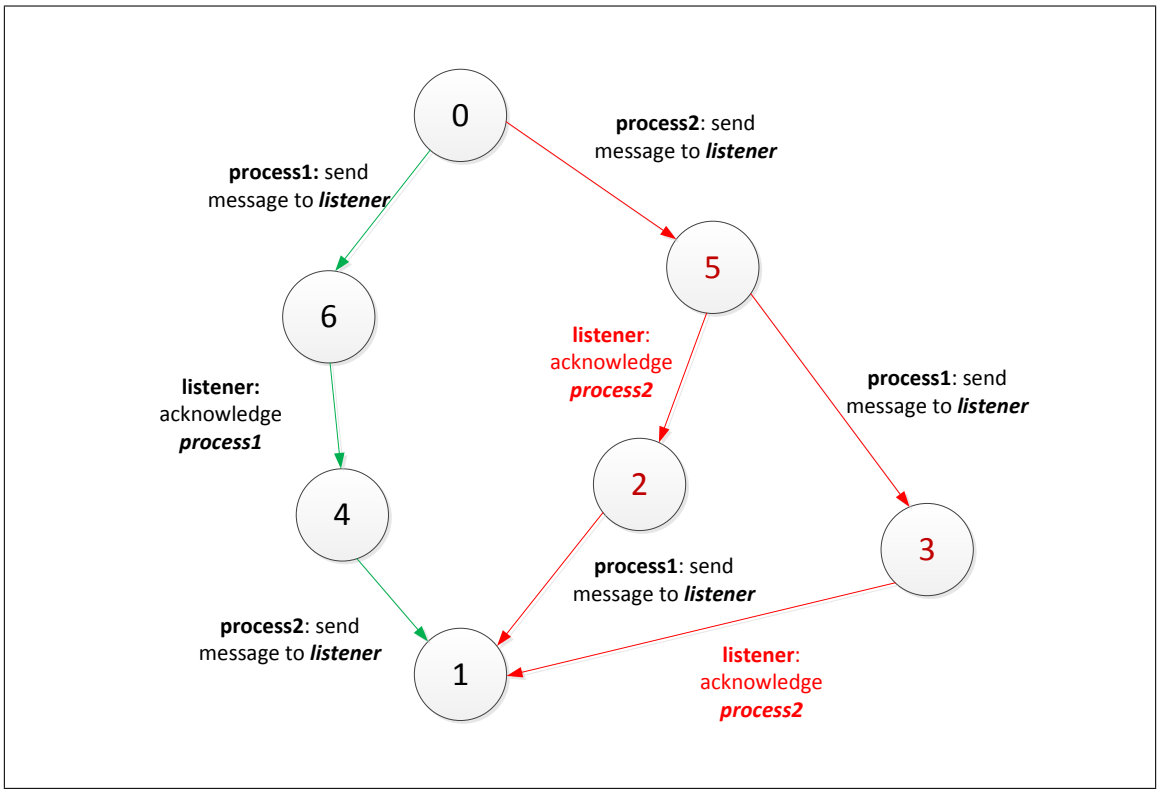

Figure 3.2: Labeled Transition System (state space) for program states for the competing processes model in Listing 6 with maximum delay as 0 for message with no timeout clauses (no after or delays in Timed Rebeca). Each state in the transition system does **not** have an equivalent state in Timed Rebeca as it is not a correct implementation.

When using this we have a different state graph depicted in Figure 3.2. This graph shows that by example using urgency, solves the problem for messages that have no timeouts (no delays or after in Timed Rebeca) but not for messages with timeouts.

Considering the first transition system (state graph) in Figure 3.1 with the second transition system (state graph) in Figure 3.2 we see that when a acknowledge message is sent, it only occurs once for each path in the graph. This is because now we only explore possible paths with 0 as the maximum timeout, when doing non-timed message passing (actions).

The second problem is still an issue. The transitions (messages) $0 \rightarrow 5$, $5 \rightarrow 2$ and $3 \rightarrow 1$ should not be possible. They are messages (actions) with timeouts in the translated Timed Rebeca code and McErlang still checks for all possible executions for them.

The formal timed semantics of McErlang is presented in (Earle & Fredlund, 2012). The operational semantics explain a transition relation for the enforcement of urgent constructs with taking into account the progress of time. An Erlang function is defined that calculates a reduced set of transitions that are also compatible with using clock references discussed in former section. This function, namely *timeRestrict* has an argument of

- list of already reduced transitions,
- current state time,
- list of non-timeout transitions,
- time of the next most urgent transition that is to be taken, and
- list of enabled transition that is to be executed before the next urgent transition.

The function is evaluated between states and providing a way to order actions marked urgent based on timeout value in enabled *receive* statements. Using urgent on message (action) with timeout is therefore supported and gives us ability to set the maximum delay for receive clauses. For Timed Rebeca models it makes us capable of ordering the exploration of states based on time units given to delays (translated to receive statement with an after clause). This means that in Timed Rebeca **delay**(2) will end one time unit before **delay**(3) if they start at the same time in the translated Erlang code.

Using this we altered the translated delay statement as depicted in Listing 7.

```
1 timedelay(Milliseconds) ->
2   mce_erl:urgent(Milliseconds), % Mark transition as urgent.
3   receive
4   after (Milliseconds) -> ok
5   end.
```

Listing 7: Erlang Program - Urgent delay

This gives us a transition system (state graph) that shows a single computational path where *process1* sends a message and gets a reply from *listener* followed by *process2* sending a message and getting no reply. All nodes in the transition system generated by McErlang then have equivalent states in Timed Rebeca. Thus giving us the correct behavior of exploring states and shows by example that we have successfully used McErlang to be able to verify the Timed Rebeca model.

Another Timed Rebeca model (non-deterministic process evaluator) is presented in Listing 37 located in Appendices A. There we explore expirations and non-deterministic behaviors of Timed Rebeca and show that our mapping works as a fine-grained mapping for models provided in Timed Rebeca.

3.3 Discussion

In this section we successfully implemented an extended translation mapping from Timed Rebeca to McErlang with timed semantics. It should be noted that the mapping proposes a fine-grained mapping of Timed Rebeca as the formal definition of Rebeca computation is to take place by taking the message from the front of the message queue and executing the corresponding message server *atomically* (Sirjani et al., 2004). The *atomicity* of the computation is not followed as demonstrated in Figure A.1, when branching to transitions (messages) $12 \rightarrow 6$ and $12 \rightarrow 14$. Both can happen at the given time and therefore both possibilities are explored. This mapping also provides us with a challenging question: whether each state generated by McErlang from the Erlang code has an equivalent state in Timed Rebeca. This could be checked formally by showing weak bisimulation for generated states for both languages.

Chapter 4

Assertion-Based Model Checking

Model-based verification techniques are based on models describing a possible behavior of a system in an accurate and unambiguous way. The design is typically constructed with a *modeling language*. Furthermore, the verification technique of our concern, *model checking*, is typically the process of exploring the state-space of a model in a brute-force manner where all reachable system states are considered. The process verifies whether the given model meets a given specification. Specifications often contain safety requirements such as the absence of deadlocks or whether states that cause the system to crash are reached. This way it can be proven whether a given model satisfies or violates a property with absolute certainty (Huth & Ryan, 2004; Baier & Katoen, 2008).

This chapter explains an approach to verify Timed Rebeca models with safety specifications. The verification is done by using the model checker McErlang that has recently been extended with timed semantics. With the mapping explained in Chapter 3 we are able to carry out a complete state-space generation of Timed Rebeca models. In Chapter 3 we explained the concept of using McErlang probes and monitors. This project uses assertive monitors that are written in the programming language Erlang. Monitors are used to build safety properties for Timed Rebeca's generated Erlang code. Checkpoints in Timed Rebeca are mapped to McErlang probes as explained in Table 3.1. To facilitate the modeler's task of building specifications, a template is proposed. This template is used for detecting checkpoints in a Timed Rebeca model.

Verifying safety properties is the first step in verifying Timed Rebeca Models.

4.1 Safety Checking with Monitors and Checkpoints

The exploration of the state-space involves building all possible states and the transitions that connect them. In McErlang, when generating a state, each state keeps some information from the program state which is observable by monitors. The information available from the program states are invariant state components, such as process mailboxes, process status, process name and process dictionaries. This is observable from the programing interface of McErlang and does not have any well-defined mechanism for retrieving the value of program variables from a program state (Earle & Fredlund, 2008).

To facilitate observation of state variables and local variables in Timed Rebeca, we use the method of probing program states in McErlang. This is done by using a *probe* action which is a function that has the arguments *label* and *term*. Probes make the internal state of a program visible to the model checker. These probes are easily accessible from monitors.

In Timed Rebeca, we use checkpoints to specify the points in the model where we want to know the values of certain variables at run-time (simulation or model checking). When doing model checking, checkpoints are translated to McErlang probes as explained in Table 3.1. Using probes gives us a way of exposing state and local variables in the Erlang code by passing them as an argument (the term) of the probe. This allows the modeler to construct monitors based on probes as shown in Listing 9. Note that probes as shown in Listing 9 are translated from checkpoints in a Timed Rebeca model as in Listing 8.

```
1  ...
2  checkpoint(id1,exp1);
3  checkpoint(id2,exp2,exp3);
4  ...
```

Listing 8: Timed Rebeca Model - Checkpoint example

```
1  ...
2  mce_erl:probe(id1,exp1),
3  mce_erl:probe(id2,{exp2,exp3}),
4  ...
```

Listing 9: McErlang translation of checkpoint example in Listings 8

In Listing 9 we have two probes; one with the label `id1`, and the other with `id2`. The expression(s) (`exp1`,`exp2` and `exp3`) passed to each checkpoint can then be retrieved by a monitor as shown in Listing 10.

The monitor code depicted in Listing 10 checks if any process is at the point with a probe with the Label `Identity`. If this is true, the monitor retrieves the values of the expressions passed to the probe (Line 3 of Listing 10). In the rest of the monitor you can see how the monitor can decide based on these values. If the Expression in the probe depicted on line 2 in Listing 9 is 5, the monitor will halt the verification process displaying a violation with the message `violation_termIsFive`.

```
1  % Check if state has any checkpoints with the Label
2  case has_probe_with_label(id1,StateActions) of
3    {true,RetrievedTerm} -> ManipulateTerm(RetrievedTerm);
4    false -> {ok,noCheckPointFoundWithLabel}
5  end.
6
7  % Function that does something with the retrived term
8  ManipulateTerm(TermRetrived) ->
9    case TermRetrived == 5 of
10    % Violate if term is equal to 5.
11    true -> {violation_termIsFive,TermRetrieved};
12    % Satisfy term if not 5.
13    false -> {ok,satisfied}
14   end.
```

Listing 10: Pseudo code of a monitor that retrieves a term of a checkpoint. When McErlang evaluates *has_probe_with_label* the `RetrievedTerm` is returned as the term of the probe. The term is then passed (if found) as an argument to the function `ManipulateTerm`. The variable *StateActions* has all actions from the current invariant state of the program, such as process mailboxes, process status, process name and process dictionaries (checkpoints are included as actions in the program state).

When a monitor is set to execute with a translated Timed Rebeca model, we are running it in parallel in a lock-step manner with the model.

$$\textit{Translated Model} \mid\mid \textit{Monitor}$$

That is, when the program takes a step, the model checker makes the monitor to take a step. The monitor will then evaluate the functions defined in the monitor with the program state as an argument. This will lead to either satisfied results, meaning that the current state is not violated by the property defined in the monitor or by halting the verification, signaling a violation. Monitors work like asserted safety observer that run in parallel with the model and are executed every time a state is generated. There are two types of monitors in McErlang, safety monitors and Büchi monitors which implement Büchi automata. Using Büchi monitors allows McErlang to check LTL properties using the automata as the specification (Svensson, 2009). However, in this project we only explore safety specifications using safety monitors.

Monitors are Erlang modules which export three functions; *init*, *stateChange* and *monitorType* of arity 1, 3 and 0, respectively (Fredlund & Svensson, 2007) (Earle & Fredlund, 2008).

- *init* is called when the monitor is started, at the beginning of the verification process.
- *stateChange* is the function that is called by McErlang (by simulation or model checking algorithm) using three parameters:
  - ProgramState
  - MonitorState
  - VerificationStack
- *monitorType* specifies what kind of monitor is running, which is in our case always a **safety** monitor.

When we do verification based on checkpoints in a Timed Rebeca model, we use the *VerificationStack* which is given to the *stateChange* function each time a translated model takes a step in an execution. From the *VerificationStack* we can then obtain all possible probes (generated from checkpoints in Timed Rebeca) in a model and evaluate them with McErlang monitors.

The verification process of a model can only have one monitor asserted at each time.

4.2 Safety Specifications

In Chapter 3 we explained how we adapted McErlang to be able to explore a state graph of a Timed Rebeca model based on the formal semantics of Timed Rebeca explained in (Aceto et al., 2011).

After extending the mapping to support the timed semantics in McErlang, we had a couple of predefined safety monitors. These monitors provide us with valuable safety checks and thus extend the operability of analyzing timed models in Timed Rebeca. Next sections explain how we use safety monitors and how we use them with our newly proposed checkpoints.

4.2.1 Deadlock Detection

An important property can be checked for *non-terminating* critical systems deadlock. A deadlock verification monitor in McErlang is shown in Listing 12. So, this monitor guards against program deadlocks by viewing the process status. If the status of the process is marked as "blocked" the process is considered deadlocked. This monitor then guarantees deadlock freedom within a non-terminating Timed Rebeca model.

```
reactiveclass DeadlockingExample(3) {
  knownrebecs { DeadlockingExample proc; }
  statevars { }

  msgsrv initial() {
   proc.go();
  }

  msgsrv go()
  {
    /* Random pick of integer */
    int rand = ?(1,2,3,4,5);

    delay(1);

    if(rand != 5) {
      sender.go();
    }
  }
}

main {
    DeadlockingExample proc1(proc2):();
    DeadlockingExample proc2(proc1):();
}
```

Listing 11: Timed Rebeca Model - Deadlocking Rebecs

```
monitorType() -> safety.

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

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

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

Listing 12: McErlang - Deadlock monitor

Example of a deadlock in Timed Rebeca. Timed Rebeca model provided in Listing 11 demonstrates a deadlock. The model presents two rebecs; namely *proc1* and *proc2*. In the initialization methods they send one another a message. Both, then send a message back only if the variable *rand* is **not** 5. This variable is randomly assigned with the integer 1 to 5, and will account for a path where none of the rebecs will send a message back. This model will therefore end in a deadlocked scenario when having all of states explored.

Verification of the example presented in Listing 11, took McErlang 0.010 runtime seconds to halt the model checking process showing a deadlock. But soon as we altered the `if`

clause to `rand` $!= 6$ the verification terminated normally with no-deadlock. The state space had 444 states and 818 transitions and was explored in 0.210 seconds. It should be noted that running the monitor on a terminating Timed Rebeca model will always end by violating the deadlock freedom. This is because each rebec in our translated code is always listening for new messages and when message passing ends, the monitor will notify that we have a deadlock.

4.2.2 Maximum Queue Detection

One of the things that was hard to implement in the mapping of Timed Rebeca was the restriction on queue of a rebec. From (Sirjani et al., 2004) it is explained that Rebeca needed to disallow unbounded message queues due to abstraction techniques to make the models finite. This safety check will however give the modeler the possibility of checking for maximum queue length of a rebec, which in some cases is useful.

Example of a queue overflow in Timed Rebeca. Presented Timed Rebeca model in Listing 13 depicts a model that has the possibility of exceeding its bounded queue which is in this example set as 2. The monitor is then presented in Listing 14.

The Timed Rebeca model has two rebecs, named *proc1* and *proc2*. Each rebec will start by sending a message to the other rebec. Then both delay with a non-deterministic choice of 1 or 2 time units. This will have the effect that at some point both of the processes could have 3 messages in its message bag. This is because when a rebec executes the message server `go()`, it sends two messages to the sender.

```
1  reactiveclass QueueViolation(2) {
2    knownrebecs { QueueViolation proc; }
3    statevars { }
4    msgsrv initial() {
5     /* Send first message */
6     proc.go();
7    }
8    msgsrv go()
9    {
10     /* Delay by 1 or 2. */
11     delay(?(1,2));
12     /* Send two messages */
13     sender.go(); sender.go();
14   }
15 }
16 main {
17     QueueViolation proc1(proc2):();
18     QueueViolation proc2(proc1):();
19 }
```

Listing 13: Timed Rebeca Model - Queue Violation

```
monitorType() -> safety.

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

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

Listing 14: McErlang - MaxQueue monitor

When using the monitor presented in Listing 14, we need to provide it with a `MaxQueueSize` parameter that defines what the maximum size of a queue is. The model provided in Listing 13 was verified with `MaxQueueSize` set as 2. The exploration violated the monitor with the shortest path of 29 states and 28 transitions after 2.180 seconds. This occurs at time 2 when one of the processes delayed by 2 and the other one by 1, therefore one of them ended by having 3 messages inside its message bag. If the model is to be checked with `MaxQueueSize` as 3, the monitor will be violated after exploring 5577 states and 7384 transitions. This took McErlang 17 seconds, to verify. This is normal as there is a finite path in the model state-space that leads to a process to have infinitely many messages in its message box.

4.2.3 Timed Rebeca Checkpoint Monitor

The main purpose of checkpoints, when model checking in Timed Rebeca is to be able to make it easier for the modeler to verify certain safety properties. To do that a monitor template is created in this project. An example of using the template is depicted in Listing 15.

```
monitorType() -> safety.

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

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

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

Listing 15: Pseudo Timed Rebeca checkpoint monitor for McErlang (Template)

The monitor depicted in Listing 15 shows how to detect if a checkpoint occurs in any path in the model. It uses one of the predefined functions `checkLabelCheckPoint` which takes an argument of a checkpoint label. The monitor then returns *satisfied* if no checkpoint with the specified label occurred in the state, otherwise it will halt with a *violation*. If the verification terminates without a violation, we are guaranteeing that the checkpoint never happens in any paths of the model.

The monitor has two parameters; *label* of the checkpoint (`CheckpointLabel`) and the *term* (`CheckpointTerm`) which is not necessarily used in all the functions predefined in the monitor. These parameters and the following predefined functions make it easier for the modeler to write their safety specifications. The following functions are defined in this project to make it easy for the modeler to write the specification without dealing with complexity of writing McErlang monitors from scratch. The functions available are;

- Checking if a dropped event happens for a message server, because of the deadline.
  - `checkDropMsgsrv`, with arity of two; actions in a state and a message server name.
- Checking if a checkpoint occurs.
  - `checkLabelCheckPoint`, with the arity of two; actions in a state and a *label* of a checkpoint.
- Compare the checkpoint *term* with an integer or boolean.
  - `checkTermMaxValue`, with the arity of four; actions in a state, checkpoint *label*, *term* and a maximum value of the *term*.
  - `checkTermMinValue`, with the arity of four; actions in a state, checkpoint *label*, *term* and a minimum value of the *term*.
  - `checkTermValue`, with the arity of four; actions in a state, checkpoint *label*, *term* andd a value which the *term* should be equal to (boolean or integer).

Full description of the monitor is presented in Listing 40 located in Appendix C. The monitor will be the basis for writing safety specifications for Timed Rebeca.

4.3 Discussion

This Chapter presents the first step in safety verifcation of models written in Timed Rebeca. Properties are written in Erlang and are hard to write without Erlang knowledge. We use checkpoints in Timed Rebeca and map them to McErlang probes when we do translation to Erlang. Probes expose program variables to be used with McErlang monitors. Timed Rebeca monitor was presented and makes it easier to write simple properties. In the future we want to have special property language that goes along with the Timed Rebeca model. The properties would then be translated to Erlang to form the monitors.

Note that a full version of the monitor template is presented in Listing 40, located in Appendix C.

Models can be faulty and result in Zeno effects, or other cycling behaviors. Static analysis of models before verifying is useful as even the smallest models could produce infinite system states in the generated Erlang code.

Chapter 5

Discrete-Event Simulation Approach

Simulating a system that has many possible outcomes is a process of generating the non-deterministic behaviors of it and gather data from the simulation over a period of time. Models that have these settings are hard to predict as they evolve with time. Simulating abstract behaviors of a system can help with estimating how the system will evolve with simple statistical methods applied on the data generated from the simulation process.

Discrete-event simulation (DES), is representing the operation of a system as a chronological sequence of events. Each event occurs at a certain time and marks a change in the system model (Robinson, 2004). In this project we define these events when a message is taken from a message bag of a rebec or when a checkpoint is executed in the model.

The generated data for a model can contain multiple simulations and each simulation can contain many traces. To evaluate these traces, we need methods for extracting information out of the generated data set. The information is then used to reason about some behavioral features of the model. This is done with tools that support analysis of the generated data, and offer what we refer to as *discrete event simulation approach* (Ross, 2006). This kind of analysis has been used in many simulation tools like Tortuga (Mitre, 2009) or SimPy (Matloff, 2011). These tools are efficient in simulating systems that are using a process-oriented object model similar as Java, C++, or Python languages. But for paradigms that include distributed event-based asynchronous design, the simulation task can imply in more challenges when modeling the system (Jacobs, Lang, & Verbraeck, 2002). Research has though been done on changing the process-oriented simulation language to use event-based architecture for being able to capture the distributed nature of a system (Snowdon & Charnes, 2002). This however often needs experienced persons when carrying out the simulation process.

We wanted to offer this kind of analysis with Timed Rebeca and be able to provide basic statistical methods on the traces generated. To attain this, the tool-set which is a combination of three components; *timedreb2erl* (translation), *timedrebsim* (simulator), and *timedrebanalyis* (analysis) are implemented to manage simulations for Timed Rebeca and to be able to apply analysis techniques on them.

5.1 Discrete-Event Simulation with McErlang Timed Semantics

From using our mapping to McErlang with timed semantics we were opening the possibility to use it for simulation as well. The advantage of using timed semantics is that we use explicit-state time passage. This means that we do not have to depend on local system time when simulating the generated Erlang code. Precise time passage is then attained because we compute the time instead of waiting for a system clock. In the previous mapping, when executing we had time-slides while running the sequential code within a message server. This caused a deviation in the expected timing of the corresponding model compared to the formal semantics of Timed Rebeca as these time-slides accumulated. For example see Listing 21, when the message server in Listing 16 is iterated for 65628.6 seconds the rounded value of time-slides from the expected time was 169.6 seconds or 0.3%. Noticeable time-slide occurred after 527 seconds as the expected time was then 526.

```
1     msgsrv calculate() {
2       int i = 0;
3       int s = ?(1,2,3,4,5);
4       int x = i*s;
5       inc = inc + 1;
6       checkpoint(incremented,inc);
7       self.calculate() after(1);
8     }
```

Listing 16: Timed Rebeca message server for testing time-slides

The newly proposed mapping is a solution for this problem as we are using explicit-state simulation where messages (actions) that do not include timeouts will not cause elapse of time. Furthermore, to show the efficiency of using simulation instead of execution we depict the gain of data generation in Table 5.1, by comparing simulation to execution on case studies from Chapter 6.

Model	Simulation using timed semantics	Simulation using execution runs	Result
Ticket Service	60 seconds	20857 seconds	347% gain
Fichers Mutual Exclusion	60 seconds	41602 seconds	693% gain
Elevator System	60 seconds	46857 seconds	747% gain

Table 5.1: Comparison of doing simulation and execution. Each simulation (using new timed semantics for McErlang) is executed with a time limit of 60 seconds and compared with how long it takes to get to the similar point with the use of executing with native Erlang. The gain depends on how much the model is delaying.

When the translation of a Timed Rebeca model to Erlang is successfully done we need to configure McErlang to do simulation or model checking. When we choose to use simulation, we use a different exploration algorithm presented in (Fredlund & Svensson, 2007) comparing to when we choose model checking. The simulation algorithm by default chooses the next program state in random fashion, and only explores one path of execution in the model.

5.2 Timed Rebeca Traceability

We can trace the simulation of Timed Rebeca model based on events and checkpoints. We defined event generation to happen when a message server takes a message out of the message box of a rebec. The information related to an event is:

- **Message sending time:** Time of when the message was sent.
- **Message arrival time:** Time of when a message is taken from the queue.
- **Message Expiration:** Information on expiration of the message.
- **Rebec Name:** Name of the rebec that the message was sent to.
- **Message Server Name:** Name of the message server corresponding to the message.
- **Parameters:** Parameters that were sent with the message.
- **Sender:** Name of the sender rebec.

These events are important to see the flow of information in the simulation and for seeing if any expired messages exist in any of the generated traces. Checkpoints are then used to identify certain scenarios that are interesting to the modeler and have more adjustable information. The information related to checkpoints is:

- **Time of checkpoint:** The time of the checkpoint.
- **Rebec Name:** Name of the rebec that includes the checkpoint.
- **Checkpoint label:** Label of the checkpoint.
- **Checkpoint terms:** Terms (additional terms) of the checkpoint.

Checkpoints can be added between statements. Therefore the modeler can analyze the flow of control inside a message server. Additional parameters of the checkpoints are used for additional data that can be used in analysis of the checkpoint.

5.3 Analysis Implementation

When performance evaluation begins for a model the usual approach is to first understand, if what you are evaluating is over finite units of time or over an uncertain or infinite time. These simulations are often categorized into (Kelton, 1997) (Law, 2010) (Goldsman, 1992),

- *terminating* simulations and
- *steady-state* simulations

For simulating a *terminating* simulation from a model the usual approach is to issue a simulation which has finitely many events and has a resulting random output V. This can be repeated n times and statistical approaches can be directly applied on the captured set of random outputs $\{V_1, ..., V_n\}$. As when measuring a *steady-state* simulation with $t \rightarrow \infty$, where t is the terminating time of the model, the simulation needs more consideration as you need to define how long you need to simulate. Different techniques have been proposed for statistically analysis of these simulations, namely replication, batch means, time-series models, standardized time series, regeneration cycles, and spectrum analysis (Kelton, 1997). These methods often boil down to breaking simulations into extents and applying the normal statistical approaches to it.

When performance evaluation of a simulation is performed we typically are looking for some quantity θ which is connected to the model. This quantity is often a segment of a *steady-state* simulation but can be interpreted as many repetitive *independent observations*. By using independent observation we are allowing ourselves to stop the simulation at any point and only using valid observation collected to this point (pair of checkpoints or one checkpoint). Thus, in this project we estimate the performance of our traces with detecting *replication* of observations in our simulations. The method involves in *replicating* the dynamic behavior of a process by simulating the model multiple times, resulting in a set of random output observations from many replicated simulations. These observations are then the main **ingredients** of the quantity θ. Each observation can be,

- a pair of checkpoint, that uses the elapsed time from the first checkpoint to the second checkpoint as a value point of interest, or
- one checkpoint with a value point that is of an interest.

This depends on what kind of analysis is being carried out. Each observation **based on their term** then provides a quantity θ that is used to reason for the performance of the system.

5.3.1 Analysis Toolset

Mining simulations

To be able to reason about some quantity connected to a model, we implemented a set of tools. The high-level architecture is depicted in Figure 5.1. The tools provide analysis methods on Timed Rebeca models. The main component, the simulation wrapper (*timedrebsim*), was implemented to keep track of the stream of generated simulation traces from McErlang. The simulation wrapper provides a Timed Rebeca model to the translation component with a combination of setting arguments, which are; model arguments, simulation timeout and the number of simulations that is to be generated. The translation component (*timedreb2erl*) handles the mapping to Erlang which was first proposed in the work (Reynisson, 2011) but was extended in this project as the former chapters explained. When the translation component has finished, *timedrebsim* executes the translated model with McErlang. While the simulation executes, McErlang will stream in the generated output from the model to the simulator wrapper.

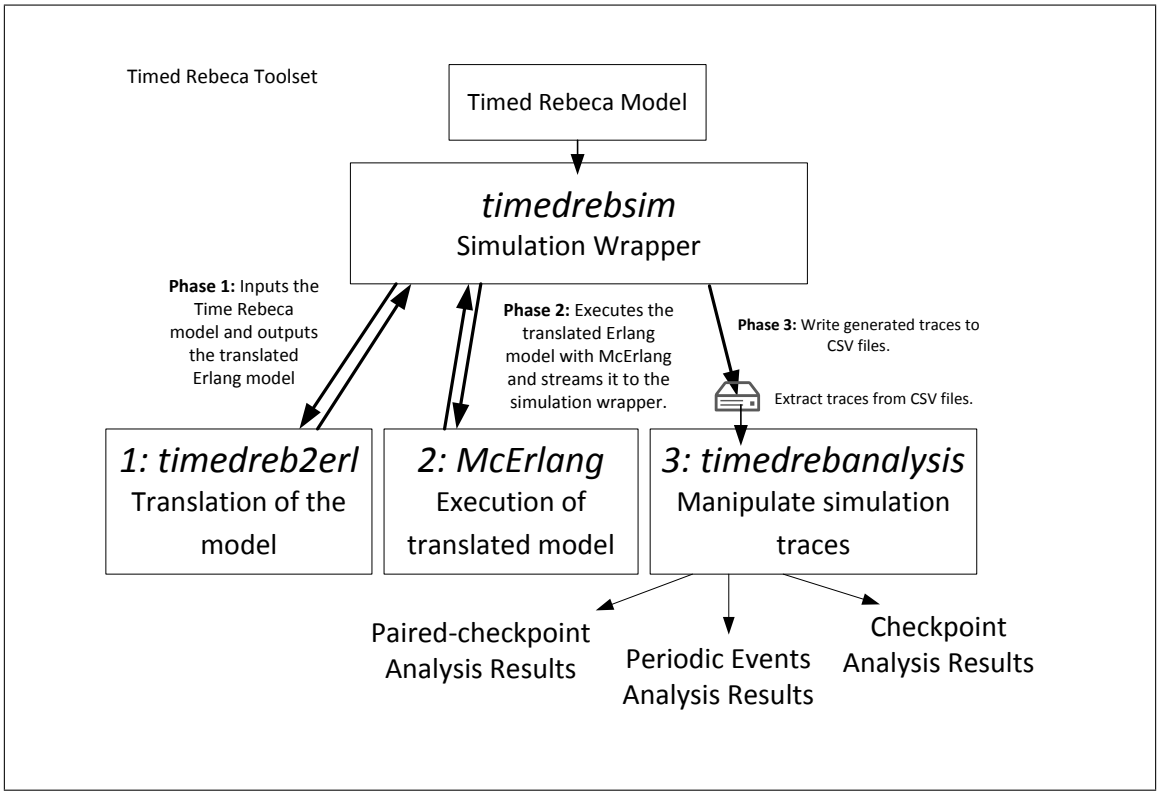

Figure 5.1: Architecture and flow of the data in the analysis tool-set. Bold arrow depicts actions that happen without user's interaction. CSV (Comma separate value) files stores all traces and are used for analyzing simulations.

While simulating, *timedrebsim* will collect all simulations; cataloging them into extents of simulation data-sets. Each data-set presents a generated output for each simulation

carried out by McErlang. When the simulation phase is over and the generated simulation output is in place, we can use the *timedrebanalysis* component to analyze the generated data.

Statistical Evaluation Methods

The main purpose of analyzing performance indicators of a model is to reason about the estimated probability of a quantity connected to the model. This helps us see an estimation of what will really happen in our system models. For processing the data we use simple statistical methods for obtaining expected values (means), variances and standard deviations. These metrics shows the modeler what is the expected value of the reviewed quantity and how the quantity distributes around the expected value. In addition we also provide the worst and best case estimate (WCET and BET in paired-checkpoint analysis).

The implementation of our analysis methods can have input of one or two (paired) checkpoints that account as one observation in a simulation data-set. All observations from a simulation are considered a *sample* which is a selection of potential results of an unknown population of results. We then want to estimate characteristics of the whole population with these samples and compute with what is observed as an estimate of the whole population.

We use a common approach to estimate the expected value and variance of our samples. Our data set consists of r simulations, each denoted as S_r. The sample size of each simulation has a sample size n_j of $X_1, X_2, ..., X_{n_j}$ checkpoint pairs, that are measuring the **same** quantity θ. Because the simulations are all from the same replicated behavior in a model we assume that we are allowed to combine the measurements to get an improved estimate of the real population. Thus we combine observations pairs with the sample size as $N_{combined} = \sum_{i=1}^{r} n_i$. We then calculate the sample mean (5.1) and the sample variance (5.2).

$$\overline{\bar{X}} = \frac{\sum_{i=1}^{r} \sum_{j=1}^{n_i} X_{ij}}{N_{combined}} \tag{5.1}$$

$$\sigma^2 = \frac{\sum_{i=1}^{r} \sum_{j=1}^{n_i} (X_{ij} - \bar{X}_i)^2}{\sum_{i=1}^{r} (n_i - 1)} \tag{5.2}$$

With this we can obtain a standard deviation (σ), which is the square root of the variance. These equations give us an estimation of the variance of the real population, giving us an estimate of the quantity θ.

Visualization Methods

Large simulated data can often offer difficulties to visualize. For this we provide linear data smoothing of the data. Following the smoothing, we reduce the data-set by picking periodically values from the data.

The smoothing is done by applying moving averages (Kenney & Keeping, 1962) to each value of the data before visualizing it. A moving average in practice is often used with time series and is to see the data trend (historical average). The method is used to smooth out short-term deviations and to create a longer-term trend of the data. The trend can be adjusted based on how large the data is. This is adjusted with a *degree* that is defined by the modeler. If each data point in the simulation is i_M where M is the number of the current value being calculated, MA_M the calculated moving average for the data point and N is the degree that has been chosen by the modeler then Formula 5.3 shows how we calculate each value in our data.

$$MA_M = \frac{1}{N}(i_M + i_{M-1} + ... + i_{M-(N-1)}) \tag{5.3}$$

When we have calculated moving averages for all values in our data we allow the modeler to choose a sample reduction period. This reduces the magnitude of the data that gives us more comprehensible visualization for the modeler. If our data contain V many value points and we have a reduction period of p we will reduce the data by V/p. This is done by taking only samples from the data every p times.

By using linear data smoothing method and then reduce the data by picking out values periodically we limit the possibility of picking values that are not showing the actual trend of the data.

5.4 Performance Evaluation of Simulations

After the generation of data from a simulation our goal is to study some quantity of the model and highlight useful information with methods proposed in the following sections. This information is used to reason about the model or discuss implementation ideas.

In this project we used events and checkpoints for evaluating certain types of performance indicators in models, namely

- *paired-checkpoint analysis*,
- *checkpoint analysis*, and
- *periodic events analysis*.

For each type of performance evaluation methods mentioned above, we use the following as value points to reason about the model:

- one checkpoint for checkpoint analysis,
- a checkpoint pair for paired-checkpoint analysis, and
- events for periodic events analysis.

For us to be able to measure these indicators we need to have the mandatory parameter of the checkpoint, meaning its label[1], and the optional term(s) (any data-type in Timed Rebeca).

- *Labels* are used to identify the checkpoints of interest:
  - In *paired-checkpoint analysis* we need to know the starting and ending checkpoint of a checkpoint pair to calculate the time difference between them.
- *Terms* are used for:
  - Grouping checkpoints with the same term in *paired-checkpoint analysis*, making each term a separate observation (but still use the label to define starting or ending checkpoints of a checkpoint pair).
  - Retrieving values from Timed Rebeca state or local variables to be used in *checkpoint analysis* and *periodic events analysis*.

Note that when doing *paired-checkpoint analysis* we need to decide if to group checkpoints by using *terms* or *labels*. Each group will then account for independent observations that are measuring the same quantity. Therefore for measuring the timing quantity

[1] A constant consisting of letter or number characters.

we collect multiple observations from the simulation, each observation being a paired checkpoint. The value of an observation is the elapsed time between checkpoints. When doing *paired-checkpoint analysis* and grouping observation with the *term*, it makes us capable of returning multiple measurements for each labeled checkpoint instead of only one when grouping with the *label*.

For the remainder of this section we go into what is involved in measuring these indicators.

5.4.1 Paired-checkpoint Analysis

Evaluation of a timing between a checkpoint pair in a model is what we call *paired-checkpoint analysis* in this project. When the data generation of a simulation has been completed we want to be able to measure the performance for a timing quantity connected to the model. For measuring the timing quantity we gather multiple observations from the simulation, each observation being a paired checkpoint. The value of an observation is the elapsed time between checkpoints. Each checkpoint pair can be initiated with a starting checkpoint of the types:

- **Restricted**: The checkpoint pair shall be inside the same rebec.
- **Global**: The checkpoint pair does not need to be inside the same rebec.

This means that each starting checkpoint in a pair needs to be defined by the modeler with either the type global or restricted. Note that ending checkpoints are not defined as global or restricted.

A good way of demonstrating types of starting checkpoints is to follow abstract traces from a simulation in Figure 5.2.

1. $\xrightarrow{S}$ ⟨ *checkpoint*, rebec1, Label: `startReq`, Term: [*Restricted*, [Term1]] ⟩ $\xrightarrow{\tau}$ $\cdots$
2. $\xrightarrow{\tau}$ ⟨ *event*, rebec1, *message server*, Parameters ⟩ $\xrightarrow{\tau}$ $\cdots$
3. $\xrightarrow{S}$ ⟨ *checkpoint*, rebec1, Label: `startReq`, Term: [*Global*, [Term1]] ⟩ $\xrightarrow{\tau}$ $\cdots$
4. $\xrightarrow{\tau}$ ⟨ *event*, rebec1, *message server*, Parameters ⟩ $\xrightarrow{\tau}$ $\cdots$
5. $\xrightarrow{E}$ ⟨ *checkpoint*, rebec2, Label: `endReq`, Term: [Term1] ⟩ $\xrightarrow{\tau}$ $\cdots$
6. $\xrightarrow{E}$ ⟨ *checkpoint*, rebec1, Label: `endReq`, Term: [Term1] ⟩ $\xrightarrow{\tau}$ $\cdots$

Figure 5.2: Abstract example of a generated simulation. We denote $\xrightarrow{S}$ as a mark for starting checkpoint of a checkpoint pair, $\xrightarrow{E}$ as a mark for ending checkpoint of a checkpoint pair, and $\xrightarrow{\tau}$ for any other transition.

In Figure 5.2, the individual observation is a pair of checkpoints with the start label `startReq` and the end label `endReq`. The term of each checkpoint contains param-

eters that is used to group checkpoint pairs that are measuring the same timing quantity together. A restricted checkpoint is denoted as *restricted* which is the first argument of a *Term*. This helps us in scenarios when we have potentially multiple ending checkpoints of an observation. When using restricted starting checkpoints we are only allowing the source rebec to match with an ending checkpoint. Global starting checkpoints on the other hand can be paired by any rebec in the model.

The two starting checkpoints in Figure 5.2 are located in line 1 and 3. The checkpoint in line 1 is *restricted* and is initiated by *rebec1*. This starting checkpoint is ended with the checkpoint in line 6. Note that the checkpoint in line 5 is also an ending checkpoint and happens before line 6, but is from a different rebec. The ending checkpoint of line 5 is originating from *rebec2* while the starting checkpoint is from *rebec1*. This makes it not possible to pair with the checkpoint initiated from the starting checkpoint in line 1. Therefore the starting checkpoint initiated from line 3 is ended with the ending checkpoint in line 6, but not in line 5.

This approach gives us valuable information for models that include non-deterministic properties and repeated behaviors. Furthermore, by adding lists to Timed Rebeca, we also can measure waiting time in a queuing system, presuming that the modeler uses the random features of Timed Rebeca to model the arrival sequence as a Poisson process (Cooper, 1981; Ross, 2006). Such studies often are represented as server queues like in Figure 5.3 and 5.6. These models are ideal for demonstrative purposes.

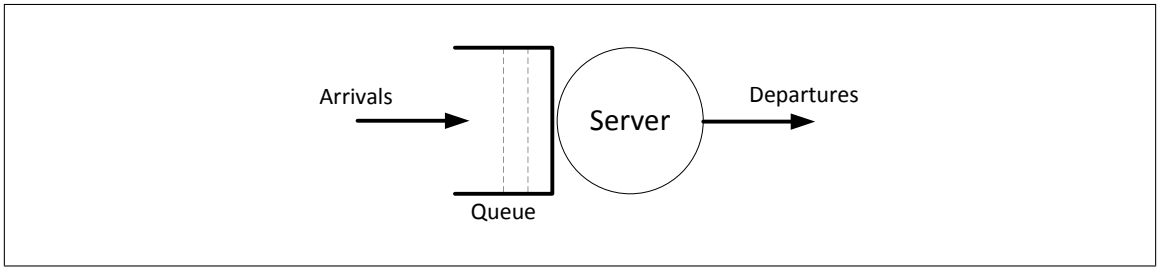

Figure 5.3: A single server queue system

For the model depicted in Figure 5.3 we have interest in estimating; *waiting times* of arrivals inside the queue, *throughput* of the system, or estimating expected values of processing timing of the server.

In Listing 17 we show a Timed Rebeca model of a single queuing system. The model consists of two reactive classes, *ArrivalProcess* and *Server*. The server is defined as rebec *srv* and is instantiated from the reactive class Server. The queue is defined as the rebec *proc* and is instantiated from the reactive class ArrivalProccess. Arrival sequences to the queue is sent with a non-deterministic delay of 1, 2, 9 or 10 (line 9). The server then

processes requests in 1 to 5 time units. It should be noted that we include checkpoints when requests are:

- sent from the *proc* (when the message server *sendRequest* is executed),
- processed by the *srv* (when the message server *serverInit* is executed),
- arriving into the queue (when the message server *queueRequest* in *srv* is executed),
- leaving the queue (finished executing the message server *queueRequest* in *srv*), and
- finished by *srv* (finished executing the message server *serverInit* in *srv*).

These checkpoints are included when these actions are taken to be able to measure the performance of the system including throughput (request initiating and finishing), queue waiting times and how long requests are being processed by the server.

The model depicted in Listing 17 was simulated with *timedrebsim* 5 times, each with a timeout of 200 seconds. The simulation resulted in a dataset of 258084 checkpoint pairs.

In Table 5.2 we measured three response indicators with *timedrebanalysis*;

- Throughput of the queuing system is measured by using the starting checkpoint in line 8 (with the label *requestStart*) and the ending checkpoint in line 40 (with the label *requestFinished*).
- Queue waiting times is measured by using the starting checkpoint in line 25 (with the label *requestInQueue*) and the ending checkpoint in line 37 (with the label *serverBegins*).
- Server performance is measured by using the starting checkpoint in line 37 (with the label *serverBegins*) and the ending checkpoint in line 40 (with the label *requestFinished*).

Response Indicator	Expected Response	Sample Standard Deviation	Minimum Value	Maximum Value	Median Value	Checkpoint Pairs
Throughput	10.3	3.7	2	42	11	258084
Queue waiting time	0.6	1.7	0	24	0	258084
Server Performance	3	1.4	1	5	3	258084

Table 5.2: Paired-checkpoint evaluation of the single queue model. Values represents time units and are estimates of elapsed times between checkpoints. Expected Responses are averages of all elapsed times.

These measurements give the modeler estimates for response, waiting or throughput times of their model as it is not trivial when simulating non-deterministic behaviors.

```
reactiveclass ArrivalProccess(2) {
 knownrebecs { Server srv; }
 statevars { int token; }

 msgsrv initial() { token = 0; self.send(); }

 msgsrv sendRequest() {
    checkpoint(requestStart,token); /* Start of Request */
    int DelayArrivalSend = ?(1,2,9,10); /* Non-Deterministic Delay */
    delay(DelayArrivalSend); /* Delay for Arrival Requests */
    srv.queueRequest(token); /* Send token to Queue of the Server */
    token = token + 1; /* Increment Token */
    self.sendRequest(); /* Iterate */
 }
}

reactiveclass Server(2) {
 knownrebecs { ArrivalProcess proc; }
 statevars { boolean serveractive; list<int> qserver; }

 msgsrv initial() { serveractive = false; }

 msgsrv queueRequest(int tok)
 {
     checkpoint(requestInQueue,tok); /* Mark Queue input */
     qserver.insert(tok); /* Insert token to Server Queue */
     if(serveractive == false) { /* Initiate Server if not running */
       self.serverInit();
     }
 }
 msgsrv serverInit() {
     serveractive = true; /* Server Active */

     if(qserver.size() > 0) { /* Safety */
       int reqToken = qserver.first(); /* FIFO, take first request out */
       qserver.remove(reqToken); /* Remove first request out of queue */
       checkpoint(serverBegins,reqToken); /* Mark request processing */
       int processtime = ?(1,2,3,4,5);
       delay(processtime); /* Non-Deterministic processing time */
       checkpoint(requestFinished,reqToken); /* Mark finished request */
       checkpoint(processQueueSize,qserver.size()); /* Mark size of the queue */
     }

     if(qserver.size() > 0) { /* Continue processing if queue not empty */
        self.serverInit();
     } else {
       serveractive = false; /* Server not active */ }
 }
}

main {
  ArrivalProccess proc(srv):();
  Server srv(proc):();
}
```

Listing 17: Timed Rebeca Model - Single Queue System

5.4.2 Checkpoint analysis

When doing checkpoint analysis we consider only one checkpoint as the observation each given time. The method uses the time and the term value of a given checkpoint. This is used for capturing some quantity which evolves over time. The method is then based on visualizing the results by plotting each simulation as an independent time series on a chart. In addition to visualization, we provide some statistical information related to the checkpoint being reviewed. This gives the modeler insights into how the observation evolves differently when simulating.

When doing checkpoint analysis on the model depicted in Listing 17 we have the interest of seeing how the queue size evolves. This is obtained by using the checkpoint *processQueueSize* placed in line 41. This checkpoint has at each given time the value *qserver.size()* assigned, which is the size of the server queue in the model.

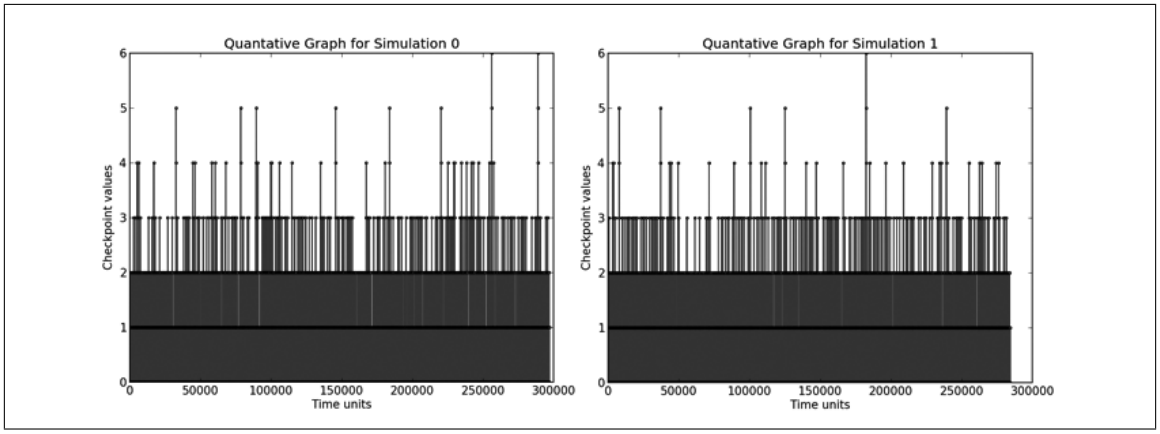

Figure 5.4: Checkpoint visualization of simulation 1 and 2 for the single queue model of Figure 5.3. Checkpoint value is the queue size of the server rebec. The presented graph shows raw data without any smoothing methods applied.

Simulation Number	Expected Value	Sample Standard Deviation	Minimum Value	Maximum Value	Median Value	Occurrences
1	0.2059	0.5254	0	6	0	53888
2	0.2037	0.5192	0	6	0	51642
3	0.2026	0.5112	0	5	0	50744
4	0.2037	0.5157	0	6	0	52781
5	0.2089	0.5263	0	7	0	49029
Total	0.2049	0.5196	0	7	0	258084

Table 5.3: Checkpoint evaluation of the single queue model. Values are estimates of the whole population.

Results of the analysis are shown partially in Figure 5.4 which shows the plots of how the queue size (values of the checkpoint) evolves over time. In addition statistical measurements are provided in Table 5.3.

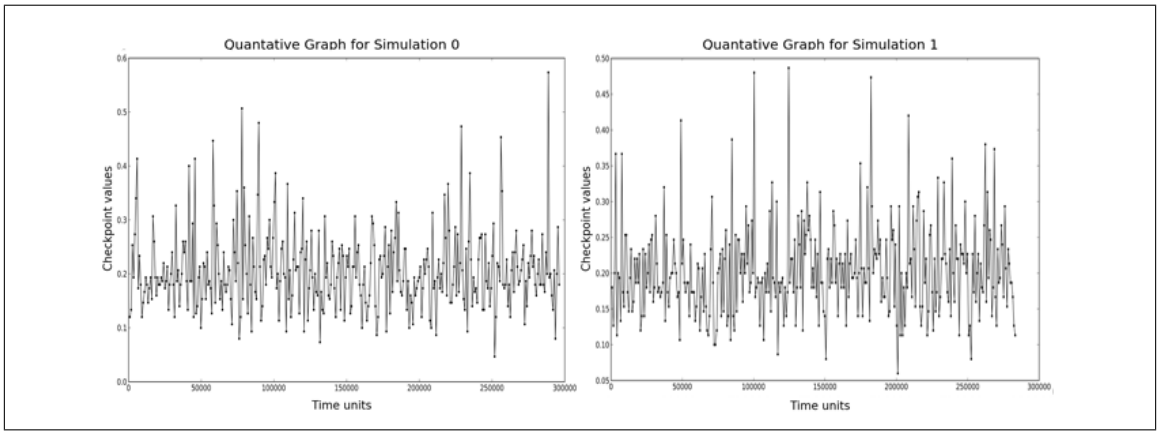

Figure 5.5: Checkpoint visualization of simulation 1 and 2 for the single queue model of Figure 5.3, with smoothed reduction. The presented graph shows data with moving average smoothing method. Smoothing degree was set to 150 and reduction period to 150.

Using checkpoint analysis shows the modeler valuable information on how the value of the checkpoint evolves as statistical measurements (like presented in Table 5.3) are often not presenting enough information. When looking at Table 5.3 and comparing it to the plots in Figure 5.4, we see that queue size is distributed between 0 and 6 which is not visible when looking at statistical information in Table 5.3. The statistical information shows that the queue size is more often near zero, and that the maximum size is around 5 to 7. By visualizing the data in addition of providing statistics we are getting more insights into how the data evolves.

Simulation results are typically very large data-set and not easily visualized. To facilitate visualization *timedrebanalysis* has the capability of reducing plots with applying Gaussian smoothing and moving average methods to the data-set. Following the smoothing we then reduce the results by choosing periodic points from the results. These setting can be configured by the modeler by defining smoothing degree and a reduce period.

Figure 5.5 presents checkpoint visualization of the same data-set shown in Figure 5.4 but with smoothing and reduction methods applied to it. This is explained in Section 5.3. This gives the modeler even more insight into the behavior of the model as the plot is more understandable when dealing with large data-sets.

5.4.3 Periodic Events Analysis

To see the frequency of behaviors in a model we use events to acquire periodic events analysis for Timed Rebeca models. We extract events from the collected data from each performed simulation of the model. We use this approach to see for example, how the load is distributed among components in a system (frequency of called message servers in rebecs). This can often offer useful information like in the system depicted in Figure 5.6 which shows an abstract multi-server queuing system. The multi-server queue has a queue that has 3 possible servers to send request to. Each server processes serves the requests. While the server is processing a request it is unable to receive any new request, therefore if server 1 is busy the queue will try server 2 and 3. If no server is free it will wait non-deterministically and try again with server 1. In this scenario we would want to know how these requests are distributed among the servers.

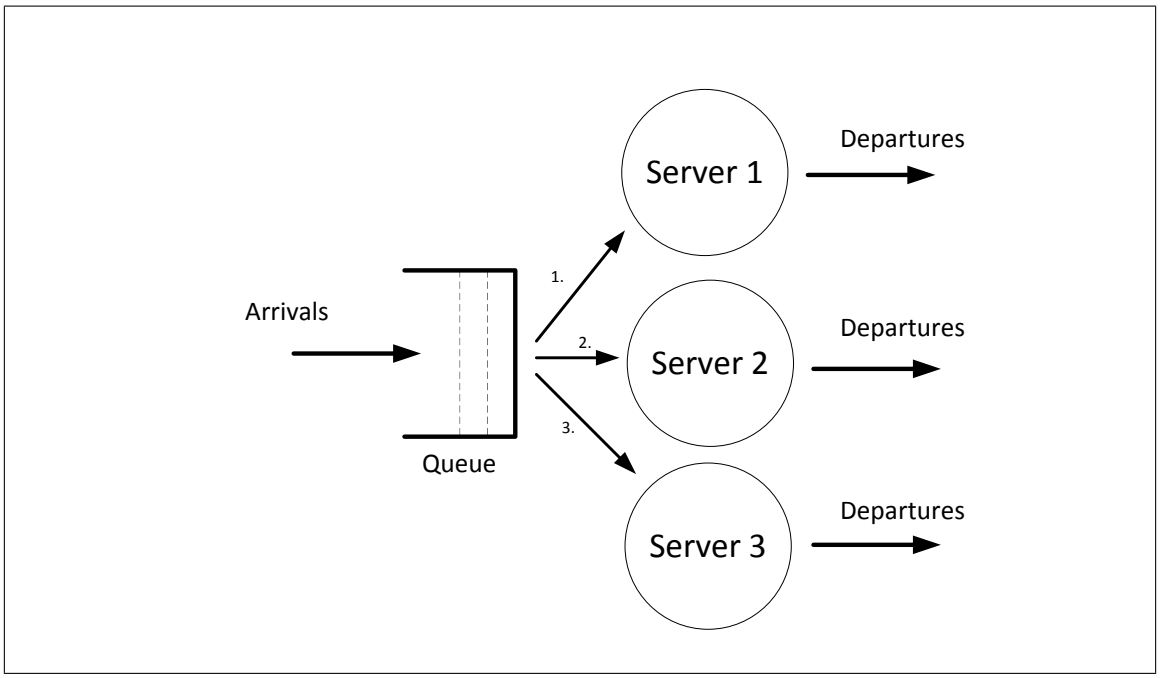

Figure 5.6: A multi-server queue system.

Timed Rebeca model for the multi queuing system in Figure 5.6 is depicted in Listing 18. The model consists of 3 reactive classes: ArrivalProcess, Queue and Server. The rebec *proc* is instantiated from the reactive class ArrivalProcess, and sends non-deterministically a request to the message server *queue* included in rebec *ques* which is instantiated from the reactive class Queue. Each request is sent with 1 to 5 time units delay. Three servers are defined as *srv1*, *srv2* and *srv3* all instantiated from the reactive class Server. These rebecs start by waiting for receiving message from the rebec *ques*. Each time a request is received by the rebec *ques*, it first tries to send it to the *srv1* rebec.

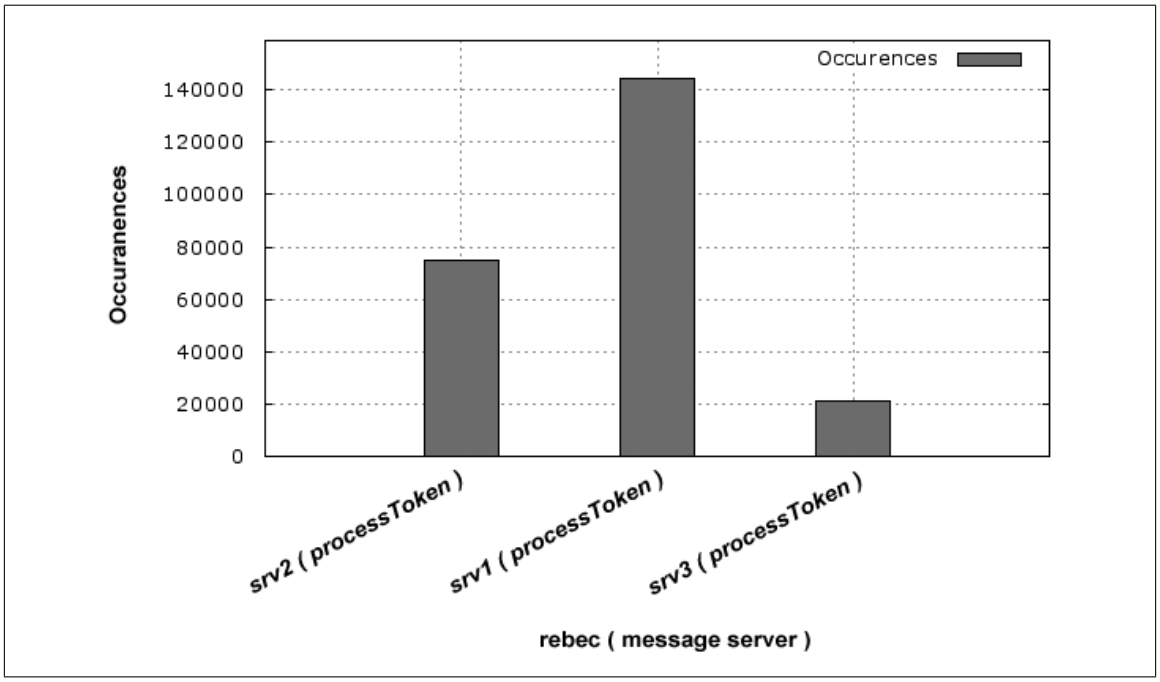

Figure 5.7: Periodic Events Analysis of all simulations of the multi queue model

If the first server is busy with another request, the rebec *ques* will try to send it to *srv2* rebec and so on. Each of the server rebecs then finishes processing the request after 1, 2, 3 or 6 time units before being again available for receiving a new request.

The model described above was simulated with *timedrebsim* 5 times, each with a timeout of 200 seconds. The simulation resulted in 240561 message server calls to *processToken* and was distributed among the three rebecs; Server1, Server2 and Server3 with values of 144391, 74826 and 21344 respectively.

The results are depicted in Figure 5.7. A noticeable difference in distribution of the requests can be seen, as the first server (rebec *srv1*) gets 60% of issued requests. This is because the rebec *ques* always asks *srv1* first, if it's free for processing, after that *srv2* and *srv3*.

```
1  reactiveclass ArrivalProccess(2) {
2   knownrebecs { Queue qa; }
3   statevars { int token; }
4   msgsrv initial() { token = 0; self.send(); }
5
6   msgsrv send() {
7      trace(requestStart,token);
8      delay(?(1,2,3,4));
9      qa.queue(token);
10     token = token + 1;
11     self.send();
12  }
13 }
14 reactiveclass Queue(2) {
```

```
15 knownrebecs { Server srv1; Server srv2; Server srv3; }
16 statevars { boolean srv1active, srv2active, srv3active; list<int> qserver; }
17 msgsrv initial() { srv1active = false; srv2active = false; srv3active = false; }
18
19 msgsrv queue(int tok) {
20   qserver.insert(tok); /* Insert request on the top of the queue */
21   self.sendrequest(); /* Send */
22   trace(queueSizeAfterAdd,qserver.size());
23 }
24 msgsrv sendrequest() {
25   if(srv1active == false) { /* Check if Server 1 is free */
26     srv1.processToken(qserver.first());
27     srv1active = true; qserver.remove(qserver.first());
28   } else if(srv2active == false) { /* Check if Server 2 is free */
29     srv2.processToken(qserver.first());
30     srv2active = true; qserver.remove(qserver.first());
31   } else if(srv3active == false) { /* Check if Server 3 is free */
32     srv3.processToken(qserver.first());
33     srv3active = true; qserver.remove(qserver.first());
34   } else {
35     delay(?(1,2,3)); /* Delay retry time */
36     self.sendrequest(); /* Retry */
37   }
38 }
39 msgsrv serverack(int Token, int ServerId) { /* Set Server as inactive */
40   if(ServerId == 1) {
41     srv1active = false;
42     trace(requestFinished,Token);
43     trace(processQueueSize,qserver.size());
44   } else if(ServerId == 2) {
45     srv2active = false;
46     trace(requestFinished,Token);
47     trace(processQueueSize,qserver.size());
48   } else if(ServerId == 3) {
49     srv3active = false;
50     trace(requestFinished,Token);
51     trace(processQueueSize,qserver.size()); }
52 }
53 }
54 reactiveclass Server(2) {
55 knownrebecs { Queue qs; }
56 statevars { int ident; }
57 msgsrv initial(int id) { ident = id; }
58
59 msgsrv processToken(int Token) {
60     trace(serverBegins,Token);
61     delay(?(1,2,3,6)); /* Non-deterministic processing time */
62    qs.serverack(Token,ident); /* Let the Queue know that the server is inactive */
63 }
64 }
65 main {
66  ArrivalProccess proc(ques):();
67  Queue ques(srv1,srv2,srv3):();
68  Server srv1(ques):(1); Server srv2(ques):(2); Server srv3(ques):(3);
69 }
```

Listing 18: Timed Rebeca Model - Multi Queue System

5.5 Discussions

This Chapter proposes three analysis methods, namely paired-checkpoint analysis, checkpoint analysis and periodic events analysis. For each method we present an example that shows the reader the scenarios where we can provide useful information.

A multitude of research question (methods) is still not answered regarding analysis of simulation for real-time systems. The methods proposed here only being few of many. These questions are left as future works.

Chapter 6

Case Studies and Experimental Results

In this section we present some case studies. For all these studies we construct a Timed Rebeca model and then perform analysis. The analysis will be based on what we have described in Chapter 4 and 5. First we carry out a verification of safety properties and then we use performance evaluation techniques to gain more insight into the dynamic behavior of each model.

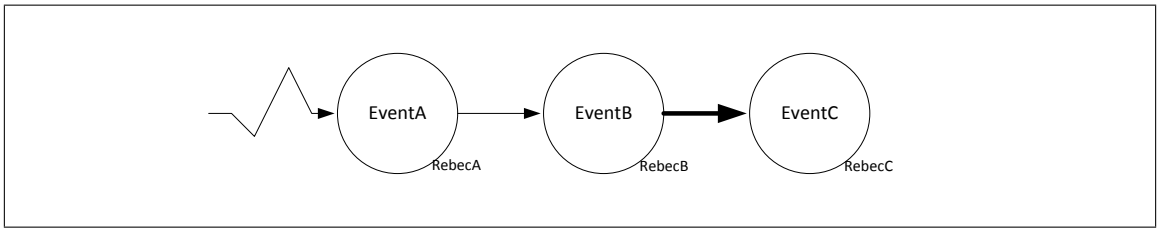

Figure 6.1: Example of an event graph. Initially the message server *EventA* in *RebecA* sends a message that triggers the message server *EventB* in rebec *RebecB*. Followed by a message sent by *EventB* that triggers the message server *EventC* of the rebec *RebecC*.

We present each model with event graphs (Buss, 1996). Event graphs are well-known for explaining discrete-event simulation models. The three elements of a discrete event system models are the state variables, the events that change the values of these state variables, and the relationships between the events (one event causing another). In an event graph the nodes represent events in the system and edges represent the scheduling of other events (causal relation between events) (Buss, 1996). Edges can be conditional (thick edge), mandatory (thin edge) or marking an initial event (jagged edge). Typically an edge can have certain guards like time or a Boolean, but here we only use reactive classes of the model as labels for an event. This label shows in what reactive class the event occurs. An example of an event graph is shown in Figure 6.1.

6.1 Ticket Service

In (Aceto et al., 2011) a ticket service example and some analysis are presented based on the execution of the Erlang code derived from the Rebeca model. The Timed Rebeca model consisted of two reactive classes: *Agent* and *TicketService*. Listing 19 shows the example written in Timed Rebeca. Two rebecs, *ts1* and *ts2*, are instantiated from the reactive class TicketService, and one rebec *a* from the reactive class Agent. The agent rebec *a* starts by sending a message and requesting a ticket from the first ticket service, *ts1*. The message has a deadline of *requestDeadline* time units. When the message is received by the ticket service *ts1* it issues the ticket in *serviceTime1* or *serviceTime2* time units. It does so by sending a message back to the agent rebec *a*. After the agent *a* sent the message to *ts1*, another message is sent to itself after *checkIssuedPeriod* time units. This is for checking if the message had been issued or not. If the ticket is issued we continue to the next customer and request a new ticket after *newRequestPeriod* time units. If the ticket has not been issued agent *a* immediately sends a message to the second ticket service *ts2* using the same approach as before. If the ticket is issued by the second ticket service *ts2*, the agent continues to his next customer after *newRequestPeriod* time units. This process then continues repeatedly. Note that the agent rebec *a* only issues the last requested ticket as some tickets could arrive sooner than prior request that was not expired by the deadline.

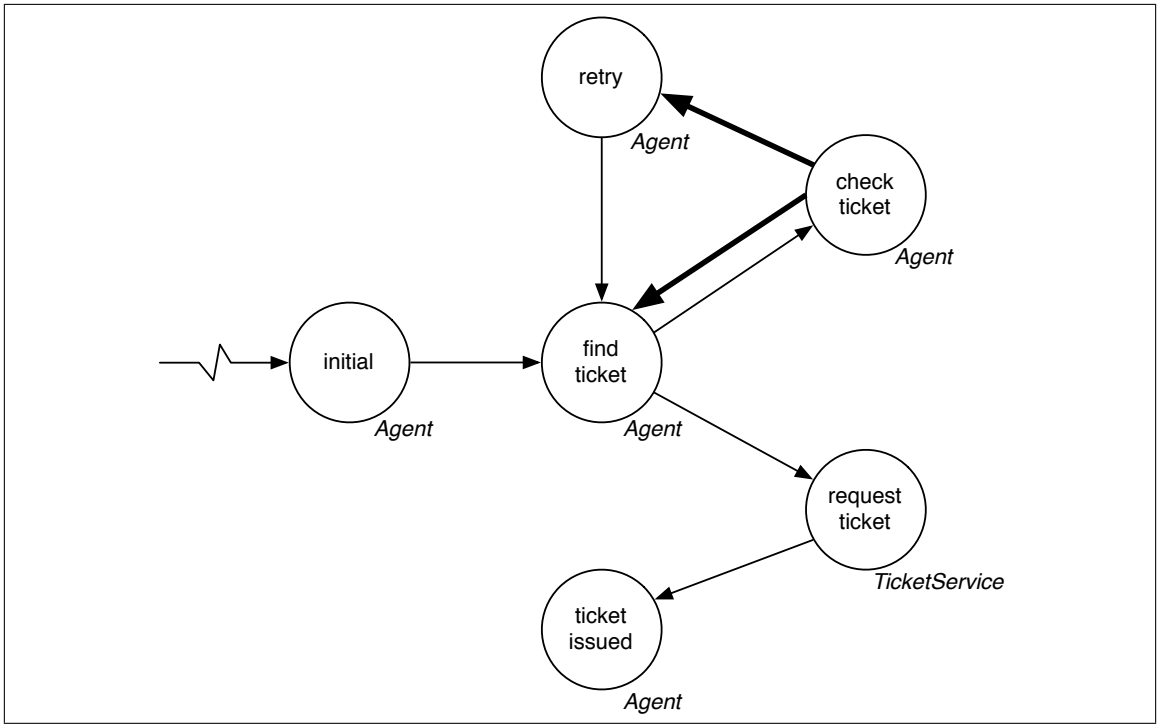

Figure 6.2: Event graph of the ticket service model.

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

Listing 19: Timed Rebeca Model - Revised ticket service example

6.1.1 Safety Verification

Experiments were carried out with different arguments passed to the model. The arguments provided are *requestDeadline*, *checkIssuedPeriod*, *retryRequestPeriod*, *newRequestPeriod serviceTime1*, *serviceTime2* and *maxIssued*. The original model from (Aceto et al., 2011) was revised and *maxIssued* was added to avoid state-explosion caused unbounded iterations. This was omitted when doing simulations.

Two safety properties were of interest:

- Are there any tickets that do not get issued (all tickets are issued).
- Are any of the tickets issued (possibility of a ticket to get issued).

The property used for verifying the model was formed as monitors (using the Timed Rebeca checkpoint template) and are presented in Listing 20 and 21. These monitors use checkpoints placed within the Timed Rebeca model in Listing 19. Checkpoints are seen in lines 18 and 20.

```
monitorType() -> safety.

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

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

  checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 20: Safety property to check if we ever get an non-issued ticket.

```
monitorType() -> safety.

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

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

  checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 21: Safety property to check if we ever get an issued ticket.

The monitors in Listings 20 and 21, show that the checkpoints of interest are using the labels *ticketissued* or *ticketnotissued*. The checkpoint of interest depends on if we are verifying if a ticket is **ever** issued or **never** issued. Safety verification is done by using

the predefined function checkLabelCheckPoint of the Timed Rebeca monitor template explained in Chapter 4.

For us to compare these experiments with ones presented in (Aceto et al., 2011) we used the property defined in Listing 21 to halt with a violation if an issued ticket occurred. Former experiments were carried out in an execution-manner and the results are presented in Table 6.1 (Aceto et al., 2011).

Setting	Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2	Result
1,2	2	1	1	1	3,4	7	Not issued
3	2	2	1	1	4	7	Not issued
4	2	2	1	1	3	7	**Ticket issued**

Table 6.1: Experimental simulation (execution-manner) results for ticket service (Aceto et al., 2011).

To confirm these results we used verification with the monitor and got the results presented in Table 6.3. The property got satisfied for settings 1, 2 and 3. These results confirmed that no ticket would ever get issued after 7 tickets being requested. The verification process took 54, 67 and 46 seconds, for settings 1, 2, and 3 respectively. Furthermore, setting 4 in Table 6.3 resulted in a violation and showed that the former results from (Aceto et al., 2011) was correct. The verification process took 2 seconds. These results confirm the former experiments.

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

Table 6.2: Verification results for ticket service. Property is satisfied if no ticket got issued.

In addition we checked if we could find a setting where all tickets get issued. For this we used the monitor in Listing 20. Table 6.3 shows the results were settings 6 and 8 had all the tickets issued.

Setting	Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2	Max Ticket Requests	Result
1	2	1	1	1	3	7	7	Violation (4584 states)
2	2	1	1	1	4	7	7	Violation (4926 states)
3	2	2	1	1	4	7	7	Violation (4440 states)
4	2	2	1	1	3	7	7	Violation (4360 states)
5	2	2	1	1	2	7	7	Violation (5227 states)
6	2	5	1	1	3	4	7	**Satisfied** (4807 states)
7	2	5	1	1	5	4	7	Violation (4807 states)
8	2	6	1	1	5	4	7	**Satisfied** (4807 states)

Table 6.3: Experimental verification results for ticket service. Property is violated if one or more tickets are not issued (all tickets have to be issued).

6.1.2 Performance Evaluation

From the verification results we now know what setting will end where some tickets got issued. For these settings we wanted to see the performance of the model. Two methods proposed in Chapter 5 can be of use, namely paired-checkpoint analysis and periodic events analysis. We used paired-checkpoint analysis to get more insight in how many tickets are issued and how long it took for a ticket to get issued. Periodic events analysis shows us how the issued tickets got distributed between ticket services. Table 6.4 shows the paired-checkpoint evaluation of the model. Each setting was simulated 5 times, each for 200 seconds. All checkpoints in the evaluation are marked as global as the starting checkpoint did not have many possible endings checkpoints. Furthermore when carrying out the paired-checkpoint evaluation we set *timedrebanalysis* to group checkpoints with labels.

Setting	Expected response	Standard Deviation	Minimum response	Maximum response	Median response	Starting checkpoints	Checkpoint pairs
4	3.0	0	3.0	3.0	3.0	519350	614
5	2.1	0.1	2	3.0	2.0	511709	51476
6	4.0	0	4.0	4.0	4.0	363891	81585
7	3.0	0	3.0	3.0	3.0	573551	286948

Table 6.4: Paired-checkpoint evaluation for Ticket Service. Settings are from Table 6.3 where all guarantee that some tickets get issued.

Given the performance analysis of the ticket service model, we see that setting 7 in Table 6.4 has the most issued tickets, around 50% of all requests. Settings 4, 5 and 6 had 0.1%, 10% and 22%. For setting 5 the expected response was 2.1 with a standard deviation of 0.1 which shows less response time than any of the other settings. Furthermore, periodic events analysis depicted in Figures 6.3 and 6.4 shows that for setting 7 we had

almost 66.8% of the issued tickets from ticket service 1 which shows that most tickets got issued immediately by the first ticket service or when retrying after *retryRequestPeriod*. Similar distribution trend is for settings 6 and 7 though setting 7 had more issued tickets. This is because the sum of *checkIssuedPeriod* and *retryRequestPeriod* is higher than both serviceTime1 and serviceTime2 making it more likely for the ticket to get issued after 2 tries.

An interesting property of the model was noticed. We got more issued tickets from setting 7, although the service time higher than settings 1 to 6. One would expect that higher service times would result in higher missed tickets. This does not seem to be the case.

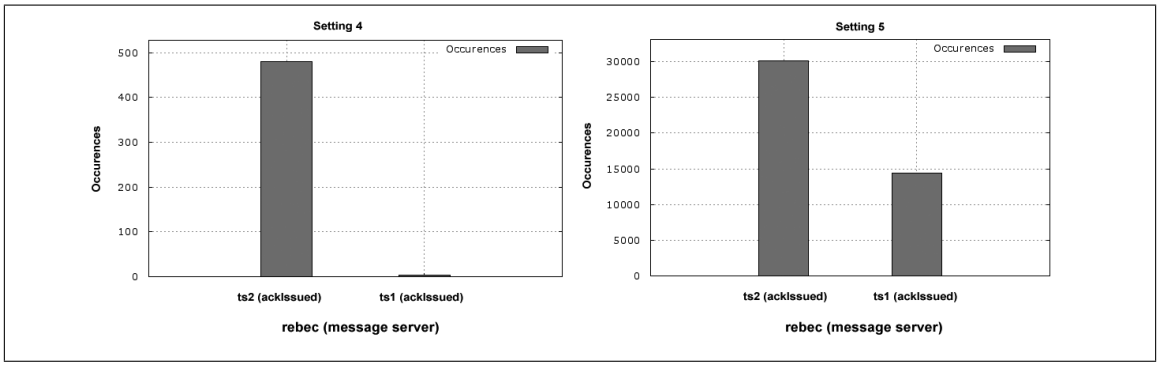

Figure 6.3: Periodic events analysis of setting 4 and 5 for the ticket service example

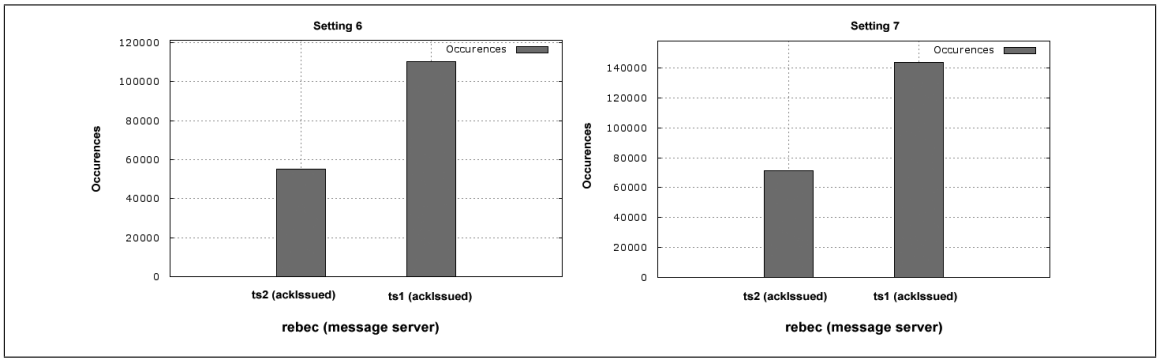

Figure 6.4: Periodic events analysis of setting 6 and 7 for the ticket service example

6.2 Elevator System

Elevator systems have been a prototypical example for presenting analysis techniques for real-time systems, like performance analysis, scheduling analysis, reliability analysis and WCET analysis (Shui, Mustafiz, Kienzle, & Dony, 2005) (Gomaa, 2000). This case study is a centralized elevator controlling system, which is an abstract model of the system presented in (Gomaa, 2000). Rather than assigning constants to all events in the system and do performance analysis by calculating if the system's hardware could carry out estimated worst case input scenarios (like the work in (Gomaa, 2000)), we proposed to use verification and simulation with Timed Rebeca and using our evaluation methods described in Chapter 4 and 5. These approaches give us a good estimation on worst case execution time (WCET) and how different scheduling policies perform.

We examined two types of elevator *movement policies* and for the most efficient one we experimented with three types of different *scheduling policies*. *Movement policies* define how an elevator moves between floors based on requests that have already been placed in the elevator queue (can be seen as an internal decision by the elevator based on the contents of its event queue). *Scheduling policies* define how floor requests are dispatched among elevators (can be seen as a decision made by a coordinator on dispatching requests).

The summary of combination of scheduling and movement policies in our experiments is presented in Table 6.5.

Implementation #	**Scheduling Policy**	**Movement Policy**
1	Shortest distance	Up priority
2	Shortest distance	Maintain movement
3	Shortest distance with movement priority	Maintain movement
4	Shortest distance with load balancing	Maintain movement

Table 6.5: Combination summary of policy implementations for the elevator system.

6.2.1 Model Design

The elevator system is presented in the form of an event graph in Figure 6.5. Abstract Timed Rebeca model is shown in Listing 22, which shows the structure and where the movement policies and scheduling policies are implemented. The full model is located on the Timed Rebeca website (Kristinsson, 2012). The model consists of four reactive classes; *Floor*, *Elevator*, *Coordinator*, and *Person*.

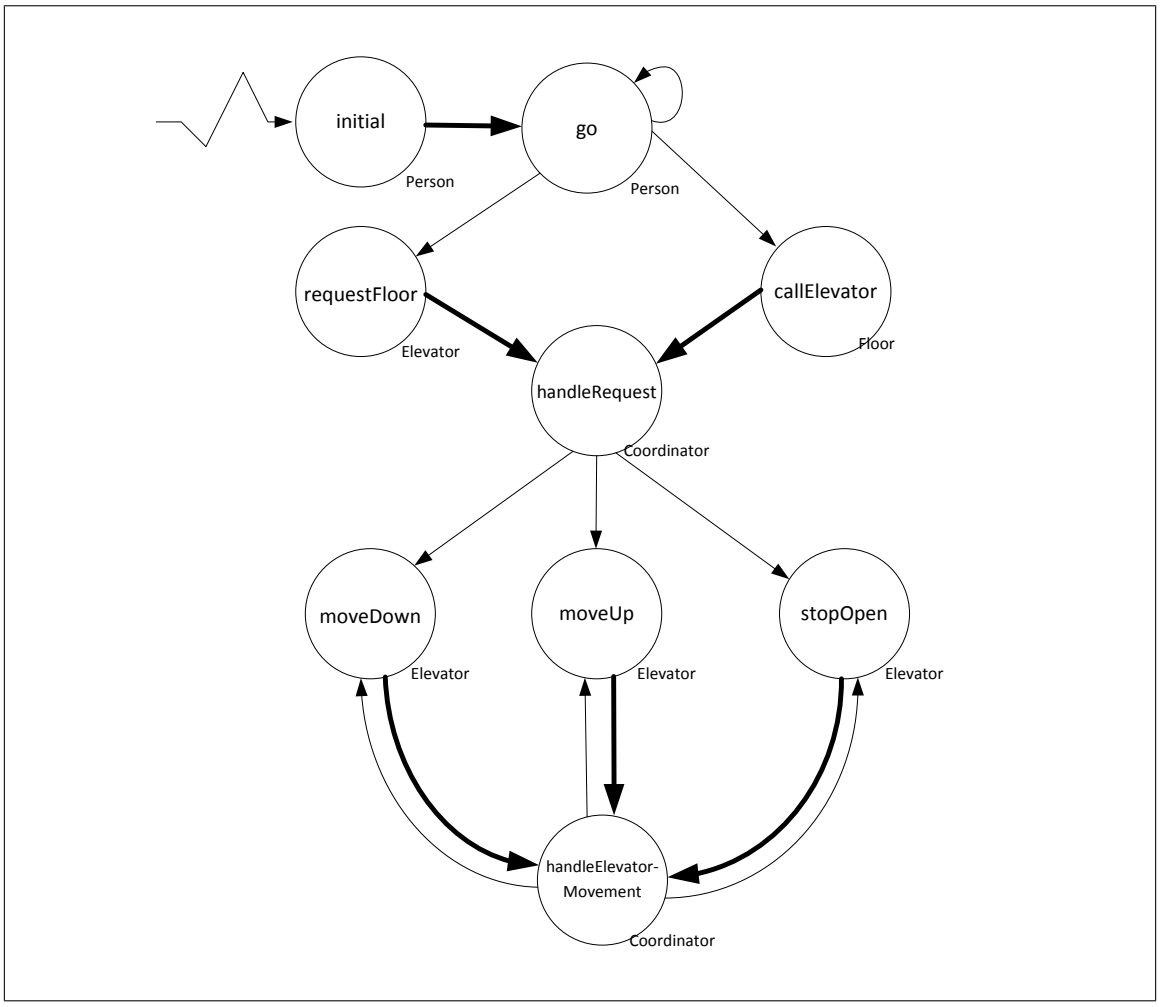

Figure 6.5: Event graph of the centralized elevator system.

Floors are defined as 10 rebecs, *floor1* to *floor10* which are instantiated from the reactive class Floor. Elevators are defined as 2 rebecs, *el1* and *el2* which are instantiated from the reactive class Elevator. Each Floor rebec represents a floor and each Elevator rebec represents an elevator in the model. The rebec *pers* instantiated from the reactive class Person is used to simulate a person who triggers periodic event and sends a message randomly to one of the floors or elevators rebecs. Random choices include sending a message *callElevator* or *requestFloor* to any of the rebecs instantiated from the Floor or Elevator reactive classes. Each random choice has a delay defined in the argument

simulationDelay which defines the time units until next iteration of sending a message *go* occurs. Number of iterations is defined in the argument *simulationIterations*.

When a message *callElevator* or *requestFloor* is sent, a message is immediately sent to the *coord* rebec which is instantiated from the reactive class Coordinator. The *coord* rebec handles all logic in the system, and decides which elevator gets the request or what movement should be made by an elevator. The two main message servers are *handleRequest* and *handleElevatorMovement*. The message server *handleRequest* decides what elevator gets the request. The first implementation of *handleRequest* is depicted in Listing 23, it presents a scheduling policy that we call: **shortest distance**. The message server *handleElevatorMovement* updates movements and queues of the elevators and decides whether to move them. The first implementation of *handleElevatorMovement* is depicted in Listing 24, it presents a movement policy that we call **up priority**.

```
1  env int simulationDelay, simulationItterations, elevatorMovementDelay,
       elevatorDoorDelay1, elevatorDoorDelay2, elevatorDoorDelay3, elevatorDoorDelay4;
2  reactiveclass Floor(4) {
3    knownrebecs { Coordinator coord; }
4    statevars { int floorIdent; boolean isRestricted; }
5
6    msgsrv initial(int floorID) {
7      floorIdent = floorID; // identity of the floor.
8      isRestricted = false; // requests from floors are not restricted.
9    }
10
11   msgsrv callElevator() {
12     checkpoint(startRequestFloor,floorIdent,isRestricted); // Global checkpoint
13     coord.handleRequest(floorIdent,true); // Send the request to the coordinator.
14   }
15 }
16
17 reactiveclass Elevator(4) {
18   knownrebecs { Coordinator coord; }
19   statevars {
20     int movementDelay; /* Delay of movement between floors. */
21     boolean isRestricted;
22   }
23
24   msgsrv initial(int mDelay) {
25     isRestricted = true; // Requests from elevators are restricted.
26     movementDelay = mDelay; // Set movement delay.
27   }
28
29   msgsrv moveUp(int floor) {
30     checkpoint(elevatorLocation,floor+1); // Checkpoint when elevator moves up.
31     delay(movementDelay); // Movement delay.
32     coord.handleElevatorMovement(1); // Send to coordinator that we have moved.
33   }
34
35   msgsrv moveDown(int floor) {
```

```
36      checkpoint(elevatorLocation,floor-1); // Checkpoint when elevator moves down.
37      delay(movementDelay); // Movement delay.
38      coord.handleElevatorMovement(-1); // Send to coordinator that we have moved.
39    }
40
41    msgsrv stopOpen(int floor) {
42      int randomdelay =
            ?(elevatorDoorDelay1,elevatorDoorDelay2,elevatorDoorDelay3,elevatorDoorDelay4);
43      delay(randomdelay); // Delay when opening and closing door.
44      checkpoint(endRequestElevator,floor); // Checkpoints that ends a request.
45      coord.handleElevatorMovement(0); // Send to coordinator that we have opened and
            closed.
46    }
47
48    msgsrv requestFloor(int floor) {
49      checkpoint(startRequestFloor,floor,isRestricted); // Requests from elevators are
            restricted.
50      coord.handleRequest(floor,false); // Send request to coordinator.
51    }
52
53    msgsrv stopRequest(int floor) { // Added due to cancelation of requests
54      checkpoint(endRequestElevator,floor); // Checkpoints that ends a request.
55    }
56  }
57
58  reactiveclass Coordinator(4) {
59    knownrebecs {
60      Floor flr1; Floor flr2; Floor flr3; Floor flr4; Floor flr5;
61      Floor flr6; Floor flr7; Floor flr8; Floor flr9; Floor flr10;
62      Elevator el1; Elevator el2;
63    }
64    statevars {
65      int el1location; int el1movement;
66      int el2location; int el2movement;
67      list<int> el1queue; list<int> el2queue;
68      int schedulingDelay;
69    }
70
71    msgsrv initial(int scDelay) {
72      el1location = 1; el2location = 1; // Start at floor 1
73      el1movement = 0; el2movement = 0; // 0 means not moving.
74      schedulingDelay = scDelay; // Scheduling delay.
75    }
76
77    msgsrv handleElevatorMovement(int movement)
78    {
79      /* Movment policies implemented here */
80    }
81
82    msgsrv handleRequest(int floor, boolean isFloor)
83    {
84      /* Scheduling policies implemented here */
85
86      checkpoint(elevator1QueueSize,el1queue); // Check Size of Elevator Queue
87      checkpoint(elevator2QueueSize,el2queue); // Check Size of Elevator Queue
88
89      /* After dispatching request to elevator queue */
```

```
90      /* Shall we start Elevator 1 - is it idle?. */
91      if(el1movement == 0 && el1queue.size() > 0) {
92        if(erlang.next(el1queue,el1location,-1) != -1 &&
              erlang.next(el1queue,el1location,-1) != el1location) {
93          el1movement = -1;
94          el1.moveDown(erlang.next(el1queue,el1location,-1));
95        }
96        else if(el1up != -1 && el1up != el1location) {
97          el1movement = 1; // Update movement
98          el1.moveUp(erlang.next(el1queue,el1location,1)); // Tell elevator to move up.
99        }
100       else { el2movement = -2; el1.stopOpen(el1location); }
101     }
102     /* Shall we start Elevator 2 - is it idle?. */
103     if(el2movement == 0 && el2queue.size() > 0) {
104       if(erlang.next(el2queue,el2location,-1) != -1 &&
              erlang.next(el2queue,el2location,-1) != el2location) {
105         el2movement = -1;
106         el2.moveDown(erlang.next(el2queue,el2location,-1);
107       }
108       else if(erlang.next(el2queue,el2location,1) != -1 &&
              erlang.next(el2queue,el2location,1) != el2location) {
109         el2movement = 1;
110         el2.moveUp(erlang.next(el2queue,el2location,1));
111       }
112       else { el2movement = -2; el2.stopOpen(el2location); }
113     }
114   }
115 }
116
117 reactiveclass Person(4) {
118   knownrebecs {
119     Floor flr1; Floor flr2; Floor flr3; Floor flr4; Floor flr5;
120     Floor flr6; Floor flr7; Floor flr8; Floor flr9; Floor flr10;
121     Elevator el1; Elevator el2;
122   }
123   statevars { int delayInSec; int iterations; }
124
125   msgsrv initial(int d, int i) {
126     delayInSec = d; iterations = i;
127     self.go(delayInSec,0);
128   }
129
130   msgsrv go(int delays, int incIter)
131   {
132     /* Randomly Select a floor or an elevator to make requests */
133     int floorCall = ?(1,2);
134     int flrNumber = ?(1,2,3,4,5,6,7,8,9,10);
135     int elvNumber = ?(1,2);
136     if(floorcall == 1) {
137       if(flr == 1) { flr1.callElevator(); } else if(flr == 2) { flr2.callElevator(); }
138       else if(flr == 3) { flr3.callElevator(); } else if(flr == 4) { flr4.callElevator();
              }
139       else if(flr == 5) { flr5.callElevator(); } else if(flr == 6) { flr6.callElevator();
              }
140       else if(flr == 7) { flr7.callElevator(); } else if(flr == 8) { flr8.callElevator();
              }
```

```
141      else if(flr == 9) { flr9.callElevator(); } else if(flr == 10) {
             flr10.callElevator(); }
142    } else {
143      if(elv == 1) { el1.requestFloor(flr); }
144      else if(elv == 2) { el2.requestFloor(flr); }
145    }
146
147    delay(delays); //Delay between simulations.
148    if(incIter < iterations) { self.go(delayInSec,incIter+1); }
149   }
150 }
151
152 main {
153   Floor flr1(coord):(1); Floor flr2(coord):(2); Floor flr3(coord):(3);
154   Floor flr4(coord):(4); Floor flr5(coord):(5); Floor flr6(coord):(6);
155   Floor flr7(coord):(7); Floor flr8(coord):(8); Floor flr9(coord):(9);
156   Floor flr10(coord):(10);
157   Elevator el1(coord):(elevatorMovementDelay); Elevator
          el2(coord):(elevatorMovementDelay);
158   Coordinator coord(flr1,flr2,flr3,flr4,flr5,flr6,flr7,flr8,flr9,flr10,el1,el2)
159     :(schedulingDelay);
160   Person pers(flr1,flr2,flr3,flr4,flr5,flr6,flr7,flr8,flr9,flr10,el1,el2)
161     :(simulationDelay,simulationItterations);
162 }
```

Listing 22: Timed Rebeca Model - Pseudo centralized elevator model. The message servers *handleElevatorMovement* and *handleRequest* implements movement and scheduling policies.

```
msgsrv handleRequest(int floor, boolean isFloor)
{
  delay(networkDelay);
  if(isFloor == true) /* Requests from floors. */
  {
    checkpoint(FloorRequestingElevator, floor);
    int choice = ?(1,2); // prepared calculations.
    if(erlang.contains(el1queue,floor) == 1 || erlang.contains(el2queue,floor) == 1) {
        // 0. Ignore the request because it is already in a queue. }
    /* 1. Either elevator on the same floor. */
    else if(el1location == floor || el2location == floor) {
      if(el1location == floor) { el1queue.insert(floor); }
      else if(el2location == floor) { el2queue.insert(floor); }
      checkpoint(endRequestElevator,floor);
    }
    /* 2. Both elevators on the same floor - Nondeterministic choice */
    else if(el1location == el2location) {
      if(choice == 1) { el1queue.insert(floor); }
      else if(choice == 2) { el2queue.insert(floor); }
    }
    /* 3. Which elevator has the least distance - First check for elevator 2 */
    else if(erlang.absolutevalue(floor-el1location) >
        erlang.absolutevalue(floor-el2location)) {
      el2queue.insert(floor);
    }
    /* 4. Which elevator has the least distance - Then elevator 1. */
    else { el1queue.insert(floor); }
  }
  /* Requests from elevators. */
  else
  {
    checkpoint(ElevatorRequestingElevator, floor);
    if(sender == el1 && el1location != floor && erlang.contains(el1queue,floor) == -1) {
       el1queue.insert(floor); }
    else if(sender == el2 && el2location != floor && erlang.contains(el2queue,floor) ==
        -1) {
       el2queue.insert(floor); }
    else if(el1location == floor || el2location == floor) {
       sender.stopRequest(floor); }
    else { sender.stopRequest(floor); }
  }

  /* Should we start any idle elevators ... */

}
```

Listing 23: Timed Rebeca Model - First implementation of request handler (message server). The message server depicts the **shortest distance** scheduling policy. Custom functions erlang.contains() and erlang.absolutevalue() are explained in Section 6.2.2

```
msgsrv handleElevatorMovement(int movement)
{
  /* Movement for elevator 1 - Up Priority Policy */
  if(sender == el1 && movement != 0) { /* If Elevator is moving */
    el1movement = movement;
    if(movement == -1) { el1location -= 1; }
    else if(movement == 1) { el1location += 1; }

    if(el1queue.size() > 0) { /* Are we on a requested floor. or should we move on? */
      if(erlang.contains(el1queue,el1location) != -1) {
        el1queue.remove(el1location);
        el1movement = -2;
        el1.stopOpen(el1location);
      } else {
        if(erlang.next(el1queue,el1location,1) != -1) { /* Try to go up first. */
          el1movement = 1; el1.moveUp(el1location); }
        else {
          el1movement = -1; el1.moveDown(el1location); } /* Then down. */
      }
    }
  }
  else if(sender == el1) { /* If elevator just stopped */
    el1movement = movement;
    if(el1movement == 0 && el1queue.size() > 0) { /* Should we restart the elevator. */
      if(erlang.next(el1queue,el1location,1) != -1) {
        el1movement = 1;
        el1.moveUp(el1location); }
      else {
        el1movement = -1;
        el1.moveDown(el1location); }
    }
  }
... /* Movement for elevator 2 - Up Priority Policy */ ...
}
```

Listing 24: Timed Rebeca Model - First implementation of movement handler (message server). The message server depicts the **up priority** movement policy. In the example we only show code for movement handling for elevator 1. Same code applies for elevator 2, but with replacement of the variables el1movement and el1location. Custom functions erlang.contains() and erlang.next() are explained in Section 6.2.2.

6.2.2 Model Extensions with Custom Functions

As discussed in Chapter 3, we extended Timed Rebeca to be able to use custom Erlang functions. This made us capable of using Erlang functions for specific behaviors that we were not able to do with Timed Rebeca. The extension consisted of three custom functions:

- Function that obtains absolute value of an integer.
- Function that checks if an element exists in a list.
- Function that checks integers as floors in a list and returns the closest floor based on location and direction.

The custom functions are shown in Listings 25, 26 and 27 and are used in *handleElevatorMovement* and *handleRequest* in Listings 23 and 24 respectively.

```
1 contains([],_) -> -1;
2 contains([H|T],E) ->
3  if
4   E == H -> 1;
5   true -> contains(T,E)
6  end.
```

Listing 25: Custom function - Check if an element exists in a given list. Returns 1 if found, otherwise -1.

```
absolutevalue(I) ->
  case is_number(I) of
   true -> abs(I);
   false -> -1
  end.
```

Listing 26: Custom function - Absolute value for an integer

These functions help us to decide which elevator is closest to a floor request and should be sent the request. Also we are capable to check if an element has already been added to a list which contained pending requests to an elevator. Double request was not allowed and will be ignored. The function *absolutevalue* shown in Listing 26 takes an integer and simply returns the absolute value using *abs* function in Erlang. This makes it easy to calculate the distance from the elevator's current floor and the request that is pending, with *erlang.absolutevalue* (requestFloor-elevatorLocation) where requestFloor and elevatorLocation are both integers. The function *contains* takes a Timed Rebeca list (Erlang lists) and an integer. It then checks if the head of the list is equal to the given integer and recursively uses the tail of the list as the next list to check. This continues until either the list is empty and returns -1 or we have a match and return 1. The function is called by using *erlang.contains* (requestList, elevatorLocation) where requestList is a list of integers and elevatorLocation is an integer.

The functions *next* with arity of 3,4, and 5 is a function that helps us in modeling the movement of the elevators. That is, they help to decide whether the elevator should go up or down, given a list of requests, current position and moving direction. The moving direction is the same as in our model, and identifies moving up with 1, moving down with -1 and not moving with 0. The function is called by using *erlang.next* (requestList, elevatorLocation, elevatorMovement) where requestList is the list of integers, elevatorLocation and elevatorMovement are also integers. If requestList contains a higher integer then elevatorLocation, with elevatorMovement as 1, it would return the closest integer that is higher than elevatorLocation. If no integer is higher or equal to the elevatorLocation it will return -1. Also if requestList contains a lower integer then elevatorLocation, with elevatorMovement as -1, it would return the closest integer that is lower than elevatorLocation.

```
% next/3
next([],_,_) -> -1;
next(List,CurrentFloor,Movement) -> next(List,CurrentFloor,Movement,List).

% next/4
next([H|T],CurrentFloor,Movement,ListCopy) ->
 if
  Movement == -1, H < CurrentFloor -> next(ListCopy,CurrentFloor,Movement,H,ListCopy);
  Movement == 1, H > CurrentFloor -> next(ListCopy,CurrentFloor,Movement,H,ListCopy);
  CurrentFloor == H -> H;
  true -> next(T,CurrentFloor,Movement,ListCopy)
 end;
next([],_,_,_) -> -1.

% next/5
next([H|T],CurrentFloor,Movement,Output,ListCopy) ->
 if
  Movement == -1, H > Output, H =< CurrentFloor ->
      next(T,CurrentFloor,Movement,H,ListCopy);
  Movement == 1, H < Output, H >= CurrentFloor ->
      next(T,CurrentFloor,Movement,H,ListCopy);
  true -> next(T,CurrentFloor,Movement,Output,ListCopy)
 end;
next([],CurrentFloor,Movement,Output,ListCopy) ->
 if
  Movement == -1, Output == CurrentFloor, ListCopy /= [] ->
      next(ListCopy,CurrentFloor,1,Output,[]);
  Movement == 1, Output == CurrentFloor, ListCopy /= [] ->
      next(ListCopy,CurrentFloor,-1,Output,[]);
  true -> Output
 end.
```

Listing 27: Custom function - Get next integer (floor) based on elevator direction (1 up and -1 down) and location (integers). The function goes through the given elevator list and checks if there is a request to finish next based on the location and direction of the elevator. The function returns the next floor as an integer or -1 if the list is empty.

6.2.3 Safety Verification

As explained in the previous chapter, we only explore safety properties in Timed Rebeca model. For this case study, we wanted to check the following conditions:

- Elevators only travel to and from the defined floors in the model.
- Elevators only stop on defined floors in the model.
- Elevators only stop on a floor if a request matching the floor is in the queue.
- Request lists of the elevators never got size over 3.

For the model to be feasible for verification we constructed a smaller model of 3 floors.

The following properties were defined as the safety monitor in Listing 28. The safety monitor is based on the Timed Rebeca Template presented in Chapter 4. The first implementation had an error in it which was detected while checking the second property in Table 6.6. The counter-example showed that one of the checkpoints had been assigned the term 4, meaning that an elevator was moving to the 4th floor. This was an error as floor 4 was not defined in the model. This was due to an error on line 99, 103, 111 and 115 seen in Listing 22. This error made it possible to send the next floor request in a list as a parameter to moveUp or moveDown messages server instead of the current floor. When floor 3 sent message to request a floor the checkpoint in line 34 and 40 ended having the term 4. This is because the next request location (obtained with the custom function in Listing 27) was sent instead of the current elevator location as the message parameter. This was corrected by replacing it with the current elevator locations (*el1location* and *el2location*).

```
monitorType() -> safety.

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

stateChange(_,satisfied,Stack) ->
 Actions = actions(Stack),
 checkTermMinValue(Actions,elevatorLocation,0),
 checkTermMaxValue(Actions,elevatorLocation,3),
 checkTermValue(Actions,elevator1StopReqInList,-1),
 checkTermValue(Actions,elevator2StopReqInList,-1),
 checkTermMaxValue(elevator1QueueSize,3),
 checkTermMaxValue(elevator2QueueSize,3).
```

Listing 28: Safety properties for the elevator system with 3 floors Safety properties for the elevator system with 3 floors. The term of checkpoint *elevatorLocation* is between 0 and 3. The term of checkpoints *elevator(1,2)StopReqInList* is not equal to -1. The term of checkpoints *elevator(1,2)QueueSize* is not higher then 3.

Table 6.6 shows the verification results, both for experiments 1 (with the error) and 2 (without the error).

The monitor defined in Listing 28 (line 7-10) checks if the checkpoint *elevatorLocation* term is always between 0 and 3. Furthermore it checks if the checkpoint *elevator1StopReqInList* and *elevator2StopReqInList* is never -1, as it means an elevator is stopping at a floor that is not in either of the request queues.

Table 6.6 shows that our requirements are satisfied, which makes it feasible for us to do the performance evaluation of our model with the current policies (presented in Listings 24 and 23). The first evaluation can be used as a baseline for implementation improvements when implementing new policies.

Experiment #	Description	Movement Delays	Open Close Delays (1,2,3,4)	Result
1	Location $0 >$	2	1,2,4,6	Satisfied (40929 states) 112.2 seconds
1	Location < 3	2	1,2,4,6	**Violation To High [4]** (4542 states) 98.0 seconds
1	Stop Queue 1	2	1,2,4,6	Satisfied (40929 states) 115.5 seconds
1	Stop Queue 2	2	1,2,4,6	Satisfied (40929 states) 112.3 seconds
1	Queue $1 < 3$	2	1,2,4,6	Satisfied (40929 states) 110.4 seconds
1	Queue $2 < 3$	2	1,2,4,6	Satisfied (40929 states) 111.0 seconds
2	Location $0 >$	2	1,2,4,6	Satisfied (40929 states) 112.4 seconds
2	Location < 3	2	1,2,4,6	Satisfied (40929 states) 111.6 seconds
2	Stop Queue 1	2	1,2,4,6	Satisfied (40929 states) 110.5 seconds
2	Stop Queue 2	2	1,2,4,6	Satisfied (40929 states) 109.5 seconds
2	Queue $1 < 3$	2	1,2,4,6	Satisfied (40929 states) 109.4 seconds
2	Queue $2 < 3$	2	1,2,4,6	Satisfied (40929 states) 109.2 seconds

Table 6.6: Safety verification for implementation 1 of the elevator system. Experiment #1 has an error which was then corrected when Experiment #2 was carried out.

6.2.4 Evaluation of Policies

For forming a baseline of how the elevator system performs we used the elevator model structure presented in Listing 22 with scheduling policy which we named *shortest distance* and a moving policy named *up policy*. These policies are presented in Listings 23 and 24. The scheduling policy dispatches requests to elevators depending on the distance between the location of the elevator and the requested floor. The movement policy moves the elevator while there is still requests pending for an elevator with priority of going up, meaning that the elevator only moves down when there is no requests for higher floors pending. For better understanding a pseudo Timed Rebeca code of the first implementation of the scheduling policy is presented in Listing 29 and the movement policy is presented in Listing 30.

Checkpoints that mark requests from floors are global while checkpoints that end a request (included in elevators) are restricted. This is because we don't want floor request from elevator 1 to be ended with elevator 2 and vice versa. Global and restricted types of checkpoint are explained in Chapter 5 and is used when doing paired-checkpoint analysis.

```
/* Scheduling policy: Shortest distance
    policy. */
...
/* Check if any elevators are already
    located on the requested floor */
int el1distance =
    erlang.absolutevalue(floor-el1location);
int el2distance =
    erlang.absolutevalue(floor-el2location);
...
else if(el1distance > el2distance) {
  el2queue.insert(floor);
} else {
  el1queue.insert(floor);
}
...
```

Listing 29: Timed Rebeca pseudo code - Scheduling policy: **Shortest distance**. The variable *floor* is an integer representing the floor request.

```
/* All Other Possibilities */
/* Movement policy: Up priority Policy. */
...
/* Check if elevators are on the requested
    floor before moving */
...
if(erlang.next(el1queue,el1location,1) !=
    -1) {
  el1movement = 1;
     el1.moveUp(el1location); }
else {
  el1movement = -1;
     el1.moveDown(el1location); }
```

Listing 30: Timed Rebeca pseudo code - Movement policy: **Up priority**.

For the elevator model, the initial performance evaluation was paired-checkpoint analysis which gives us more insight in how the first implementation of the system performs.

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

The simulation was then followed by executing *timedrebanalysis* with the option of grouping checkpoints by their terms. This allowed us to return multiple results for the label of the starting checkpoint, each result representing a floor. The starting checkpoint was configured with the label *startRequestFloor* and is included in the message server requestFloor in all of the floor rebecs, or the message server callElevator in either of the elevator rebecs. Term of the checkpoint is the floor requested by the *pers* rebec or the elevator identity (floor number). Ending checkpoint configured was the label *endRequestElevator*. The ending checkpoint is always included when either of the elevator gets a message to the message server stopOpen. Term of the checkpoint is always the current floor of the elevator.

Response analysis of implementation 1 is presented in Table 6.7. As expected the results was noticeably better for the higher floors in the model.

Floor	Expected Response	Standard Deviation	Median Response	Max (WCET) Response	Min (BET) Response	Checkpoint pairs
1	58.5	76.2	29.0	683	1	4772
2	44.4	61.0	18.0	564	1	5591
3	33.1	46.1	14.0	467	1	6568
4	24.5	30.6	12.0	317	1	7361
5	20.6	21.6	13.0	196	1	7880
6	17.5	14.6	13.0	131	1	8182
7	14.6	10.9	12.0	85	1	8615
8	13.4	10.6	11.0	82	1	8966
9	14.7	12.3	11.0	89	1	8777
10	18.0	13.3	15.0	99	1	8442

Table 6.7: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement Policy: **Up priority**.

The results in Table 6.7 called for another policy for movement of the elevators as we wanted more fairness in serving requests coming from floors. Implementation 2 of the model included a new movement policy named *maintain movement*. The policy is presented in pseudo Timed Rebeca code in Listing 31. This implementation maintains movement of the elevator and continues in the former direction.

```
/* All Other Possibilities */
/* Movement policy: Maintain movement Policy. */
...
/* Check if elevators are on the requested floor before moving */
...
/* If elevator queue is not empty: */
/* If movement is UP and there is a request higher then the
    current floor then go up. Otherwise go down. */
if(movement == 1) {
 if(erlang.next(el1queue,el1location,1) != -1) { el1movement = 1;
     el1.moveUp(el1location); }
 else { el1movement = -1; el1.moveDown(el1location); }
}
/* ElseIf movement is DOWN and there is a request lower then the
    current floor then go down. Otherwise go up. */
else {
 if(erlang.next(el1queue,el1location,-1) != -1) { el1movement =
     -1; el1.moveDown(el1location); }
 else { el1movement = 1; el1.moveUp(el1location); }
}
...
```

Listing 31: Timed Rebeca pseudo code - Movement policy: **Maintain movement**. Example code for elevator 1.

Safety verification of an abstracted version of the model (3 elevators) for implementation 2 was executed with same properties as for implementation 1. The properties are defined in Listing 28. All the properties were satisfied and stored 40929 states with 99243 transitions. The average time of verification process for each property was 113.9 seconds.

For performance evaluation of implementation 2, we again used *timedrebsim* with the same options as before. The simulation was then followed by executing *timedrebanalysis* to carry out the paired-checkpoint analysis. Results for paired-checkpoint analysis are presented in Table 6.8.

Implementation 2 of the model showed more fairness in response times of serving requests coming from floors. In addition we got more floor requests finished (higher number of checkpoint pairs), and less max response times for all floors.

Floor	Expected Response	Standard Deviation	Median Response	Max (WCET) Response	Min (BET) Response	Checkpoint pairs
1	21.6	15.5	18.0	95	1	9004
2	17.3	14.1	12.0	87	1	9508
3	14.8	11.7	11.0	68	1	9926
4	14.6	10.5	12.0	72	1	9915
5	14.7	9.5	12.0	65	1	9762
6	14.6	9.7	12.0	62	1	9915
7	14.3	10.4	11.0	77	1	9919
8	14.8	11.8	11.0	80	1	9930
9	17.1	13.9	12.0	81	1	9555
10	21.7	15.5	17.0	86	1	9021

Table 6.8: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement policy: **Maintain movement**.

Based on the results presented in Table 6.8 we wanted to propose a new implementation to reduce the response time of processing requests and increase the finished requests (more requests served because of floor request being served faster). The new implementation (implementation 3) involves in changing the policy of scheduling requests between elevators. Implementation of the scheduling policy is presented in Listing 32. The policy takes into account the movement of both elevators. Like in the initial scheduling policy we check for the shortest distance and in addition then check if it is moving in the right direction, if not we try the other elevator.

```
/* Scheduling policy: Shortest distance policy with movement priority. */
...
/* Check if any elevators are already located on the requested floor */
...
else if(el1distance > el2distance) {
  /* if floor request is higher then current el2 location and moving towards */
  if(floor > el2location && el2movement == 1) {
    el2queue.insert(floor); }
  /* if floor request is lower then current el2 location and moving towards */
  else if(floor < el2location && el2movement == -1) {
    el2queue.insert(floor); }
  /* el2 not moving towards request try the other elevator */
  else if(floor > el1location && el1movement == 1) {
    el1queue.insert(floor); }
  else if(floor < el1location && el1movement == -1) {
    el1queue.insert(floor); }
  else { el2queue.insert(floor); }
} else {
  /* if floor request is higher then current el1 location and moving towards */
  if(floor > el1location && el1movement == 1) {
    el1queue.insert(floor); }
  /* if floor request is lower then current el1 location and moving towards */
  else if(floor < el1location && el1movement == -1) {
    el1queue.insert(floor); }
  /* el1 not moving towards request try the other elevator */
  else if(floor > el2location && el2movement == 1) {
    el2queue.insert(floor); }
  else if(floor < el2location && el2movement == -1) {
    el2queue.insert(floor); }
  else { el1queue.insert(floor); }
}
...
```

Listing 32: Timed Rebeca pseudo code - Scheduling policy: **Shortest distance with movement priority**. Triple dots *[...]* denotes skipped code which is already written in Listings 22 and 23. The variable *floor* is the requested floor number sent by the *pers* rebec.

For implementation 3 we safety checked an abstracted version of the model (3 elevators). The verification used same properties as for implementation 1 and 2. The properties are defined in Listing 28. All the properties were satisfied and stored 41239 states with 101343 transitions. The average time of verification process for the properties was 119.1 seconds.

Before being able to do performance evaluation we again used *timedrebsim* with the same settings as before and followed by using *timedrebanalysis* to carry out the paired-checkpoint analysis. Results for paired-checkpoint analysis of implementation 3 are presented in Table 6.9.

The third implementation did not yield better results, as response times were higher on average. In addition to higher response time we had less finished requests.

Floor	Expected Response	Standard Deviation	Median Response	Max (WCET) Response	Min (BET) Response	Checkpoint pairs
1	28.3	19.9	24.0	99	1	6767
2	22.4	17.9	17.0	92	1	7420
3	18.7	15.0	14.0	90	1	8168
4	16.7	12.5	14.0	78	1	8444
5	16.3	11.0	14.0	67	1	8457
6	16.2	10.9	14.0	63	1	8688
7	16.8	12.3	14.0	73	1	8449
8	18.6	15.0	14.0	79	1	8142
9	21.6	17.6	17.0	92	1	7691
10	28.1	19.9	24.0	103	1	6843

Table 6.9: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with movement priority**. Movement policy: **Maintain movement**.

A checkpoint analysis of checkpoints with the label *elevator1QueueSize* and *elevator2QueueSize* that is included in the beginning of *handleRequest* message server showed that the elevator lists (queues) were not balanced. Figures 6.6 and 6.7 present checkpoint analysis for both checkpoints. It should be noted that the presented plots are only for the first simulation.

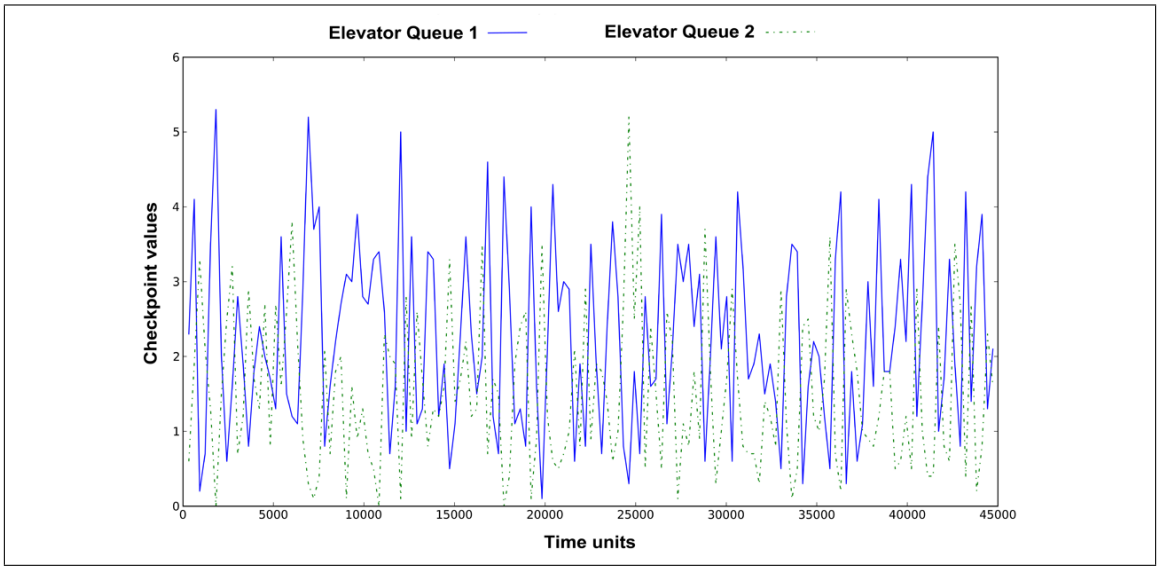

Figure 6.6: Checkpoint Analysis for elevator queue sizes (Implementation 2). Showing result for simulation 1. Moving Average smoothing with degree of 10 and sample reduce period of 100

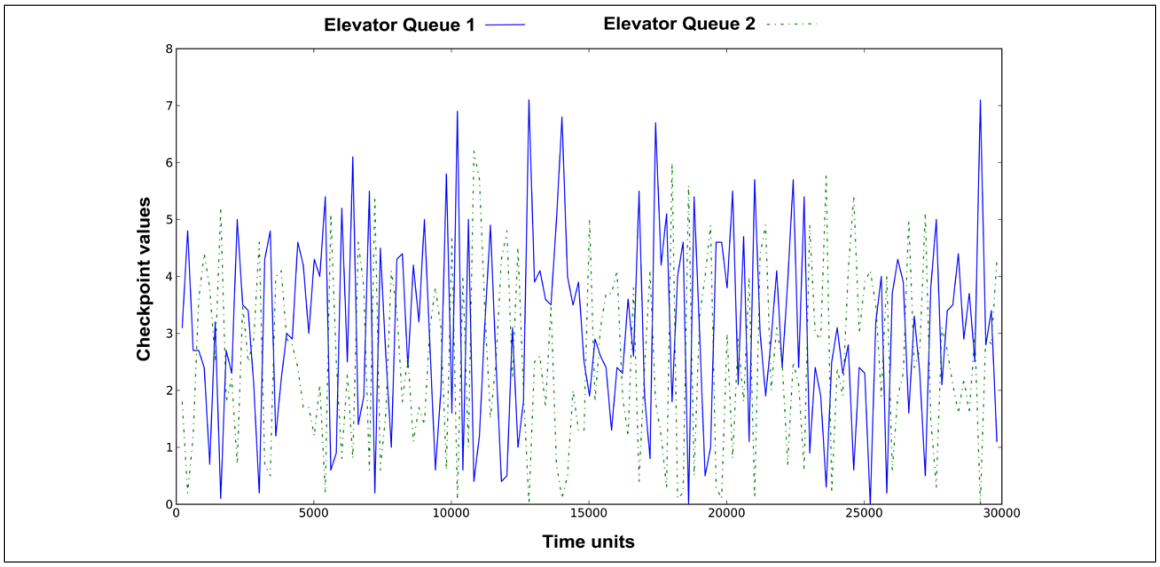

Figure 6.7: Checkpoint Analysis for elevator queue sizes (Implementation 3). Showing result for simulation 1. Moving Average smoothing with degree of 10 and sample reduce period of 100

```
/* Scheduling policy: Shortest distance policy with load balancing. */
...
/* Check if any elevators are already located on the requested floor */
...
else if(el1distance > el2distance) {
  if(el2queue.size() < el1queue.size() || el2queue.size() == el1queue.size()) {
    el2queue.insert(floor);
  } else {
    el1queue.insert(floor); }
}
else {
  if(el1queue.size() < el2queue.size() || el1queue.size() == el2queue.size()) {
    el1queue.insert(floor);
  } else {
    el2queue.insert(floor); }
}
...
```

Listing 33: Timed Rebeca pseudo code - Scheduling policy: **Shortest distance with load balancing**. Triple dots *[...]* denotes skipped code which is already written in Listings 22 and 23. The variable *floor* is the requested floor number sent by the *pers* rebec.

Following the results of checkpoint analysis of implementation 2 and 3, we proposed a new scheduling policy. This policy was to take into account the size of the elevator queue before dispatching the request to it. Dispatching of requests are then based on the queue size of an elevator. This policy was to give each elevator more time to carry out pending requests as the queues were to be more balanced and to have less requests

pending than before. Therefore it was to yield for better response times and more finished requests.

The results for implementation 4 showed some better results than implementation 3 as the maximum response was noticeably lower and more requests were served on average. Furthermore, comparing checkpoint analysis of implementation 2 and 3 depicted in Figures 6.6 and 6.7 with implementation 4 in Figure 6.8 we see that the elevator queues are more balanced in most cases.

Floor	Expected Response	Standard Deviation	Median Response	Max (WCET) Response	Min (BET) Response	Checkpoint pairs
1	28.1	16.4	28.0	79	1	7096
2	22.9	15.3	21.0	76	1	7554
3	18.9	13.0	16.0	67	1	8161
4	16.8	10.9	14.0	64	1	8354
5	15.5	9.2	14.0	53	1	8600
6	15.6	9.5	14.0	52	1	8695
7	16.5	10.9	14.0	63	1	8457
8	19.2	13.2	16.0	66	1	8071
9	22.7	15.2	21.0	68	1	7627
10	28.4	16.7	28.0	85	1	7140

Table 6.10: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with load balancing**. Movement policy: **Maintain movement**.

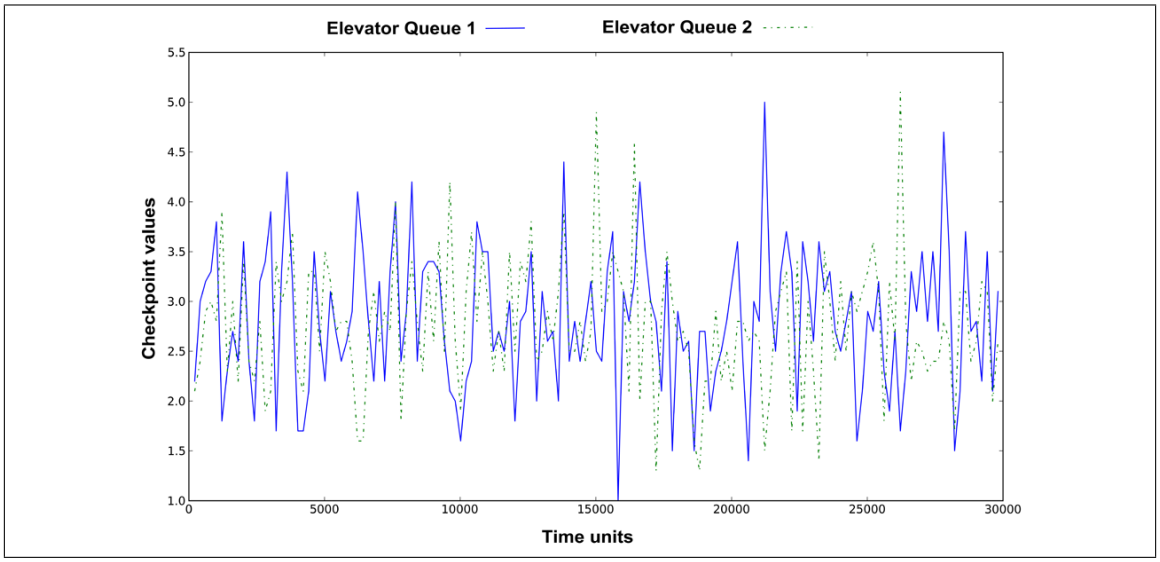

Figure 6.8: Checkpoint Analysis for elevator queue sizes (Implementation 4). Showing result for simulation 1. Moving Average smoothing with degree of 10 and sample reduce period of 100

6.2.5 Analysis Conclusion

The results presented in this case study was from 4 implementations, where implementation 2 had the best response time and most requests finished. Implementation 4 which used load balancing to dispatch requests to elevators had the lowest worst case response time but unexpectedly did not result in better results comparing to implementation 2. The most inefficient implementation was the first one, which was expected because of an unreasonable movement policy. An interesting finding was that implementation 3 did not yield better results (less average time of the elevator finishing a request) than implementation 2 and 4 as one would expect thinking that taking into account the direction of movement of the elevator in dispatching requests will be helpful. That is, if the elevator is moving towards or away from the requested location. This presents how difficult it can be to predict complex systems behavior without proper analysis. Summary of the results are shown in Table 6.11.

Note that each experiment simulated $10 * 15000\ (150000)$ requests which showed that 49.9%, 35.7%, 47.2%, and 46.8% for implementations 1, 2, 3 and 4 respectively were already pending in the queue at the starting time for some random floor requests. If a floor request is made that is already in either of the elevators queues, we ignore it (think of it like a person pushing the same button often, which accounts for one request only). Having less response time (elevator finishing floor request in less time) makes the elevator able to serve more request, because we ignore less floor requests. Table 6.11 shows how average response correlates with total of finished requests.

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

Table 6.11: Experimental results summary for the elevators.

Chapter 7

Related Work

This thesis presents two methods of analyzing models, one using formal verification to verify if the specification for a model is satisfied and one for validating if it meets the intended performance requirements.

This calls for two classes of related works:

- Modeling languages that provide tools to do formal verification, and
- simulation tools that offer analyzing methods for performance evaluation.

7.1 UPPAAL

UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays etc.). The tool is currently the most well-known model checker for real-time systems. While tools and specification formalisms like UPPAAL are quite successful at real-time verification, there is still a need to reason about timed behavior in other specification and programming languages, using dedicated model checkers. However, many of these model checkers do not implement tailored real-time verification algorithms (Region Graphs and Zones) like UPPAAL. Instead of that, they are used to check timed behavior by discretizing time, and by using normal model checking algorithms for the LTL and CTL logics (Larsen, Pettersson, & Yi, 1995; Earle & Fredlund, 2012).

Timed Rebeca and UPPAAL differ greatly in what is captured by the model in each tool. UPPAAL allows us to model synchronous time varying behaviors while Timed Rebeca focuses on distributed and asynchronous agents.

7.2 Real-Time Maude

Real-Time Maude is a language for formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, emphasizes generality and ease of specification, and is particularly suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques, including timed rewriting for simulation purposes, and time-bounded linear temporal logic model checking. It has been used to model and analyze sophisticated communication protocols and scheduling algorithms. Real-Time Maude is an extension of Maude and a major redesign of an earlier prototype (Ölveczky & Meseguer, 2007).

Timed Rebeca and Real-Time Maude are different in the computational paradigms that they naturally support. Real-Time Maude is a lower level language than Timed Rebeca. 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 suspectible to get used by modellers without intimate knowledge of the theory behind modelling. Translating Timed Rebeca to Real-Time Maude is a interesting project as Real-Time Maude tool has various ways to analyze timed systems.

7.3 Traviando

Traviando is a project that emphasizes on trace analysis of discrete event simulations (Kemper & Tepper, 2009). The design goal for Traviando is to make a simple and straightforward way to shed light on what happens over a series of simulation experiments. This is for the modeler to understand what really happens when simulating. The name Traviando is for the corresponding software package that implements these concepts.

Like in our project, Traviando uses generated traces from a simulation to analyze models. It provides some statistical information about variable changes and occurrences of method executions in a model. In addition to the statistical methods mentioned it provides progress information (detecting cycling behaviors), visualization, and model checking based on traces (Kemper, 2007; Klock & Kemper, 2010).

As a comparison to Timed Rebeca, we see that we are providing similar methods as Traviando does (statistical information and visualization of the simulation). Furthermore, a project that was introduced in parallel to our current project presented a property language

TeProp (Magnusson, 2012). The modeler can use TeProp to specify temporal properties and then check the properties on the simulation traces that are stored in a relational database.

Traviando uses models written with the ProC/B and Möbius modeling languages. ProC/B is a modeling language that is especially designed for the needs of logistics networks. It is the common specification language of the collaborative research center "Modeling of Large Logistics Networks" at the university of Dortmund (Kemper & Tepper, 2005). Möbius is a multi-paradigm multi-solution frame-work for the performance and dependability assessment of systems. It supports multiple ways to create large and complex models in a compositional manner (Deavours et al., 2002). These languages support wide area of paradigms but seem to lack the ability to model systems with asynchronous and distributed features.

7.4 RapidRT

RapidRT (Lu, Nolte, Kraft, & Norstrom, 2010) is based on Extreme Value Theory (EVT) (Beirlant, 2004), which was first introduced in 1958 and is a separate branch of statistics for dealing with the tail behavior of a distribution. Extreme Value Theory is used to model the risk of the extreme, rare events, without the vast amount of sample data required by a brute-force approach. Example applications of EVT include risk management, insurance, hydrology, material sciences, and telecommunications.

The tool emphasizes on finding worst-case response times (WCRT) by using Extreme Value Theory, capturing these rare events. This is done by an algorithm that combines Extreme Value Theory and other statistical methods in order to produce a probabilistic WCRT estimate. This algorithm uses Monte Carlo simulations to be able to guide simulations to be more probable in catching these behaviors.

For comparison, this project presents worst case response times in a simpler fashion (comparing with paired-checkpoint analysis). We only simulate with an algorithm that chooses next transition in each state of th model randomly instead of guiding it differently. This however proposes valuable future work for analyzing Timed Rebeca models.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

This thesis presents a number of results by using examples and experimental case studies. We introduced some extensions that improve the usability of Timed Rebeca. In addition we present a modified mapping to Erlang which has support for McErlang newly introduced timed semantics.

For verification of Timed Rebeca we examined how to utilize McErlang by using safety monitors to verify properties for Timed Rebeca models. This was done using case studies and examples. Furthermore, we proposed a template that makes it easier for the modeler to write properties as it can be troublesome to write them without Erlang programming knowledge.

For validation of models we introduced performance evaluations methods. These methods were applied on simulations generated from McErlang. The method showed how informative it can be to have statistical analysis and visualization over the simulation results. All provided methods were explained with examples to give the reader better understanding.

In addition to the examples throughout the thesis we presented typical case studies that showed the applicability of our methods and tools.

We find that Timed Rebeca and our tools can offer great value in analyzing systems with asynchronous message based patterns. Research for these kind of systems seem to be somewhat unattended, yet can describe many real life implementations such as modern messaging (Facebook, Twitter and Google Talk) or the use of web services in general.

8.2 Future Work

This thesis is a part of a larger project of analyzing asynchronous event-based models. The project offers opportunities for multiple interesting research questions in terms of how to analyze these models. Next paragraphs proposes some future works that we find interesting follow up from this project.

- **Detection of invalid models.** Detection of invalid models is of a high value before starting any analysis of a model. Effects like the Zeno-behaviors (infinite number of events happening in a finite amount of time) and other cycling behaviors can show itself as a major flaw in models. Certain modeling scenarios can also produce invalid models, for example one that can produce infinite clock reference in the translated code due to the use of *deadlines* in Timed Rebeca. These kind of faults can make the model infeasible for model checking or simulation.

  The use of static analysis would limit some of these problems and could provide interesting work.

- **LTL properties and monitors.** Automatic monitor generation for verification based on a simple property language that can support linear temporal logic (LTL) properties. This monitor generation could create monitors that allow McErlang to verify LTL properties based on checkpoints within a Timed Rebeca model. For extending the verification process one could add a new syntax marking a permanent probe generation in the translated Erlang code. This is done in the translated code by using the McErlang function *mce_erl:probe_state(Label::term,Term::term)*. The function marks all future states with the label and the term until it is deleted with *mce_erl:del_probe_state(Label::term())* (Earle & Fredlund, 2008). Temporal logic (LTL) properties then can be encoded to a Büchi automaton that can interface with a monitor in McErlang. Given a program and such an automaton, McErlang will run them in lockstep letting the automaton investigate each new program state generated. With this we could check with temporal logic formulas whether probes (generated with our new syntax) will hold in certain states or not.

  An automated generation of these monitors should be done as they typically need experienced individual to carry them out.

- **Custom simulation algorithms.** New algorithms for simulations to be more "clever", could open up new research questions that involves validating models (or even partially verifying) as we could get more coverage of the model's state-space without using brute-force manners to explore the state-space. This may not be interesting

in regards to some performance analysis methods as we want repeated behaviors to happen randomly and not be guided differently. However, it could give us better estimates on methods like worst case response time (WCET) using Monte Carlo simulation with the use of statistical techniques like in the project Traviando discussed in Related Works (Kemper & Tepper, 2009).

- **Real-time analysis**. Real-time on-the-fly calculation while simulating could provide the modeler a richer experience. A user interface could be implemented based on our simulation and analysis components in this project. We should then provide on-the-fly statistical calculations and visualization of simulation results.
- **Support for other analysis tools.** Multiple analysis tools have been introduced that we could benefit from. Tools like NS2, ProC/B toolset, APNN, and Möbius provide with the same type of simulation generated output making other tools able to do analysis on them. By making our generated trace output compatible with other analysis tools we would expand the analysis possibilities.
- **Translation refinements.** A research question that is related to provide a more efficient translation to Erlang, could be an interesting one. Not much has been done on explore new ways to make the translation produce smaller state-space within McErlang. Ideas like exchanging dictionaries in the translated code with records (two dictionaries with the same set of elements can be nonequivalent due to order of insert) in Erlang could be an example and could provide us with a better normalization (finding equivalent states) of states in McErlang.

8.2.1 Timed Rebeca Discussion.

We belive that Timed Rebeca is a natural and easy to use language for modeling distributed and asynchronous systems. Despite of that, we want to propose ideas of new constructs and extensions for Timed Rebeca that can make the language more efficient. Ideas brought up here are based on experience while modeling systems and examples in this project. The constructs and extension that could ease and extend the operability of Timed Rebeca can be:

- **A construct for periodic behavior.** Rather then modeling periodic behavior, we can add a construct for periodic behavior. This construct could have parameters to define the message server to be sent periodically, its parameters, and the time period to send it.

  periodic(rebec.messageServer, period, parameters);

- **Enumeration types.** While only having the types integer and boolean, we frequently use a sub-range of integers in our models. Example could be like the elevator case study in Chapter 6. There we present movements as 1 (up), -1 (down), and 0 (idle). Instead we should define an enumerable variable with the types up, down, and idle. This would simplify the syntax of the model and make it more understandable.

  enum floors = up, down, idle;

- **Dynamic creation of rebecs.** One thing missing in the mapping is the support of dynamic creation of rebecs. This addition would extend the usability of the language and could for example allow us to model P2P protocols or the Gossip Communication protocol, both which need dynamic creation of nodes to be successfully modeled.

Bibliography

Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A. H., Sigurdarson, S. H., & Sirjani, M. (2011). Modelling and simulation of asynchronous real-time systems using timed rebeca. In *Proceedings of foclasa 2011* (Vol. 58, p. 1-19).

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press.

Alur, R., & Dill, D. L. (1994). A theory of timed automata. *Theor. Comput. Sci.*, *126*(2), 183-235.

Armstrong, J. (2007). A history of erlang. In *Hopl* (p. 1-26).

Baier, C., & Katoen, J.-P. (2008). *Principles of model checking (representation and mind series)*. The MIT Press.

Beirlant, J. (2004). *Statistics of extremes: Theory and applications*. Wiley.

Buss, A. H. (1996). Modeling with event graphs. In *Proceedings of the 28th conference on winter simulation* (pp. 153–160). Washington, DC, USA: IEEE Computer Society.

Calleam, C. C. (2008). *Denver international airport baggage handling system case study. an illustration of ineffectual decision making.*

Cooper, R. B. (1981). *Introduction to queueing theory* (second ed.). New York, NY: North-Holland.

Deavours, D. D., Clark, G., Courtney, T., Daly, D., Derisavi, S., Doyle, J. M., ... Webster, P. G. (2002, October). The möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, *28*(10), 956–969.

Earle, C. B., & Fredlund, L.-Å. (2008, September). Mcerlang user manual [Computer software manual].

Earle, C. B., & Fredlund, L.-A. (2012). *Verification of timed erlang programs using mcerlang.* (Accepted for publication, FORTE 2012)

Fredlund, L. Å. (2012). *Home page.* Retrieved from `http://babel.ls.fi.upm.es/~fred/`

Fredlund, L. Å., & Svensson, H. (2007). Mcerlang: a model checker for a distributed functional programming language. In *Icfp* (p. 125-136).

Goldsman, D. (1992). Simulation output analysis. In *Proceedings of the 24th conference on winter simulation* (pp. 97–103). New York, NY, USA: ACM.

Gomaa, H. (2000). *Designing concurrent, distributed, and real-time applications with uml* (1st ed.). Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Hewitt, C. (1972). *Description and theoretical analysis (using planner: A language for proving theorems and manipulating models in a robot).* Unpublished doctoral dissertation, Department of Computer Science, MIT.

Hewitt, C. (2010). Actor model for discretionary, adaptive concurrency. *CoRR*, *abs/1008.1459*.

Holloway, C. M. (1997). Why engineers should consider formal methods. In *In 1997 aiaa/ieee 16th digital avionics systems conference* (p. 9).

Huth, M., & Ryan, M. (2004). *Logic in computer science: Modelling and reasoning about systems*. New York, NY, USA: Cambridge University Press.

Jacobs, P. H. M., Lang, N. A., & Verbraeck, A. (2002). Web-based simulation 1: D-sol; a distributed java based discrete event simulation architecture. In *Winter simulation conference* (p. 793-800).

Kelton, W. D. (1997). Statistical analysis of simulation output. In *Winter simulation conference* (p. 23-30).

Kemper, P. (2007). A trace-based visual inspection technique to detect errors in simulation models. In *Winter simulation conference* (p. 747-755).

Kemper, P., & Tepper, C. (2005). Visualizing the dynamic behavior of proc/b models. In *Simvis* (p. 63-74).

Kemper, P., & Tepper, C. (2009). Automated trace analysis of discrete-event system models. *IEEE Transactions on Software Engineering*, *35*, 195-208.

Kenney, J., & Keeping, E. (1962). *"moving averages" 14.2 in mathematics of statistics* (No. pt. 1). Van Nostrand.

Klock, S. K., & Kemper, P. (2010). An automated technique to support the verification and validation of simulation models. In *Dsn* (p. 595-604).

Kristinsson, H. (2012). *Elevator case study with timed rebeca modelling.* Retrieved from `http://rebeca.cs.ru.is/files/Elevators/Elevators.htm`

Lamport, L. (2002). *Specifying systems: The tla+ language and tools for hardware and software engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Lamport, L. (2005). Real-time model checking is really simple. In *Charme* (p. 162-175).

Larsen, K. G., Pettersson, P., & Yi, W. (1995, August). Model-Checking for Real-Time Systems. In *Proc. of fundamentals of computation theory* (pp. 62–88).

Law, A. M. (2010). *Proceedings of the 2010 winter simulation conference, wsc 2010,*

baltimore, maryland, usa, 5-8 december 2010. WSC.

Lu, Y., & högskola, M. (2010). *Approximation techniques for timing analysis of complex real-time embedded systems*. School of Innovation, Design and Engineering, Mälardalen University.

Lu, Y., Nolte, T., Kraft, J., & Norstrom, C. (2010). A statistical approach to response-time analysis of complex embedded real-time systems. In *Proceedings of the 2010 ieee 16th international conference on embedded and real-time computing systems and applications* (pp. 153–160). Washington, DC, USA: IEEE Computer Society.

Magnusson, B. (2012). *Simulation-based analysis of timed rebeca using teprop and sql*. Unpublished master's thesis, Reykjavik University.

Matloff, P. N. (2011). *A discrete-event simulation course based on the simpy language.* University of California. Retrieved from `http://simpy.sourceforge.net/documentation.htm`

Mitre. (2009). *Tortuga simulation language.* MITRE Corporation. Retrieved from `http://code.google.com/p/tortugades/`

Ölveczky, P. C., & Meseguer, J. (2007). Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, *20*(1-2), 161-196.

Reynisson, A. H. (2011). *Timed rebeca: Refinement and simulation*. Unpublished master's thesis, Reykjavik University.

Robinson, S. (2004). *Simulation: The practice of model development and use*. John Wiley & Sons.

Ross, S. (2006). *Simulation*. Elsevier Academic Press.

Schmidt, D. C. (2006, February). Model-driven engineering. *IEEE Computer*, *39*(2). Retrieved from `http://www.truststc.org/pubs/30.html`

Shui, A., Mustafiz, S., Kienzle, J., & Dony, C. (2005). Exceptional use cases. In *Models* (p. 568-583).

Sirjani, M., & Jaghoori, M. M. (2011). Ten years of analyzing actors: Rebeca experience. In *Formal modeling: Actors, open systems, biological systems* (p. 20-56).

Sirjani, M., Movaghar, A., Shali, A., & de Boer, F. S. (2004, June). Modeling and verification of reactive systems using rebeca. *Fundam. Inf.*, *63*(4), 385–410.

Snowdon, J. L., & Charnes, J. M. (Eds.). (2002). *Proceedings of the 34th winter simulation conference: Exploring new frontiers, san diego, california, usa, december 8-11, 2002.* ACM.

Svensson, H. (2009). Implementing an ltl-to-büchi translator in erlang: a protest experience report. In *Proceedings of the 8th acm sigplan workshop on erlang* (pp. 63–70). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1596600.1596610`

Typesafe. (2012). *Akka, scalable real-time transaction processing.* Typesafe Inc. Retrieved from `http://akka.io/docs/`

Appendix A

Timed Rebeca Additional Examples

An example of using lists in Timed Rebeca

Adding lists in Timed Rebeca is an extension that provides us to model behaviors that are frequently used in today's systems where buffers and queues are essential. As seen in Table 3.1 lists can contain the data type integer and are converted to list in Erlang. Lists can be stated as a local variable or a state variable inside a reactive class or a message server. Lists in Timed Rebeca are like arrays but without boundaries. They may grow and shrink and are able to return the first or the last value of its containing integers.

Listing 34 presents an example of the functionality of lists in Timed Rebeca. The model consists of one reactive class ListExample. Two rebecs *listprocess1* and *listprocess2* are instantiated from the reactive class. Both rebecs are initialized by sending a message to themselves. The message is received by the message server *go* with the parameter 1. The parameter will then be inserted to the list *numbericlist* followed by assigning the size of the list, first element of the list, and last element of the list to the local variables *listsize*, *firstelement*, and *lastelement* respectively. Before the message server sends another message to self and incrementing the parameter *intAdd*, it will remove the last integer of the list. But only if the size of the list is equal or higher then 10.

```
reactiveclass ListExample(3) {
  knownrebecs { ListExample listproc }

  statevars {
    list<int> numericlist;
    int index;
  }

  msgsrv initial() { self.go(1); }

  msgsrv go(int intAdd)
  {
    //Insert into the list
    numericlist.insert(intAdd);
    // Operations on lists:
    int listsize = numericlist.size();
    int firstelement = numericlist.first();
    int lastelement = numericlist.last();
    //Remove a integer of the value lastelement from the list if the size is 10.
    if(listsize >= 10) { numericlist.remove(lastelement); }
    listproc.go(intAdd+1);
  }
}

main {
  ListExample listprocess1(listprocess2):();
  ListExample listprocess2(listprocess1):();
}
```

Listing 34: Timed Rebeca Model - List example. The example models a list with upper bound 10, the last item of the list will be deleted if trying to insert the 11th integer to the list *numbericlist*.

Lists are one of the mostly used data structures in Erlang (and all other functional programming languages). Lists are used to solve many problems efficiently and easily by using for example, higher-order functions. A given function can be easily applied to all elements of a list and then return a new list with the applied results or we can simply go through them recursively like in the presented custom function in Listing 36. It is very convenient for the modeler to have the ability to use lists in Timed Rebeca. It gives us the ability to use lists with manipulative features in native Erlang by using custom function which is discussed in the next section.

An example of using Custom Functions in Timed Rebeca

To maintain the simplicity of the language we added the construct *erlang*, which allows us to define custom functions in the form of a native Erlang code. Many behaviors often need complicated operations, as the model presented in Listing 35. The model consists of the reactive class CustomFunctionExample and has one rebec *customprocess* instantiated from it. In the initial message server of *customprocess*, a message go is sent to itself. The message then iterates sending a message *go* to itself. An insertion of a random integer to the list *numericlist* is done by calling the insert statement (line 15). After 50 iterations the custom function *MaxNumberOfList* [1] is called with the *numericlist* as the parameter. This will result in assigning the highest integer in the list to the integer variable *max*. The custom function *MaxNumberOfList* is presented in Listing 36.

```
1  reactiveclass CustomFunctionExample(3) {
2     knownrebecs {}
3     statevars { list<int> numericlist; int itterations; int max; }
4
5     msgsrv initial() {
6       itterations = 0;
7       max = -1;
8       self.go();
9     }
10
11    msgsrv go()
12    {
13      itterations = itterations + 1;
14      int number = ?[1:100];
15      numericlist.insert(number);
16
17      if(itteration < 50) { self.go(); }
18      else {
19        max = erlang.MaxNumberOfList(numericlist);
20        checkpoint(HighestRandomValue,max);
21      }
22    }
23 }
24 main {
25    CustomFunctionExample customprocess():();
26 }
```

Listing 35: Timed Rebeca Model - Get the maximum random number added to a list.

Calling Erlang functions from a Timed Rebeca model, increases the ability of modeling more complicated behaviors that cannot be done with very simple constructs in Timed Rebeca. Erlang functions gives Timed Rebeca all the features of Erlang while maintain-

[1] The custom function is called with ***erlang***.*MaxNumberOfList*(L(,...,L_n)), where L is a variable or a set of variables

ing the simplicity of the modeling language.

```
MaxNumberOfList([Head|Rest]) ->
  MaxNumberOfList(Rest, Head).

MaxNumberOfList([], Res) -> Res;
MaxNumberOfList([Head|Rest], Max) when Head > Max ->
  MaxNumberOfList(Rest, Head);
MaxNumberOfList([_Head|Rest], Max) -> MaxNumberOfList(Rest, Max).
```

Listing 36: Custom Erlang Function - Get the maximum number from a list.

Non-deterministic Process Evaluator

To show that the mapping and McErlang are providing us with a correct state space when using expirations and non-deterministic constructs of Timed Rebeca we present another example. The example is presented in Listing 37.

The formerly mentioned timed semantics of McErlang in Chapter 3 had the Erlang function *timeRestricts* that computed what transition should be enabled from a program state giving us correct order of delayed messages (actions). This function is to take into account the usage of clock references mentioned in the former sections of Chapter 3. This being correct is crucial for us if we need to be able to evaluate expiration of a messages.

The model consists of two reactive classes, Process and Listener. One rebec *listener* is instantiated from the reactive class Listener and two rebecs *process1* and *process2* are instantiated from the reactive class Process. In the initialization of the rebec *process1* a message is sent to *listener* with after(2) and deadline(3). The rebec *process2* is initialized by sending a message to *listener* with an after(3) and deadline(4). The *listener* listens for message from *process1* and *process2*.

```
1  env boolean delayEnabled;
2
3  reactiveclass Process(2) {
4    knownrebecs { Listener lsnr; }
5    statevars { }
6
7    msgsrv initial(int evalutationDelay, int messageDeadline)
8    {
9      lsnr.receive(delaybeforesend) after(evalutationDelay) deadline(messageDeadline);
10   }
11   msgsrv ack(boolean processed)
12   {
13     % Safety Property: All processed gets processed.
14     trace(isProcessProcessed, processed);
15   }
16 }
17 reactiveclass Listener(2) {
18   knownrebecs { Process process1 Process process2; }
19   statevars { bool processdelay; }
20
21   msgsrv initial(boolean enableProcessDelay) {
22     processdelay = enableProcessDelay;
23   }
24   msgsrv.receive(int delayLabel)
25   {
26     trace(processDelay,delayLabel);
27     if(processdelay == true) {
28       int nonDetDelay = ?(1,3);
29       delay(nonDetDelay);
30     }
31     sender.ack(true);
32   }
33 }
34 main {
35   Listener lsnr(proc1,proc2):(delayEnable); % Listener
36   Process proc1(lsnr):(2,3); % Delays by 2 before sending with a deadline of 3.
37   Process proc2(lsnr):(3,4); % Delays by 3 before sending with a deadline of 4.
38 }
```

Listing 37: Timed Rebeca Model - Non-deterministic Process Evaluator

For simplification purposes for the reader an abstracted Erlang translation was created and presented in Listing 39 located in the Appendices B. The translation has clock references defined before spawning and delaying for both processes. We demonstrates expiration by replying to the sender process with "*{deadlined,*}*" otherwise with "*{reply}*" message.

The model has an environment argument *delayEnabled*, that enables or disables a non-deterministic delay of 1 or 3 time units before *listener* sends a message back to the sender rebecs (*process1* or *process2*).

If the model has *delayEnabled* set to **false**, the *listener* rebec will not delay before sending a message back to the sender after having received a message. McErlang showed a state graph with one computation path when *delayEnabled* was set as false. The computation path started with *process1* to send a message *listener* and getting a reply message. Then *process2* sent a message *listener* and also got a reply message. This is the correct interpretation when comparing it to formal semantics of Timed Rebeca as there is no expiration of messages involved when setting *delayEnabled* to false.

To explore the non-deterministic behavior and expiration of messages in Timed Rebeca we generated another state graph with McErlang, by setting *delayEnabled* to **true**. This makes *listener* non-deterministically delay by 1 or 3 time units before sending a reply message to the sender (*process1* or *process2*). It should be noted that no messages can be taken out of *listener* message bag while the delay occurs. With these settings McErlang generated the state graph (depicted as a labeled transition system) presented in Figure A.1.

As depicted in Figure A.1 we see that all paths start with the transition *process1* to send the message *listener*. When reaching state labeled 15, we branch as we then are exploding the non-deterministic behavior shown in line 34 of the model presented in Listing 37.

When we branch, one transition (message), namely $(15 \rightarrow 10)$ explores the possibility of having *listener* delay by 3 time units and having *process1* get the reply message. Followed by expiring the message from *process2* and (replying it with *{deadline,three}* message in the abstracted translation in Listing 39). The other transition, namely $(15 \rightarrow 12)$, explores the possibility of delaying by 1 time unit that results in message from *process2* to achieve its deadline and get the reply message from *listener*.

It should be noted that after the transition (message) $(15 \rightarrow 12)$ we branch again with two available transitions (messages). This is because the program has two possible transitions (messages) available during this time.

- Firstly, *listener* can send a reply message to *process1*,
- Secondly, *process2* can send a message to *listener*.

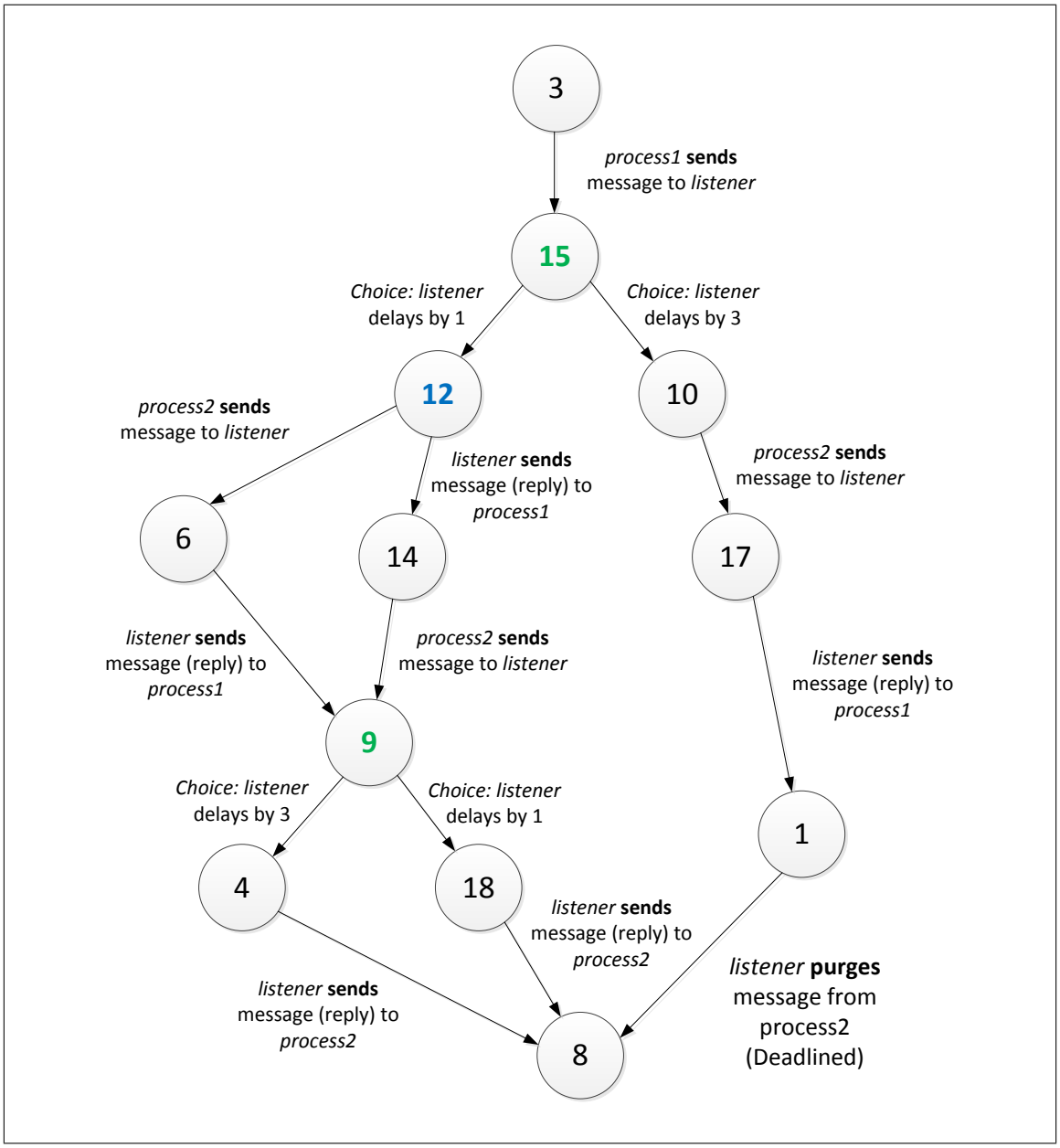

Figure A.1: Labeled Transition System (state graph) for the non-deterministic process evaluator model with delay enabled in listing 39. {...} brackets are message parameters.

This is a normal behavior as when the message from *process1* is sent it happens at time 2. Following the delay(1) of *listener* the time has elapsed by 3 time units. This is the exact time when *process2* can send a message to *listener*. Therefore we explore both possibilities that both have the same results.

Appendix B

Abstract Erlang Translations

This chapter presents abstract Erlang translations for some Timed Rebeca models that were discussed in this thesis. Translation presented in Listing 38 is of the competing processes model in Listing 6. We abstract the initialization process to the Erlang method *start*, which spawns three processes (Listener, Process 1 and Process 2). After the method *start* has finished spawning the three processes, it waits for one message to show that a reply message has arrived. Process 1 delays by 2 time units before sending the message "*two*" to Listener and Process 2 delays by 3 time units before sending the message "*three*" to Listener.

The second translation is presented in Listing 39. It shows an abstract translation of the non-deterministic process evaluator model in Listing 37. This translation is based on the first abstract translation, but with the addition of creating clock references before sending a message to the listener. Note that we send the relative deadline value with messages that come from Process 1 and Process 2. The Listener (*deadliner* in the code) evaluates the clock references when a message is received with the relative deadline value. This is done by using *mce_erl_time:was()* that reads the time of the reference. We then decide if to send a "*deadlined*" or "*ok*" message back depending on if the clock reference elapsed time was more than the relative deadline value. Note that we spawn a special process for receiving messages (*reaceivelistener* in the code) as we want to receive more then one message.

```
1  start () ->
2    Pid = spawn(fun () -> Listener() end),                    % Listener
3    spawn(fun () -> timedelay(2), Pid!{self(),two} end),      % Process1 - Delay 2.
4    spawn(fun () -> timedelay(3), Pid!{self(), three} end),   % Process2 - Delay 3.
5    receive
6      {reply,Msg} -> (io:format("~p~n", [Msg])) % Got Reply.
7    end.
8
9  Listener () ->
10   receive % Receives only one message.
11     {Pid, Msg} -> Pid!{reply,Msg} % Reply to the Process.
12   end.
13
14 timedelay(Milliseconds) ->
15   receive
16   after (Milliseconds) -> ok
17   end.
```

Listing 38: Abstract Erlang Translation - Competing Processes Model

```
start () ->
  Pid = spawn(fun () -> deadliner() end), % Listener
  TTone = mce_erl_time:nowRef(), % Clock Reference that is used when sending message with
      after from Process 1.
  spawn(fun () -> timedelay(2), Pid!{self(),TTone,3,two} end), % Process 1
  TTtwo = mce_erl_time:nowRef(), % Clock Reference that is used when sending message with
      after from Process 2.
  spawn(fun () -> timedelay(3), Pid!{self(),TTtwo,4,three} end), % Process 2
   receivelistener(). % Listener starts listening.

receivelistener() ->
  receive
    {ok,Msg} -> (io:format("Processed! ~p ~n", [Msg]));
    {deadlined,Msg} -> (io:format("Deadlined! ~p ~n", [Msg]))
  end,
   receivelistener(). % Restart the receive for Listener.

deadliner() ->
  mce_erl:urgent(),
  receive
    {Pid, TT, DL, Msg} ->
      case compare(addTimeStampsD(milliSecondsToTimeStamp(DL), mce_erl_time:was(TT)),
          mce_erl_time:now()) of
        true -> nondetdelay(1,3), Pid ! {ok, Msg}, mce_erl_time:forget(TT);
        false -> Pid ! {deadlined, Msg}, mce_erl_time:forget(TT)
      end
  end,
   deadliner(). % Restart the deadliner listener.

timedelay(Milliseconds) ->
  mce_erl:urgent(Milliseconds),
  receive
  after (Milliseconds) -> ok
  end.

nondetdelay(Delay1,Delay2) ->
  mce_erl:urgent(),
  mce_erl:choice
  ([fun () -> mce_erl:urgent(), timedelay(Delay1) end,
    fun () -> mce_erl:urgent(), timedelay(Delay2) end]).
```

Listing 39: Abstract Erlang Translation - Non-deterministic Process Evaluator Model

Appendix C

Monitors

Monitor Template for Timed Rebeca Checkpoints

The monitor presented in Listing 40 is a complete code for the Timed Rebeca Checkpoint template. The template is explained in Section 4.2.3 and has predefined function to make it easy for the modeler to write simple safety properties.

```
-module(monitor).
-export([init/1, stateChange/3, monitorType/0]).

-compile(nowarn_shadow_vars).
-compile(nowarn_unused_vars).

-include("$MCERLANG_HOME/src/include/stackEntry.hrl").

-behaviour(mce_behav_monitor).

-include("state.hrl").
-include("process.hrl").
-include("node.hrl").

monitorType() -> safety.

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

stateChange(_, satisfied, Stack) ->

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

  Actions = actions(Stack),
  checkDropMsgsrv(Actions, CheckpointTerm).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Timed Rebeca Functions for Checkpoints %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Check if Message Server Has an Expired Message
checkDropMsgsrv(Actions,MessageServer) ->
  Value = [MessageServer],
  case has_probe_with_tag(drop,Actions) of
```

```
39      {true,MsgSrv} -> checkDropMsgsrvReturn(MsgSrv,Value);
40      false  -> {ok, satisfied }
41    end.
42  checkDropMsgsrvReturn(MsgSrv,Value) ->
43    case [MsgSrv] == Value of
44     true  -> {droppedmessage,MsgSrv};
45     false  -> {ok, satisfied }
46    end.
47
48  % Check if an checkpoint occur with Label
49  checkLabelCheckPoint(Actions,Label) ->
50    LabelL = [Label],
51    case has_probe_with_tag(Label,Actions) of
52      {true,GetTerm} -> {foundlabel,{LabelL,GetTerm}};
53      false  -> {ok, satisfied }
54    end.
55
56  % Check if an checkpoint Term is higher then some number.
57  checkTermMaxValue(Actions,Label,Term) ->
58    Value = [Term],
59    case has_probe_with_tag(Label,Actions) of
60      {true,GetTerm} -> checkTermMaxValueReturn(GetTerm,Value);
61      false  -> {ok, satisfied }
62    end.
63
64  checkTermMaxValueReturn(GetTerm, Value) ->
65    case GetTerm > Value of
66     true  -> { violation_tohigh ,GetTerm};
67     false  -> {ok, satisfied }
68    end.
69
70  % Check if an checkpoint Term is lower then some number.
71  checkTermMinValue(Actions,Label,Term) ->
72    Value = [Term],
73    case has_probe_with_tag(Label,Actions) of
74      {true,GetTerm} -> checkTermMinValueReturn(GetTerm,Value);
75      false  -> {ok, satisfied }
76    end.
77
78  checkTermMinValueReturn(GetTerm, Value) ->
79    case GetTerm < Value of
80     true  -> { violation_tohigh ,GetTerm};
81     false  -> {ok, satisfied }
82    end.
83
```

```
84  % Check if an checkpoint Term is equal to some type.
85  checkTermValue(Actions,Label,Term) ->
86    Value = [Term],
87    case has_probe_with_tag(Label,Actions) of
88      {true,GetTerm} -> checkTermValueReturn(GetTerm,Value);
89       false  -> {ok, satisfied }
90    end.
91
92  checkTermValueReturn(GetTerm, Value) ->
93    case GetTerm == Value of
94     true  -> { violation_tohigh ,GetTerm};
95      false  -> {ok, satisfied }
96    end.
97
98  %%%%%%%%%%%%%%%%%%%%%
99  % Support functions : %
100 %%%%%%%%%%%%%%%%%%%%%
101
102  actions(Stack) ->
103   {Entry,_} = mce_behav_stackOps:pop(Stack),
104   Entry#stackEntry.actions.
105
106 has_probe_with_tag(Tag,Actions) ->
107    lists : foldl
108      (fun (Action, false) ->
109            mce_erl_actions:is_probe(Action) andalso
110              case mce_erl_actions:get_probe_label(Action) of
111                {Tag,Id} -> {true,Id};
112                _ -> false
113              end;
114            (_Action,Other) -> Other
115       end, false, Actions).
```

Listing 40: Monitor Template for Timed Rebeca Checkpoints

Appendix D

Revised Timed Rebeca Language Description

The lexical structure of Timed Rebeca

Identifiers

Identifiers ⟨*Ident*⟩ are unquoted strings beginning with a letter, followed by any combination of letters, digits, and the characters _ ', reserved words excluded.

Literals

Integer literals ⟨*Int*⟩ are nonempty sequences of digits.

Reserved words and symbols

The set of reserved words is the set of terminals appearing in the grammar. Those reserved words that consist of non-letter characters are called symbols, and they are treated in a different way from those that are similar to identifiers. The lexer follows rules familiar from languages like Haskell, C, and Java, including longest match and spacing conventions.

The reserved words used in Rebeca are the following:

```
after   boolean        deadline
delay   else           env
erlang  false          first
if      initial        insert
int     knownrebecs    last
list    main           msgsrv
now     reactiveclass  remove
size    statevars      time
trace   true
```

The symbols used in Rebeca are the following:

```
;    (        )
{    }        ,
=    <        >
.    else if  ||
&&   |        ^
&    ==       !=
<=   >=       <<
>>   +        -
*    /        %
?    :        ~
!    *=       /=
%=   +=       -=
```

Comments

Single-line comments begin with `//`.
Multiple-line comments are enclosed with `/*` and `*/`.

The syntactic structure of Timed Rebeca

Non-terminals are enclosed between $\langle$ and $\rangle$. The symbols ::= (production), | (union) and ϵ (empty rule) belong to the BNF notation. All other symbols are terminals.

⟨*Model*⟩ ::= ⟨*ListEnvVar*⟩ ⟨*ListReactiveClass*⟩ ⟨*Main*⟩

⟨*EnvVar*⟩ ::= `env` ⟨*TypedParameter*⟩ `;`

⟨*ListEnvVar*⟩ ::= ϵ
| ⟨*EnvVar*⟩ ⟨*ListEnvVar*⟩

⟨*ReactiveClass*⟩ ::= `reactiveclass` ⟨*Ident*⟩ `(` ⟨*Integer*⟩ `)` `{` ⟨*KnownRebecs*⟩ ⟨*StateVars*⟩ ⟨*MsgSrvInit*⟩ ⟨*ListMsgSrv*⟩ `}`
| `reactiveclass` ⟨*Ident*⟩ `(` ⟨*Integer*⟩ `)` `{` ⟨*KnownRebecs*⟩ ⟨*StateVars*⟩ ⟨*ListMsgSrv*⟩ `}`

⟨*ListReactiveClass*⟩ ::= ϵ
| ⟨*ReactiveClass*⟩ ⟨*ListReactiveClass*⟩

⟨*KnownRebecs*⟩ ::= ϵ
| `knownrebecs` `{` ⟨*ListTypedVarDecl*⟩ `}`

⟨*StateVars*⟩ ::= ϵ
| `statevars` `{` ⟨*ListTypedVarDecl*⟩ `}`

⟨*MsgSrvInit*⟩ ::= `msgsrv initial` `(` ⟨*ListTypedParameter*⟩ `)` `{` ⟨*ListStm*⟩ `}`

⟨*MsgSrv*⟩ ::= `msgsrv` ⟨*Ident*⟩ `(` ⟨*ListTypedParameter*⟩ `)` `{` ⟨*ListStm*⟩ `}`

⟨*ListMsgSrv*⟩ ::= ϵ
| ⟨*MsgSrv*⟩ ⟨*ListMsgSrv*⟩
| ϵ
| ⟨*MsgSrv*⟩ ⟨*ListMsgSrv*⟩

⟨*VarDecl*⟩ ::= ⟨*Ident*⟩

⟨*ListVarDecl*⟩ ::= ϵ
| ⟨*VarDecl*⟩
| ⟨*VarDecl*⟩ `,` ⟨*ListVarDecl*⟩

⟨*TypedVarDecl*⟩ ::= ⟨*TypeName*⟩ ⟨*Ident*⟩
| ⟨*TypeName*⟩ ⟨*Ident*⟩ = ⟨*Exp*⟩

⟨*ListTypedVarDecl*⟩ ::= ϵ
| ⟨*TypedVarDecl*⟩
| ⟨*TypedVarDecl*⟩ `;` ⟨*ListTypedVarDecl*⟩

⟨*TypedParameter*⟩ ::= ⟨*TypeName*⟩ ⟨*Ident*⟩

⟨*ListTypedParameter*⟩ ::= ϵ
| ⟨*TypedParameter*⟩
| ⟨*TypedParameter*⟩ `,` ⟨*ListTypedParameter*⟩

⟨*BasicType*⟩ ::= `int`
| `time`
| `boolean`
| `list` < ⟨*BasicType*⟩ >

⟨*TypeName*⟩ ::= ⟨*BasicType*⟩
| ⟨*Ident*⟩

⟨*Stm*⟩ ::= ⟨*Stm*⟩ `;`
| ⟨*Ident*⟩ ⟨*AssignmentOp*⟩ ⟨*Exp*⟩ `;`
| ⟨*TypedVarDecl*⟩ `;`
| ⟨*Ident*⟩ `.` ⟨*Ident*⟩ `(` ⟨*ListExp*⟩ `)` ⟨*After*⟩ ⟨*Deadline*⟩ `;`
| `delay (` ⟨*Exp*⟩ `) ;`
| `if (` ⟨*Exp*⟩ `)` ⟨*CompStm*⟩ ⟨*ListElseifStm*⟩ ⟨*ElseStm*⟩
| ⟨*Ident*⟩ `. insert (` ⟨*Exp*⟩ `)`
| ⟨*Ident*⟩ `. remove (` ⟨*Exp*⟩ `)`
| `trace (` ⟨*Ident*⟩ `,` ⟨*ListExp*⟩ `) ;`

⟨*ListStm*⟩ ::= ϵ
| ⟨*Stm*⟩ ⟨*ListStm*⟩

⟨*CompStm*⟩ ::= ⟨*Stm*⟩
| `{` ⟨*ListStm*⟩ `}`

⟨*After*⟩ ::= ϵ
| `after (` ⟨*Exp*⟩ `)`

⟨*Deadline*⟩ ::= ϵ
| `deadline (` ⟨*Exp*⟩ `)`

⟨*ElseifStm*⟩ ::= `else if (` ⟨*Exp*⟩ `)` ⟨*CompStm*⟩

⟨*ListElseifStm*⟩ ::= ϵ
| ⟨*ElseifStm*⟩ ⟨*ListElseifStm*⟩

⟨*ElseStm*⟩ ::= ϵ
| `else` ⟨*CompStm*⟩

⟨*ListIdent*⟩ ::= ⟨*Ident*⟩
| ⟨*Ident*⟩ `.` ⟨*ListIdent*⟩

⟨*Exp*⟩ ::= ⟨*Exp*⟩ || ⟨*Exp2*⟩
| ⟨*Exp1*⟩

⟨*Exp2*⟩ ::= ⟨*Exp2*⟩ `&&` ⟨*Exp3*⟩
| ⟨*Exp3*⟩

⟨*Exp3*⟩ ::= ⟨*Exp3*⟩ | ⟨*Exp4*⟩
| ⟨*Exp4*⟩

⟨*Exp4*⟩ ::= ⟨*Exp4*⟩ ^ ⟨*Exp5*⟩
| ⟨*Exp5*⟩

⟨*Exp5*⟩ ::= ⟨*Exp5*⟩ `&` ⟨*Exp6*⟩
| ⟨*Exp6*⟩

⟨*Exp6*⟩ ::= ⟨*Exp6*⟩ == ⟨*Exp7*⟩
| ⟨*Exp6*⟩ != ⟨*Exp7*⟩
| ⟨*Exp7*⟩

⟨*Exp7*⟩ ::= ⟨*Exp7*⟩ < ⟨*Exp8*⟩
| ⟨*Exp7*⟩ > ⟨*Exp8*⟩
| ⟨*Exp7*⟩ <= ⟨*Exp8*⟩
| ⟨*Exp7*⟩ >= ⟨*Exp8*⟩
| ⟨*Exp8*⟩

⟨*Exp8*⟩ ::= ⟨*Exp8*⟩ << ⟨*Exp9*⟩
| ⟨*Exp8*⟩ >> ⟨*Exp9*⟩
| ⟨*Exp9*⟩

⟨*Exp9*⟩ ::= ⟨*Exp9*⟩ + ⟨*Exp10*⟩
| ⟨*Exp9*⟩ − ⟨*Exp10*⟩
| ⟨*Exp10*⟩

⟨*Exp10*⟩ ::= ⟨*Exp10*⟩ * ⟨*Exp11*⟩
| ⟨*Exp10*⟩ / ⟨*Exp11*⟩
| ⟨*Exp10*⟩ % ⟨*Exp11*⟩
| ⟨*Exp11*⟩

⟨*Exp11*⟩ ::= (⟨*Exp*⟩)
| ? (⟨*ListExp*⟩)
| ? (⟨*Exp*⟩ : ⟨*Exp*⟩)
| ⟨*Exp12*⟩

⟨*Exp12*⟩ ::= ⟨*UnaryOperator*⟩ ⟨*Exp11*⟩
| ⟨*Exp13*⟩

⟨*Exp13*⟩ ::= now ( )
| erlang . ⟨*Ident*⟩ (⟨*ListExp*⟩)
| ⟨*Ident*⟩ . size ( )
| ⟨*Ident*⟩ . first ( )
| ⟨*Ident*⟩ . last ( )
| ⟨*Constant*⟩
| ⟨*Exp14*⟩

⟨*Exp14*⟩ ::= ⟨*ListIdent*⟩
| (⟨*Exp*⟩)

⟨*ListExp*⟩ ::= ϵ
| ⟨*Exp*⟩
| ⟨*Exp*⟩ , ⟨*ListExp*⟩

⟨*Exp1*⟩ ::= ⟨*Exp2*⟩

⟨*Constant*⟩ ::= ⟨*Integer*⟩
| true
| false

⟨*ListConstant*⟩ ::= ϵ
| ⟨*Constant*⟩
| ⟨*Constant*⟩ , ⟨*ListConstant*⟩

⟨*UnaryOperator*⟩ ::= +
| −
| ~
| !

⟨*AssignmentOp*⟩ ::= =
| *=
| /=
| %=
| +=
| −=

⟨*Main*⟩ ::= `main {` ⟨*ListInstanceDecl*⟩ `}`

⟨*InstanceDecl*⟩ ::= ⟨*TypedVarDecl*⟩ `(` ⟨*ListVarDecl*⟩ `) : (` ⟨*ListExp*⟩ `)`

⟨*ListInstanceDecl*⟩ ::= ϵ
| ⟨*InstanceDecl*⟩
| ⟨*InstanceDecl*⟩ `;` ⟨*ListInstanceDecl*⟩

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539