TIMED REBECA: REFINEMENT AND SIMULATION

Árni Hermann Reynisson

Master of Science

Software Engineering

June 2011

School of Computer Science

Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

Timed Rebeca: Refinement and Simulation

by

Árni Hermann Reynisson

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**

June 2011

Research Thesis Committee:

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

Dr. Anna Ingólfsdóttir
Professor, Reykjavík University, Iceland

Dr. Carolyn L. Talcott
Program Manager, Computer Science Laboratory, SRI International

Copyright
Árni Hermann Reynisson
June 2011

Timed Rebeca: Refinement and Simulation

Árni Hermann Reynisson

June 2011

Abstract

With the the popularity of web services and applications in wireless networks, distributed computing is becoming ubiquitous. The software can be time critical and as such must respond to requests in a timely fashion. Analyzing timing behaviour of distributed and asynchronous systems is a particularly challenging task. We present an extension of the actor-based Rebeca language that can be used to model distributed and asynchronous systems with timing constraints. We provide an automated translation from Timed Rebeca to Erlang. Translation schemes for both refined programs and simulation are included. The translation tool is built upon formal mapping from Timed Rebeca to Erlang and the Structural Operational Semantics of Timed Rebeca. A few examples are studied and experimental results are provided.

Timed Rebeca: Nákvæmni og Hermun

Árni Hermann Reynisson

Júní 2011

Útdráttur

Vinsældir vefþjónusta og hugbúnaðar sem hefur þráðlaus samskipti gera það að verkum að dreifð tölvukerfi má finna víða. Algengt er að slík kerfi setji skilyrði fyrir svartíma sín á milli og því getur tími getur skipt sköpum. Greining á dreifðum ósamstilltum kerfum gagnvart tíma er einkar áskorandi verkefni. Við kynnum viðbót við gerendabundna málið Rebeca til þess að hanna líkön af dreifðum og ósamstilltum kerfum með tímahömlum. Við útbúum sjálfvirka þýðingu frá Timed Rebeca yfir í Erlang. Þýðingar fyrir bæði útfærslu og hermun eru innifaldar. Þýðingartólið er byggt á formlegri skilgreiningu á Timed Rebeca yfir í Erlang og formlegri merkingarfræði (SOS) Timed Rebeca. Nokkur dæmi eru skoðuð og niðurstöður kynntar.

To my girlfriend Kolbrún Ragna Ragnarsdóttir
and my son Gabríel Freyr Árnason.

Acknowledgements

Dr. Marjan Sirjani, my supervisor and teacher. I am very grateful to her for introducing me to research in computer science and all her support with the project. This work would have never been done without her.

Steinar Hugi Sigurðarson, M.Sc. student and friend. He has been my partner in crime for the last few years. I am grateful for the discussions we have had about my project, life, the universe and everything. He is my Hasselhoff, always cool and steady as a rock, when I go into Hulk mode, which frequently happens.

A group of people from ICE-TCS and ICE-ROSE met regularly to work on specifying Timed Rebeca in the early stages of development. For all the discussions we had, I thank Luca Aceto, Matteo Cimini and Anna Ingólfsdóttir. Specifically, I thank Matteo Cimini for assisting in making the mapping in the thesis more concise and for always being ready to help.

I am very grateful for the work I have done with fellow students and for all the discussions I have had both with students and faculty at Reykjavík University. This experience has been very fruitful for me.

I thank Dr. Ralf Lämmel for quickly responding to my questions and engaging in online discussions about fold algebras.

For proofreading this thesis, I thank Björgvin Reynisson and Kjartan Þór Ragnarsson.

The work in this thesis is partially supported by the projects "Timed Asynchronous Reactive Objects in Distributed Systems: TARO" (project nr. 110020021) and "New Developments in Operational Semantics" (project nr. 080039021), of the Icelandic Research Fund.

Publications

Part of the material in this thesis was submitted to an international conference. Co-authors are Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Steinar Hugi Sigurdarson and Marjan Sirjani. The co-authors contributed significantly to the writing of the conference submission, as well as defining Timed Rebeca and its semantics. However, the implementation and experimentation is my own work.

Contents

Contents x

List of Figures xiii

List of Tables xiv

List of Listings xv

1 Introduction 1
- 1.1 Modelling with Actors and Time . . . 2
- 1.2 Contributions . . . 3
- 1.3 Overview of the Thesis . . . 3

2 Background 5
- 2.1 Model Driven Development . . . 5
- 2.2 Actor Model . . . 6
- 2.3 Rebeca . . . 6
- 2.4 Simulation . . . 7

3 Timed Rebeca 9
- 3.1 Timing Features . . . 9
- 3.2 Timing Constructs . . . 10
- 3.3 Modelling Timing Features . . . 12
- 3.4 Progress of Time . . . 12
- 3.5 Formal Semantics for Timed Rebeca . . . 13

4 Refinement and Simulation of Timed Rebeca using Erlang 17
- 4.1 Erlang Language . . . 17
- 4.2 Informal Mapping of Timed Rebeca to Erlang . . . 18
- 4.3 Formal Mapping . . . 23
- 4.4 Simulating Timed Rebeca with McErlang . . . 28

4.5 Extended Formal Mapping 29
4.6 Discussion 31

5 Experimental Results 33
5.1 Simple Communication Protocol 34
5.2 Ticket Service 35
5.3 Sensor Network 38
5.4 Discussion 42

6 Related Work 45
6.1 RT-synchronizer 45
6.2 Real-Time Maude 46
6.3 Creol 46
6.4 UPPAAL 47
6.5 Schedulability for Rebeca Models 47

7 Conclusion 49
7.1 Summary 49
7.2 Future Work 50
7.3 Discussion on Rebeca 50

Bibliography 53

Appendices 55

A Timed Rebeca Language Description 55
A.1 Lexical Structure of Timed Rebeca 55
A.2 Syntactic Structure of Timed Rebeca 56

B Implementation of the Translation Tool 61
B.1 First Implementation 62
B.2 Revised Implementation 65
B.3 Discussion 69

C Design Decisions: Imperative to Functional 71
C.1 Approach 1: Derive Functions from Control Flow Graph 72
C.2 Approach 2: Static Single Assignment Form and Records for State Variables 73
C.3 Approach 3: State Transformer Functions and Records for State Variables 74

C.4 Approach 4: State Transformer Functions and Dictionaries for State Variables 75

List of Figures

3.1 Abstract syntax of Timed Rebeca 11
3.2 SOS Rules for Timed Rebeca 16

4.1 Abstract syntax of a relevant subset of Erlang 19

5.1 Example of an event graph 33
5.2 Event graph of the simple communication protocol model 34
5.3 Event graph of the ticket service model 36
5.4 Event graph of the sensor network model 39

C.1 Erlang Emulator session showing how pattern matching in Erlang works 71
C.2 Control Flow Graph of Listing C.1 73

List of Tables

4.1 Structure of the mapping from Timed Rebeca to Erlang 19

5.1 Experimental simulation results for ticket service. 37
5.2 Experimental simulation results for sensor network. 40

List of Listings

4.1 Syntax of a receive with timeout 18
4.2 Pseudo Erlang code capturing the behaviour of the ticketService process 20
4.3 Example of a message send after 15 time units in Erlang 21
4.4 Example of delay of 10 time units in Erlang 22
4.5 Pseudo Erlang code showing statement execution 22
4.6 Translation functions from Timed Rebeca to Erlang 26
4.7 Mapping for a reactive class 30
4.8 Mapping for a message server 30

5.1 A Timed Rebeca model of the simple communication protocol example 35
5.2 A Timed Rebeca model of the ticket service example 37
5.3 A Timed Rebeca model of the sensor network example 40

B.1 Example of generated template for code generation 62
B.2 Example of how to type the translation to Erlang 63
B.3 Suggestion to the extended mapping problem 64
B.4 Haskell data types for Erlang syntax 65
B.5 Algebra interface (partial listing) 66
B.6 Fold algebra (partial listing) 67
B.7 Monoidal algebra for collecting identifier names (partial listing) . . . 67
B.8 Updated monoidal algebra for collecting known rebec names 68
B.9 Instance of an identity algebra (partial listing) 68
B.10 Assignment simplification algebra instance 69
B.11 Instance of an algebra that translates Timed Rebeca to refined Erlang code (partial listing) 70
C.1 Example message server for translation attempts 72
C.2 Erlang code derived from the CFG in Figure C.2 73
C.3 Erlang code derived from Listing C.1 using SSA 74
C.4 Erlang code derived from Listing C.1 using state transformers 75

Chapter 1

Introduction

Networked computer systems are everywhere and we use them every day. The systems which largely go unnoticed are embedded systems such as mobile phones, televisions, digital cameras and coffee machines. These systems are also called reactive systems, they progress by means of interacting with their environment which might include inputs from humans or other devices. Reactive systems are also being increasingly used in safety critical systems such as heart rate monitors, nuclear reactors and car brakes. Failure in any of these embedded systems can lead to financial loss for the manufacturer or worse, loss of lives.

With the rise of the Internet, distributed systems have become the norm perhaps without much consideration to the fact. Asynchronous and distributed systems like web services can make or break companies. For example, if an airline loses the ability to sell tickets online due to a web service failure, the company will suffer great loss of revenue. Probability of it going out of business increases as the problem persists.

Despite advances in software engineering, producing reliable software is a challenging task. Surprisingly, it seems that consumers have come to terms with the fact that software is unreliable; most people simply reboot in case of software failure and continue their work and accept that updates will, up to a point, fix the flaws they experience. Software can suffer from devastating quality issues as history has shown us. The Intel Pentium floating point bug in 1994 cost the company $475 million and damage in reputation (Pratt, 1995). Ariane 5's integer overflow bug in 1996 resulted in a explosion 40 seconds into mid air and cost European tax payers an estimated $370 million (Dowson, 1997).

Formal methods are a mathematically precise way to specify, develop and verify software and hardware systems. Proponents of formal methods like to compare software

engineering to bridge construction. They say that before the time of engineering principles, bridges used to collapse and now they don't. In reality, modern bridges collapse due to errors in construction[1] but in general they are more reliable than without engineering principles during construction. One way to build software based on formal methods is to create a formal model which represents the behaviour of the software. The model can be verified by checking if the behaviour conforms to a specification, it can be simulated which offers some validation of the behaviour, and it can be refined to a runnable program, sometimes automatically.

Models can incorporate features such as time constraints and probability. Time constraints allow a modeller to specify that a computation takes time. Probabilistic models allow a modeller to have random or probablistic behaviour in the model. Various reactive systems are time critical, such as communication protocols, traffic lights as well as the aforementioned radiation machines in which if timing constraints are not met lives may be at stake. The need to be able to analyze timed, distributed and asynchronous systems is clear and this thesis is a part of a larger project that attempts to address that need.

1.1 Modelling with Actors and Time

The actor model is well suited for distributed and asynchronous systems. It is based on asynchronous messages passing between agents in which actors can only respond to messages by changing behaviour, creating more actors or sending more messages. Actors are becoming ever more popular in industry as companies are building highly concurrent software with the actor model. Example of this are the Facebook chat system and Twitter message queue. Although actors are attracting more and more attention both in academia and industry, little has been done on timed actors and even less on analyzing timed actor-based models.

In this thesis we present an extension of the actor-based Rebeca language (Sirjani, Movaghar, Shali, & Boer, 2004) that can be used to model distributed and asynchronous systems with timing constraints. This extension of Rebeca is motivated by the ubiquitous presence of real-time computing systems, whose behaviour depends on timing as well as functional requirements.

[1] Tuo River bridge in China 2007, Kota Chambal Bridge in India 2009.

1.2 Contributions

The contribution of this thesis is to extend Rebeca with time constraints, introduce Timed Rebeca as a language for designing and analyzing timed systems in distributed and asynchronous settings and a tool to simulate Timed Rebeca models. This work is a part of a larger project in which the entire life cycle of software development will be supported by verifying and simulating the model, and refining the model to a program. Part of the motivation to focus on simulation is that a translation from Timed Rebeca to UPPAAL exists (Izadi, 2010). However, even simple examples result in a state explosion when model checked.

To summarize, the contributions are:

- Define Timed Rebeca language and its formal semantics (together with Aceto, Cimini, Ingolfsdottir, Sigurdarson and Sirjani).
- A formal mapping to Erlang for refinement and simulation of Timed Rebeca models using McErlang.
- A tool to translate Timed Rebeca models to Erlang, both refined models and simulation models.

1.3 Overview of the Thesis

In Chapter 2 we introduce the main concepts behind Rebeca. This includes the actor model, model checking and additionally simulation which is crucial for this thesis. Chapter 3 introduces Timed Rebeca, its language definition and informal semantics as well as the formal semantics by means of Structural Operational Semantics. Chapter 4 presents both informally and formally a translation to Erlang which is based on the semantics of Timed Rebeca. There we also give extended mapping in order to be able to simulate Timed Rebeca models with McErlang. In Chapter 5 we look at three case studies and analyze their behaviour using simulations. Related work is then discussed in Chapter 6. We conclude with conclusions and future work in Chapter 7.

The work on defining Timed Rebeca language is done as a team work within the ICE-ROSE (ICEROSE, 2011) and ICE-TCS (ICETCS, 2011) groups at Reykjavik University. The work on mapping to Erlang and developing the tool is done by myself. The technical report (Aceto et al., 2011) is a condensed version of this thesis report.

Chapter 2

Background

In this chapter we introduce the model driven software methodology. We then introduce the actor model and Rebeca which the work in this thesis extends. Finally, we outline the simulation process.

2.1 Model Driven Development

Model driven development is a software development methodology. The methodology focuses on creating models of some aspects of a system. The models have the benefit of being more abstract than the actual implementation and as such can be used to reason about the behaviour offered by the model even before it is built. However, too abstract models might not capture the behaviour of the system faithfully.

Modelling offers various benefits to the life cycle of software development. Making a model in the first place, prior to implementing it, can lead to discoveries about problems that were not immediately visible before. Models are typically built from a specification which can at a later phase in development be used to verify the behaviour of the model. Different verification techniques exist. We have theorem proving and model checking which offer the most solid guarantees about the behaviour in question. On the other hand we have simulation and testing which are more lightweight and can be used complementary to heavier methods. Once a model has been verified it can be refined (implemented). Refinement is building the code based on the (verified) model.

2.2 Actor Model

A well-established paradigm for modelling distributed and asynchronous systems is the actor model. This model was originally introduced by Hewitt as an agent-based language (Hewitt, 1972), and later established as a mathematical model of concurrent computation that treats *actors* as the universal primitives of concurrent computation (Agha, 1985). In response to a message that it receives, an actor can make local decisions:

- Create more actors,
- send more messages,
- and change their behaviour.

There is no order assigned to the decisions above. They can be carried out concurrently. Actors have encapsulated states and behaviour, and are both capable of creating new actors, as well as redirecting communication links through the exchange of actor identities.

A number of systems can be modelled naturally by the actor system. For instance email, where an account is an actor and the email address is the actor identity. Another example is a web service, whose URL can be modelled as an actor identity.

Different interpretations, dialects and extensions of actor models have been proposed in several domains and are claimed to be the most suitable model of computation for the most dominating applications, such as multi-core programming and web services (Hewitt, 2007).

2.3 Rebeca

Reactive objects language, Rebeca (Sirjani et al., 2004), is an operational interpretation of the actor model with formal semantics and model checking tools. Rebeca is designed to bridge the gap between formal methods and software engineers. The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and embodied in verification tools (Jaghoori, Sirjani, Mousavi, Khamespanah, & Movaghar, 2010a; Sirjani, Movaghar, Shali, & Boer, 2005; Sirjani et al., 2004). With its simple, message-driven and object-based computational model, Java-like syntax, and set of verification tools, Rebeca is an interesting and easy-to-learn

model for practitioners. The Rebeca toolkit ships with an integrated development environment based on Eclipse. Part of the environment is the ability to model check with Modere (Jaghoori, Sirjani, Mousavi, Khamespanah, & Movaghar, 2010b) which does symmetry and partial order reduction on the models.

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*, *state 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 atomically (Sirjani et al., 2004).

2.4 Simulation

Simulation is a method to imitate the behaviour of a system. A model of the system is created, which represents a set of assumptions made about the system. A simulation is used to evaluate a model quantitatively and gather data about it in order to estimate some characteristics about the model. The simulation method can complement model checking since it does not require building the state space and can discover errors before model checking begins, which is a much more computationally expensive process. Moreover, simulation can be applied to models which are out of reach for model checkers due to state space explosion, but gives no guarantees.

Chapter 3

Timed Rebeca

The contents of this chapter is based on joint work with Luca Aceto, Matteo Cimini, Anna Ingolfsdottir, Steinar Hugi Sigurdarson and Marjan Sirjani.

In this chapter we introduce an extension of Rebeca with real-time features. We present the timing features we're interested in modelling, changes made to Rebeca to realize these features and the formal semantics of the extension.

3.1 Timing Features

Modelling real-time aspects is different than that of regular modelling. By real-time, we want to describe *when* something takes place in the system, not only *how*. The features we are interested in describing are the following:

1. **computation time**: the time that is taken by a computation
2. **message delivery time**: network delay, when messages are sent from one location to another
3. **messages expiration**: a request for service can expire if a deadline is not met for delivering a reply for a request
   a. **request**: the request can expire after a certain amount of time
   b. **reply**: when a reply is sent, i.e. a request has been served, there may be an expiration time for the reply
4. **periodic events**: represents events that occur periodically

With this basic set of features we can build complex timing behaviour into models. It is not hard to imagine a model of networked devices at different locations in the world

which might require these features for modelling. Some processing might be done in one device that normally takes longer in other devices. Sending messages between the devices takes different amounts of time based on their geographical location. A device might send out progressive information to other devices but that information is only valid for a specific amount of time before it is renewed and resent, hence the old information should be discarded after the timeout. A device might request information from another device and expect a reply before some fixed deadline. This requires the ability to read local time before a request and checking the time of reply to see if the reply is valid. A device might periodically read sensory input and act upon them.

3.2 Timing Constructs

We consider synchronized local clocks for rebecs in the timed Rebeca models. The time domain is the set of natural numbers, yielding a discrete time domain. Methods are still executed atomically, but we can model passing of time while executing a method. Instead of a message queue for each rebec, we have a bag containing the messages that are sent. Each rebec knows about its local time and can put deadlines on the service requests (messages) that are sent declaring that the request will not be valid after the deadline (modelling the timeout for a request). When a message is sent there can also be a constraint on the earliest time at which it can be served (taken from the message bag by the receiver rebec). The modeller may use these constraints for various purposes, such as modelling the network delay or modelling a periodic event.

The timing primitives that are added to the syntax of Rebeca are *delay*, *now*, *deadline* and *after*. See Figure 3.1.

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. Delays are used to model computations that take time.

Now: $now()$ returns the time of the local clock of the rebec from which it is called.

Deadline: $r.m()$ $deadline(t)$, where r denotes a rebec name, m denotes a method name of r and t is a positive natural number, means that the message m is sent to the rebec r and is put in the message bag. After t units of time the message is not valid any more and is purged from the bag.

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

mod	::=	$\overline{ev}\ \overline{rc}\ main$	Rebeca model
ev	::=	$\mathbf{env}\ t\ \overline{v};$	environment variables
rc	::=	$\mathbf{reactiveclass}\ c\ \{kr\ sv\ \overline{msg}\ \}$	reactive class definition
kr	::=	$\mathbf{knownrebecs}\ \{\ \overline{t}\ \overline{v}\ \}$	known rebecs
sv	::=	$\mathbf{statevars}\ \{\ \overline{t}\ \overline{v}\ \}$	state variables
msg	::=	$\mathbf{msgsrv}\ m(\overline{t}\ \overline{v})\ \{\ \overline{stmt}\ \}$	message servers
$stmt$	::=	$v = e;$	assignment
	|	$r = \mathbf{new}\ c(\overline{e});$	new rebec
	|	$r.m(\overline{e})\ [\mathbf{after}(e_a)]\ [\mathbf{deadline}(e_d)];$	message send
	|	$\mathbf{if}\ (e)\ stmt\ [\mathbf{else}\ stmt]$	branching
	|	$\mathbf{delay}(e)$	delay;
	|	$\{\ \overline{stmt}\ \}$	block
e	::=	$\mathbf{now}()$	now
$main$	::=	$\mathbf{main}\ \{\ \overline{id}\ \}$	main block
id	::=	$c\ r(\overline{r}) : (\overline{k});$	instance declaration

Figure 3.1: Abstract syntax of Timed Rebeca. Words with line over it are shorthand for sequences, like $\overline{ev}$ is for $ev_1 \dots ev_n$. Pairs of sequences are abbreviated s.t. $\overline{t}\ \overline{v}$ denotes $t_1 v_1 \dots t_n v_n$. Identifiers c, t, m, v, and r denote class, type, method, variable, and rebec names, respectively; k denotes constants, and e denotes an (arithmetic, boolean or nondetermistic choice) expression.

time units have passed. After statements can be used to model network delays in delivering a message to the destination, and also a periodic event. The difference between after and delay is not immediately obvious but will be explained later in this chapter.

The *delay* statement models the passing of time for a rebec during execution of a method. The *now* expression returns the local time of the rebec. The keywords *after* and *deadline* can only be used in conjunction with a method call. The messages that are sent are put in the message bag together with their time tag and *deadline* tag. The scheduler decides which message is to be executed next, based on the time tags of the messages. The time tag of a message is the value of *now* when the message was sent, with the value of the argument of the *after* added to it when the message is augmented with an *after*. The intuition is that a message cannot be taken (served) before the time that the time tag determines and it shall be discarded after the deadline.

Should After be Modelled with Delay?

At first, intuition suggests that after can be modelled with delay. This turns out not to be the case. If $r.m()\ after(t)$ were modelled as $delay(t);\ r.m()$, the local clock of the

sender would be incremented by t before sending the message, while the message send with after would not.

3.3 Modelling Timing Features

We will now show how we can model the timing features described in Section 3.1 by using the timing primitives from Section 3.2 in Timed Rebeca.

- Computation time is modelled by means of the *delay* construct.
- Message delivery time is modelled with the *after* construct (in conjunction with message sends).
- Message expiration is modelled in two different ways:
  - Request expiration is modelled with the *deadline* construct.
  - Response deadline is modelled by sending the deadline as parameter with the request such that the responding agent sets that as a *deadline* when the reply is sent.
- Periodic events are modelled by repeatedly sending the same message to a rebec in a loop with the *after* construct.

3.4 Progress of Time

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

A scheduler takes a message from the message bag, executes the corresponding message server atomically, and then takes another message. Every time the scheduler takes a message for execution, it chooses a message with the least time tag. Before the execution of the corresponding method starts, the local time (*now*) of the receiver rebec is set to the maximum value between its current time and the time tag of the message. The current local time of each rebec is the value of *now*. This value is frozen when the method execution ends until the next method of the same rebec is taken for execution.

The arguments of *after* and *delay* are relative to the local time, but when the corresponding messages are put in the message bag their tags are absolute values, which are computed by adding the relative values of the arguments to the value of the expression *now* of the sender rebec (where the messages are sent).

3.5 Formal Semantics for Timed Rebeca

In this section we provide an SOS (Plotkin, 1981) semantics for Timed Rebeca in the style of Plotkin. The behaviour of Timed Rebeca programs is described by means of the transition relation $\rightarrow$ that describes the evolution of the system.

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

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

Scheduler

The system transition relation $\rightarrow$ is defined by the rule *scheduler* Figure in 3.2. The scheduler rule allows the system to progress by picking up messages from the bag and executing the corresponding methods. The first side condition of the rule, namely $\sigma_{A_i}(now) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The second side condition is the predicate $TT = min(B)$, which is satisfied whenever the time tag TT is the smallest time tag for the messages of all the rebecs A_i in the bag B. The premise executes the method m, as described by the transition relation $\xrightarrow{\tau}$, which will be defined below. The method body is looked up in the environment of A_i and is executed in the environment of A_i modified as follows.

- The variable *sender* is set to the sender of the message.

- In executing the method m, the formal parameters $\overline{arg}$ are set to the values of the actual parameters $\overline{v}$. Methods of arity n are indeed supposed to have $arg_1, arg_2, \ldots, arg_n$ as formal parameters. This is not a lack of generality since such a change of variable names can be performed in a pre-processing step for any program.
- The variable *now* is set to the maximum between the current time of the rebec and the time tag of the selected message.

Method Execution

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

The transition relation $\overset{\tau}{\rightarrow}$ describes the execution of methods in the style of natural semantics (Kahn, 1987). See Figure 3.2 for the set of rules. Since in this kind of semantics the whole computation of a method is performed in a single step, this choice perfectly reflects the atomic execution of methods underlying the semantics of the Rebeca language. The general form of this type of transition is $(S, \sigma, Env, B) \overset{\tau}{\rightarrow} (\sigma', Env', B')$. A single step of $\overset{\tau}{\rightarrow}$ consumes all the code S and provides the value resulting from its execution. Carrying the bag B is important because new messages may be added to it during the execution of a statement S. Also Env is required because *new* statements create new rebecs and may therefore add new environments to it. In the semantics, the local environment σ is separated from the environment list Env for the sake of clarity. The result of the execution of the method thus amounts to the modified private store σ', the new list of environments Env' and the new bag B'.

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

- Rule *msg* describes the effect of method invocation statements. For the sake of brevity, we limit ourselves to presenting the rule for method invocation statements that involve both the *after* and *deadline* keywords. The semantics of instances of that statement without those keywords can be handled as special cases of that rule by setting the argument of *after* to zero and that of *deadline* to $+\infty$, meaning that this message never expires. Method invocation statements put a new message in the bag, taking care of properly setting its fields. In particular

the time tag for the message is the current local time, which is the value of the variable *now*, plus the number d that is the parameter of the *after* keyword.

- *Delay* statements change the private variable *now* for the considered rebec.

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

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

$$\text{if } \sigma_{A_i}(now) \leq DL \text{ and } TT = min(B)$$

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

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

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

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

$$\textsc{cond}_1\ \frac{eval(e, \sigma) = true \quad (S_1, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

$$\textsc{cond}_2\ \frac{eval(e, \sigma) = false \quad (S_2, \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}{(if\ (e)\ then\ S_1\ else\ S_2,\ \sigma, Env, B) \xrightarrow{\tau} (\sigma', Env', B')}$$

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

Figure 3.2: SOS Rules for Timed Rebeca. The first side condition of the *scheduler* rule, $\sigma_{A_i}(now) \leq DL$, checks whether the selected message carries an expired deadline, in which case the condition is not satisfied and the message cannot be picked. The second side condition is the predicate $TT = min(B)$, which is satisfied whenever the time tag TT is the smallest time tag for the messages of all the rebecs A_i in the bag B. In rule *create*, the rebec name A should not appear in the range of the environment σ. The function *eval* evaluates expressions in a given environment in the expected way.

Chapter 4

Refinement and Simulation of Timed Rebeca using Erlang

In this chapter, we present a translation from the fragment of Timed Rebeca without rebec creation to Erlang. The motivation for translating Timed Rebeca models to Erlang code is to have an executable code for Timed Rebeca (refinement) and to be able to simulate Timed Rebeca models.

4.1 Erlang Language

Erlang is a dynamically-typed general-purpose programming language, which was designed for the implementation of distributed, real-time and fault-tolerant applications (Armstrong, 1997). Originally, Erlang was mostly used for telephony applications such as switches. Its concurrency model is based on the actor model.

Concurrency Primitives of Erlang

Erlang is an actor-based concurrent language. It is intended for real-time systems where response time in order of milliseconds is required. Erlang runtime has a real-time garbage collector in which heap spaces between processes are not shared. Hence, garbage collection in a process does not affect another process. Erlang has no shared memory and all interaction between processes takes place as asynchronous message passing. All values in a message are copied before sending them, which makes it easy for Erlang to distribute processes onto a network instead of a single machine. The primitives Erlang offers to do concurrent computations are **`spawn`**, `!` and **`receive`**.

- `Pid = spawn(Fun)` creates a new process that evaluates the given function `Fun` in parallel with the process that invoked spawn. A process corresponds to an actor.

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

Listing 4.1: Syntax of a receive with timeout

- `Pid ! Msg` sends the given message `Msg` to the process with the identifier `Pid`.
- **`receive`** ... **`end`** receives a message that has been sent to a process; message discrimination is based on pattern matching.

Timing Primitives of Erlang

Erlang comes with a set of features for real-time computations. They are **`after`** and **`now`**.

- **`after`** is used in conjunction with a **`receive`** in which the receive has a timeout block as shown in Listing 4.1
- `erlang:now()` returns the current time of the process

Execution of Erlang Code

When a process reaches a **`receive`** expression it looks in the queue and takes the message that matches the pattern if the corresponding guard is true. A guard is a boolean expression, which can include the variables of the same process. The process looks in the queue each time a message arrives until timeout occurs.

4.2 Informal Mapping of Timed Rebeca to Erlang

The abstract syntax for a fragment of Erlang, that is required to present the translation, is shown in Figure 4.1. Table 4.1 offers an overview of how a construct in one language relates to one in the other. We discuss the general principles behind our translation in more detail below.

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

Figure 4.1: Abstract syntax of a relevant subset of Erlang. Angle brackets ⟨...⟩ are used as meta parenthesis, superscript + for repetition more than once, superscript * for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Identifiers v, p and g denote variable names, patterns and guards, respectively, and e denotes an expression.

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

Table 4.1: Structure of the mapping from Timed Rebeca to Erlang. Notice that dictionaries is also known as associative array or map.

```
ticketService() →
  receive
    % wait for a message with a set of known rebecs
    {Agent} →
      % proceed to the next behaviour
      ticketService(dict:from_list([agent, Agent]))
  end.
ticketService(KnownRebecs) →
  LocalVars = dict:new(),
  receive
    % wait for the 'initial' message
    initial →
      % process message 'initial' and proceed to the next behaviour
      ticketService(KnownRebecs, dict:from_list([]))
  end.
ticketService(KnownRebecs, StateVars) →
  LocalVars = dict:new(),
  receive
    % wait for each message servers
    requestTicket →
      % process message 'requestTicket' and make a recursive call (loop)
      ticketService(KnownRebecs, StateVars)
  end.
```

Listing 4.2: Pseudo Erlang code capturing the behaviour of the ticketService process

Mapping a Reactive Class

Reactive classes are translated into three functions, each representing a possible behaviour of an Erlang process:

1. the process waits to get references to known rebecs,
2. the process reads the initial message from the queue and executes it,
3. the process reads messages from the queue and executes them.

Once processes reach the last function they enter a loop. Erlang pseudo code for the reactive class *TicketService* in the Rebeca model in Listing 5.2 is shown in Listing 4.2.

Mapping a Message Server

A message server is translated into a *match* expression (see Figure 4.1), which is used inside **`receive`** ... **`end`**. On line 18 in Listing 4.2, `requestTicket` is the *pattern* that

```
Sender = self(),
spawn(fun() →
  receive after 15 →
    TicketService ! {{Sender, now(), inf}, requestTicket}
  end
end)
```

Listing 4.3: Example of a message send after 15 time units in Erlang. The value of inf shows the deadline for the message, and now() is used to tag the message with the current time. The sender is captured outside the scope of the thread that waits for 15 time units. The last item, requestTicket, is the name of the message that is being sent.

is *matched* on, and the body of the message server is mapped to the corresponding expression.

Mapping a Message Send

Message send is implemented depending on whether *after* is used. If there is no *after*, the message is sent like a regular message using the send operator of Erlang, denoted as `!`, as shown on line 4 in Listing 4.3. However, if the keyword *after* is present a new process is spawned which sleeps for a specified amount of time before sending the message as described before. Setting a deadline for the delivery of a message is possible by changing the value `inf`, which denotes no deadline (as shown on line 3 in Listing 4.3), to an absolute point in time. Messages are tagged with the time at which they were sent. For the simulation we use the system clock to find out the current time by calling the Erlang function `now()`.

Moreover, since message servers can reply to the sender of the message, we need to take care of setting the sender as part of the message as seen on lines 1 and 4 in Listing 4.3.

As there is no pattern to match with, the *delay* statement is implemented as a `receive` consisting only of timeout that makes the process wait for a certain amount of time. For example, *delay(10)* is translated to the code in Listing 4.4.

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

```
receive
after 10 →
  ok
end
```

Listing 4.4: Example of delay of 10 time units in Erlang

```
ticketService(KnownRebecs, StateVars) →
  LocalVars = dict:new(),
  receive
    requestTicket →
      {NewStateVars, _} = fn({StateVars, LocalVars})
      ticketService(KnownRebecs, NewStateVars)
  end.
```

Listing 4.5: Pseudo Erlang code showing statement execution

Mapping of Sequential Statements

Erlang is a functional language and does not allow assignments like imperative languages do. Once a value is bound to a variable, the variable cannot be reassigned. Rebeca on the other hand follows an imperative paradigm which creates a problem for the mapping of assignments. We solved this problem by modelling each statement of Timed Rebeca as an anonymous function. The functions take a single parameter, a tuple, `{StateVars, LocalVars}`. Assignment statements will make the function return a new tuple with the updated value. Otherwise, the same tuple is returned, for example for message sends. Each translated statement is then composed by means of function composition. Erlang pseudo code for execution of statements is shown in Listing 4.5. The function *fn* on line 5 is a single function, the result of composing the translated statements. See Appendix C for discussions on design decisions regarding this mapping.

Modelling Timing Features

In this section we show how we can model the timing features presented in Section 3.1 in Erlang.

- Computation time is modelled by doing an empty **`receive`** with an empty **`after`** block, thus blocking the process for the amount of time specified in the after block

- Message delivery time is modelled by spawning a new process. In the newly spawned process, we delay the execution by using the same method as described in computation time, above, except that the `after` block contains the message send.
- Message expiration is modelled by sending an expiration time as parameter with the message. Upon receiving the message, the process checks if the parameter is greater than his current time. If so, he must not process the message. Notice that Erlang has no construct for a message send with deadline as Timed Rebeca does. For reply, we use the same method as described in Section 3.3.
- Periodic events are modelled by repeatedly sending the same message to a process in a loop, and using the same method as described in message delivery time above, to make it happen at a specific interval.

4.3 Formal Mapping

Mapping Outline

We provide the encoding by means of several functions, one for each relevant syntactic category of Timed Rebeca from Figure 3.1 in Chapter 3. For instance encoding of *if* statements and almost all expressions are not shown since their implementation is trivial or subsumed by the encoding. Some of the functions are parametrized by the information contained in a structure we call *conf*, which stands for configuration. The structure *conf* contains three fields: *knownrebecs*, *statevar* and *localvars*. Intuitively, the field *knownrebecs* contains the set of known rebecs, and *statevar* and *localvars* contain the pairs variable-value for both state and local variables. The structure *conf* is created in a preprocessing step and then passed to the encoding mappings. Our functions use the standard dot notation in order to access the structure *conf*, e.g. *conf.statevar* to access the field *statevar* of the structure *conf*. Below we give a brief description of the relevant parts of the mappings.

$\mathcal{MO}(ev_1 \dots ev_n\ rc_1 \dots rc_n\ main)$ Encoding of a Rebeca model. Where $ev_1 \dots ev_n$ are environmental variables, $rc_1 \dots rc_n$ are reactive classes and $main$ is the code in the main block. This function, computes the structure $conf$ for each rebec and encodes each rebec passing this structure as parameter. Next, it encodes the code in $main$.

$\mathcal{R}(\texttt{reactiveclass}\ c\ \texttt{\{}kr\ sv\ m_0\ m_1\ m_2 \dots m_n\texttt{\}})$ Encoding of reactive classes. Where c is the name of the reactive class, kr is a set of knownrebecs, sv is a set of state

variables, m_0 is the *initial* method and $m_1 \dots m_n$ are methods. This function, encodes the reactive class in three Erlang functions with same name, but accepting different formal parameters, so with different signatures.

1. The first Erlang function accepts the known rebecs and call the second function.

2. The second function accepts the initial message, and once arrived, it runs the corresponding code, obtained with the mapping $\mathcal{B}$ which is explained below. This mapping returns a new set of state variables, since variables might have been changed during the execution of the inital message. Because structures are immutable in Erlang, they cannot be modified directly. In our encoding, following standard solutions, we create a new structure and return it as value. After the execution of the code for *initial*, this function calls the third function, passing this new set of state variables as well as the set of known rebecs.

3. The third function waits for incoming messages, which correspond to method calls. Once arrived, it runs the corresponding code, obtained again with $\mathcal{B}$. After that, it becomes ready to reaccept messages by calling itself with the modified set of state variables. Indeed variables might have been changed during the execution of a method.

$\mathcal{B}$(**msgsrv** m**(**$t_1 v_1 \dots t_n v_n$**){**$stmt_1 \dots stmt_n$**}**) Encoding of methods. Where m is the name of the method, $t_1 \dots t_n$ are type names, $v_1 \dots v_n$ are identifiers, and $stmt_1 \dots stmt_n$ are the statements of the body of the method. This function, using pattern matching, makes the reactive class wait for a message

$$\{\{Sender, TT, DL\}, m, \{v_1 \dots v_n\}\}$$

which is basically the messages exchanged in the actual Timed Rebeca. When such a message has arrived, we check if the deadline of the message is not expired. If not, we execute the statements of the method, otherwise a null action is executed. A few peculiarities of this encoding deserve a word.

- Performing a null action corresponds to that which in Timed Rebeca is the discarding of the message. Indeed, in the Erlang system the message has been delivered and it will be not processed again.

- The execution of statements is complex procedure. Indeed, structures are immutable in Erlang, but in Timed Rebeca the execution of some state-

ments might change variable values. Successive statements then should be executed knowing the new values for variables. Our solution is to execute every statement as a function that receive the set of variables as an argument, and return a new set of variable. The auxiliary function `AP` composes these functions.

- The function `tr_now()` recovers the current time from the Erlang primitive `erlang:now()`.

$\mathcal{S}$ Encoding of statements. This function encodes statements from Timed Rebeca into anonymous functions in Erlang. Functions receive as input the set of variables and return a new set of variables. The two relevant cases to discuss are the method invocation when it involves *after* and *deadline* constructs, and the *delay* statement.

- $\mathcal{S}(r.m(e_1 \ldots e_n)$ `after(`e_a`) deadline(`e_d`);`): It creates a new process using the primitive **spawn**. This new process, uses a **receive** with an empty body and the Erlang **after** to send the message. As stated above, differently from Timed Rebeca where messages are sent immediately but carrying the time tag from when they become retrievable, here the Erlang system takes care of this aspect for us, by waiting the expected amount of time before sending the message. Thanks to the primitive **spawn** the sender process does not stop its execution by the effect of **receive**. Instead the new process waits. Also, notice that the message sent by this new process contains the parent process as sender, not itself, which would be the expected behaviour.

- $\mathcal{S}$(`delay(`e`);`): It simply performs a **receive** with an empty body and **after**, in order to let the time pass, performing a null action, afterwards.

other mappings $\mathcal{I}$ and $\mathcal{T}$ translate name of the identifiers and types from Timed Rebeca to Erlang. $\mathcal{E}$ encodes expression and it is implemented in the conventional way. $\mathcal{K}$ translates constants from Timed Rebeca to Erlang. $\mathcal{M}$ encodes the *main* block. $\mathcal{PC}$ encodes rebec instance declarations and $\mathcal{L}$ encodes rebec setup by passing their known rebecs and initial message.

auxiliary functions we define the functionality of the following as a function of a reactive class to identifiers such that we can refer to these names at a later stage in the translation:

- $knownrebecnames$: **ReactiveClass** → **Ident** × **Ident**

- $statevarnames$: **ReactiveClass** → **Ident** × **Ident**

- *localvarnames* : **ReactiveClass → Ident × Ident**

Additionally, we need two functions to manipulate identifiers. An identifier in Erlang is a variable if it starts with an uppercase character. Otherwise, it is an atomic value.

- var_E : **Ident → Ident**
- $atom_E$: **Ident → Ident**

Finally, we need a function which gives initial values for a given type name such as 0 for integers and *false* for booleans

- *initval* : **TypeName → Value**

Code Translation

Listing 4.6 shows the translation functions described in this chapter.

```
1  MO(ev_1 ... ev_n  r_1 ... r_n  main) = R(r_1)  conf_1
2                    ⋮
3                    R(r_n)  conf_n
4                    M(main)  env
5    where  conf_n.knownrebecs = knownrebecnames(r_n)
6           conf_n.envvars = ev_1 × ev_n
7           conf_n.statevars = statevarnames(r_n)
8           conf_n.localvars = localvarnames(r_n)
9           env = {ev_1 ... ev_n}
10
11 R(reactiveclass c {kr sv m_0 m_1 ... m_n}) conf =
12   c(Env, InstanceName) →
13     receive
14       {B_var(kr) conf} →
15         c(Env, InstanceName, dict:from_list([B(kr) conf]))
16     end.
17   c(Env, InstanceName, KnownRebecs) →
18     StateVars = dict:from_list([B(sv) conf]),
19     LocalVars = dict:from_list([]),
20     {NewStateVars, _} = receive
21       B(m_0) conf
22     end,
23     c(Env, InstanceName, KnownRebecs, NewStateVars).
24   c(Env, InstanceName, KnownRebecs, StateVars) →
25     LocalVars = dict:from_list([]),
26     {NewStateVars, _} = receive
27       B(m_1) conf
28       ⋮
29       B(m_n) conf
```

```
30      end,
31      c(Env, InstanceName, KnownRebecs, NewStateVars).
32    where B_var = B without atomic names
33
34  B(knownrebecs{t_1 v_1 ... t_n v_n}) conf = atom_E(v_1), var_E(v_1) ... atom_E(v_n), var_E(v_n)
35
36  B(statevars{t_1 v_1 ... t_n v_n}) conf = atom_E(v_1), initval(t_1) ... atom_E(v_n), initval(t_n)
37
38  B(msgsrv m(t_1 v_1 ... t_n v_n){stmt_1 ... stmt_n}) conf =
39    {{Sender, TT, DL}, m, {w_1 ... w_n}} →
40      case DL == inf orelse tr_now() =< DL of
41        true →
42          AP(S(stmt_1) conf ... S(stmt_n) conf);
43        false →
44          % dropping message
45          {StateVars, LocalVars}
46      end
47
48  S(v=e;) conf = fun({StateVars, LocalVars}) →
49      v ∈ conf.statevars ⟶ {dict:store(v, E(e) conf, StateVars), LocalVars}
50      v ∈ conf.localvars ⟶ {StateVars, dict:store(v, E(e) conf, LocalVars)}
51      otherwise ⟶ error
52    end
53
54  S(r.m(e_1 ... e_n);) conf = fun({StateVars, LocalVars}) →
55      tr_send(I(r), I(m), {E(e_1) conf ... E(e_n) conf}),
56      {StateVars, LocalVars}
57    end
58
59  S(r.m(e_1 ... e_n) after(e_a) deadline(e_d);) conf = fun({StateVars, LocalVars}) →
60      tr_sendafter(E(e_a), I(r), I(m), {E(e_1) conf ... E(e_n) conf}, E(e_d) conf),
61      {StateVars, LocalVars}
62    end
63
64  S(delay(e);) conf = fun({StateVars, LocalVars}) →
65      tr_delay(E(e) conf),
66      {StateVars, LocalVars}
67    end
68
69  M(main{id_1 ... id_n}) env = main(X_var(env)) →
70      Env = dict:from_list([X(env)]),
71      PC(id_1),
72      ⋮
73      PC(id_n),
74      L(id_n),
75      ⋮
76      L(id_n).
77    where X(t_1 v_1 ... t_n v_n) = atom_E(v_1), var_E(v_1) ... atom_E(v_n), var_E(v_n)
78          X_var = B without atomic names
```

```
79
80  𝒫𝒞(t_r v_r (t_1 v_1 … t_n v_n):(k_1 … k_n)) =
81    ℐ(v_r) = spawn(fun() → ℐ(t_r)(Env, list_to_atom("ℐ(v_r)")) end)
82
83  ℒ(t_r v_r (t_1 v_1 … t_n v_n):(k_1 … k_n)) =
84    ℐ(v_r) ! {v_1 … v_n},
85    tr_send(ℐ(v_r), initial, {𝒦(k_1) … 𝒦(k_n)})
```

Listing 4.6: Translation functions from Timed Rebeca to Erlang

4.4 Simulating Timed Rebeca with McErlang

In this section we extend the mapping given in Section 4.3. The extended mapping allows us to simulate Timed Rebeca models with McErlang. Doing so, we can perform time based analysis on models. The analysis will give us some satisfaction of whether a model is correct given the time constraints.

McErlang

McErlang is a model checking tool to verify distributed programs written in Erlang. The tool itself is also written in Erlang and supports Erlang data types, process communication, fault detection and fault tolerance and the Open Telecom Platform (OTP) library, which is used by most Erlang programs. The verification methods range from complete state-based exploration to simulation, with specifications written as LTL formulae or hand-coded runtime monitors. This thesis focuses on simulation since model checking with real-time semantics is not yet supported by McErlang.

Runtime Monitoring

One of the reasons for using McErlang is to be able to write code that monitors the state of the simulation and either let the simulation continue running or stop the simulation due to an erroneous state or unexpected behaviour in the program.

McErlang only supports LTL expressions as properties in model checking, while hand-coded runtime monitors can be used in both model checking and simulation. Monitors are Erlang modules which export three functions; `init`, `stateChange` and `monitorType` of arity 1, 3 and 0, respectively.

- `init` is called when the monitor is started, at the beginning of the verification process.

- `stateChange` is called by McErlang at runtime whenever the program moves from one state to another. The arguments are `ProgramState`, `MonitorState` and `VerificationStack`. The program state contains information about the program which is running such as state of all processes and if any process is deadlocked. The monitor state gives us the action that lead to that state. The verification stack is a list of all the actions up to the current state. Actions can be receiving a message, executing the message and custom actions which the Erlang program can be instrumented with. We use that feature for dropped messages.
- `monitorType` specifies what kind of monitor is running. We use the `safety` monitor which is the only monitor that works for simulation.

4.5 Extended Formal Mapping

We extend the mapping given in Section 4.3 so that we can utilize McErlang to simulate Timed Rebeca models. The extension needs only a few changes for McErlang to monitor the simulation.

- First, McErlang has to be used as a library.
- Second, at the end of execution in each method McErlang needs to be notified of the state of the process. We want to be able to write monitors using McErlang API, hence we need to extend the mapping for it. The API call to notify McErlang of the state is called `mce:probe_state(Key, Value)`.
- Last, whenever a message is dropped due to deadline not being met, we need to notify McErlang of it. We might want to write a monitor that treats dropped messages as an error. The API call to notify McErlang of these kind of actions is `mce:probe_action(Key, Value)`.

Probing State Variables

We need to change the translation function $\mathcal{R}$ from Section 4.3 to incorporate the `mce:probe_state` calls before we make a recursive call. Two lines need to be added to the original translation in Listing 4.6 (see lines 11—32). Listing 4.7 shows the updated mapping.

```
1  R(reactiveclass c {kr sv m0 m1 ...mn}) conf =
2    c(Env, InstanceName) →
3      receive
4        {Bvar(k) conf} →
5          c(Env, InstanceName, dict:from_list([B(kr) conf]))
6      end.
7    c(Env, InstanceName, KnownRebecs) →
8      StateVars = dict:from_list([B(sv) conf]),
9      LocalVars = dict:from_list([]),
10     {NewStateVars, _} = receive
11       B(m0) conf
12     end,
13     mce:probe_state(InstanceName, NewStateVars),
14     c(KnownRebecs, NewStateVars).
15   c(Env, InstanceName, KnownRebecs, StateVars) →
16     LocalVars = dict:from_list([]),
17     {NewStateVars, _} = receive
18       B(m1) conf
19       ⋮
20       B(mn) conf
21     end,
22     mce:probe_state(InstanceName, NewStateVars),
23     c(Env, InstanceName, KnownRebecs, NewStateVars).
24   where Bvar = B without atomic names
```

Listing 4.7: Mapping for a reactive class

```
1  B(msgsrv m(t1v1 ...tnvn){stmt1 ...stmtn}) conf =
2    {{Sender, TT, DL}, m, w1 ...wn} →
3      case DL == inf orelse tr_now() =< DL of
4        true →
5          AP(S(stmt1) conf ...S(stmtn) conf);
6        false →
7          % dropping message
8          mce:probe(drop, {I(m), tr_now()}),
9          {StateVars, LocalVars}
10     end
```

Listing 4.8: Mapping for a message server

Probing Actions

We need to change the translation function $\mathcal{B}$ from Section 4.3 to notify McErlang of dropped messages due to message that deadline has been exceeded. We add a line to Listing 4.6 (see lines 38—47) in which we call `mce:probe` with the name of the message server whose message was dropped and the time of the dropping of the message. That will give us the flexibility we need to monitor specifically for dropped messages using McErlang runtime monitors. Listing 4.8 shows the updated mapping.

4.6 Discussion

In this chapter we presented Erlang and its concurrency and timing features. We associated syntactical elements from Timed Rebeca to corresponding ones in Erlang. The translation is given informally to convey the basic idea behind the methods used. Additionally, we give a formal translation which helps when creating a tool to automate the translation, as well as for reasoning about it. The formal translation also helps in finding possible inconsistencies or ambiguities.

Moreover, we have presented an extension to the formal mapping for simulating Timed Rebeca. It is as little as updating two translation functions. The extension as presented here turns out to map nicely to implementation as we will see in the next chapter.

In Appendix B we describe the tool which automates the translation presented in this chapter.

Chapter 5

Experimental Results

In this chapter we present three case studies. For each study we construct a model and use Timed Rebeca to analyze their timing behaviour. We describe each model, give a graph representation based on event graphs and then give results from simulating the models with the technique described in Chapter 4.

Before we present the models used in the experimental results we need to explain the notation we use, event graphs. Event graphs have a single type of node and two types of edges. The nodes represent events in a system. Edges correspond to the scheduling of other events (Buss, 1996). Jagged incoming edges denote an initial event. Edges can optionally be associated with a boolean condition for scheduling an event and/or a time delay which means that an event will be scheduled after the delay. Figure 5.1 shows an example of an event graph where event *B* is scheduled by *A* after *t* amount of time has been delayed and if condition *(i)* is true.

Event graphs are widely used in simulation and analysis of complex systems within the engineering community. More specifically, they're used to graphically represent discrete-event simulation models. We use event graphs in this thesis only to give a highly abstracted view of how events are scheduled in our case studies. We adopt an alternative notation where conditional edges are thicker, even if the conditions are not specified (Law, 2007). Additionally, we add a label below each node that shows in

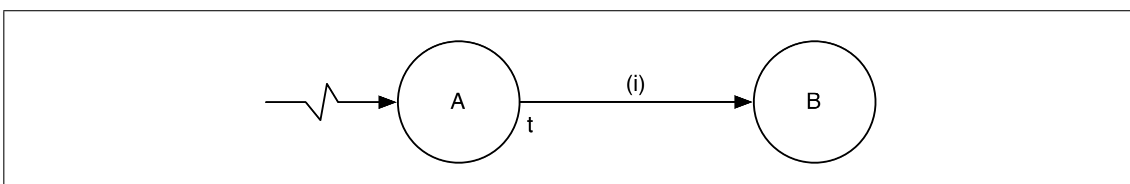

Figure 5.1: Example of an event graph

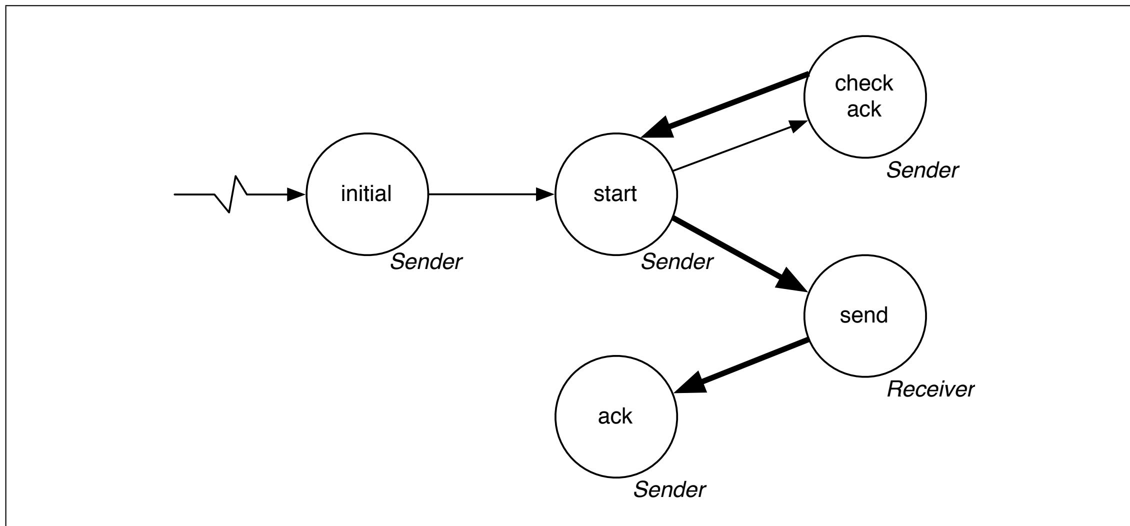

Figure 5.2: Event graph of the simple communication protocol model

which reactive class the event occurs. We decided to draw the graph based on reactive classes, and not rebecs, to have a simpler view.

5.1 Simple Communication Protocol

The simple communication protocol is an example from (Satoh & Tokoro, 1995) that consists of a sender agent and a receiver agent. The sender agent sends a message to the receiver and waits for acknowledgement. If acknowledgement is not received within 8 time units it resends the message. The receiver agent receives the message and replies with acknowledgement. A successful outcome is that the sender agent receives an ack message before 8 time units have passed. Communications from sender agent to receiver agent takes 3 ± 1 time units and may fail, while a message from receiver agent to sender agent takes 2 ± 1 time units and may also fail. This is a simple model where the execution terminates after the sender agent receives the ack.

Figure 5.2 shows the event graph of the simple protocol. The graph shows how the system is initialized from the sender agent. Moreover, there is a loop in the graph (start → check ack → start) which tells us if we select certain time constraints in the model it might result in an imbalance situation and an infinite computation. Listing 5.1 shows the Timed Rebeca code for the model.

Repeated simulations of the model show us that the model behaves as expected. They reveal that the system sometimes drops messages due to the nondeterministic choice of the network delay and this will execute the loop (start → check ack → start). The loop

was not repeated indefinitely which is not surprising since nondeterministic choices are implemented as random selections in the Erlang mapping.

```
1
2  reactiveclass SenderAgent(3) {
3    knownrebecs { ReceiverAgent receiverAgent; }
4
5    statevars { boolean receivedAck; }
6
7    msgsrv initial() { self.start(); }
8
9    msgsrv start() {
10     time sendDelay = ?(-1,2,3,4); // -1=fail -- 2,3,4=delays
11     if (sendDelay != -1) {
12       receiverAgent.send() after(sendDelay);
13     }
14     self.checkAck() after(8);
15   }
16
17   msgsrv ack() { receivedAck = true; }
18
19   msgsrv checkAck() {
20     if (!receivedAck) self.start();
21   }
22 }
23
24 reactiveclass ReceiverAgent(3) {
25   knownrebecs { SenderAgent senderAgent; }
26
27   statevars {}
28
29   msgsrv initial() {}
30
31   msgsrv send() {
32     time sendDelay = ?(-1,1,2,3); // -1=fail -- 1,2,3=delays
33     if (sendDelay != -1) {
34       senderAgent.ack() after(sendDelay);
35     }
36   }
37 }
38
39 main {
40   ReceiverAgent receiverAgent(senderAgent):();
41   SenderAgent senderAgent(receiverAgent):();
42 }
```

Listing 5.1: A Timed Rebeca model of the simple communication protocol example

5.2 Ticket Service

The ticket service model consists of two reactive classes: Agent and TicketService. Listing 5.2 shows this example written in Timed Rebeca. Two rebecs, *ts1* and *ts2*, are instantiated from the reactive class TicketService, and one rebec *a* is instantiated from the reactive class Agent. The agent *a* is initialized by sending a message *findTicket* to itself in which a message *requestTicket* is sent to the ticket service *ts1* or *ts2* based on the parameter passed to *findTicket*. The deadline for the message *requestTicket* to be

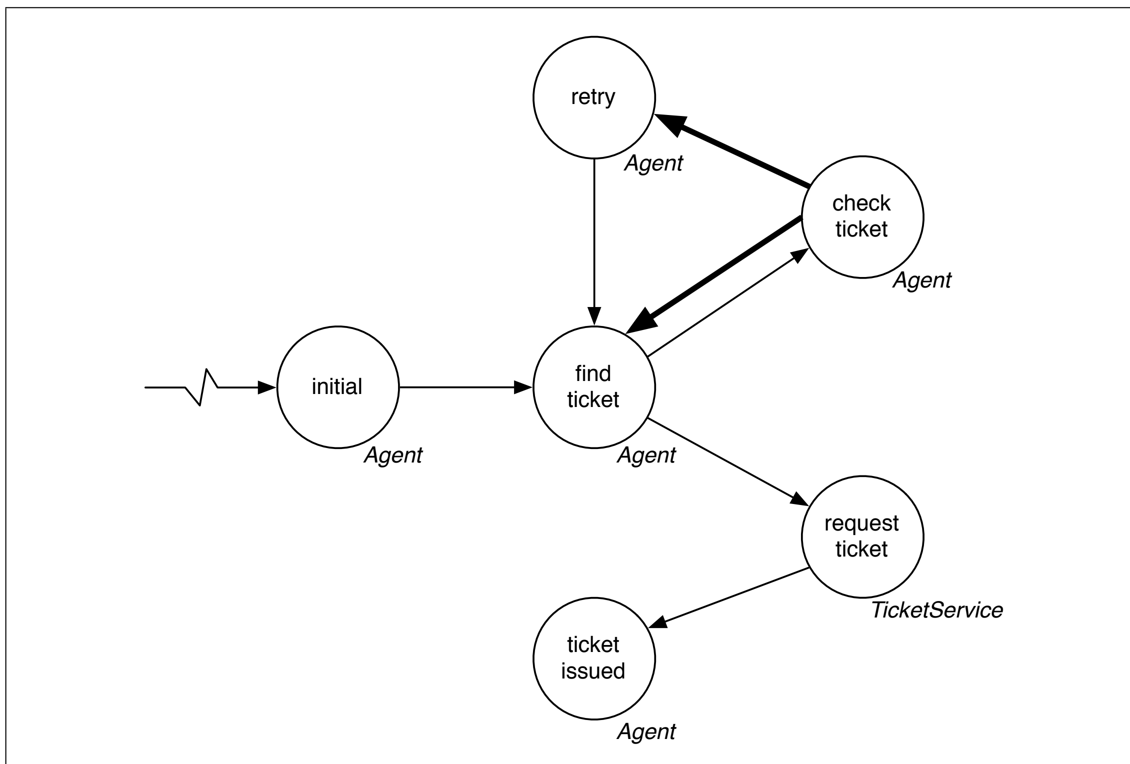

Figure 5.3: Event graph of the ticket service model

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

Figure 5.3 shows the event graph of the ticket service model. The graph shows how the system is initialized in the Agent class by scheduling a find ticket event.

The find ticket event will always schedule two simultaneous events, request ticket and check ticket. Request ticket event will schedule a ticket issued event after a delay. However, there is a network delay on the scheduling of check ticket, which means that

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

Table 5.1: Experimental simulation results for ticket service.

it may be scheduled later than ticket issued event is. In that case, either a find ticket event or retry event may be scheduled. Notice that the model is reactive, and will continue to schedule find ticket event to start the cycle all over again.

For each simulation, we change one of the following parameters: the amount of time that is allowed to pass before a request is processed, the amount of time that passes before agent checks if he has been issued a ticket, the amount of time that passes before agent tries the next ticket service if he did not receive a ticket, the amount of time that passes before agent restarts the ticket requests in case neither ticket service issued a ticket and two different service times, which are nondeterministically chosen as delay time in a ticket service and model the processing time for a request. Table 5.2 shows different settings of those parameters for which the ticket services never issue a ticket to the agent because of tight deadlines, as well as settings for which a ticket is issued during a simulation of the model.

```
1  env int requestDeadline, checkIssuedPeriod, retryRequestPeriod, newRequestPeriod, serviceTime1,
       serviceTime2;
2
3  reactiveclass Agent {
4    knownrebecs { TicketService ts1; TicketService ts2; }
5
6    statevars { int attemptCount; boolean ticketIssued; int token; }
7
8    msgsrv initial() {
9      self.findTicket(ts1);   // initialize system, check 1st ticket service
10   }
11
12   msgsrv findTicket(TicketService ts) {
13     attemptCount += 1;
14     token += 1;
15     ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
16     self.checkTicket() after(checkIssuedPeriod);      // check if the request is replied
17   }
18
19   msgsrv ticketIssued(int tok) {
20     if (token == tok) ticketIssued = true;
21   }
22
23   msgsrv checkTicket() {
24     if (!ticketIssued && attemptCount == 1) {        // no ticket from 1st service,
25       self.findTicket(ts2);                          // try the second TicketService
26     } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
27       self.retry() after(retryRequestPeriod);        // restart from the first TicketService
28     } else {                                         // the second TicketService replied,
29       self.retry() after(newRequestPeriod);          // new request by a customer
30     }
```

```
31    }
32
33    msgsrv retry() {
34      attemptCount = 0;
35      self.findTicket(ts1);       // restart from the first TicketService
36    }
37  }
38
39  reactiveclass TicketService {
40    knownrebecs { Agent a; }
41
42    msgsrv initial() { }
43
44    msgsrv requestTicket(int token) {
45      int wait = ?(serviceTime1,serviceTime2); // the ticket service sends the reply
46      delay(wait);                     // after a non-determinstic delay of
47      a.ticketIssued(token);           // either serviceTime1 or serviceTime2
48    }
49  }
50
51  main {
52    Agent a(ts1, ts2):();       // instantiate agent, with two known rebecs
53    TicketService ts1(a):();    // instantiate 1st ticket service, with
54                                // the agent as its known rebecs
55    TicketService ts2(a):();    // instantiate 2nd ticket service, with
56  }
```

Listing 5.2: A Timed Rebeca model of the ticket service example

5.3 Sensor Network

We model a simple sensor network using Timed Rebeca. See Listing 5.3 for the complete description of the model. A distributed sensor network is set up to monitor levels of toxic gasses. The sensor rebecs (*sensor0* and *sensor1*), announce the measured value to the admin node (*admin* rebec) in the network. If the admin node receives reports of dangerous gas levels, it immediately notifies the scientist (*scientist* rebec) on the scene about it. If the scientist does not acknowledge the notification within a given time frame, the admin node sends a request to the rescue team (*rescue* rebec) to look for the scientist. The rescue team has a limited amount of time units to reach the scientist and save him.

Figure 5.4 shows the event graph of the sensor network model. The graph shows how the system is initialized in the Sensor and Admin classes. The sensor class schedules *do report* events and continues to do so through out the life cycle of the system. The admin class schedules an event that checks the sensor values repeatedly. These events make the model reactive. The *check sensors* event may set off a routine that checks if a scientist has acknowledged about dangerous gas levels. Additionally, it may schedule the scientist to abort whatever is being perfomed. If scientist is instructed to abort,

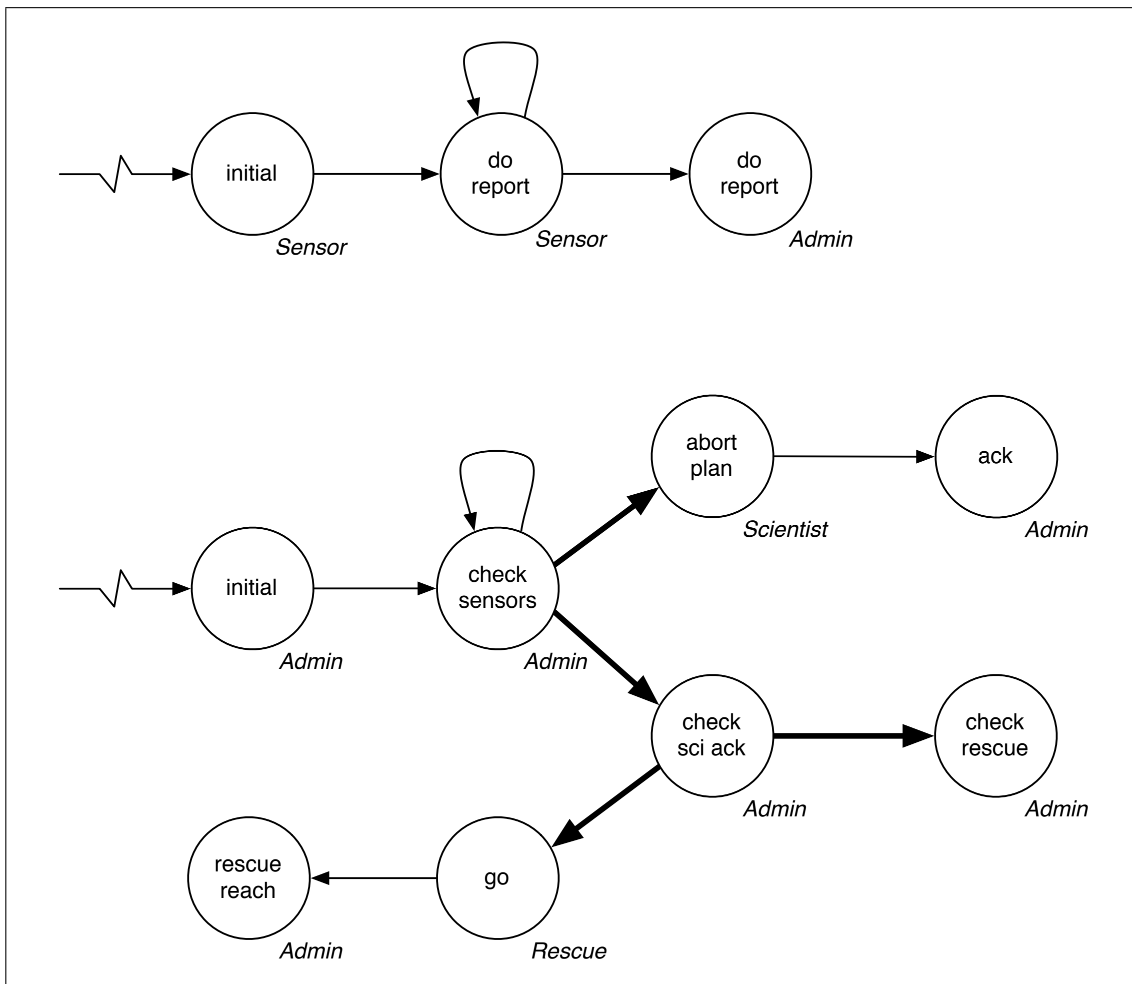

Figure 5.4: Event graph of the sensor network model. Notice there are two events that start the system.

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

The rebecs *sensor0* and *sensor1* will periodically read the gas-level measurement, modelled as a nondeterministic selection between *GAS_LOW* and *GAS_HIGH*, and send their values to *admin*. The *admin* continually checks, and acts upon, the sensor values it has received. When the *admin* node receives a report of a reading that is life threatening for the *scientist* (*GAS_HIGH*), it notifies him and waits for a limited amount of time units for acknowledgement.

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

Table 5.2: Experimental simulation results for sensor network.

The *rescue* rebec represents a rescue team that is sent off, should the *scientist* not acknowledge the message from the *admin* in time. We model the response speed of the rescue team with a nondeterministic delay of 0 or 1 time units.

The *admin* keeps track of the deadlines for the *scientist* and the *rescue* team as follows:

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

Otherwise we consider the mission failed.

The model can be parametrized over the values of network delay, *admin* sensor-read period, *sensor0* read period, *sensor1* read period, *scientist* reply deadline and *rescue* team reply deadline, as shown in Table 5.3.

In that table, we can see two different cases in which we go from mission failure to mission success between simulations. In the first scenario, we go from mission failure to success as we increase the rescue deadline, as expected.

In the second scenario, we changed the parameters to model a faster sensor update and we observed mission failure. In this scenario, increasing the rescue deadline further (from 5 to 7) is insufficient. Upon closer inspection, we observe that our model fails to cope with the rapid sensor updates and admin responses because it enters an unstable state.

The admin node initiates a new rescue mission while another is still ongoing, eventually resulting in mission failure. This reflects a design flaw in the model for frequent updates that can be solved by keeping track of an ongoing rescue mission in the model. Alternatively, increasing the value of *admin* sensor-read period above half the rescue deadline eliminates the flaw and the simulation is again successful.

```
1  env int netDelay, adminCheckDelay, sensor0period, sensor1period, scientistDeadline, rescueDeadline;
2
3  reactiveclass Sensor(3) {
```

```
4    knownrebecs { Admin admin; }
5
6    statevars { int period; }
7
8    msgsrv initial(int myPeriod) {
9      period = myPeriod;
10     self.doReport();
11   }
12
13   msgsrv doReport() {
14     int value;
15     value = ?(2, 4); // 2=safe gas levels, 4=danger gas levels
16     admin.report(value) after(netDelay);
17     self.doReport() after(period);
18   }
19 }
20
21 reactiveclass Scientist(3) {
22   knownrebecs { Admin admin; }
23
24   msgsrv initial() {}
25
26   msgsrv abortPlan() {
27     admin.ack() after(netDelay);
28   }
29 }
30
31 reactiveclass Rescue(3) {
32   knownrebecs { Admin admin; }
33
34   msgsrv initial() {}
35
36   msgsrv go() {
37     int msgDeadline = now() + (rescueDeadline-netDelay);
38     int excessiveDelay = ?(0, 1); // unexpected obstacle might occur during rescue
39     delay(excessiveDelay);
40     admin.rescueReach() after(netDelay) deadline(msgDeadline);
41   }
42 }
43
44 reactiveclass Admin(3) {
45   knownrebecs { Sensor sensor0; Sensor sensor1; Scientist scientist; Rescue rescue; }
46
47   statevars {
48     boolean reported0;
49     boolean reported1;
50     int sensorValue0;
51     int sensorValue1;
52     boolean sensorFailure;
53     boolean scientistAck;
54     boolean scientistReached;
55     boolean scientistDead;
56   }
57
58   msgsrv initial() { self.checkSensors(); }
59
60   msgsrv report(int value) {
61     if (sender == sensor0) {
62       reported0 = true;
63       sensorValue0 = value;
64     } else {
65       reported1 = true;
66       sensorValue1 = value;
67     }
68   }
69
70   msgsrv rescueReach() { scientistReached = true; }
```

```
71
72  msgsrv checkSensors() {
73    if (reported0) reported0 = false;
74    else sensorFailure = true;
75
76    if (reported1) reported1 = false;
77    else sensorFailure = true;
78
79    boolean danger = false;
80    if (sensorValue0 > 3) danger = true;
81    if (sensorValue1 > 3) danger = true;
82
83    if (danger) {
84      scientist.abortPlan() after(netDelay);
85      self.checkScientistAck() after(scientistDeadline); // deadline for the scientist to answer
86    }
87
88    self.checkSensors() after(adminCheckDelay);
89  }
90
91  msgsrv checkRescue() {
92    if (!scientistReached) scientistDead = true; // scientist is dead
93    else scientistReached = false;
94  }
95
96  msgsrv ack() { scientistAck = true; }
97
98  msgsrv checkScientistAck() {
99    if (!scientistAck) {
100     rescue.go() after(netDelay);
101     self.checkRescue() after(rescueDeadline);
102   }
103   scientistAck = false;
104 }
105 }
106
107 main {
108   Sensor sensor0(admin):(sensor0period);
109   Sensor sensor1(admin):(sensor1period);
110   Scientist scientist(admin):();
111   Rescue rescue(admin):();
112   Admin admin(sensor0, sensor1, scientist, rescue):();
113 }
```

Listing 5.3: A Timed Rebeca model of the sensor network example

5.4 Discussion

This chapter presented three case studies. It is natural to model distributed and asynchronous systems with Timed Rebeca. The language offers easy to use primitives for modelling timed scenarios that is demonstrated by the case studies. As with most formal analysis, we could see that the models show more complex behaviour than expected at first. The complexity stems from the asynchrony, which does not assume order of events in the system. Time constraints recuded the complexity by ordering

some of the events. On the other hand other, complexities are introduced by having real-time limitations.

Chapter 6

Related Work

Different approaches are used in designing formal modelling languages for real-time systems. In this chapter we briefly survey related work on real-time actor-based modelling languages. We also compare our work with UPPAAL, a well established tool for analyzing real-time systems.

6.1 RT-synchronizer

Ren and Agha proposed a real-time actor model, RT-synchronizer, where a centralized synchronizer is responsible for enforcing real-time relations between events (Ren & Agha, 1995). Actors are extended with timing assumptions, and the functional behaviours of actors and the timing constraints on patterns of actor invocation are separated. Nielsen and Agha gave semantics for the timed actor-based language (Nielsen & Agha, 1996). Two positive real-valued constants, called *release time* and *deadline*, are added to the *send* statement and are considered as the earliest and latest time when the message can be invoked.

In Timed Rebeca, we have the constructs *after* and *deadline*, which are representing the same concepts, respectively. In our language, it is also possible to consider a time *delay* in the execution of a computation. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, our model allows us to work at a lower level of abstraction. Using Timed Rebeca, a modeller can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various points of view.

6.2 Real-Time Maude

Maude is a high level declarative programming language. It supports executable specification and programming in rewriting logic. Moreover, it supports equational logic and algebraic specification. It can deal with nondeterministic concurrent computations and has support for concurrent object oriented computation models (Clavel et al., 2005). SOS semantics can be easily mapped to rewrite rules in Maude, making it an efficient tool for prototyping systems that are described in SOS.

Real-Time Maude is an extension to Maude. It supports both discrete and dense time domains. As with Maude, 0-time transtitions are defined with rewrite rules while time elapse is defined by *tick* rewrite rules. The Real-Time Maude offers timed rewriting for simulations, timed search for reachability analysis and time bounded LTL model checking (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 an interesting avenue to explore the Real-Time Maude tool has various ways to analyze timed systems.

6.3 Creol

Creol is a concurrent object-oriented language with operational semantics written in an actor-based style, and supported by a language interpreter in the Maude system. Boer, Chothia, and Jaghoori extended Creol by adding best-case and worst-case execution time for each statement, and a deadline for each method call (Boer et al., 2010). In addition, an object is assigned a scheduling strategy to resolve the nondeterminism in selecting from the enabled processes.

Bjørk, Johnsen, Owe, and Schlatte presented a timed version of Creol in which the only additional syntax is read-only access to the global clock, plus adding a data-type *Time* together with its accompanying operators to the language (Bjørk et al., 2010). Timed behaviour is modelled by manipulating the *Time* variables and via the *await* statement in the language.

6.4 UPPAAL

The model of timed automata, introduced by Alur and Dill (Alur, 1994), has established itself as classic a formalism for modelling real-time systems. The theory of Timed Automata is a timed extension of automata theory, using clock constraints on both locations and transitions.

UPPAAL is a toolbox which consists of three components: a modelling language, simulator and model checking tool. The language describes systems as networks of timed automata with extension of data variables (Larsen, Pettersson, & Yi, 1997). The simulator allows the modeller to examine the state space during the early stages of development. The model checking tool provides verification by means of exhaustive checking of the state space generated by the model. The model checking tool is supported by a specification language to check for reachability properties.

Timed Rebeca and UPPAAL differ greatly in what is accomplished by the model in each tool. UPPAAL allows us to model synchronous time varying behaviours while Timed Rebeca focuses on distributed and asynchronous agents. There has been some work to do verification on Timed Rebeca models by means of translation to UPPAAL but simple models run into state explosion problem (Izadi, 2010).

6.5 Schedulability for Rebeca Models

Recently, there have been some studies on schedulability analysis for Rebeca models (Jaghoori, Boer, Chothia, & Sirjani, 2009). This work is based on mapping Rebeca models to Timed Automata and using UPPAAL to check the schedulability of the resulting models. Deadlines are defined for accomplishing a service and each task spends a certain amount of time for execution. In these works, modelling of time is not incorporated in the Rebeca language.

There is also some work on schedulability analysis of actors (Nigro & Pupo, 2001), but this is not applied on a real-time actor language. Time constraints are considered separately.

Chapter 7

Conclusion

7.1 Summary

This thesis presents a number of results. First, we have developed a real-time extension to the modelling language Rebeca. The extension includes formal representation of syntax and semantics.

Next, we presented a formal mapping of refinement from Timed Rebeca to Erlang. The mapping is presented in a mathematical way which is helpful when reasoning about the translation and for creating the tool which automates the translation. The formal mapping is then extended to include code that makes it possible to simulate Timed Rebeca models with McErlang.

Third, we give a tool to automatically translate Timed Rebeca models to Erlang. The tool is based on the formal mapping, it operates directly on the syntax of Timed Rebeca. Moreover, it is written in a modular way to support extensions and other languages which future work might wish to target.

Last, we give results on using the simulation tool on three case studies. The case studies show that it is possible to use Timed Rebeca to describe real-time, distributed and asynchronous systems and using the tool to do analysis on the models. Even relatively simple models can become complex with different time constraints and difficult to analyze.

7.2 Future Work

As a part of a larger project which will address the issue of the entire modelling life cycle with verification, simulation and refinement, the work in this thesis lays the foundation for future work.

Branching out, we can translate to Maude which is a tool to prototype languages and perform analysis and verification. A translation to Real-Time Maude would allow designers to use the analysis tools supported by Maude in the verification and validation of Timed Rebeca models. It would also be interesting to see how Maude copes with the large state space that Timed Rebeca will generate when model checking Timed Rebeca models, since UPPAAL is known to suffer from state explosion.

Going deep, an interesting avenue to explore is to store the local time of each process and write a custom-made scheduler in McErlang that simulates the way the Timed Rebeca scheduler operates. In fact, this work was already under way few months ago but did not advance to a usable state.

The experiments in this thesis are not executed in a distributed environment. Experimenting with the generated Erlang code in a distributed setting would be interesting. No changes to the mapping are required, other than a new main function. It would need to establish connections to a number of Erlang shells (equal to the number of rebecs in the model). The shells would have to be running before this new main function is run. Running the experiments would require an infrastructure to do so and some way to measure the impact of the timing constraints.

Another interesting way, is to verify that the mapping is semantics preserving or to construct a bisimulation between Timed Rebeca and Erlang. This would give the mapping more credibility.

And last, we see the need to investigate a specification language to write Timed Rebeca properties with time constraints. Ideally, it should support specifying functional behaviour as well as time constraints, such that verification with time need not be separated from functional verification.

7.3 Discussion on Rebeca

Rebeca has been a part of our tool tool set for the last two years. We feel that it is a good language to model distributed and asynchronous systems and would like to propose a few extensions that might make it an even better language for modelling.

- Make actor identities to be first class citizens. Actor identities, according to the original actor model, are first class values and can be sent as a part of the messages. Rebeca as presented in (Sirjani et al., 2004) does not support this behaviour while the work in this thesis does. Doing so, we can remove the implicit variable *sender* from the context of message servers. The ticket service case study makes use of this to reduce boilerplate in the model.
- Remove the syntactical distinctions between state variables and known rebecs. The distinction in Rebeca code between state variables and known rebecs is not necessary at the syntax level. It is important to differentiate between state variables and known rebecs when simulating or model checking Timed Rebeca but the distinction does not have to be made by the modeller, it can be made automatically by a tool.
- Allow richer data structures such as finite polymorphic lists and iteration over them. Data structures like the classic cons list, which can destruct lists into head and tail, and create new lists by prepending an element in front of another list would make the modelling capabilities of Rebeca much richer. Then modelling structured data would be easier. It is possible to make sure that all computations that involve these structure are terminating by restricting the operations over them, which might otherwise be an issue for a language used for formal modelling. Allowing structures like this in model checking can increase the likelihood of state explosion. However, the modeller must already be aware of the state explosion problem and we reason that this structure should be allowed when it can be used. Otherwise, more abstraction is needed and the modeller should refrain from using it.
- Allow enumeration types. When modelling a variable which is a set of finite size and fixed domain the modeller often turns to using *int* type to store possible values of the variable. Example of this could be the days of the week, where the variable $int\ days$ should hold values from $0 - 6$. Working with an enumeration type where a modeller can specify a new type

  $$type\ Days = Sun|Mon|Tue|Wed|Thu|Fri|Sat$$

  would make the models clearer and less prone to error.

Bibliography

Aceto, L., Cimini, M., Ingolfsdottir, A., Reynisson, A. H., Sigurdarson, S. H., & Sirjani, M. (2011). Modelling and simulation of real-time systems using timed rebeca.

Agha, G. A. (1985, Jan). Actors: a model of concurrent computation in distributed systems. *AITR-844*.

Alur, R. (1994, Jan). A theory of timed automata. *Theoretical Computer Science*.

Appel, A. W. (1987, Jan). A standard ml compiler. *Functional Programming Languages and Computer Architecture*.

Armstrong, J. (1997, Jan). The development of erlang. *ACM SIGPLAN Notices*.

Bjørk, J., Johnsen, E. B., Owe, O., & Schlatte, R. (2010, Jan). Lightweight time modeling in timed creol. *Arxiv preprint arXiv:1009.4262*.

Boer, F. S. de, Chothia, T., & Jaghoori, M. M. (2010, Jan). Modular schedulability analysis of concurrent objects in creol. *Fundamentals of Software ...*.

Buss, A. H. (1996, Jan). Modeling with event graphs. *Proceedings of the 28th conference on Winter ...*.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al. (2005, Jan). Maude manual (version 2.4). *Menlo Park*.

Dowson, M. (1997, Jan). The ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes*.

Hewitt, C. (1972, Jan). Description and theoretical analysis (using schemata) of planner: A language for proving theorems and manipulating models in a robot. *dspace.mit.edu*.

Hewitt, C. (2007, Jan). What is commitment? physical, organizational, and social (revised). *Coordination*.

ICEROSE. (2011). *ICE-ROSE Homepage.* (http://en.ru.is/icerose/applying-formal-methods/)

ICETCS. (2011). *ICE-TCS Homepage.* (http://www.icetcs.ru.is/)

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

Jaghoori, M. M., Boer, F. S. de, Chothia, T., & Sirjani, M. (2009, Jan). Schedulability of asynchronous real-time concurrent objects. *Journal of Logic and Algebraic Programming*.

Jaghoori, M. M., Sirjani, M., Mousavi, M. R., Khamespanah, E., & Movaghar, A. (2010a, Jan). Symmetry and partial order reduction techniques in model checking rebeca. *Acta Informatica*.

Jaghoori, M. M., Sirjani, M., Mousavi, M. R., Khamespanah, E., & Movaghar, A. (2010b, Jan). Symmetry and partial order reduction techniques in model checking rebeca. *Acta Informatica*.

Jones, S. L. P., Hall, C., Hammond, K., Partain, W., & Wadler, P. (1992, Jan). The glasgow haskell compiler: a technical overview.

Kahn, G. (1987, Jan). Natural semantics. *STACS 87*.

Lämmel, R., Visser, J., & Kort, J. (2000, Jan). Dealing with large bananas.

Larsen, K. G., Pettersson, P., & Yi, W. (1997, Jan). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*.

Law, A. M. (2007). *Simulation modeling and analysis* (4nd ed.). McGraw-Hill.

Nielsen, B., & Agha, G. (1996, Jan). Semantics for an actor-based real-time language. *wpdrts*.

Nigro, L., & Pupo, F. (2001, Jan). Schedulability analysis of real time actor systems using coloured petri nets. *Concurrent object-oriented programming and petri nets*.

Ölveczky, P. C., & Meseguer, J. (2007, Jan). Semantics and pragmatics of real-time maude. *Higher-Order and Symbolic Computation*.

Pellauer, M., Forsberg, M., & Ranta, A. (n.d.). Bnf converter: Multilingual front-end generation from labelled bnf grammars.

Plotkin, G. D. (1981, Jan). A structural approach to operational semantics.

Pratt, V. (1995, Jan). Anatomy of the pentium bug. *TAPSOFT'95: Theory and Practice of Software Development*.

Ren, S., & Agha, G. A. (1995, Jan). Rtsynchronizer: language support for real-time specifications in distributed systems. *ACM SIGPLAN Notices*.

Satoh, I., & Tokoro, M. (1995, Jan). Time and asynchrony in interactions among distributed real-time objects. *ECOOP'95—Object-Oriented Programming*.

Sirjani, M., Movaghar, A., Shali, A., & Boer, F. S. de. (2004, Jan). Modeling and verification of reactive systems using rebeca. *Fundamenta Informaticae*.

Sirjani, M., Movaghar, A., Shali, A., & Boer, F. S. de. (2005, Jan). Model checking, automated abstraction, and compositional verification of rebeca models. *Journal of Universal Computer Science*.

Appendix A

Timed Rebeca Language Description

A.1 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 Timed Rebeca are the following:

```
after       boolean       deadline
delay       else          env
false       if            initial
int         knownrebecs   main
msgsrv      now           reactiveclass
statevars   time          true
```

The symbols used in Timed Rebeca are the following:

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

Comments

Single-line comments begin with `//`.

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

A.2 Syntactic Structure of Timed Rebeca

Non-terminals are enclosed between ⟨ and ⟩. 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*⟩ `}`

⟨*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*⟩

⟨*VarDecl*⟩ ::= ⟨*Ident*⟩ = ⟨*Exp*⟩
| ⟨*Ident*⟩

⟨*ListVarDecl*⟩ ::= ⟨*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`

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

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

⟨*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⟩ )
             |  ⟨Exp12⟩

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

⟨Exp13⟩    ::=  now ( )
             |  ⟨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*⟩

Appendix B

Implementation of the Translation Tool

In this section we present the tool *timedreb2erl* which translates Timed Rebeca models to Erlang. This tool can output code both based on refinement and simulation mappings. Additionally, if simulation is chosen as the output target it will output template for monitor code. The tool is built with Haskell and BNFC.

Haskell

Haskell is a purely functional programming language with non-strict evaluation semantics by default. Functional programming languages are widely used in compiler construction (Appel, 1987; Jones, Hall, Hammond, Partain, & Wadler, 1992) as they generally support algebraic data types and compositional reasoning very well. This applies particularly well to Haskell whose expressions are referentially transparent and make equational reasoning safe and useful for the programmer.

BNF Converter

BNF Converter (BNFC) (Pellauer, Forsberg, & Ranta, n.d.) is a multi-lingual compiler tool. It takes as input a grammar written in Labelled BNF notation (LBNF) and generates a compiler front end (lexer, parser, data types), a case skeleton for the back end compiler code which is useful as a starting point for code generation, and LaTeX document which contains the language specification. BNFC is able to generate code for Java, C, C++, C#, OCaml, F# and Haskell.

This allows for rapid prototyping of language constructs. Having only to deal with the syntax of the source language and get a whole front end generated, one can focus on,

```
transModel :: Model → Result
transModel x = case x of
  Model envvars reactiveclasss main → failure x

transEnvVar :: EnvVar → Result
transEnvVar x = case x of
  EnvVar typedparameter → failure x

transReactiveClass :: ReactiveClass → Result
transReactiveClass x = case x of
  ReactiveClass id n knownrebecs statevars msgsrvinit msgsrvs → failure x

transKnownRebecs :: KnownRebecs → Result
transKnownRebecs x = case x of
  NoKnownRebecs → failure x
  KnownRebecs typedvardecls → failure x

transStateVars :: StateVars → Result
transStateVars x = case x of
  NoStateVars → failure x
  StateVars typedvardecls → failure x
```

Listing B.1: Example of generated template for code generation

arguably, the more time consuming work of the back end instead of boilerplate work at the front end.

B.1 First Implementation

The first implementation used BNFC for rapid prototyping of the source language, Timed Rebeca. Example output of the case skeleton for Timed Rebeca can be seen in Listing B.1. The data types of Timed Rebeca constructs are directly derived from the syntax of the language as seen in Appendix A. Notice that `failure x` is a function representing unimplemented functionality.

In this setting the result of the translation was `String` which means that each translation function built up the translated code as concatenated strings. This is not considered to be very good software engineering practice, and we can summarize the reasons in the following:

1. The output language is untyped,

```haskell
data BasicValue = Atom String
data Expr = Var String
```

Listing B.2: Example of how to type the translation to Erlang

2. we cannot reuse the traversal functions since they're intertwined with the code generation,
3. we cannot, in a clean way, swap one translation function out for another as we did in Chapter 4 when extending the mapping to include McErlang API calls.

Untyped Target Language

The problem with collecting the code of the generated Erlang program as strings is that it is error prone. For example, there is a semantic different between strings in Erlang programs that start with lower case character and upper case character. The difference is that upper case means that the string is a *variable*. If it is lower case it could be a *value*. This means that much care must be taken whenever working with strings to preserve the meaning of the string.

However, if we type the target language we can at least contain the problem although it does not solve it completely. Listing B.2 shows an example of how we can implement data types for the language of Erlang as given in Figure 4.1.

If we only ever construct atomic values with the `Atom` data constructor and variables in expressions with the `Var` data constructor we can make sure when we print the syntax tree of the constructed Erlang programs that atomic values will start with a lower case character and variable with upper case characters. This is possible due to the fact that no *strings* are generated for the target language until the entire syntax tree has been built. Therefore, when we do output the tree as Erlang code, the function that prints values of type `BasicValue` transforms the string to be lower case.

Program Analysis and Transformation

Let's give a motivating example for item 2: We need to find all variable names of state variables before we generate code for each reactive class. Let's capture this functionality in a function with the following declaration:

```haskell
statevars :: ReactiveClass → [String]
```

```
transModel :: Model → (ReactiveClass → Result) → Result
transModel x r = case x of
  Model envvars reactiveclasss main → failure x
```

Listing B.3: Suggestion to the extended mapping problem

The statevars function returns a list of variable names for a given reactive class. We need to take care of traversing the same amount of syntax tree as in Listing B.1 but we cannot use the same traversal code and must duplicate the traversal for the simple task of collecting variable names instead of generating code.

Another example is a program transformation that must be performed before we translate a Timed Rebeca model to Erlang. Timed Rebeca supports assignments operators such as $x+ = e$ which adds the value of e to the variable x. These kind of composite assignment operators are not supported in Erlang which only supports simple assignments. We must therefore translate all instances of $x \oplus e$ where $\oplus \in$ {+=, -=, *=, /=, %=}.

Extended Mapping

The generated traversal code from BNFC is not equipped for reusing parts of the translation functions. One solution would be to make the top level translation parametric for the functions from the extended mapping. One possible implementation of that is shown in Listing B.3 where the function r would be passed down the tree and called whenever we need to translate a reactive class.

This would require adding these hooks and changing the code in unpredictable manner as the mapping is extended.

Another solution is to devise a *fold algebra* which is a combination of generic folds for tree traversal and algebras for instantiations of the data types in question (Lämmel, Visser, & Kort, 2000). This provides a nice balance of modularity for the problems we defined earlier and is the subject of next section along with the definitions of Erlang syntax as Haskell data types.

```haskell
type Name = String

data Program = Program Attribute [Attribute] [Attribute] [Function] -- module,
    export, import

data Attribute = Module Name | Export Name | Import Name

data Function = Function Name [Pattern] Exp

data BasicValue = AtomicLiteral String | StringLiteral String | NumberLiteral
    Integer | ProcessLiteral String

data InfixOp = OpLT | OpLEq | OpGT | OpGEq | OpEq | OpNEq | OpLAnd | OpLOr |
    OpMul | OpDiv | OpMod  | OpSub | OpBAnd | OpBXor | OpBOr | OpAdd

data Exp = InfixExp InfixOp Exp Exp | Apply Name [Exp] | Call Exp Exp | Case Exp
    [Match] | FunAnon [Pattern] Exp | Receive [Match] | If [Match] | Send Exp Exp
    | Seq Exp Exp | Assign Pattern Exp | ExpT [Exp] | ExpL [Exp] | ExpVal
    BasicValue | ExpVar Name

data Match = Match Pattern (Maybe Guard) Exp

data Pattern = PatVar Name | PatT [Pattern] | PatL [Pattern] | PatVal BasicValue
    | PatE Exp

data Guard = InfixGuard Guard Guard | GuardVal BasicValue | GuardVar Name |
    GuardCall Guard [Guard] | GuardT [Guard] | GuardL [Guard]
```

Listing B.4: Haskell data types for Erlang syntax

B.2 Revised Implementation

Data Types for Erlang

To make the code generation safer we created data types based on the subset of Erlang presented in Section 4.2. The data types are shown in Listing B.4 and are used as a foundation in the fold algebra. They are expressive enough for our translation scheme although they don't capture entire Erlang syntax.

Fold Algebra

The fold algebra will consist of two components, a generic fold algebra and behaviours for particular constructors. This allows us to separate two concerns, walking the syntax tree and specific behaviour by an algebra. Examples of processing can be finding state

```
1  data RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs aft dea eli
       el exp con uop aop mai ins
2      = RebecaAlgebra {
3      identF :: String → id
4    , modelF :: [env] → [rc] → mai → mod
5    , envVarF :: tp → env
6    , reactiveClassF :: id → Integer → kr → sv → msi → [ms] → rc
7    , noKnownRebecsF :: kr
8    , knownRebecsF :: [tvd] → kr
9    , ⋮
10 }
```

Listing B.5: Algebra interface (partial listing)

variables, transforming assignment operators or generating corresponding syntax tree in Erlang.

An algebra has the interface outlined in Listing B.5. We can see that it is a record, parametric in number of data types in the language. Each data constructor is then given a field in the record.

The fold operation encodes the tree walk over the types in the language. It is parametrized by the fold algebra. Listing B.6 shows how fold operation is represented by a multi parameter typeclass `Fold` and how `fold` is implemented by making a recursive call into the sub terms of a type and combining the intermediate results by calling a specific component in the algebra. Notice how `f` is passed along the entire computation which is an instance of an algebra.

Program Analysis with Fold Algebra

In this section we present a fold algebraic solution to the problem of finding state variable names from Section B.1. We start by defining a monoidal algebra which collects all identifiers in a tree. Listing B.7 shows an outline of the algebra. Notice how we do not traverse further into the structure of known rebecs. This is because the monoid algebra is a template for collecting identifiers and we need to instantiate it for a specific set of variables.

To collect the names of all known rebecs of reactive class is now easy. We update the monoid algebra such that the field `knownRebecsF` no longer returns the identity

```
1  class Fold f t r | f t → r where
2      fold :: f → t → r
3
4  instance Fold (RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs
       aft dea eli el exp con uop aop mai ins) Ident id where
5      fold f (Ident s) = identF f s
6
7  instance Fold (RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs
       aft dea eli el exp con uop aop mai ins) Model mod where
8      fold f (Model vars classes mainbody) = modelF f (map (fold f) vars) (map
           (fold f) classes) (fold f mainbody)
9
10 instance Fold (RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs
       aft dea eli el exp con uop aop mai ins) EnvVar env where
11     fold f (EnvVar tp) = envVarF f (fold f tp)
12
13 instance Fold (RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs
       aft dea eli el exp con uop aop mai ins) ReactiveClass rc where
14     fold f (ReactiveClass name qs kr sv msi ms) = reactiveClassF f (fold f name)
           qs (fold f kr) (fold f sv) (fold f msi) (map (fold f) ms)
15
16 instance Fold (RebecaAlgebra id mod env rc kr sv msi ms vd tvd tp bt tn stm cs
       aft dea eli el exp con uop aop mai ins) KnownRebecs kr where
17     fold f NoKnownRebecs = noKnownRebecsF f
18     fold f (KnownRebecs tvds) = knownRebecsF f (map (fold f) tvds)
19 ⋮
```

Listing B.6: Fold algebra (partial listing)

```
1  monoidAlgebra = RebecaAlgebra {
2      identF = λs → [s]
3    , modelF = λenvs rcs mai → (mconcat envs) `mappend` (mconcat rcs)
4    , envVarF = λ_ → mempty
5    , reactiveClassF = λid _ kr sv msi ms → kr `mappend` sv `mappend` msi `mappend`
         (mconcat ms)
6    , noKnownRebecsF = mempty
7    , knownRebecsF = λ_ → mempty
8    , ⋮
```

Listing B.7: Monoidal algebra for collecting identifier names (partial listing)

```
knownRebecsAlgebra = monoidAlgebra {
    knownRebecsF = λtvds → mconcat tvds
}
```

Listing B.8: Updated monoidal algebra for collecting known rebec names

```
identityAlgebra = RebecaAlgebra {
    identF = Ident
  , modelF = Model
  , envVarF = EnvVar
  , reactiveClassF = ReactiveClass
  , noKnownRebecsF = NoKnownRebecs
  , knownRebecsF = KnownRebecs
  , ⋮
}
```

Listing B.9: Instance of an identity algebra (partial listing)

element of the monoid but the concatenated result of its sub terms. Listing B.8 shows the updated algebra.

Program Transformation with Fold Algebra

We give the outline of the simplification of assignment operators such that only = is used in assignments. First, we devise an instance of an algebra called *identity* algebra that returns the same that it is given back. Outline of the algebra is given in B.9.

The identity algebra takes care of simple tree walk that does not change anything in the tree. We can then update the identity algebra to obtain a new instance of an algebra with specific behaviour for simplifying the assignment operators. Listing B.10 shows how easy it is to create a new algebra based on existing ones.

Translation to Erlang with a Monadic Fold Algebra

The translation from Timed Rebeca to Erlang is done with a fold algebra. The translation is based on the formal mapping from Section 4.3. In the formal mapping there is a configuration passed down to some translation functions, which contains information on the names of variables of a reactive class. Notice how the interface for the fold algebra does not have this configuration as a parameter anywhere. Instead of changing the algebra to be suitable for this translation we use monads to implicitly thread the

```
1 simplifyAssignmentAlgebra = identityAlgebra {
2   assF = λid aop exp → case aop of
3                            Assign → Ass id aop exp
4                            AssignMul → Ass id Assign (Etimes (Evar [id]) exp)
5                            AssignDiv → Ass id Assign (Ediv (Evar [id]) exp)
6                            AssignMod → Ass id Assign (Emod (Evar [id]) exp)
7                            AssignAdd → Ass id Assign (Eplus (Evar [id]) exp)
8                            AssignSub → Ass id Assign (Eminus (Evar [id]) exp)
9 }
```

Listing B.10: Assignment simplification algebra instance

configuration around the translation functions. More specifically, we use the State monad in Haskell to implement the monadic fold algebra. Listing B.11 shows when state variables block is translated, we put the collected variable names into the state monad via `setStateVars` function and when assignments are translated, we retrieve the names of state variables and local variables and return different values based on which set the identifier is found in.

An algebra which translates Timed Rebeca models to the extended mapping from Chapter 4 is now easy to implement. It requires updating `refinementAlgebra` and overriding the behaviour that generates Erlang code for reactive classes and message servers.

B.3 Discussion

We have presented a tool which is capable to translating Timed Rebeca models into Erlang. Moreover, the tool allows for good reuse and composition by using fold algebras for manipulating data types of Timed Rebeca language. This makes the foundations for the tool a good option for translating Timed Rebeca to other languages. The tool can be found at http://github.com/arnihermann/timedreb2erl.

```
1  refinementAlgebra = RebecaAlgebra {
2      identF = λid → return id
3    , ⋮
4    , noKnownRebecsF = return []
5    , knownRebecsF = λtvds → do
6          tvds’ ← sequence tvds
7          setKnownRebecs (map snd tvds’)
8          return (map snd tvds’)
9    , noStateVarsF = return []
10   , stateVarsF = λtvds → do
11         tvds’ ← sequence tvds
12         setStateVars (map snd tvds’)
13         return tvds’
14   , ⋮
15   , assF = λid aop exp → do
16         id’ ← id
17         aop’ ← aop
18         exp’ ← exp
19         sv ← getStateVars
20         lv ← getLocalVars
21         let assignment
22                 | id’ ‘elem‘ sv = ExpT [Apply ”dict:store” [ExpVal $
                       AtomicLiteral id’, exp’, ExpVar ”StateVars”], ExpVar
                       ”LocalVars”]
23                 | id’ ‘elem‘ lv = ExpT [ExpVar ”StateVars”, Apply ”dict:store”
                       [ExpVal $ AtomicLiteral id’, exp’, ExpVar ”LocalVars”]]
24                 | otherwise = error $ ”unknown variable name ” ++ id’
25         return (stm assignment)
26   , ⋮
```

Listing B.11: Instance of an algebra that translates Timed Rebeca to refined Erlang code (partial listing)

Appendix C

Design Decisions: Imperative to Functional

Certain care needs to be taken when translating imperative code, like Rebeca, to a language like Erlang. Erlang does not allow multiple assignments to a variable. In fact, Erlang does not do assignments at all. What looks like an assignment operator (=) is really a pattern matching operator. Let's discuss each executed expression in an Erlang Emulator from Figure C.1. Expressions are prefixed with a number and the result is in the next line.

1. `Foo = true.`, shows that the pattern matching is successful because no value was previously bound to the variable `Foo` and therefore Erlang assigns the value `true` to it.
2. Erlang is unable to match the value `false` to the value that is currently bound to `Foo`, therefore resulting in runtime exception.
3. The pattern matching is successful if we try to match it with a value that is currently bound to the variable `Foo`.

```
1> Foo = true.
true
2> Foo = false.
** exception error: no match of right hand side value false
3> Foo = true.
true
```

Figure C.1: Erlang Emulator session showing how pattern matching in Erlang works

```
msgsrv foo() {
  c = false;
  if (a || b) {
    c = true;
    d = !c;
  } else {
    d = true;
  }
  e = c || d;
}
```

Listing C.1: Example message server for translation attempts

Additionally, since state in Erlang processes is kept as parameters in functions and updated by making a recursive call, we cannot translate imperative code with multiple assignment statements (see Listing C.1) that executes conditionally, into one function in Erlang. We can see how a *direct* translation does not work out if we try to capture the imperative code in an Erlang function called `foo` that is defined as `foo(A,B,C,D,E)`. In the first statement, we try to assign `X` to be 1 and would need to make a recursive call to update the value of `X` for the following computation. In the following sections we discuss the options we looked at and which we ended up using.

C.1 Approach 1: Derive Functions from Control Flow Graph

First solution that we came up with was to create a control flow graph of statements in a given Rebeca code. A branch (`if` statement) in the code would generate a new function for the rest of the graph which the current statement would call. Figure C.2 shows the control flow graph of Listing C.1. To derive functional behaviour in Erlang equivalent to the one in Timed Rebeca we translate each node to a function which recursively calls the next the function that represents the next node in the graph. The functions are all mutually recursive. Listing C.2 shows the Erlang code for the control flow graph in Figure C.2 with nodes 2–3 and 2–4 compacted such that the transition of the first statement after entering the branch is executed immediately. The resulting translation is 5 functions but could be reduced to fewer functions by compressing the graph based on data flow analysis.

Formalizing a translation for Timed Rebeca to Erlang in this way appears to be complicated and hence we looked at different options.

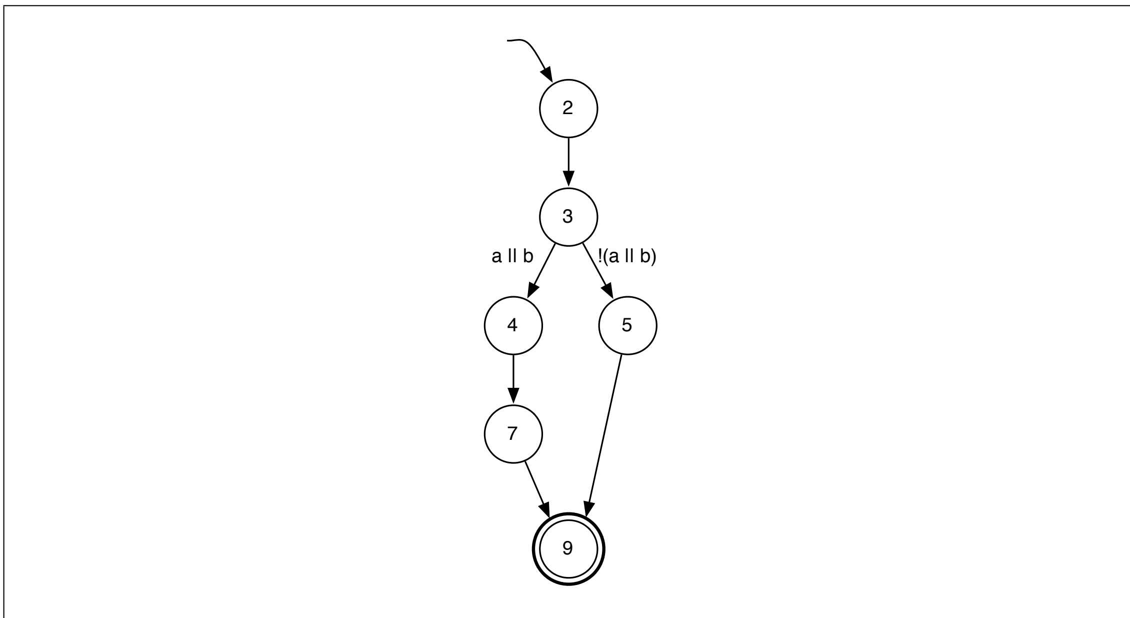

Figure C.2: Control Flow Graph of Listing C.1. Node numbers correspond to line numbers.

```
1  foo(A,B,C,D,E) → foo_1(A,B,C,D,E).
2  foo_1(A,B,C,D,E) → foo_2(A,B,false,D,E).
3  foo_2(A,B,C,D,E) →
4    if
5      A orelse B → foo_3(A,B,true,D,E);
6      true       → foo_4(A,B,C,true,E)
7    end.
8  foo_3(A,B,C,D,E) → foo_4(A,B,C,not C,E).
9  foo_4(A,B,C,D,E) → foo(A,B,C,D,C orelse D)
```

Listing C.2: Erlang code derived from the CFG in Figure C.2

C.2 Approach 2: Static Single Assignment Form and Records for State Variables

Before we present a translation using SSA, note that we use Erlang records for keeping state variables of a reactive class in a single Erlang variable. Erlang records are a compiler trick which stores the values of the record in a tuple but gives access to the tuple by name (instead of positional pattern matching).

With that in place, we thought about using SSA to thread the state through the computation of a message server. In SSA, variables are only assigned once, which fits well with Erlang. When translating statements that modify state variables, the Erlang code would create a new state variable record with an updated value for the respective field

```
-record(statevars, {a=false, b=false, c=false, d=false, e=false}).

foo(SV) →
  SV1 = SV#statevars{c=false},
  SV2 = if
          a orelse b →
            SV3 = SV2#statevars{c=true},
            SV4 = SV3#statevars{d=not SV3#statevars.c};
          true →
            SV3 = SV2#statevars{d=true}
        end,
  SV3 = SV2#statevars{e=SV2#statevars.c orelse SV2#statevars.d}
  foo(SV3).
```

Listing C.3: Erlang code derived from Listing C.1 using SSA

in the record. Listing C.3 shows how a record is defined for the message server in Listing C.1 and SSA is used to thread the state around in the function `foo`.

Formalism for this translation requires quite a bit of bookkeeping of the variable name and make the translation more complex.

C.3 Approach 3: State Transformer Functions and Records for State Variables

Previous approaches both required much work in the translation of Timed Rebeca to Erlang. The biggest drawback is the amount of context which is needed for which function to call or which variable name to use. Hence came the idea about using state transformer functions where each statement in Timed Rebeca is translated to an anonymous function in Erlang. The functions have the type `State` → `State`. Thus the first function is called with the current state of the process and returns a possibly updated state for the next function to use. The anonymous functions can be composed via ordinary function composition, such as given two functions `f` and `g` the composition `f ∘ g` is a new function `h = fun(S)→f(g(S))end`. A minor issue is that the innermost function in an `if`-branch must be applied with the currently scoped `SV` variable, otherwise the branch would return a function instead of the result of the function inside the branch. Listing C.4 shows how this works in practice, albeit without intermediate functions for the composition.

This approach has a certain simplicity to it that the previous ones lack.

```
foo(SV) →
  fun(SV) → SV#statevars{c=SV#statevars.c orelse SV#statevars.d} end (
    fun(SV) →
      if
        a orelse b →
          fun(SV) → SV#statevars{d=not c} end (
            fun(SV) → SV#statevars{c=true} end (SV));
        true →
          fun(SV) → SV#statevars{d=true} end (SV)
      end (
        fun(SV) → SV#statevars{c=false} end (StateVars))
    end)
```

Listing C.4: Erlang code derived from Listing C.1 using state transformers

C.4 Approach 4: State Transformer Functions and Dictionaries for State Variables

There is a slight annoyance with the last approach. A record declaration had to be generated and put in a header file (outside the generated code) and the unique name of the record had to be passed around in a configuration throughout the translation phase.

This is easily remedied by not storing state variables in records, but instead simply store them as dictionaries. Little work is required to change the mapping. One unfortunate exception is that `if` statements no longer translate directly since reading the value of variable with the `dict:fetch` function is considered a side-effecting operation in Erlang, and side-effecting operations are not allowed in guards. This is a minor annoyance which we solve by translating `if` statements into nested `case` expressions in Erlang.

We expect dictionaries take up more memory than tuples and accessing elements from them might not be as fast as from a tuple. That is a sacrifice we are willing to make in order to keep the mapping and the implementation of the translation simpler. We suspect that if this foundation gets used in model checking with Erlang, this could become a real issue.

Record accessors and updates are therefore swapped for a `dict:fetch(Key, Dict)` and `dict:store(Key, Value, Dict)` function applications, respectively.

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