SIMULATION-BASED ANALYSIS OF TIMED REBECA USING TEPROP AND SQL

Brynjar Magnússon
Master of Science
Software Engineering
June 2012
School of Computer Science
Reykjavík University

M.Sc. RESEARCH THESIS

ISSN 1670-8539

Simulation-based Analysis of Timed Rebeca using TeProp and SQL

by

Brynjar Magnússon

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 2012

Research Thesis Committee:

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

Dr. Luca Aceto
Professor, Reykjavík University, Iceland

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

Simulation-based Analysis of Timed Rebeca using TeProp and SQL

Brynjar Magnússon

June 2012

Abstract

As software systems get larger, more complex and time critical, we need formal methods to get them right. In order to get practitioners to use formal methods we need easy to use modeling and specification languages and powerful and scalable verification tools. We present TeProp, a timed event-based property language, which is designed to reason about timed occurrence of events in a natural way and be easy to use for specifying properties.

We also present TRSim, a simulation tool-kit for working with simulations of Timed Rebeca models. McErlang is used to simulate the models while the occurrences of events are stored in a relational database. TeProp properties can then be checked against multiple simulation runs, using a query client that transforms a TeProp property to an SQL query. The SQL query is then executed on the database which determines whether the property is satisfied over the simulation runs in the database.

To show the capability of TeProp and the TRSim tool-kit we provide few case studies and show that we are able to detect a flaw in previously analyzed model.

Hermunargreining á Timed Rebeca með TeProp og SQL

Brynjar Magnússon

Júní 2012

Útdráttur

Jafnfram því sem hugbúnaður verður stærri, flóknari og háðari tíma þurfum við formlegar aðferðir til að vel takist til. En til að fá iðkendur til að nýta sér formlegar aðferðir þurfum við aðgengileg líkana-og skilgreiningarmál ásamt öflugum og skalanlegum sannreyningar-tólum. Við kynnum TeProp, tíma og atburðabundið skilgreiningarmál, sem var hannað til að takast á við tímaháða röðun atburða á eðlilegan hátt, jafnfram því að vera auðvelt í notkun við skilgreiningu eiginleika.

Við kynnum einnig TRSim, samansafn forrita til að vinna með hermanir af Timed Rebeca líkönum. McErlang er notað til að herma líkönin og upplýsingar um atburði skráð í venslagagnagrunn. TeProp eiginleikar eru svo sannreyndir á móti fjöldi hermanna með hjálp fyrirspurnarbiðlara sem þýðir TeProp eiginleika yfir í SQL fyrirspurn. SQL fyrirspurnin er svo keyrð á móti gagnagrunninum og ákvarðar hvort eiginleikinn er sannur fyrir hermunar keyrslurnar í gangagrunninum.

Til að sýna fram á möguleika TeProp og TRSim kynnum við nokkur dæmi og sýnum að við getum fundið villu í líkani sem hafði verið kannað áður.

To my fiancé Aðalheiður Kristín Jónsdóttir

Acknowledgements

Dr. Marjan Sirjani, for convincing me to do a research thesis and introducing me to research in software engineering. This work would never have been done without her great vision and expertise.

Dr. Luca Aceto and Dr. Carolyn L. Talcott for being in my committee and for their helpful comments.

Haukur Kristinsson for all the discussions and collaboration during the last year.

My family for all their love and support.

Högni Eyjólfsson for offering interesting problems to work on when I was in a need for a context switch.

For proofreading this thesis, I thank Guðmundur Narfi Magnússon.

I would also like to thank my fellow students and faculty at Reykjavík University.

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

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In recent years, there has been an explosion in the number of embedded systems we interact with each day. Without even realising it we depend on these systems and even trust them for our lives. These embedded systems range from smart phones, traffic lights, elevators and even cars; and we often refer to them as reactive systems since they react to certain events. A failure in such a system can have catastrophic effects, dissatisfied customers, damaged reputation, financial loss and even loss of lives.

Imagine driving your car on a sunny summer day when you suddenly notice that the brakes are not working and within seconds you hit the car in front of you. As your head hits the steering wheel it is clear that the airbag controller malfunctioned, and then 5 seconds later the airbag inflates doing you no good. During recovery you are told that a software bug caused your accident.

The rise of internet services during the same time also emphasizes the importance of correct systems. Systems capable of handling requests from millions of people tend to be complex, requiring distributed computing and asynchronous communication for the best user experience. History has told us that a failure in even one component of a distributed system can lead to major service disruption, as was the case for Microsoft Azure Services Platform (Microsoft, 2012).

Formal methods are mathematically based techniques for modeling, specification and verification of software and hardware systems. They are meant to help engineers develop reliable systems using mathematical approaches similar to those used in other engineering disciplines. Just as a civil engineer makes precise calculations before constructing a building, software / hardware engineers should build models of their systems before development and analyze the models in order to assess whether they meet the specified requirements.

But in order to get practitioners to use formal methods we need easy to use modeling and specification languages and powerful verification tools. Nowadays most of the systems have timing constraints which are as important as their functionality, so, it is important that our methods cover timing constraints.

We use Timed Rebeca as our modeling language, as it is an actor-based language providing a natural concurrency model, object-based computation and real-time primitives.

For the specification language we looked for an easy to use property language capable of expressing properties of timed event-driven systems.

For the analysis we use simulation, as the de-facto formal verification technique, model checking, tends to suffer from the state explosion problem.

1.1 Contributions

This thesis is an attempt to provide a suitable property language and analysis techniques for Timed Rebeca.

The contributions are:

- A property language for Timed Rebeca
- A tool for storing simulation information in a relational database
- Mapping of the property language to SQL
- A tool for checking properties against simulations stored in a relational database, using the mapping to SQL
- Experimental results

1.2 Overview of the Thesis

The thesis is structured as follows: Chapter 2 introduces the main concepts behind Timed Rebeca and timed property languages. Chapter 3 introduces TeProp, its language definition and semantics. Chapter 4 presents how we carry out simulations and the TRSim tool-kit, its architecture and implementation. In Chapter 5 we analyze four case studies using TeProp and the TRSim tool-kit. Related work is then

discussed in Chapter 6. Finally, Chapter 7 presents conclusions and outlines future work.

Chapter 2

Background

2.1 Actor Model

The actor model is a model of computation originally proposed by Hewitt as an agent-based language (Hewitt, 1972), that was later developed into a concurrent object-based language by Agha (Agha, 1986) and formalized by Talcott et al. (Agha, Mason, Smith, & Talcott, 1997).

In the actor model, *actors* are the universal primitives of concurrent computation: in response to a message that it receives, an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message that it receives. Actors have encapsulated states and behavior, and are capable of creating new actors and redirecting communication links through exchange of actor identities (Sirjani & Jaghoori, 2011).

All the actors in the system run concurrently and the message passing between them is asynchronous.

Different interpretations, dialects and extensions of actor models have been proposed in several domains and are claimed to be the most suitable model of computation for some of the dominating applications, such as multi-core programming and web services (Hewitt, 2007; Aceto et al., 2011).

2.2 Timed Rebeca

Reactive Objects Language, Rebeca, is an actor-based language with formal semantics and model checking tools (Sirjani, Movaghar, Shali, & Boer, 2004). Rebeca was designed to bridge the gap between formal verification approaches and real applications. Its simple, message-driven and object-base computational model and Java like syntax makes it easy to use for software engineers.

A Rebeca model consists of a set of *reactive classes* and the *main* program in which we declare reactive objects, or rebecs, as instances of *reactive classes*. A reactive class has an argument of type integer, which denotes the length of its message queue. The body of the reactive class includes the declaration for its *known rebecs*, variables, and methods (also called message servers). Each method body consists of the declaration of local variables and a sequence of statements, which can be assignments, *if* statements, rebec creation (using the keyword *new*), and method calls. Method calls are sending asynchronous messages to other rebecs (or to self) to invoke the corresponding message server (method). Message passing is fair, and messages addressed to a rebec are stored in its message queue. The computation takes place by taking the message from the front of the message queue and executing the corresponding message server (Sirjani et al., 2004; Aceto et al., 2011).

Timed Rebeca is an extension of Rebeca adding time-related primitives to the language (Aceto et al., 2011). In a Timed Rebeca model, each rebec has its own local clock and message bag instead of a message queue. The timing primitives *delay*, *now*, *deadline* and *after* were added to the syntax. In Figure 2.1 each timing primitive is explained and Figure 2.2 shows the syntax of Timed Rebeca.

All messages that are sent are put in the receiving rebecs message bag along with their time tag (the value of *now* in addition to the argument of *after*, if provided) and deadline tag. The computation then takes place by taking the message with the least time tag and executing the corresponding message server. But before the execution of the corresponding message server the local clock of the receiver rebec is set to the maximum value between its current value and the time tag of the message.

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

Figure 2.1: Timing primitives of Timed Rebeca (Aceto et al., 2011).

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

Figure 2.2: Abstract syntax of Timed Rebeca (Aceto et al., 2011). 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. Brackets [...] indicates that the text within the brackets is optional. Identifiers *C*, *T*, *M*, *v*, *c*, and *r* denote class, type, method, variable, constant, and rebec names, respectively; and *e* denotes an (arithmetic, boolean or nondetermistic choice) expression.

2.3 Timed Property Languages

For the specification of models using formal methods, property languages are used to describe the desired behaviors.

TCTL

Timed CTL, TCTL (Alur, Courcoubetis, & Dill, 1990) is a real-time extension of computational tree logic (Clarke, Emerson, & Sistla, 1986). TCTL is interpreted over a dense time line and contains time-constrained version of the always **G**, eventually **F**, strong until **U**, and weak until **W** operators, which are either existentially **E** or universally **A** quantified over computation paths (Konrad & Cheng, 2005). It was designed for use with finite set of real valued clocks that may be reset. As it was proposed along with the real-time specification formalism time graphs, similar to timed automata (Alur & Dill, 1994).

TCTL has been used as a property language in HyTech (Henzinger, Ho, & Toi, 1997), Kronos (Bozga et al., 1998), and a subset in UPPAAL (Larsen, Pettersson, & Yi, 1997),

MTL

Metric Temporal Logic (Koymans, 1990) is an extension of Linear Temporal Logic (LTL) (Pnueli, 1977) adding relative time and optional real-time constraints to the temporal operators, sometime **F**, always **G**, and strong until **U**. MTL assumes a global clock that progresses at a fixed rate.

MTL has been used as a property language in Temporal Rover (Drusinsky, 2000) and Real-Time Maude (Wirsing, Bauer, & Schroeder, 2010) .

TILCO

TILCO (Mattolini & Nesi, 2001) is a logic language which can be used to specify temporal constraints in either a qualitative or a quantitative way (Bellini, Giotti, Nesi, & Rogai, 2003). A TILCO formula is given with respect to current time and has four basic operators, universal quantification **@**, existential quantification **?**, **until**, and **since**. All working with time intervals that support future and past.

TILCO has been implemented in the theorem prover Isabelle/HOL (Bellini et al., 2003).

Chapter 3

Property Language

In this chapter we introduce our property language, timed event-based property language, TeProp. First we motivate introducing a new language and the new operators of our language.

We set out to find a suitable property language for Timed Rebeca (Aceto et al., 2011), which is a timed actor based modeling language. Since we are working with the actor model (Hewitt, 1972) and its asynchronous message passing, the property language needs to be able to reason about the timing and occurrence of the messages more than the values of some variables inside the actors. This leads us towards an event-based language where an event occurs each time a message is received by an actor, rather than a language working with state propositions.

After looking at a selection of timed property languages in the literature such as Metric Temporal Logic MTL (Koymans, 1990), Timed Computation Tree Logic TCTL (Alur et al., 1990), TILCO (Mattolini & Nesi, 2001) and Timed Propositional Temporal Logic TPTL (Alur & Henzinger, 1994), we found that none of them does naturally fit our event-based properties. These languages can be used to specify your properties, but it would result in long and complicated formulas with high likelihood of mistakes. This is mainly because they are all designed around state proposition where we can say that something lasts for some time. For example, we can say that the value of α will be the same for the next 5 time units. This is not possible in the event based setting where an event only occurs at one time unit and tells us nothing about what will happen in the future, and we cannot say an event will stay the same for x time units.

This leads us to define a new language based on the language we found to be closest to our needs, MTL, along with influence from the property patterns in (Abid, Dal Zilio, & Le Botlan, 2011), (Bellini, Nesi, & Rogai, 2009) and (Konrad & Cheng, 2005).

The reason we found MTL to be closest to our needs instead of for example TCTL (used in the state of the art timed model checker UPPAAL (Larsen et al., 1997)) is the timing model. MTL uses relative intervals while TCTL uses multiple clocks that can be reset. Since we are working with event based systems using the notion of a single global clock the relative intervals are a natural fit.

As for the design of TeProp our goal was to create a language that was easy to use by practitioners, applicable for model checking, simulation and execution and capable of specifying properties about the timed occurrence of events in a natural way. This focus has restricted TeProp compared to MTL, which was in line with our goal to have an easy to use language. At the same time using timed property patterns we show that we can specify a wide range of properties using TeProp.

Below we give a brief explanation of MTL and its syntax and semantics, and the patterns that led us to TeProp.

MTL (Koymans, 1990) is an extension of Linear Temporal Logic (LTL) (Pnueli, 1977) adding optional real-time constraints to the temporal operators.

MTL Syntax (Ouaknine & Worrell, 2008)
Given a set P of atomic propositions, the formulas of MTL are built from P using Boolean connectives, and time-constrained versions of the until operator **U** as follows:

$$\phi ::= p \mid \neg\, \phi \mid \phi \ \wedge \ \phi \mid \phi \ U_I \ \phi$$

where $I \subseteq (0, \infty)$ is an interval over the non-negative reals with endpoints in $\mathbb{N} \cup \{\infty\}$. Further connectives can be defined following standard conventions. In addition to propositions $\top$ (true) and $\bot$ (false), and to disjunction $\vee$, the *constrained eventually* operator $\mathbf{F}_I\ \phi \equiv \top\ \mathbf{U}_I\ \phi$ and the *constrained always* operator $\mathbf{G}_I\ \phi \equiv \neg\ \mathbf{F}_I \neg\ \phi$ are defined.

MTL Semantics (Ouaknine & Worrell, 2008)
Here we show the *pointwise semantics* of MTL, for the interval semantics see (Ouaknine & Worrell, 2008). In the *pointwise semantics*, MTL formulas are interpreted over timed words. Given an alphabet of events Σ, a timed word ρ is a finite or infinite sequence $(\sigma_0\ ,\ \tau_0)(\sigma_1\ ,\ \tau_1)\ .\ .\ .$ where $\sigma_i \in \Sigma$ and $\tau_i \in \mathbb{R}_+$, such that the sequence (τ_i) is strictly increasing and non-Zeno (i.e., it is either finite or it diverges to infinity). The requirement of *non-Zenoness* is closely related to the condition of finite variability in the continuous semantics. It reflects the intuition that a system has only finitely many state changes in bounded time interval. Given a (finite or infinite) timed

word $\rho = (\sigma, \tau)$ over alphabet 2^P and an MTL formula ϕ, the satisfaction relation $\rho, i \models \phi$ (read ρ satisfies ϕ at position i) is defined inductively, with the classical rules for Boolean operators, and with the following rule for the "until" modality:

$\rho, i \models \phi_1 \mathbf{U}_I \phi_2$ iff there exists j such that $i < j < |\rho|$, $\rho, j \models \phi_2$, $\tau_j - \tau_i \in I$, and $\rho, k \models \phi_1$ for all k with $i < k < j$.

While defining the language we looked into the literature of timed property patterns (Abid et al., 2011), (Bellini et al., 2009) and (Konrad & Cheng, 2005) along with papers related to the untimed property languages (Dwyer, Avrunin, & Corbett, 1999) and decided on the following properties as the ones most important for us to be able to express. The first five are from (Koymans, 1990) and the sixth one is from (Bellini et al., 2009). Variations of these patterns also appear in (Abid et al., 2011) and (Konrad & Cheng, 2005) but are mainly focusing on state-based models.

- Maximal distance between events
- Exact distance between events
- Minimal distance between events
- Periodicity of an event
- Bounded response time for an event
- Precedence of an event before another event

Since the standard Until operator in temporal logic is expressing that a state-proposition should hold until something happens and we are only concerned with the order and occurrence of instantaneous events we introduce the Before operator.
For example for stating that: e_1 precedes e_2 in the next 10 time units, we say $e_1 \mathbf{B}[0, 10]\, e_2$, while in MTL this would be $\neg((\neg e_1)\ \mathbf{U}[0, 10]\ e_2) \wedge \mathbf{F}[0, 10]\ e_2$.

3.1 Patterns

Below are the property patterns that influenced TeProp along with textual description, and the MTL and TeProp formulas. You can clearly see how MTL influenced TeProp.

Maximal distance
Maximal distance between an event and its reaction, for example, every e_1 is followed by an e_2 within x time units.
MTL: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0, x]\ e_2)$
TeProp: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0, x]\ e_2)$

Exact distance
Exact distance between events, for example, every e_1 is followed by an e_2 in exactly x time units.
MTL: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[x, x]\ e_2)$
TeProp: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[x, x]\ e_2)$

Minimal distance
Minimal distance between events, for example, two consecutive events e are at least x units apart.
MTL: $\mathbf{G}(e \rightarrow \neg\ \mathbf{F}[0, x]\ e)$
TeProp: $\mathbf{G}(e \rightarrow \neg\ \mathbf{F}[0, x]\ e)$

Periodicity
Periodicity, for example, event e occurs regularly with a period of x time units.
MTL: $\mathbf{F}\ e \wedge \mathbf{G}(e \rightarrow (\mathbf{F}[x, x]\ e \wedge \mathbf{G}[0, x-1] \neg\ e))$
TeProp: $\mathbf{F}\ e \wedge \mathbf{G}(e \rightarrow (\mathbf{F}[x, x]\ e \wedge \neg\ \mathbf{F}[0, x-1]\ e))$

Bounded response
Bounded response, for example, each occurrence of an event e is responded within a maximum number of time units.
MTL: $\exists x\ \mathbf{G}(e_1 \rightarrow \mathbf{F}[0, x]\ e_2)$
TeProp: $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0, x]\ e_2)$ - Note: the tool can be extended such that the user can define a range of values for x.

Precedence
Precedence, for example, within the next x time units, the occurrence of e_1 precedes the occurrence of e_2.
MTL: $\neg((\neg e_1)\ \mathbf{U}[0,x]\ e_2) \wedge \mathbf{F}[0,x]\ e_2$
TeProp: $e_1\ \mathbf{B}[0,x]\ e_2$

Note: Although from the above formulas it seems that TeProp and MTL have minor differences, when we have more complicated formula TeProp can be easier to use in the event based setting. For example, stating that at least once after an occurrence of e_1, within the next 8 time units, e_2 precedes e_3. In MTL it will be $\mathbf{F}(e_1 \rightarrow (\neg((\neg e_2)\ \mathbf{U}[0,8]\ e_3) \wedge \mathbf{F}[0,8]\ e_3 \wedge e_1))$, while in TeProp this would be $\mathbf{F}(e_1 \rightsquigarrow e_2\ \mathbf{B}[0,8]\ e_3)$. In the state space setting MTL is more expressive as not all MTL formulas can be expressed in TeProp, as it was designed for use in the event based setting.

3.2 Syntax

$$\begin{aligned} \phi &::= \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid (\phi) \mid F_I\ e \mid F_I\ (e \rightsquigarrow \phi) \mid G_I\ (e \rightarrow \phi) \mid e\ B_I\ e \\ I &::= \epsilon \mid [\langle Integer \rangle,\ \langle Integer \rangle] \mid [\langle Integer \rangle,\ end] \end{aligned}$$

Figure 3.1: Syntax of TeProp. *e* stands for event and can include additional information that can be used to determine if an matching event has occurred. When used with Timed Rebeca the syntax for an event is *instance.msgsrv(conditions)* where *conditions* is a Boolean formula referring to the parameters of the event.

3.3 Semantics

Informal Description

An event in Timed Rebeca happens when a message is taken from the front of the message queue and executed. Inside each event the following information is stored:

- **Instance name:** the instance name of the rebec called.
- **Message server:** the name of the message server called.
- **Parameters:** the parameters passed to the message server.
- **Time:** the time when the message was taken off the message queue.
- **Sender:** the instance name of the sender rebec.

The property language is designed to reason about the timing and occurrence of these events and has five operators, **G**, **F**, **B**, $\rightsquigarrow$, and $\rightarrow$ where the first three operators work with time intervals. The interval defines when the formula should hold in respect to the current time and consists of two non-negative integers inside brackets [from, to]. The current time instant is represented by 0, positive integer represents the future and the symbol *end* can be used to refer to the last time of any event in the system, this is useful to be able to check if something holds from some point in time till the end. Omitting an interval for an operator is the same as using the interval $[0, end]$.

Event: *instance.msgsrv*(*conditions*). An event is selected with its instance name, message server and time. Optional conditions are given as a Boolean formula that can refer to the event's parameters as well as to the instance name of the sender using the keyword *sender*. In the following when we say an event matching e, we mean an event matching the instance name, message server and conditions of e.

Finally in combination with **Event**[1]: $\mathbf{F}[i_1, i_2]\ e$. An event matching e will happen somewhere on the interval $[i_1, i_2]$.

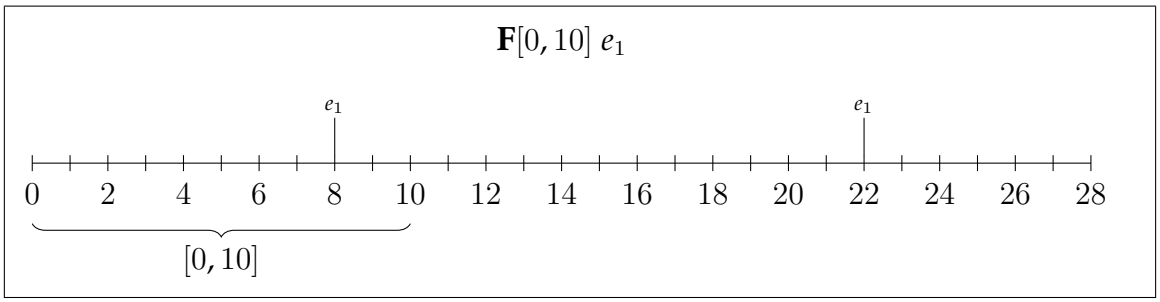


Figure 3.2: Visualization of **Finally**: The formula is satisfied by the first event on the time line, but not by the second event.

Examples:

Formula	Description
F X.call()	An event with instance name X and message server *call* will happen at some point.
F$[0, 10]$ X.call()	An event with instance name X and message server *call* will happen at some point between 0 and 10 time units.
F$[5, 20]$ X.call(n==7)	An event with instance name X and message server *call* and the message parameter n with value 7 will happen at some point between 5 and 20 time units.

Before: $e_1\ \mathbf{B}[i_1, i_2]\ e_2$. Within the interval $[i_1, i_2]$ an event matching e_1 happens at least once before an event matching e_2.

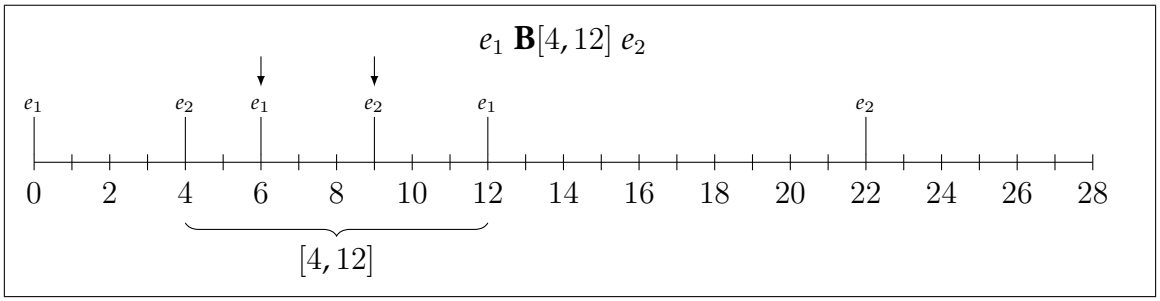


Figure 3.3: Visualization of **Before**: The formula is satisfied using the events the arrows point to but not using the other events.

[1] We allow **Finally** with **Event** since it's useful to be able to say that something will or will not happen, we do however not allow **Globally** with **Event** since saying that event happens all the time is of limited use and can be formulated as a periodic formula using **Globally** with **Implies**.

Example:

Formula	Description
X.call() **B**$[0, 5]$ Y.call()	Within 0 and 5 time units an event with instance name X and message server *call* will happen before an event with instance name Y and message server *call*.

The formula implies, $e \rightarrow \phi$, is used in combination with globally **G**$[i_1, i_2]$. For every event matching e the formula ϕ is evaluated using the timing of the event e as its current time. The formula is also satisfied at time instance where there is no event that matches e, so the formula is satisfied even if no event matches e on the interval $[i_1, i_2]$.

Globally with **Implies**: **G**$[i_1, i_2](e \rightarrow \phi)$. For every occurrence of an event matching e on the interval $[i_1, i_2]$ the evaluation of the formula ϕ using the timing of the event e as its current time must be satisfied. The formula is also satisfied if no event matches e on the interval $[i_1, i_2]$.

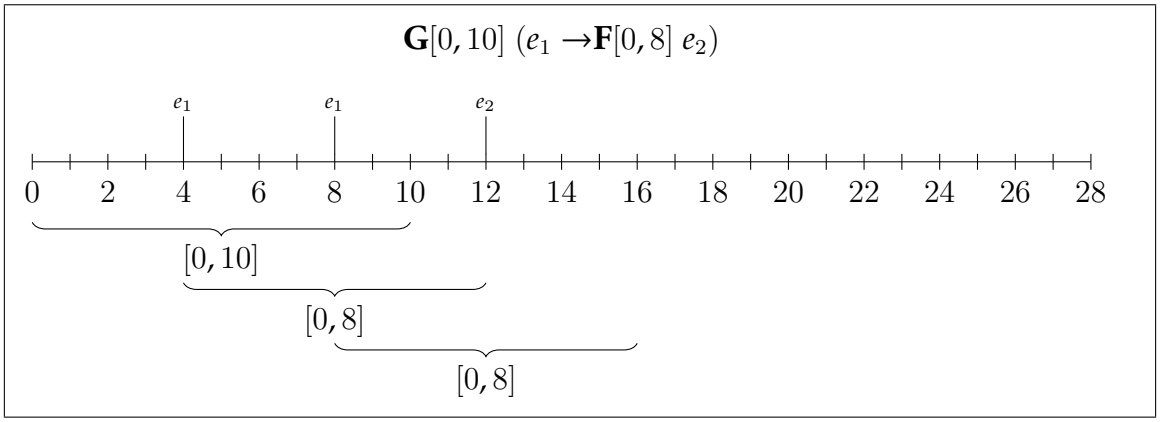


Figure 3.4: Visualization of **Globally** with **Implies**: The formula is satisfied by the events shown in the figure above, but if the event e_1 also happens at time zero then the formula will no longer be satisfied.

Examples:

Formula	Description
$\mathbf{G}(X.call() \rightarrow \mathbf{F}[0,10]\ Y.call())$	Every occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* within 10 time units of event X.
$\mathbf{G}(X.call() \rightarrow Y.call()\ \mathbf{B}[0,9]\ Z.call())$	Every occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* before an event with instance name Z and message server *call* , both within 9 time units of event X.

The formula leads-to, $e \rightsquigarrow \phi$, is used in combination with finally $\mathbf{F}[i_1, i_2]$. For every event matching e the formula ϕ is evaluated using the timing of the event e as its current time.

Finally with **Leads-to**: $\mathbf{F}[i_1, i_2](e \rightsquigarrow \phi)$. At least for one occurrence of an event matching e on the interval $[i_1, i_2]$ the evaluation of the formula ϕ using the timing of the event e as its current time must be satisfied.

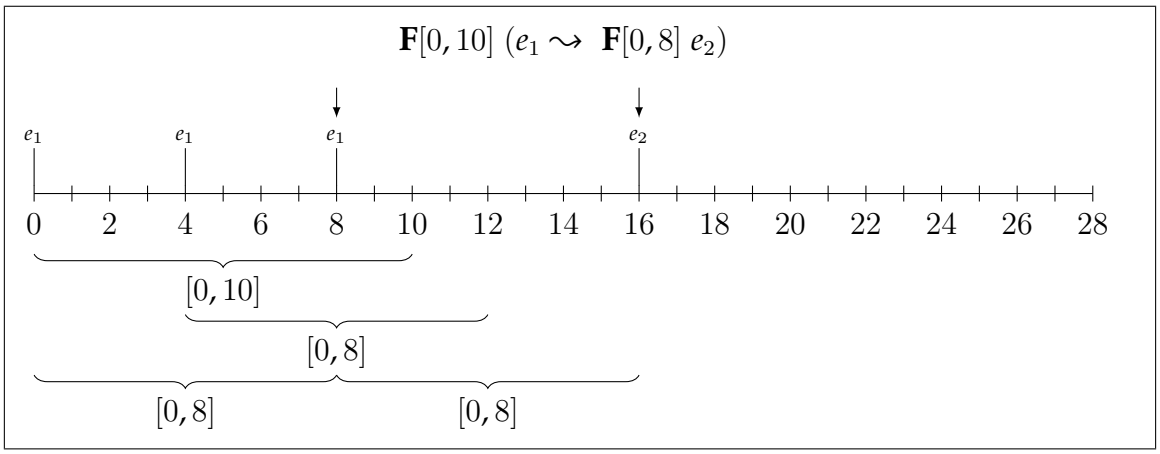


Figure 3.5: Visualization of **Finally** with **Leads-to**: The formula is satisfied using the events the arrows point to, the other events have no effect.

Examples:

Formula	Description
$\mathbf{F}(X.call() \rightsquigarrow \mathbf{F}[0,10]\ Y.call())$	At least one occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* within 10 time units of event X.
$\mathbf{F}(X.call() \rightsquigarrow Y.call()\ \mathbf{B}[0,9]\ Z.call())$	At least one occurrence of an event with instance name X and message server *call* is followed by an event with instance name Y and message server *call* before an event with instance name Z and message server *call*, both within 9 time units of event X.

Formal Description

Given the alphabet Σ of all events of a model. Let π be a sequence $(e_0,\tau_0),(e_1,\tau_1),\ldots$ of timed events where $e_i \in \Sigma$ and τ_i is increasing over time and in $\mathbb{N}$. Whether π, i satisfies an formula at position i is defined by the satisfaction relation $\models$ as follows:

1. $\pi, i \models e$ iff e is the same event as e_i
2. $\pi, i \models \neg\phi$ iff $\pi, i \not\models \phi$
3. $\pi, i \models \phi_1 \wedge \phi_2$ iff $\pi, i \models \phi_1$ and $\pi, i \models \phi_2$
4. $\pi, i \models \phi_1 \vee \phi_2$ iff $\pi, i \models \phi_1$ or $\pi, i \models \phi_2$
5. $\pi, i \models \mathbf{F}_I\, e$ iff $\pi, j \models e$ for some j with $i < j < |\pi|$ and $\tau_j - \tau_i \in I$
6. $\pi, i \models \mathbf{G}_I\, (e \rightarrow \phi)$ iff $\pi, j \not\models e$ or $\pi, j \models \phi$ for all j with $i < j < |\pi|$ and $\tau_j - \tau_i \in I$
7. $\pi, i \models \mathbf{F}_I\, (e \rightsquigarrow \phi)$ iff $\pi, j \models e$ and $\pi, j \models \phi$ for some j with $i < j < |\pi|$ and $\tau_j - \tau_i \in I$
8. $\pi, i \models e_1\ \mathbf{B}_I\ e_2$ iff $\pi, j \models e_2$ for some j with $i < j < |\pi|$, $\tau_j - \tau_i \in I$ and $\pi, k \models e_1$ for some k with $i < k < j$ and $\tau_k - \tau_i \in I$

Figure 3.6: Formal description of TeProp.

3.4 Mapping to SQL

After designing TeProp we looked into ways of integrate TeProp into our existing Timed Rebeca tools rather than developing a new tool from scratch.

We soon realized that storing the results from our Timed Rebeca simulator in a SQL database and create a mapping from TeProp to SQL would be a convenient way of accomplishing our goal. As by using a database for storing the simulations outputs we get an established way of storing the simulations in an durable, organized and structured way as well as taking advantage of the industry standard technology for data analysis. The alternative would have been writing the simulation outputs to a file and creating an analysis program for checking TeProp properties, which would be more work and not necessary give better results.

Another advantage of using a database for storing the simulations, is the possibility for the modeler to write his own SQL queries in addition to checking TeProp properties. Example of such a query could be checking if every message server was called at least once.

3.4.1 Database Design

But before we start explaining the mapping we first need to introduce the database design the mapping is based on. After reviewing the information we could extract from a model simulation run in relation to TeProp we decided on storing the following information in the database.

- The order of the events in the simulation
- The time of the event
- The instance name of the rebec called
- The message server called
- The instance name of the sender rebec
- The parameters passed to the message server

To store the information in a structural way to the database as well as giving the modeler and possible other application access to the simulation information in a convenient form, we decided that for each rebec instance we create a separate table per message server. Each table then has the following columns:

Column name	Column data type	Description
ID	big integer	Global counter over the simulation indication the order of the event within the simulation
Time	timestamp	The time of the event within the simulation
Sender	varchar(50)	The instance name of the sender rebec

Then for message servers with parameters we also create a separated column for each parameter passed to the server with a data type matching the Timed Rebeca type. As for indexing of the tables an index is set on the ID and Time columns, as both columns are essential in retrieving the relevant events from the database in the SQL mapping.

Since each TeProp formula is either relative to another event or the beginning of the simulation it was essential for us to be able to retrieve the starting time of the simulation, for this purpose we created a database view for each simulation that selects the minimum time from all the tables described above. The PrepareDB tool in Subsection 4.2.2 on page 34 creates both the tables and the database view from a Timed Rebeca model.

3.4.2 The Mapping

We do not provide a formal proof of the mapping but we explain how the mapping is similar to the model checking algorithms; which in an essence are either finding a path in the model's state space, where a property is not satisfied in case of a globally defined property, or a path where the property is satisfied in the case of a finally defined property. If such a path is not found for the globally defined property or found for the finally defined property the property is satisfied. Our mapping does the same but instead of finding a path in a state space we only check a part or all of a simulation trace. Please note that since we are only checking a set of execution paths, and not the whole state space, we can not claim that we prove a property even though it is satisfied for all our simulation traces. But if a property is not satisfied for one simulation trace, then the simulation trace is a so called counter example that proves that the property is not satisfied for all runs.

To give a top-down view of the mapping we start with the base SQL query that is the same for all properties and then go into how we retrieve information about events and the recursive mapping of valid TeProp property expressions. The purpose of the

base query is providing a starting point for the *time* in the recursive mapping. It also provides the result of the satisfiability of the property over a simulation run in a uniform way. The base query is as follows:

- select 'satisfied' from [database view storing the starting time of the simulation] where ([the outcome of the mapping of a TeProp property to SQL])

The way the mapping works is that if the result of the SQL statement for a TeProp property is satisfied then one record with the text "satisfied" is returned. We therefore map each valid TeProp property expression to an SQL statement that returns whether it is satisfied or not.

When a TeProp formula is nested within another TeProp formula its *current time* is set to the *time* of the associated event in the parent formula. We call the associated event of the outer formula the parent event. Example of a nested formula is:

- $\mathbf{F}[0, 10](e_1 \rightsquigarrow \mathbf{F}[0, 5]e_2)$

In the example we have the parent formula $\mathbf{F}[0, 10](e_1 \rightsquigarrow \phi)$, the parent event e_1 and the inner formula $\mathbf{F}[0, 5]e_2$. When the nested formula is evaluated the inner formula is evaluated for each parent event e_1 found on the interval [0, 10] and the *current time* for the inner formula is set to the *time* of the parent event. So, the time interval [0,5] of the inner formula is relative to the occurrence time of the parent event e_1 which is now the *current time*.

In the mapping we incorporate this behavior by passing a reference to the parent formula to each inner formula, such that each formula knows its *current time*. The starting *time* of the simulation is then passed as the *current* to the outermost formula.

But before we start explaining the mapping of the TeProp property expressions to SQL we first need to determine what information we need to retrieve from the database for the mapping:

- Retrieve all occurrences of event e_2 that happened after e_1 at a specific time interval$[i_1, i_2]$ relative to e_1

This is essential in order to reason about the occurrence of events, and can be set forward in an abstract SQL query as:

- select [the ID of the event e_2] from [the table storing events e_2] where [the ID of event e_2] > [the ID of event e_1] and [the Time of event e_2] is between [the Time of e_1 + i_1] and [the Time of e_1 + i_2]

The non-abstract version is shown in Table 3.1 for $e[i_1, i_2]$, note that the event expression $e[i_1, end]$ is just a special case of $e[i_1, i_2]$ when there is no upper limit on the interval.

Now that we are able to retrieve from the database whether an event occurred in relation to another event we can define the mapping for the TeProp property expressions[2]:

For **Finally** in combination with **Event**, $\mathbf{F}[i_1, i_2]\ e$, we check if at least one occurrence of event e happens within the interval $[i_1, i_2]$, relative to the formula's *current time*. Here we pass the SQL statement for retrieving all occurrences of event e relative to the *parent* event on interval $[i_1, i_2]$ to the SQL statement **exists**; that is satisfied if the SQL statement passed to it returns at least one row.

For **Before**, $e_1\ \mathbf{B}[i_1, i_2]\ e_2$, we check if at least one occurrence of event e_1 happens within the interval $[i_1, i_2]$, relative to the formula's *current time*, that is then followed by an event e_2 within the rest of the same interval. Here the mapping is similar to the mapping for **Finally** but in addition we have an extra check that event e_2 follows somewhere between e_1 and the end of the interval.

For **Finally** with **Leads-to**, $\mathbf{F}[i_1, i_2](e \rightsquigarrow \phi)$, we check if at least one occurrence of e exists in the interval $[i_1, i_2]$, relative to the formula's *current time*, where the formula ϕ is satisfied using the timing of event e as its *current time*. Here we use the **exist** statement to check if there is an event e within the interval $[i_1, i_2]$, relative to the formula's *current time*, and where the result of the SQL mapping of ϕ is satisfied with e as its parent.

For **Globally** with **Implies**, $\mathbf{G}[i_1, i_2](e \rightarrow \phi)$, we check that no occurrence of e exists in the interval $[i_1, i_2]$, relative to the formula's *current time*, where the formula ϕ is not satisfied using the timing of event e as its *current time*. Here we use the **not exits** statement to check that no event e exists within the interval $[i_1, i_2]$, relative to the formula's *current time*, and where the result of the SQL mapping of ϕ is not satisfied with e as its parent.

[2] The full non-abstract version of the whole mapping is provided in Table 3.1

TeProp		SQL
$\neg\phi$	$\rightarrow$	**not** (ϕ)
$\phi_1 \wedge \phi_2$	$\rightarrow$	(ϕ_1) **and** (ϕ_2)
$\phi_1 \vee \phi_2$	$\rightarrow$	(ϕ_1) **or** (ϕ_2)
$e[i_1, end]$	$\rightarrow$	**select** alias_e.id **from** event_e alias_e **where** alias_e.id > alias_{parent}.id **and** alias_e.time >= alias_{parent}.time + **interval** i_1 **second**
$e[i_1, i_2]$	$\rightarrow$	**select** alias_e.id **from** event_e alias_e **where** alias_e.id > alias_{parent}.id **and** alias_e.time **between** alias_{parent}.time + **interval** i_1 **second** **and** alias_{parent}.time + **interval** i_2 **second**
$\mathbf{F}[i_1, i_2]\ e$	$\rightarrow$	**exists**$(e[i_1, i_2])$
$\mathbf{F}[i_1, i_2]\ (e \rightsquigarrow \phi)$	$\rightarrow$	**exists**$((e[i_1, i_2])$ **and** $(\phi))$
$\mathbf{G}[i_1, i_2]\ (e \rightarrow \phi)$	$\rightarrow$	**not exists**$((e[i_1, i_2])$ **and not**$(\phi))$
$e_1\ \mathbf{B}[i_1, i_2]\ e_2$	$\rightarrow$	**exists**$((e_1[i_1, i_2])$ **and** **exists**(**select** alias_{e_2}.id **from** event_{e_2} alias_{e_2} **where** alias_{e_2}.id > alias_{parent}.id **and** alias_{e_2}.time **between** alias_{parent}.time **and** $\text{alias}_{grandparent}$.time + **interval** i_2 **second**))

Table 3.1: Mapping from TeProp to SQL. The final SQL for the formula ϕ is the result of the mapping, shown in this table, wrapped inside the query "**select** 'satisfied' **from** *base* alias_{base} **where** (ϕ)" where *base* is a database view that returns the starting time of the simulation. The table aliases are used for distinguishing between result sets in case of multiple selects on the same table, the mapping tool is responsible for assigning unique table alias to each select statement.

3.5 SQL Examples

To give a clear idea how the SQL queries look like we have here below some TeProp formulas and their SQL counterpart. The formulas are for the simple communication protocol model that you will see later in the thesis in Listings 5.1 on page 41.

```
select 'satisfied' from "base" t0_0 where (
  not exists(
    select t1_0.ID from "senderAgent_start" t1_0 where t1_0.ID > t0_0.ID and
      t1_0.time >= t0_0.time and not (exists(
        select t1_1.ID from "receiverAgent_send" t1_1 where t1_1.ID > t1_0.ID and
          t1_1.time between t1_0.time and t1_0.time + interval '10' second
      ))
  )
)
```

Listing 3.1: SQL for $\mathbf{G}(\text{senderAgent.start}() \rightarrow \mathbf{F}[0,10]\text{receiverAgent.send}())$

```
select 'satisfied' from "base" t0_0 where (
  exists(
    select t1_0.ID from "senderAgent_start" t1_0 where t1_0.ID > t0_0.ID and
      t1_0.time >= t0_0.time and (exists(
        select t1_1.ID from "receiverAgent_send" t1_1 where t1_1.ID > t1_0.ID and
          t1_1.time between t1_0.time and t1_0.time + interval '8' second
      ))
  )
)
```

Listing 3.2: SQL for $\mathbf{F}(\text{senderAgent.start}() \rightsquigarrow \mathbf{F}[0,8]\text{receiverAgent.send}())$

```
select 'satisfied' from "base" t0_0 where (
  exists(
    select t1_0.ID from "senderAgent_ack" t1_0 where
      t1_0.ID > t0_0.ID and t1_0.time >= t0_0.time
  )
)
```

Listing 3.3: SQL for $\mathbf{F}$ receiverAgent.ack()

```
select 'satisfied' from "base" t0_0 where (
  exists(
    select t1_0.ID from "senderAgent_start" t1_0 where
        t1_0.ID > t0_0.ID and t1_0.time between t0_0.time and
        t0_0.time + interval '5' second and exists(
          select t1_1.ID from "senderAgent_ack" t1_1 where
          t1_1.ID > t1_0.ID and t1_1.time between t1_0.time and
          t0_0.time + interval '5'
        )
  )
)
```

Listing 3.4: SQL for senderAgent.start() $\mathbf{B}[0, 5]$senderAgent.ack()

Chapter 4

Simulation and the TRSim Tool-set

In this chapter we discuss how Timed Rebeca models are simulated and introduce the Timed Rebeca Simulation, TRSim, tool-set. In Section 4.1, we discuss simulation of Timed Rebeca models and important matters to keep in mind when performing the simulations. In Section 4.2, we discuss the architecture of TRSim.

4.1 Simulation of Timed Rebeca Models

For simulating Timed Rebeca models we use McErlang (Fredlund & Svensson, 2007), a model checker for Erlang, that until recently supported only simulation of timed Erlang programs. Meaning that the simulation just executes the Erlang program while allowing the use of hand-coded runtime monitors capable of stopping the simulation. This means that each simulation takes as much time as the real execution, since during the simulation, time is passing by according to the delays in the model. For the sake of clarity we call this kind of simulation execution.

A new version of McErlang supports discrete-time semantics (Earle & Fredlund, 2012), allowing us to simulate timed models without requiring the time to pass based on the delays in the model. Here we respect the causality and ordering of the events. This simulation has the advantage of being fast in comparison with execution by removing the time delays, and also more precise since we can align the passing of time to the semantic of Timed Rebeca and therefore remove the time drift that occurs during execution. For the rest of the thesis we refer to this variant as simulation.

Before we can start simulating our Timed Rebeca model, we have to create SQL database tables using the TRSim PrepareDB tool (Subsection 4.2.2) and translate the

model to an Erlang program. For mapping the Timed Rebeca model to Erlang we use the mapping given in (Aceto et al., 2011), adapted for the discrete-time version of McErlang. The resulting Erlang program can then be simulated using McErlang. During the simulation, information about each message server call, along with its parameters, sender, and time the message was executed is sent to the TRSim Logger (Subsection 4.2.3). TRSim Logger stores the information in the SQL database tables, created by the TRSim PrepareDB tool, for future analysis.

4.1.1 When to Stop the Simulation

For credible analysis of reactive systems using simulation data, the importance of choosing the right place to stop the simulation is paramount. Since we are reasoning about the timing and occurrence of the events, we are unable to distinguish between the case of an event missing its deadline right before the end of simulation and a simulation that ended right before the occurrence of the event. See an example in Figure 4.1.

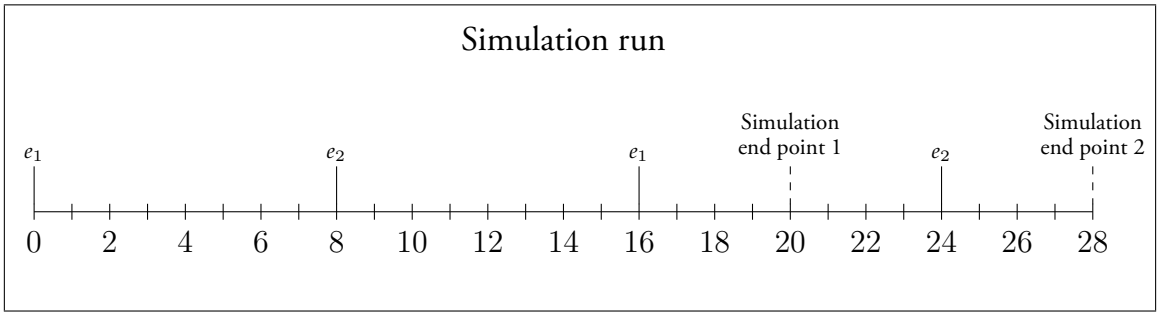


Figure 4.1: Example of a simulation run where we get different results for the property $\mathbf{G}(e_1 \rightarrow \mathbf{F}[0,8]\ e_2)$ depending on where we stop the simulation. If we stop the simulation at time unit 20 we get that the property is not satisfied, as we have no knowledge whether e_2 will occur within 8 time units of the last e_1. But if we stop the simulation at time unit 28 we get that the property is satisfied, as every occurrence of e_1 is followed by e_2 within 8 time units.

But the best place to stop the simulation is not always clear and depends heavily on the model and the properties the modeler intends to check. Our suggestion is to use any of the following ways to decide where to stop the simulation:

1. Changing the model such that it stops after few iterations. This is an ideal solution for reactive systems that have a non terminating behavior, this can be done with a local counter inside a reactive class. See example in Listing 4.1.

2. Using the time limit option in McErlang that allows specifying how long the simulation should be in seconds, along with the filtering option of the TRSim

Logger, that allows you to specify one event as an end mark for the simulation. So, we collect the data up to the time limit and then remove the data that is collected after the last occurrence of the end mark event. This solution is ideal for reactive systems where one event shows you that simulation iteration is finished.

3. Using the time limit option in McErlang that allows specifying how long the simulation should be in seconds, along with executing a hand-coded SQL query against the database to remove the unwanted events. This is the most complicated solution and only intended in situations where the other two are not applicable.

For the simulations in this thesis we used the first two options.

Note: Even though the end event of the simulation is known it is still possible to run into situations where the timing of the last occurrence of an event is crucial. A good example is when we want to check for a periodic behavior in a simulation. Since the simulation is finite we cannot say that every event *e* is followed by *e*, as that is not true for the last occurrence of *e*. In this case we want to specify that all *e* except the last one will be followed by *e*. For this reason we added to our Query Tool the option to view the latest occurrence of any event. We can then specify that all *e* from time 0 up to the time unit before the last occurrence of *e* is followed by *e*.

```
1  reactiveclass Sensor(3) {
2    knownrebecs { Computer computer; }
3
4    statevars { int counter; }
5
6    msgsrv initial() { self.start(); }
7
8    msgsrv start() {
9         int data = ?(1,2,3,4);
10        computer.send(data);
11        counter = counter + 1;
12        if(counter <= 100) {
13              self.start() after(8);
14        }
15 }
```

Listing 4.1: Example of a reactive class modified to stop after 100 iterations (option 1).

4.1.2 The Right Number of Simulation Runs

No matter how simple our models are, their behavior can be complex as a result of concurrency and non-determinism. It is therefore important to run multiple simula-

tions to try to cover all the behaviors of the model, but deciding on the right number of simulation runs that is needed to cover all the cases for all the models is a research project on its own. Therefore we do not offer a simple formula for telling how many simulations should be run. Instead we introduce a way for the modeler to define his own method for deciding whether sufficient numbers of simulations have been run, using a hand-coded SQL function that is called after each simulation. An example of such a function is given in Listing 4.2.

```
1  create function EnoughSimulations()
2  returns boolean
3  as $$
4  declare
5      table_row record;
6      tmp_row   record;
7      counter   int;
8  begin
9      counter := 0;
10     for table_row in
11         select table_name
12         from  information_schema.tables
13         where table_name like '%senderobj_ack'
14     loop
15         execute 'select count(*) as c from ' ||
16                quote_ident(table_row.table_name) into tmp_row;
17         if(tmp_row.c > 0)
18                then counter := counter + 1;
19         end if;
20     end loop;
21     return (counter > 10);
22 end;
23 $$ language plpgsql;
```

Listing 4.2: Example of a SQL function for deciding when to stop simulating: This function stops the simulation process when 10 simulation runs include at least one event with instance name *senderobj* and message server *ack*. The function is written in the SQL Procedural Language used by the PostgreSQL (PostgreSQL, 2012) database.

We do not require the modeler to write such a function and he is free to decide upon the number of simulations based on his expertise, but keep in mind that in most cases you want as many simulations as possible within your time frame.

4.1.3 Models With Zeno Behavior

When simulating using the discrete time semantic of McErlang, the modeler must be aware of the Zeno behavior (Lynch, 1996). That is an infinite number of messages sent at the same time unit, without any after or delay, which in turn shall cause the

simulation to stop. This can happen if the model includes, for example, a reactive class calling itself without a delay.

Our TRSim Logger can be configured to help with the detection of the Zeno behavior within a model. When starting the logger the modeler specifies a limit of events allowed to occur on the same time unit, if the number of events goes above this limit a warning is shown to the user, who can then stop the simulation and check his model.

4.2 TRSim Architecture

We implemented a set of tools to support the analysis of Timed Rebeca simulations using our property language TeProp. We built upon the already established simulation method of Timed Rebeca models by mapping them to Erlang and then use McErlang for simulation. The mapping and a tool, timedreb2erl, for the translation was introduced in (Aceto et al., 2011).

The TRSim tool-kit includes three separate programs (PrepareDB, Logger and a Query Tool) in addition to a modified version of timedreb2erl and a PostgreSQL database to store the events of the simulation. We decided on using a SQL database for storing the simulation events after looking at viable options for checking TeProp properties over the simulation events. So, we developed a mapping from TeProp to SQL and we automated our mapping by implementing it as a part of our Query Tool. As for the database system itself we chose PostgreSQL (PostgreSQL, 2012) as it is a powerful open source database system available for multiple operating systems with long history and good support for subqueries which is essential for the mapping of TeProp to SQL.

The reason for creating a set of tools instead of just one program is to provide the flexibility to run the simulations on a server. Multiple simulations can take quite some time, while running the query tool on a workstation. It also makes it easy for other programs to integrate with our tools without having to use a special interface, since they can call them with command line parameters. It also makes it possible for the modeler to create his own program to control the simulation process, including the environment variables in each simulation and the number of simulations.

Figure 4.2 shows the overall architecture of the TRSim tool-kit, including the life cycle from modeling to specification.

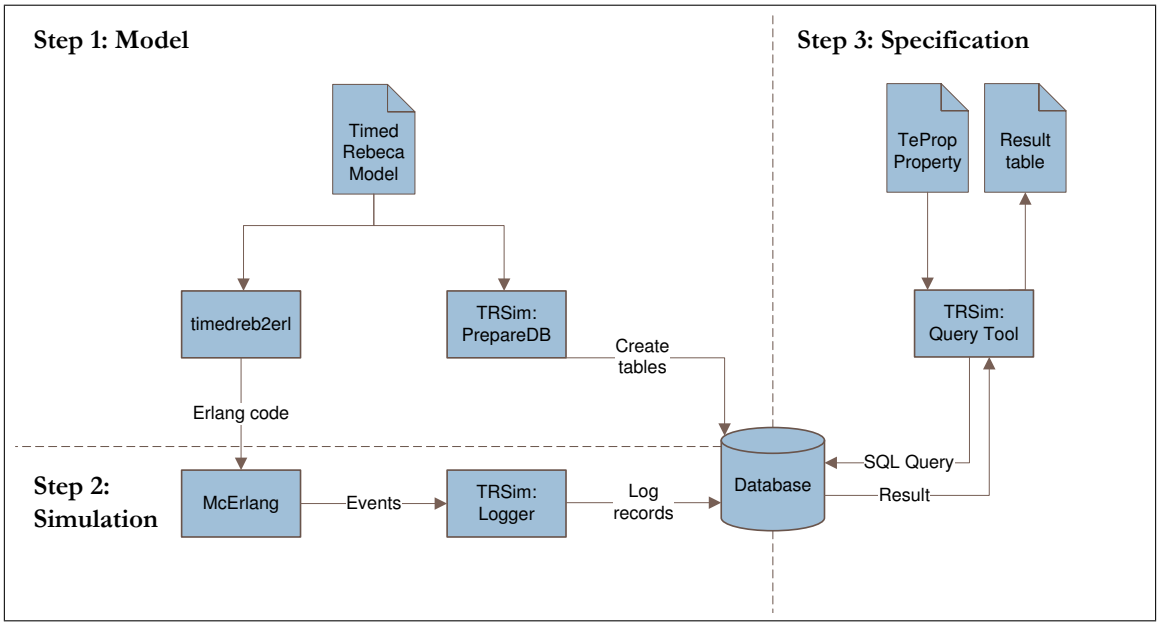


Figure 4.2: Architectural overview of the TRSim tool-set.

In the following Subsections we describe the individual programs in the TRSim tool-set.

4.2.1 Timedreb2erl

Timedreb2erl is the mapping tool introduced in (Aceto et al., 2011) for transforming Timed Rebeca models to Erlang, supporting code generation for both the Erlang and McErlang runtime. Here we use a version where the McErlang output has been adapted (Kristinsson, 2012) for the new discrete timed semantics. We made small change to the mapping such that the resulting Erlang code sends a message to the TRSim Logger each time a message server is executed including the instance name of the rebec, the message server name, the parameters of the message, the sender of the message and the local time of the rebec. The environment variables of the simulations are also sent to the logger in the beginning of the simulation.

4.2.2 PrepareDB

PrepareDB is a command line Java program that creates the database tables required for the Logger based on a Timed Rebeca model. A separate table is created for every message server of each rebec instance using the following naming rule: SimulationID, InstanceName and MessageServerName separated by an underscore. Each table then includes the following columns:

Column name	Column data type	Description
ID	bigint	The global id of the event
Time	timestamp	The time of the event
Sender	varchar(50)	The sender of the event

In addition a column is created for each parameter using the same name and data type equivalent to its Timed Rebeca data type. The tool also creates a table for storing information about the simulation such as the environment variables. And a database view that selects the lowest time in any of the tables created, used to find at which time the simulation began.

4.2.3 Logger

Logger is a command line Java program that logs all messages it receives to the database. It uses the Erlang Jinterface to communicate with Erlang, allowing us to send and receive messages as any other actor in the Erlang environment. The reason we chose to implement the logger in Java rather than Erlang is the database support provided in Java.

Due to the high number of messages the logger receives during the simulation we are unable to keep up by inserting each record in the database at the time it arrives. We therefor build up a data file on disk for each table in a compatible format for bulk insert, then when the simulation run is over each file is bulk inserted into the database.

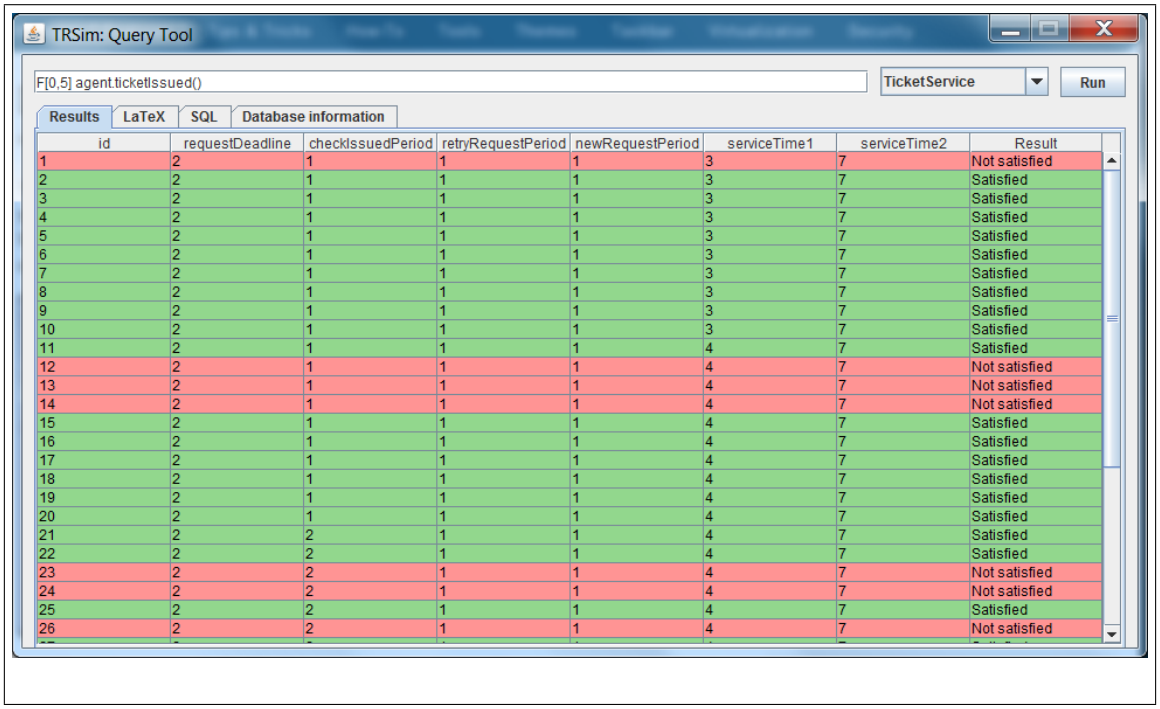
The logger is able to detect possible Zeno behavior (Subsection 4.1.3) by counting the number of messages received in a row tagged with the same time. If this number is higher than the threshold passed to the program a warning of possible Zeno behavior is printed out on the screen.

4.2.4 Query Tool

The Query Tool is a Java program with graphical interface for the analysis of simulations stored in a database using the TeProp property language. The modeler can select the database he wants to work with, write a TeProp property and check its satisfiability over all the simulations in the database. The results are displayed in a grid showing the number of the simulation runs, its environment parameters and whether the property is satisfied or not for that simulation. The grid can be exported as a LaTeX table. The

modeler can also view information about the simulations in the database such as the last time unit of each event type.

For checking the TeProp property against the simulation runs in the database we use the mapping of TeProp to SQL from Section 3.4. A SQL query is created for each simulation and executed on the database, the result of the query determines whether the property is satisfied or not for that simulation.



id	requestDeadline	checkIssuedPeriod	retryRequestPeriod	newRequestPeriod	serviceTime1	serviceTime2	Result
1	2	1	1	1	3	7	Not satisfied
2	2	1	1	1	3	7	Satisfied
3	2	1	1	1	3	7	Satisfied
4	2	1	1	1	3	7	Satisfied
5	2	1	1	1	3	7	Satisfied
6	2	1	1	1	3	7	Satisfied
7	2	1	1	1	3	7	Satisfied
8	2	1	1	1	3	7	Satisfied
9	2	1	1	1	3	7	Satisfied
10	2	1	1	1	3	7	Satisfied
11	2	1	1	1	4	7	Satisfied
12	2	1	1	1	4	7	Not satisfied
13	2	1	1	1	4	7	Not satisfied
14	2	1	1	1	4	7	Not satisfied
15	2	1	1	1	4	7	Satisfied
16	2	1	1	1	4	7	Satisfied
17	2	1	1	1	4	7	Satisfied
18	2	1	1	1	4	7	Satisfied
19	2	1	1	1	4	7	Satisfied
20	2	1	1	1	4	7	Satisfied
21	2	2	1	1	4	7	Satisfied
22	2	2	1	1	4	7	Satisfied
23	2	2	1	1	4	7	Not satisfied
24	2	2	1	1	4	7	Not satisfied
25	2	2	1	1	4	7	Satisfied
26	2	2	1	1	4	7	Not satisfied

Figure 4.3: Screen capture of the Query Tool interface: here we are checking the TeProp property "**F**$[0,5]$ agent.ticketIssued()" over simulations of the ticket service model in Listing 5.2 on page 46.

Chapter 5

Experimental Results

Parallel to this project (and as a positive response to our suggestion), McErlang has been extended (Earle & Fredlund, 2012) to support time. So, we can now use simulation. In the execution the exact number of time units mentioned in the parameters of after and delay must pass in real-time, but in simulation we can progress time respecting the causality and ordering of the events. Clearly, it makes running large number of simulations much faster and more convenient than having the same number of executions, and hence we can derive more precise conclusions by having larger number of runs.

In this chapter we analyze four case studies using TeProp and the TRSim tool-kit. For each study we ran multiple simulations and checked interesting timing properties. In Section 5.1 we describe the experimental setup used for the simulations. In Section 5.2 we show the time benefit of simulation over execution. In Section 5.3 we describe the graphical notation we use to give an abstracted view of the scheduling of events in a model. In Sections 5.4 to 5.7 we introduce the case studies.

5.1 Experimental Setup

The experiments were executed on a server with Intel(R) Xeon(TM) Quad CPU 2.66GHz processor and 32GB of RAM. The machine ran 64-bit Ubuntu 10.04.4 LTS (Lucid Lynx) with Linux kernel 2.6.32-38-server, PostgreSQL 9.1.3 and Erlang R13B03.

5.2 The time benefit of simulation

To show the time benefit of using the simulation over the execution we did 10 runs of the ticket service model in Listing 5.2, using the environment variables for setting 1 in Table 5.3, for 1 minute. In Table 5.1 we show the time required for the same runs using the execution. In (Aceto et al., 2011) execution was used as McErlang was not yet supporting time.

	Simulation	Execution
Simulation 1	00:01:00	13:06:47
Simulation 2	00:01:00	13:04:13
Simulation 3	00:01:00	11:41:52
Simulation 4	00:01:00	13:05:52
Simulation 5	00:01:00	11:51:19
Simulation 6	00:01:00	12:51:20
Simulation 7	00:01:00	11:53:03
Simulation 8	00:01:00	12:46:09
Simulation 9	00:01:00	12:54:35
Simulation 10	00:01:00	13:00:58

Table 5.1: Comparison of the time required for the same runs using simulation and execution. The time format is hours:minutes:seconds.

As you can see from the results the time saving of using the simulation is significant. But keep in mind that the speed up is never the same for all models and environment variables since it is relative to the delays and afters used.

5.3 Event graph

Before presenting the case studies, we first explain the notation we use to show an abstracted view of the scheduling of events within a model. Event graph is a graphical notation used to represent graphically discrete-event simulation models, with a single type of node and two types of edges. The nodes represent events in a system and edges correspond to the scheduling of other events (Buss, 1996). Each edge can be associated with a Boolean condition and/or a time delay. Jagged incoming edges denote an initial event. Here, we use an alternative notation introduced in (Aceto et al., 2011) where we omit the time delays and replace the Boolean conditions with conditional edges, drawn as thick arrows. Label is also placed next to each node to show in which reactive class the event occurs. Figure 5.1 shows an example of an event graph using our

notation where event e_2 is scheduled by e_1, that then schedules e_3 if the condition is fulfilled.

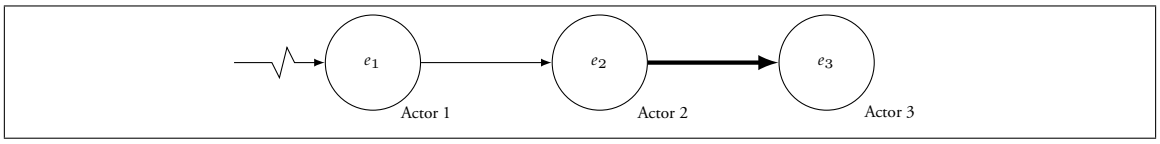


Figure 5.1: Example of an event graph.

5.4 Simple Communication Protocol

Our first case study is a simple communication protocol from (Satoh & Tokoro, 1995) that was modeled and analyzed in (Aceto et al., 2011). The protocol consists of a sender agent and a receiver agent at different locations using unreliable communication channel. The sender agent sends a message to the receiver agent and waits for an acknowledgment. If the acknowledgment is not received by the sender agent within 8 time units, it retransmits the message. When the receiver agent receives a message it sends an acknowledgment back. Communications over the communication channel from the sender agent to the receiver agent take 2 to 4 time units and may occasionally fail, while from the receiver agent to the sender agent it takes 1 to 3 time units and may also occasionally fail. If the sender agent receives the acknowledgement within 8 time units the communication was successful and we terminate the execution of the model.

Note: The sender agent retransmits the message until a successful communication can be made.

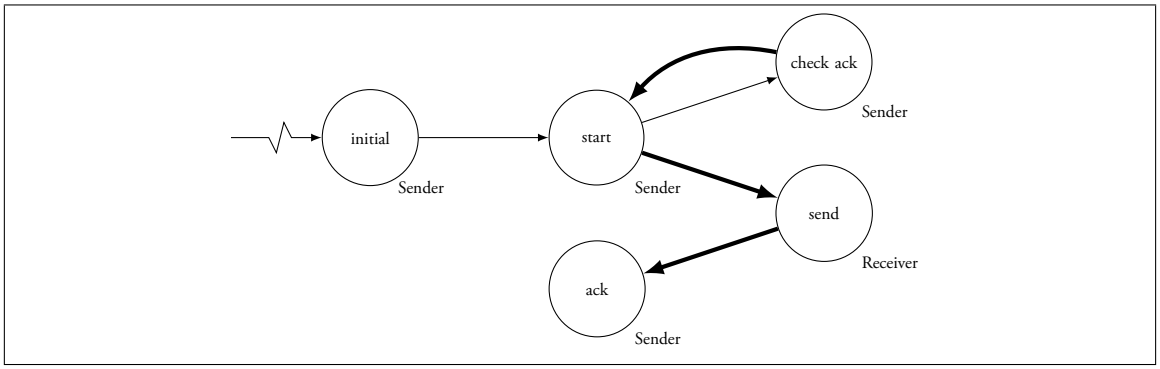


Figure 5.2: Event graph of the simple communication protocol model. One interesting property of the model is the possibility of infinite computation. That is if message is dropped in every try we end up in a loop, *start* → *check ack* → *start* → …

The Timed Rebeca code for the model is shown in Listing 5.1 and the accompanying event graph in Figure 5.2. In addition, we also include sequence diagrams showing few simulation runs in Section A.1.

For the analysis of the model we ran 100 simulations, each one until the model terminated successfully. In Table 5.2 we show the TeProp properties we checked and the results.

Description	Property	Result
Acknowledgment was received.	**F** senderAgent.ack()	Satisfied for 100%
At least one send was acknowledged within 8 time units.	**F**(senderAgent.start() $\leadsto$ **F**$[0, 8]$ senderAgent.ack())	Satisfied for 100%
The delivery time is at least 2 time units.	**G**(senderAgent.start() $\rightarrow \neg$ **F**$[0, 1]$ receiverAgent.send())	Satisfied for 100%
The delivery time of the acknowledgment was exactly 3 time units.	**F**(receiverAgent.send() $\leadsto$ **F**$[3, 3]$ senderAgent.ack())	Satisfied for 33%

Table 5.2: Overview of TeProp properties checked for the simple communication protocol.

From the results of the properties it seems that the model is behaving as intended; as the sender receives an acknowledgement in every simulation. That is not all we can read from the results, property 4 gives us insight into the non-determinism behavior of the model. It shows that the time it takes from the sending of message until it is acknowledged varies between simulations and is exactly 3 time units for 33% of the simulation runs.

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

Listing 5.1: Timed Rebeca code for the simple communication protocol model.

5.5 Ticket Service

Our second case study is the ticket service introduced, modeled and tested in (Aceto et al., 2011). The model consists of two ticket services and one agent. The agent is ordering a ticket for a customer that is outside the model. The agent starts by sending a message, requesting a ticket from the first ticket service with a deadline of *requestDeadline* time units. When the ticket service receives a ticket request, it issues the ticket by sending a message to the agent after processing the ticket. Processing the ticket takes *serviceTime1* or *serviceTime2* time units. Following the ticket request the agent checks, after *checkIssuedPeriod* time units, if the ticket was issued. In the case the ticket was issued the agent continues to his next customer after *newRequestPeriod* time units. Otherwise the agent tries the second ticket service immediately and as before checks if the ticket was issued after *checkIssuedPeriod* time units. If the ticket was issued from the second ticket service, the agent continues to his next customer after *newRequestPeriod* time units. Otherwise the agent tries the first ticket service again after *retryRequestPeriod* time units, repeating the process. The agent only accepts the last requested ticket.

The Timed Rebeca code for the model is shown in Listing 5.2 and the accompanying event graph in Figure 5.3. In addition we also include sequence diagrams showing few simulation runs in Section A.2.

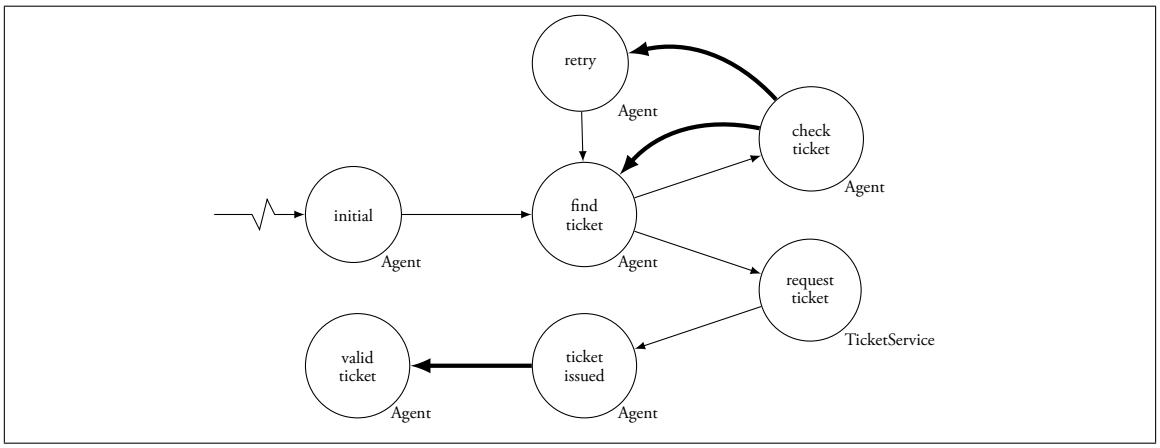


Figure 5.3: Event graph of the ticket service model.

One important implementation detail of the model for the analysis, is the *token* variable used to keep track of the last requested ticket.

For the analysis of the model we ran 600 simulations, using the 6 environment settings in Table 5.3 (100 simulations for each setting). Each one until the *token* count was

Setting	Request deadline	Check issued period	Retry request period	New request period	Service time 1	Service time 2
1	2	1	1	1	3	7
2	2	1	1	1	4	7
3	2	2	1	1	4	7
4	2	2	1	1	3	7
5	2	4	1	1	3	7
6	2	8	1	1	3	7

Table 5.3: Environment settings used for the simulation of the ticket service model. The first four settings are the same as in (Aceto et al., 2011).

equal to 10.000. In Table 5.4 we show the TeProp properties we checked and their results.

Property	Setting	Result
Valid ticket is issued: **F** agent.validTicket()	1 2 3 4 5 6	Not satisfied for all runs Not satisfied for all runs Not satisfied for all runs Satisfied for 100% Satisfied for 100% Satisfied for 100%
Ticket request with token 5 is delivered: **F** ts1.requestTicket(token == ”5”) ∨ **F** ts2.requestTicket(token == ”5”)	1 2 3 4 5 6	Satisfied for 74% Satisfied for 76% Satisfied for 71% Satisfied for 69% Satisfied for 86% Satisfied for 100%
At least 4 time units between retries: **G**(agent.retry() → ¬**F**$[0, 4]$ agent.retry())	1 2 3 4 5 6	Not satisfied for all runs Not satisfied for all runs Satisfied for 100% Not satisfied for all runs Satisfied for 100% Satisfied for 100%
An issued ticket should not be accepted by the agent while the agent is in the the process of requesting a new ticket: ¬**F**(agent.checkTicket() ⇝ agent.validTicket() **B**$[0, 2]$ agent.findTicket())	1 2 3 4 5 6	Satisfied for 100% Satisfied for 100% Satisfied for 100% Not satisfied for all runs Not satisfied for all runs Satisfied for 100%

Table 5.4: Overview of TeProp properties checked for the ticket service. The setting column refers to the environment variables in Table 5.3.

The results of property four in Table 5.4 indicates a flaw in the Timed Rebeca code for the model. As a ticket issued after the expiration (when *checkTicket* is called) of the ticket request should not be accepted.

A review of the code confirmed the flaw. After two unsuccessful attempts to order a ticket the agent waits *retryRequestPeriod* time units before requesting a new ticket. During that time the *token* is not updated. Since the *token* is not updated the agent will accept a ticket during this waiting time as a valid one. Even though he never checks for the ticket and will request a new ticket as soon as waiting period ends. This flaw can be fixed by updating the *token* immediately in the *checkTicket* message server instead of the *findTicket* message server, invalidating the last requested ticket before the waiting time starts. Figure A.7 shows a sequence diagram with the flaw.

We repeated all the simulations using the revised model in Listing B.1. The results for the same properties as in Table 5.4 are shown in Table 5.5.

Property	Setting	Result
Valid ticket is issued: **F** agent.validTicket()	1	Not satisfied for all runs
	2	Not satisfied for all runs
	3	Not satisfied for all runs
	4	Not satisfied for all runs
	5	Satisfied for 100%
	6	Satisfied for 100%
Ticket request with token 5 is delivered: **F** ts1.requestTicket(token == ”5”) ∨ **F** ts2.requestTicket(token == ”5”)	1	Satisfied for 74%
	2	Satisfied for 74%
	3	Satisfied for 78%
	4	Satisfied for 65%
	5	Satisfied for 87%
	6	Satisfied for 100%
At least 4 time units between retries: **G**(agent.retry() → ¬**F**$[0, 4]$ agent.retry())	1	Not satisfied for all runs
	2	Not satisfied for all runs
	3	Satisfied for 100%
	4	Satisfied for 100%
	5	Satisfied for 100%
	6	Satisfied for 100%
An issued ticket should not be accepted by the agent while the agent is in the the process of requesting a new ticket: ¬**F**(agent.checkTicket() ⇝ agent.validTicket() **B**$[0, 2]$ agent.findTicket())	1	Satisfied for 100%
	2	Satisfied for 100%
	3	Satisfied for 100%
	4	Satisfied for 100%
	5	Satisfied for 100%
	6	Satisfied for 100%

Table 5.5: Overview of TeProp properties checked for the revised ticket service. Compared to Table 5.4, here you see that the forth property is satisfied for all settings. The setting column refers to the environment variables in Table 5.3.

When we compare the results in Table 5.4 and Table 5.5 we see that for the revised model the main difference is that no valid ticket is ever issued for setting 4 and that

no ticket is ever accepted while the agent is in process of requesting a new ticket as expected. The reason for no ticket being issued for setting 4 is that the *checkIssued-Period* is too short compared to the *serviceTime1* and *serviceTime2*, it takes the ticket service longer to process the ticket request than the agent is willing to wait for a ticket. The only reason for a valid ticket being issued in the previous run was the flaw in the model.

When we review the results in Table 5.5 it shows that the revised model is behaving as intended; given the right timings a valid ticket is issued in all simulation runs. We say that a ticket is valid if it is issued by the ticket service and sent to the agent while it is waiting for the ticket. Property 2 shows us how the different timings affect the delivery of a single ticket, here we check when the token is 5 meaning the 5^{th} request. The percentage of simulation runs where ticket request for ticket number 5 is delivered varies between 65% and 100% for settings 1 to 6. It is interesting to compare the results from property 1 and 2 for settings 5 and 6. Property 1 shows that a valid ticket is always issued for at least 1 of the 10.000 tickets requested in each simulation run using setting 5. Property 2 shows that the ticket request number 5 does not always get delivered to the ticket service using setting 5, meaning that the ticket request is dropped. Setting 6 on the other hand always delivers the ticket request to the ticket service. The reason for the difference between settings 5 and 6 is that for setting 5 we have *checkIssuedPeriod* that is only long enough if we non-deterministically get a ticket service with process time of 3 time units, while setting 6 has *checkIssuedPeriod* long enough for both processing time's, *serviceTime1* and *serviceTime2*. As a result of this in setting 5 the agent can start requesting a new ticket from a ticket service while it is still processing a previous ticket request from the agent. If the processing time left on the previous request is more than the deadline of the current request, then the ticket request is dropped, and is never processed by the ticket service.

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

reactiveclass Agent {
  knownrebecs { TicketService ts1; TicketService ts2; }

  statevars { int attemptCount; boolean ticketIssued; int token; }

  msgsrv initial() { self.findTicket(ts1); } // initialize system, check 1st ticket service

  msgsrv findTicket(TicketService ts) {
    attemptCount += 1;
    token += 1;
    ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
    self.checkTicket() after(checkIssuedPeriod); // check for issued ticket
  }

  msgsrv validTicket(int tok) {
    //Event for TeProp
  }

  msgsrv ticketIssued(int tok) {
    if (token == tok) {
       ticketIssued = true;
       self.validTicket(tok);
    }
  }

  msgsrv checkTicket() {
    if (!ticketIssued && attemptCount == 1) {      // no ticket from 1st service,
      self.findTicket(ts2);                        // try the second TicketService
    } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
      self.retry() after(retryRequestPeriod);      // restart from the first TicketService
    } else if (ticketIssued) {                     // the second TicketService replied,
      ticketIssued = false;
      self.retry() after(newRequestPeriod);        // new request for a customer
    }
  }
  msgsrv retry() {
    attemptCount = 0;
    self.findTicket(ts1);                          // restart from the first TicketService
  }
}

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

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

Listing 5.2: Timed Rebeca code for the ticket service model. Notice that we added one extra message server to the model that is called when a ticket is issued with the current token. This is done to be able to check when a valid ticket is issued using **events**. Aceto et al. used a McErlang monitor working with **states** and could therefore use the variable *ticketIssued*.

5.6 Sensor Network

Our third case study is a sensor network that is introduced, modeled and analyzed in (Aceto et al., 2011). The model consists of two sensors, a scientist, an admin and a rescue team. The sensors are set up to monitor levels of toxic gasses near the scientist. Each sensor sends the measured values periodically to the admin. With a period of *sensor0period* time units for the first sensor and *sensor1period* for the second sensor. The admin checks the received measurements periodically with a period of *adminCheckDelay* time units. When the measurements show dangerous toxic levels the admin immediately notifies the scientist by a message. If the scientist does not acknowledge the message within *scientistDeadline* time units, the admin sends the rescue team to save the scientist. The scientist dies without a rescue within *rescueDeadline* time units.

There is a communication delay of *netDelay* time units between all participants in the model.

The Timed Rebeca code for the model is shown in Listing 5.3 and the accompanying event graph in Figure 5.4. In addition we also include sequence diagrams showing few simulation runs in Section A.3.

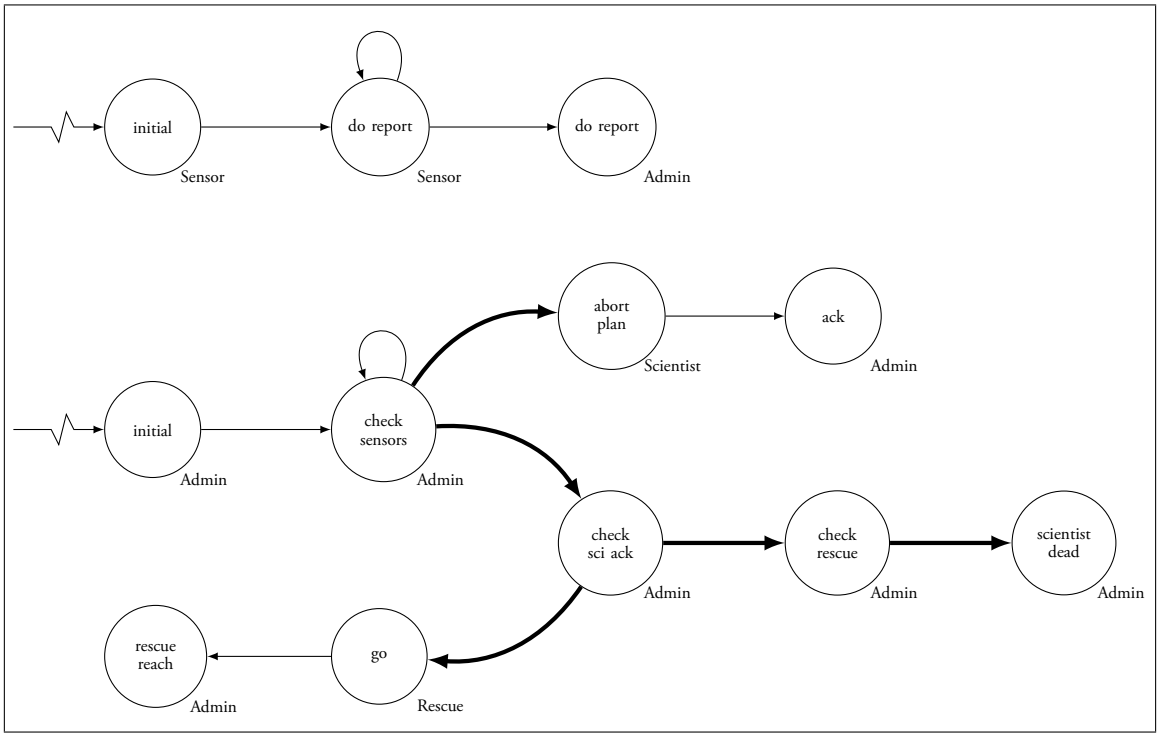


Figure 5.4: Event graph of the sensor network model.

For the analysis of the model we ran 700 simulations, using the 7 environment settings in Table 5.6 (100 simulations for each setting). Each one for 60 seconds. In Table 5.7 we show the TeProp properties we checked and their results.

Setting	Network delay	Admin period	Sensor 0 period	Sensor 1 period	Scientist deadline	Rescue deadline
1	1	4	2	3	2	3
2	1	4	2	3	2	4
3	2	1	1	1	4	5
4	2	1	1	1	4	6
5	2	1	1	1	4	7
6	2	4	1	1	4	7
7	2	4	1	1	5	7

Table 5.6: Environment settings used for the simulation of the sensor network model. The first six settings are the same as in (Aceto et al., 2011).

Property	Setting	Result
The scientist will not die: ¬**F** admin.scientistDead()	1	Not satisfied for all runs
	2	Satisfied for 100%
	3	Not satisfied for all runs
	4	Not satisfied for all runs
	5	Not satisfied for all runs
	6	Satisfied for 100%
	7	Satisfied for 100%
The rescue team never sent: ¬**F** rescue.go()	1	Not satisfied for all runs
	2	Not satisfied for all runs
	3	Not satisfied for all runs
	4	Not satisfied for all runs
	5	Not satisfied for all runs
	6	Not satisfied for all runs
	7	Satisfied for 100%
The admin never misses an acknowledgement as result of ordering of events within a time unit: **G**(admin.checkScientistAck() → ¬**F**[0, 0] admin.ack())	1	Not satisfied for all runs
	2	Not satisfied for all runs
	3	Not satisfied for all runs
	4	Not satisfied for all runs
	5	Not satisfied for all runs
	6	Not satisfied for all runs
	7	Satisfied for 100%

Table 5.7: Overview of TeProp properties checked for the sensor network. The setting column refers to the environment variables in Table 5.6.

From the results of the properties it seems that the model is behaving as intended; the rescue team is sent when the scientist does not acknowledge within the time limit and sadly the scientist dies in case the rescue team does not reach him in time. The model shows well how timings can have dramatic effects on the results. It is interesting to compare the results for property 1 and 2, where we see that for settings 2 and 6 the

scientist does not die as the rescue team is sent for him. But for setting 7 the scientist does not die although no rescue team is sent, as the scientist acknowledges the warning message in time. We also see from property 3 that the timings for setting 7 avoids sending the rescue team on the same time unit as the scientist acknowledges.

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

reactiveclass Sensor {
    knownrebecs {
        Admin admin;
    }

    statevars {
        int period;
    }

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

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

reactiveclass Scientist {
    knownrebecs {
        Admin admin;
    }

    msgsrv initial() {}

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

reactiveclass Rescue {
    knownrebecs {
        Admin admin;
    }

    msgsrv initial() {}

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

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

```
63      }
64
65      statevars {
66          boolean reported0;
67          boolean reported1;
68          int sensorValue0;
69          int sensorValue1;
70          boolean sensorFailure;
71          boolean scientistAck;
72          boolean scientistReached;
73          boolean scientistDead;
74      }
75
76      msgsrv initial() {
77          self.checkSensors();
78      }
79
80      msgsrv report(int value) {
81          if (sender == sensor0) {
82              reported0 = true;
83              sensorValue0 = value;
84          } else {
85              reported1 = true;
86              sensorValue1 = value;
87          }
88      }
89
90      msgsrv rescueReach() {
91          scientistReached = true;
92      }
93
94      msgsrv checkSensors() {
95          if (reported0) reported0 = false;
96          else sensorFailure = true;
97
98          if (reported1) reported1 = false;
99          else sensorFailure = true;
100
101         boolean danger = false;
102         if (sensorValue0 > 3) danger = true;
103         if (sensorValue1 > 3) danger = true;
104
105         if (danger) {
106             scientist.abortPlan() after(netDelay);
107             self.checkScientistAck() after(scientistDeadline); // deadline for the scientist to
                    answer
108         }
109
110         self.checkSensors() after(adminCheckDelay);
111     }
112
113     msgsrv checkRescue() {
114         if (!scientistReached) {
115             self.scientistDead();
116             scientistDead = true; // scientist is dead
117         } else {
118             scientistReached = false;
119         }
120     }
121
122     msgsrv ack() {
123         scientistAck = true;
124     }
125
126     msgsrv checkScientistAck() {
127         if (!scientistAck) {
128             rescue.go() after(netDelay);
129             self.checkRescue() after(rescueDeadline);
130         }
131         scientistAck = false;
```

```
132     }
133
134     msgsrv scientistDead() {
135         //Event for TeProp
136     }
137 }
138
139 main {
140     Sensor sensor0(admin):(sensor0period);
141     Sensor sensor1(admin):(sensor1period);
142     Scientist scientist(admin):();
143     Rescue rescue(admin):();
144     Admin admin(sensor0, sensor1, scientist, rescue):();
145 }
```

Listing 5.3: Timed Rebeca code for the sensor network model. Notice that we added one extra message server to the model that is called when the scientist dies. This is done to be able to check properties related to the death of the scientist using **events**. Aceto et al. used a McErlang monitor working with **states** and could therefore use the variable *scientistDead*.

5.7 Multi Flight Booking

Our forth case study is a model of a multi flight booking which can be thought of as an extended version of the ticket service where timing is crucial. Here we have related requests which have to be served by two different servers in an atomic transaction. But because of distribution without synchronization, non-desirable case can happen when we end up with one successful request while the other was unsuccessful.

The model consists of two airlines websites and a customer. The customer starts by finding flights; that involves finding the first flight and requesting a reservation of a ticket from airline_1 website and then finding and reserving the second flight from airline_2 website. The time it takes to find a flight is *findFlightTime1* or *findFlightTime2* time units. The airline websites reserves the flight for either *reservationTimeout1* or *reservationTimeout2* time units. After the customer receives messages that both flights have been reserved he starts to book the flights by sending a request to the airlines websites. It takes the customer *bookingTime1* or *bookingTime2* time units to prepare the booking order before sending it. If the airline website receives the booking order before the reservation expires it sends a confirmation that the flight was successfully booked, otherwise a message indicating that the flight was not booked is sent.

There is a communication delay of *networkDelay* time units between all participants in the model.

The Timed Rebeca code for the model is shown in Listing 5.4 and the accompanying event graph in Figure 5.5. In addition we also include sequence diagrams showing few simulation runs in Section A.4.

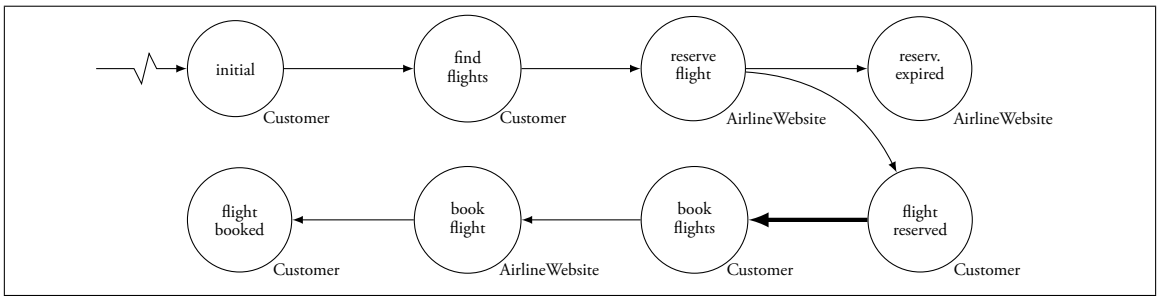


Figure 5.5: Event graph of the multi flight booking model.

For the analysis of the model we ran 700 simulations, using the 7 environment settings in Table 5.8 (100 simulations for each setting). In Table 5.9 we show the TeProp properties we checked and their results.

Setting	Network delay	Find flight time 1	Find flight time 2	Booking time 1	Booking time 2	Reservation timeout 1	Reservation timeout 2
1	1	1	2	1	2	2	4
2	1	2	2	2	2	5	10
3	2	1	3	0	0	2	5
4	2	2	1	0	3	4	8
5	2	2	1	2	1	2	4
6	2	3	2	1	1	8	10
7	2	3	2	1	1	15	16

Table 5.8: Environment settings used for the simulation of the multi flight booking model.

Property	Setting	Result
The first ticket is successfully booked: **F** customer.flightBooked(f == ”1” ∧ successful == ”true”)	1	Satisfied for 7%
	2	Satisfied for 52%
	3	Satisfied for 13%
	4	Satisfied for 28%
	5	Not satisfied for all runs
	6	Satisfied for 90%
	7	Satisfied for 100%
The second ticket is successfully booked: **F** customer.flightBooked(f == ”2” ∧ successful == ”true”)	1	Satisfied for 9%
	2	Satisfied for 54%
	3	Satisfied for 50%
	4	Satisfied for 48%
	5	Not satisfied for all runs
	6	Satisfied for 100%
	7	Satisfied for 100%
All tickets are successfully booked: ¬**F** customer.flightBooked(successful == ”false”)	1	Satisfied for 2%
	2	Satisfied for 31%
	3	Satisfied for 7%
	4	Satisfied for 19%
	5	Not satisfied for all runs
	6	Satisfied for 90%
	7	Satisfied for 100%
Booking occurred 3 or more time units before the reservation ran out: **F** (ws1.bookFlight() ⇝ **F**[3, *end*] ws1.reservationExpired()) ∨ **F** (ws2.bookFlight() ⇝ **F**[3, *end*] ws2.reservationExpired())	1	Not satisfied for all runs
	2	Satisfied for 75%
	3	Not satisfied for all runs
	4	Satisfied for 30%
	5	Not satisfied for all runs
	6	Satisfied for 57%
	7	Satisfied for 100%

Table 5.9: Overview of TeProp properties checked for the multi flight booking. The setting column refers to the environment variables in Table 5.8.

From the results in Table 5.9 it seems that the model is behaving as intended; just as in the ticket service model valid tickets are booked given the right timings. When we compare the results for properties 1 to 3 we clearly see the impact of an unsuccessfull

booking for a flight, as both flights must be booked for a successful multi-flight booking. Even though our success rate of booking a ticket for each flight is close to 50% we can end up with only 30% of successful multi-flight bookings. Property 4 then gives us indication whether the reservation times are too generous, this is important since in reservation systems we want to minimize the reservation times to fulfill as many requests as possible.

```
env int networkDelay;
env int findFlightTime1;
env int findFlightTime2;
env int bookingTime1;
env int bookingTime2;
env int reservationTimeout1;
env int reservationTimeout2;

reactiveclass Customer {
	knownrebecs {
		AirlineWebsite ws1;
		AirlineWebsite ws2;
	}

	statevars {
		boolean reservedFlight1;
		boolean reservedFlight2;
		boolean bookedFlight1;
		boolean bookedFlight2;
	}

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

	msgsrv findFlights() {
		int findDelay1 = ?(findFlightTime1, findFlightTime2);
		delay(findDelay1);
		ws1.reserveFlight(1) after(networkDelay);
		int findDelay2 = ?(findFlightTime1, findFlightTime2);
		delay(findDelay2);
		ws2.reserveFlight(2) after(networkDelay);
	}

	msgsrv flightReserved(int f) {
		if(f == 1) {
			reservedFlight1 = true;
		} else {
			reservedFlight2 = true;
		}

		if(reservedFlight1 && reservedFlight2) {
			self.bookFlights();
		}
	}

	msgsrv bookFlights() {
		int bookingDelay1 = ?(bookingTime1, bookingTime2);
		delay(bookingDelay1);
		ws1.bookFlight(1) after(networkDelay);
		int bookingDelay2 = ?(bookingTime1, bookingTime2);
		delay(bookingDelay2);
		ws2.bookFlight(2) after(networkDelay);
	}

	msgsrv flightBooked(int f, boolean successful) {
		if(successful && f == 1) {
			bookedFlight1 = true;
		} else if(successful) {
```

```
60                      bookedFlight2 = true;
61              }
62      }
63  }
64
65  reactiveclass AirlineWebsite {
66      knownrebecs {
67              Customer c;
68      }
69
70      statevars {
71              boolean flightReserved;
72      }
73
74      msgsrv reservationExpired(int f) {
75              flightReserved = false;
76      }
77
78      msgsrv reserveFlight(int f) {
79              flightReserved = true;
80              int reservedFor = ?(reservationTimeout1, reservationTimeout2);
81              self.reservationExpired(f) after(reservedFor);
82              c.flightReserved(f) after (networkDelay);
83      }
84
85      msgsrv bookFlight(int f) {
86              c.flightBooked(f, flightReserved) after (networkDelay);
87      }
88  }
89
90  main {
91      Customer customer(ws1, ws2):();
92      AirlineWebsite ws1(customer):();
93      AirlineWebsite ws2(customer):();
94  }
```

Listing 5.4: Timed Rebeca code for the multi flight booking model.

Chapter 6

Related Work

There are two dimensions to this thesis, and therefore two types of related work. In the background we presented timed property languages that are related to TeProp, and here we present related work on simulations using databases.

For the analysis of simulations using databases different techniques have been applied. In this chapter we present a brief survey of related work on simulation analysis using databases and compare them to our work.

6.1 Temporal-SQL

Böhlen et al. introduced a transformation from Linear Temporal Logic (LTL) to TSQL2, a temporal extension to the SQL language (Böhlen, Chomicki, Snodgrass, & Toman, 1996). They showed that the transformation from LTL to SQL is rather straight forward.

Our transformation from TeProp to SQL differs from theirs in the way that we are also working with time and use a standard SQL database.

6.2 XAV

XAV is a tracing framework for exploring large network simulation outputs (Ben-El-Kezadri, Pujolle, & Kamoun, 2008). It uses a XML enabled database to store the simulation data and has been implemented in the network simulator NS-2. Trace files from the simulator are loaded into the database in an XML based XAV format that

utilizes XML pointers to reduce duplication of data. The analysis of the simulation is then done by extracting data from the database, using queries in the SQL-like query language XQuery.

The main difference of our approach is that the end user of TRSim writes TeProp properties instead of SQL-like queries. We believe that writing properties in TeProp is much easier for the user than writing SQL-like queries, as the user does not need to know the underlying data structure.

6.3 Trace Server

Trace Server is an extension of the Java PathFinder model checking tool for storing, querying and processing data describing the execution of a Java program being verified (Andjelkovic & Artho, 2011). The user defines trace filters that are used to decide on what events are sent to the trace storage, which can either be an in-memory database or a graph database (Neo4j, 2012). For the analysis of the trace data a Java query interface is provided.

The main difference of our approach is the underlying database and the query options. While in TRSim the user writes a TeProp property that is converted to SQL and executed on a standard SQL database, the user of Trace Server writes a Java code that traverses a graph stored in a graph database.

There is also the difference of Trace Server working with traces of a full programming language, while we are working with a modeling language that is more abstract.

6.4 TLtoSQL

The most similar approach to us. TLtoSQL is a tool-set for rapid post-mortem verification of systems using temporal logic to SQL (Drusinsky, 2009). Post-mortem execution log files are read into the tool-kit and automatically converted into JUnit test cases. The JUnit test cases are then executed and the event sequence during the run is stored in a database along with the events relative order and time.

The tool-kit then offers a graphical editor for Linear Temporal Logic (LTL) and Metric Temporal Logic (MTL) formal specifications, the output of the editor is SQL query code that the user can then execute on the database.

While both TRSim and TLtoSQL make use of a database and conversion from a property language to SQL, they differ in their intended use. TLtoSQL is meant to be a verification framework for verifying system implementations using execution logs, where TRSim is an integrated environment for simulation and verification of Timed Rebeca models; making it easy to run verification queries over multiple simulations. LTtoSQL also stores all the information in one database table, while TRSim uses a separate table for each message server.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, we presented two main contributions. First we introduced the timed event-based property language, TeProp, for reasoning about the timing and occurrence of events. We presented its syntax and semantics as well as the timed property patterns we based the language on. TeProp was designed for use with Timed Rebeca but can be used by other event-based systems as well. As the first implementation of TeProp we provided a mapping to SQL.

Second, we presented the TRSim tool-set for working with Timed Rebeca simulations using a relational database. During the simulation of a Timed Rebeca model information about the timing and occurrence of events are saved into a database. The TRSim query tool can then be used to check TeProp properties by mapping them to a SQL query and check their validity over multiple simulations. Although the TRSim tool-kit makes use of TeProp it can easily be extended for analysis using other approaches.

We also showed results of four case studies using the TeProp property language and the TRSim tool-kit. We showed that using the TRSim tool-kit along with TeProp properties we were able to find a flaw in previously analyzed model.

7.2 Future Work

- Establish a model checking algorithm for TeProp, such that TeProp can be used as a property language in Timed Modere; a timed version of the Modere model checker for Rebeca that is being developed. As well as investigate whether TeProp can be used as a property language when model checking Timed Rebeca models using McErlang.
- Create a run-time monitor for TeProp, such that the implementation of the model could be tested with the same properties.
- Extend the mapping of TeProp to SQL, to allow the modeler to use one interval variable in a property, like $\mathbf{F}[0, x]\ e$. The modeler could then ask for the maximum or minimum value for x such that the property is satisfied, for each simulation run, if one exists within a specified threshold.
- Look into ways of guiding the simulations and try to maximize the coverage, it would also be interesting to see if we could estimate the number of simulations required to reach certain coverage.
- Add support for other relational databases in TRSim, such as SQLite (SQLite, 2012) and MonetDB (MonetDB, 2012), and compare their performance.
- Add static code analysis to TRSim to detect invalid models before simulating them, for example models with the Zeno-behavior.

Bibliography

Abid, N., Dal Zilio, S., & Le Botlan, D. (2011). *A Real-Time Specification Patterns Language* (Tech. Rep. No. LAAS 11364). Available from `http://hal.archives-ouvertes.fr/hal-00593965`

Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A. H., Sigurdarson, S. H., & Sirjani, M. (2011). Modelling and Simulation of Asynchronous Real-Time Systems using Timed Rebeca. In M. R. Mousavi & A. Ravara (Eds.), *Foclasa* (Vol. 58, p. 1-19).

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

Agha, G., Mason, I. A., Smith, S. F., & Talcott, C. L. (1997). A Foundation for Actor Computation. *J. Funct. Program.*, *7*(1), 1-72.

Alur, R., Courcoubetis, C., & Dill, D. (1990, jun). Model-checking for real-time systems. In *Logic in computer science, 1990. lics '90, proceedings., fifth annual ieee symposium on e* (p. 414 -425).

Alur, R., & Dill, D. L. (1994). A Theory of Timed Automata. *Theoretical Computer Science*, *126*, 183–235.

Alur, R., & Henzinger, T. A. (1994, January). A really temporal logic. *J. ACM*, *41*(1), 181–203.

Andjelkovic, I., & Artho, C. (2011). Trace Server: A Tool for Storing, Querying and Analyzing Execution Traces. In *Proc. JPF workshop 2011.* Lawrence, USA.

Bellini, P., Giotti, A., Nesi, P., & Rogai, D. (2003). TILCO Temporal Logic for Real-Time Systems Implementation in C++. In *Seke* (p. 166-173).

Bellini, P., Nesi, P., & Rogai, D. (2009, February). Expressing and organizing real-time specification patterns via temporal logics. *J. Syst. Softw.*, *82*(2), 183–196.

Ben-El-Kezadri, R., Pujolle, G., & Kamoun, F. (2008). XAV: a tracing framework for exploring large network simulation outputs. In *Proceedings of the 3rd international conference on performance evaluation methodologies and tools* (pp. 76:1–76:9). ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer

Sciences, Social-Informatics and Telecommunications Engineering).

Böhlen, M. H., Chomicki, J., Snodgrass, R. T., & Toman, D. (1996). Querying TSQL2 Databases with Temporal Logic. In *Proceedings of the 5th international conference on extending database technology: Advances in database technology* (pp. 325–341). London, UK, UK: Springer-Verlag.

Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., & Yovine, S. (1998). Kronos: A Model-Checking Tool for Real-Time Systems. In A. J. Hu & M. Y. Vardi (Eds.), *Cav* (Vol. 1427, p. 546-550). Springer.

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

Clarke, E. M., Emerson, E. A., & Sistla, A. P. (1986, April). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, *8*(2), 244–263.

Drusinsky, D. (2000). The Temporal Rover and the ATG Rover. In *Spin* (p. 323-330).

Drusinsky, D. (2009, 30 2009-june 3). TLtoSQL: Rapid post-mortem verification using temporal logic to SQL code generation in the Eclipse PDE. In *System of systems engineering, 2009. sose 2009. ieee international conference on* (p. 1 -5).

Dwyer, M. B., Avrunin, G. S., & Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of the 21st international conference on software engineering* (pp. 411–420). New York, NY, USA: ACM.

Earle, C. B., & Fredlund, L. Åke. (2012). *Verification of Timed Erlang Programs using McErlang.* (Accepted for publication, FORTE 2012)

Fredlund, L.-A., & Svensson, H. (2007, October). McErlang: a model checker for a distributed functional programming language. *SIGPLAN Not.*, *42*(9), 125–136.

Henzinger, T. A., Ho, P. H., & Toi, H. W. (1997). HYTECH: A Model Checker for Hybrid Systems. *International Journal on Software Tools for Technology Transfer*, *1*(1-2), 110–122.

Hewitt, C. (1972). *Description and Theoretical Analysis (Using Schemata) of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot* (Tech. Rep. No. 258). MIT AI Laboratory.

Hewitt, C. (2007). Coordination, Organizations, Institutions, and Norms in Agent Systems II. In P. Noriega et al. (Eds.), (pp. 293–307). Berlin, Heidelberg: Springer-Verlag.

Konrad, S., & Cheng, B. (2005, may). Real-time specification patterns. In *Software engineering, 2005. icse 2005. proceedings. 27th international conference on* (p. 372

- 381).

Koymans, R. (1990). Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, *2*(4), 255-299.

Kristinsson, H. (2012). *Event-based Analysis of Real-time Actor Models*. Unpublished master's thesis, Reykjavik University, Iceland.

Larsen, K. G., Pettersson, P., & Yi, W. (1997, October). Uppaal in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, *1*(1–2), 134-152.

Lynch, N. A. (1996). *Distributed Algorithms*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Mattolini, R., & Nesi, P. (2001, mar). An interval logic for real-time system specification. *Software Engineering, IEEE Transactions on*, *27*(3), 208 -227.

Microsoft. (2012). *http://blogs.msdn.com/b/windowsazure/archive/2012/03/01/windows-azure-service-disruption-update.aspx.*

MonetDB. (2012). *http://www.monetdb.org.*

Neo4j. (2012). *http://neo4j.org/.*

Ouaknine, J., & Worrell, J. (2008). Some Recent Results in Metric Temporal Logic. In *Proceedings of the 6th international conference on formal modeling and analysis of timed systems* (pp. 1–13). Berlin, Heidelberg: Springer-Verlag.

Pnueli, A. (1977, 31 1977-nov. 2). The temporal logic of programs. In *Foundations of computer science, 1977., 18th annual symposium on* (p. 46 -57).

PostgreSQL. (2012). *http://www.postgresql.org.*

Satoh, I., & Tokoro, M. (1995). Time and Asynchrony in Interactions among Distributed Real-Time Objects. In *Proceedings of the 9th european conference on object-oriented programming* (pp. 331–350). London, UK, UK: Springer-Verlag.

Sirjani, M., & Jaghoori, M. M. (2011). Formal modeling. In G. Agha, J. Meseguer, & O. Danvy (Eds.), (pp. 20–56). Berlin, Heidelberg: Springer-Verlag.

Sirjani, M., Movaghar, A., Shali, A., & Boer, F. S. de. (2004, June). Modeling and Verification of Reactive Systems using Rebeca. *Fundam. Inf.*, *63*(4), 385–410.

SQLite. (2012). *http://www.sqlite.org/.*

Wirsing, M., Bauer, S. S., & Schroeder, A. (2010). Modeling and Analyzing Adaptive User-Centric Systems in Real-Time Maude. In P. C. Ölveczky (Ed.), *Rtrts* (Vol. 36, p. 1-25).

Appendix A

Sequence Diagrams

A.1 Simple Communication Protocol

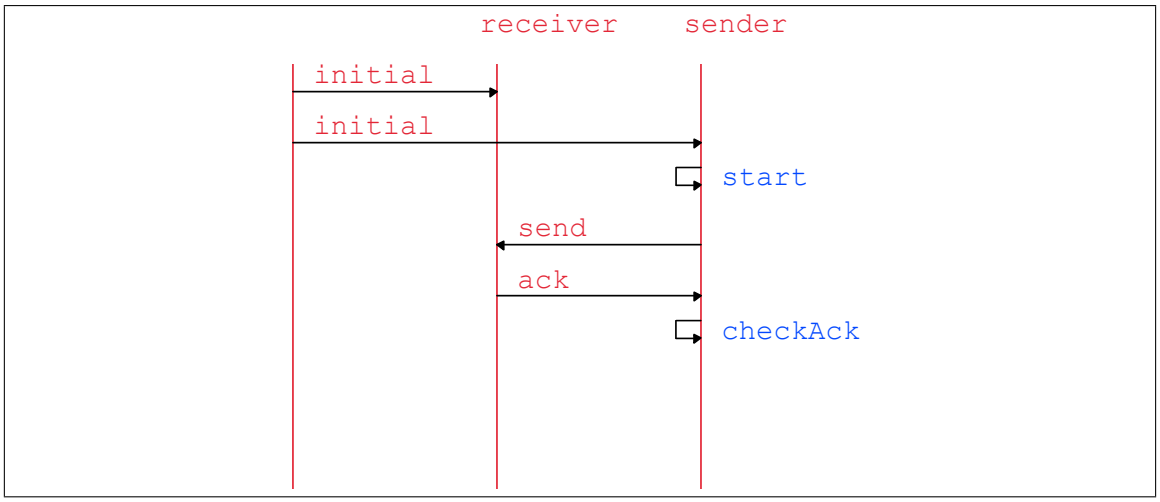


Figure A.1: Sequence diagram of a run of the simple communication protocol where the send and acknowledge messages where delivered in first try.

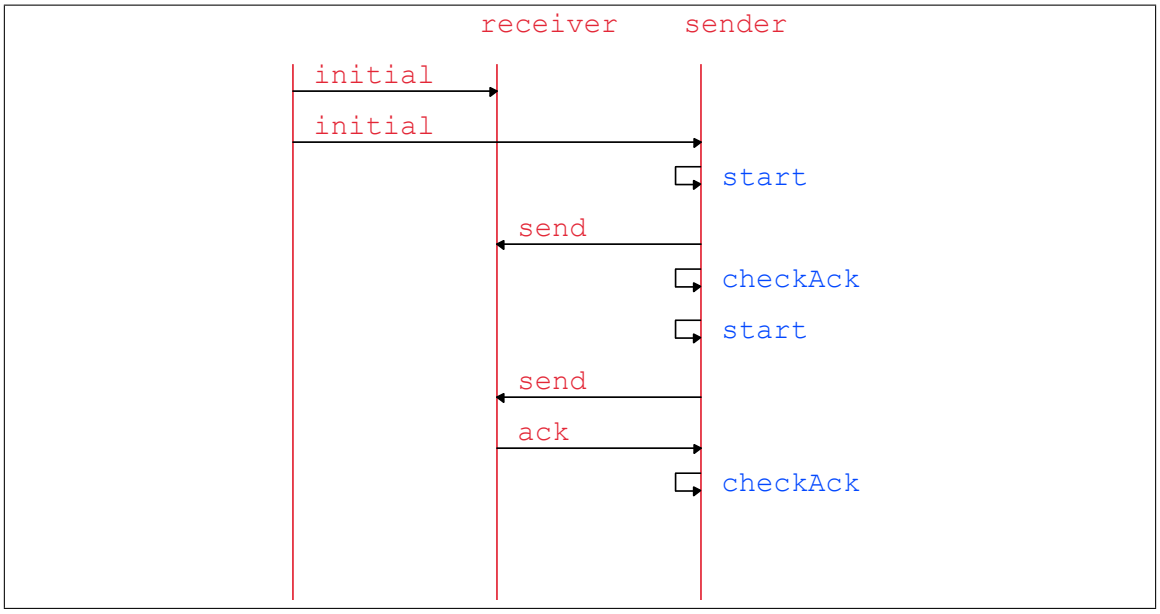


Figure A.2: Sequence diagram of a run of the simple communication protocol where the first acknowledge messages is dropped and the sender agent retransmits the message.

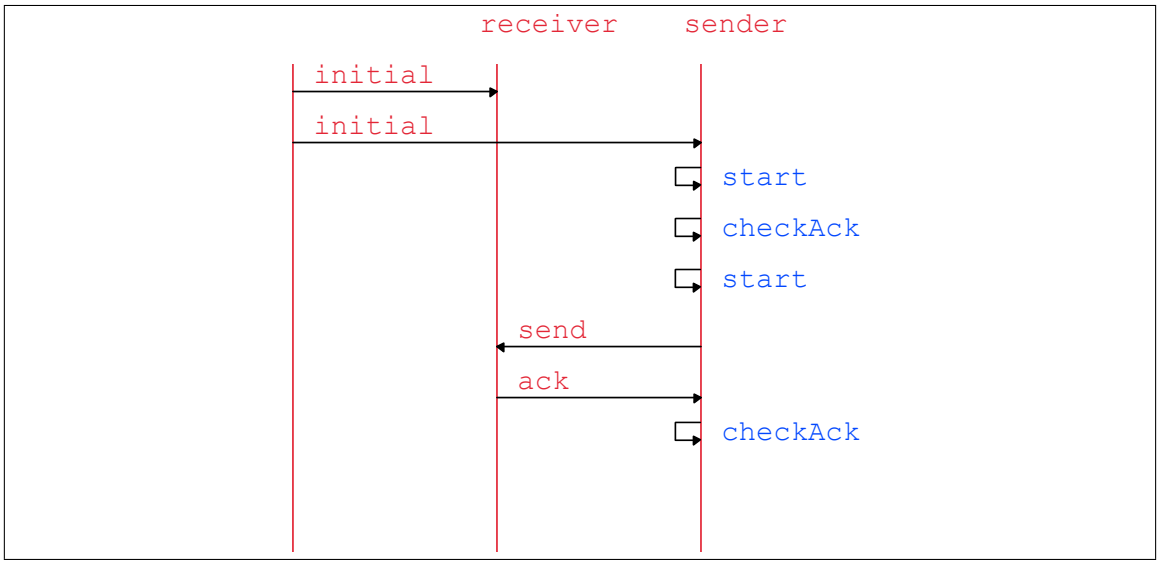


Figure A.3: Sequence diagram of a run of the simple communication protocol where the first send messages is dropped and the sender agent retransmits the message.

A.2 Ticket Service

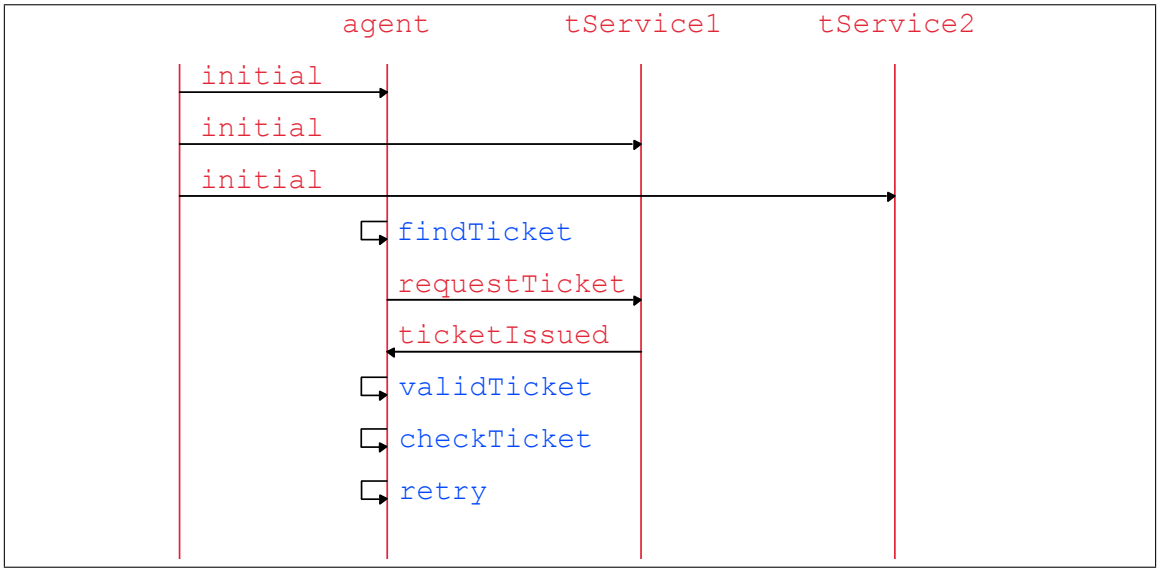


Figure A.4: Sequence diagram of a run of the ticket service where a ticket was issued for the first request before the agent checked. Note: this is only the beginning of the simulation run.

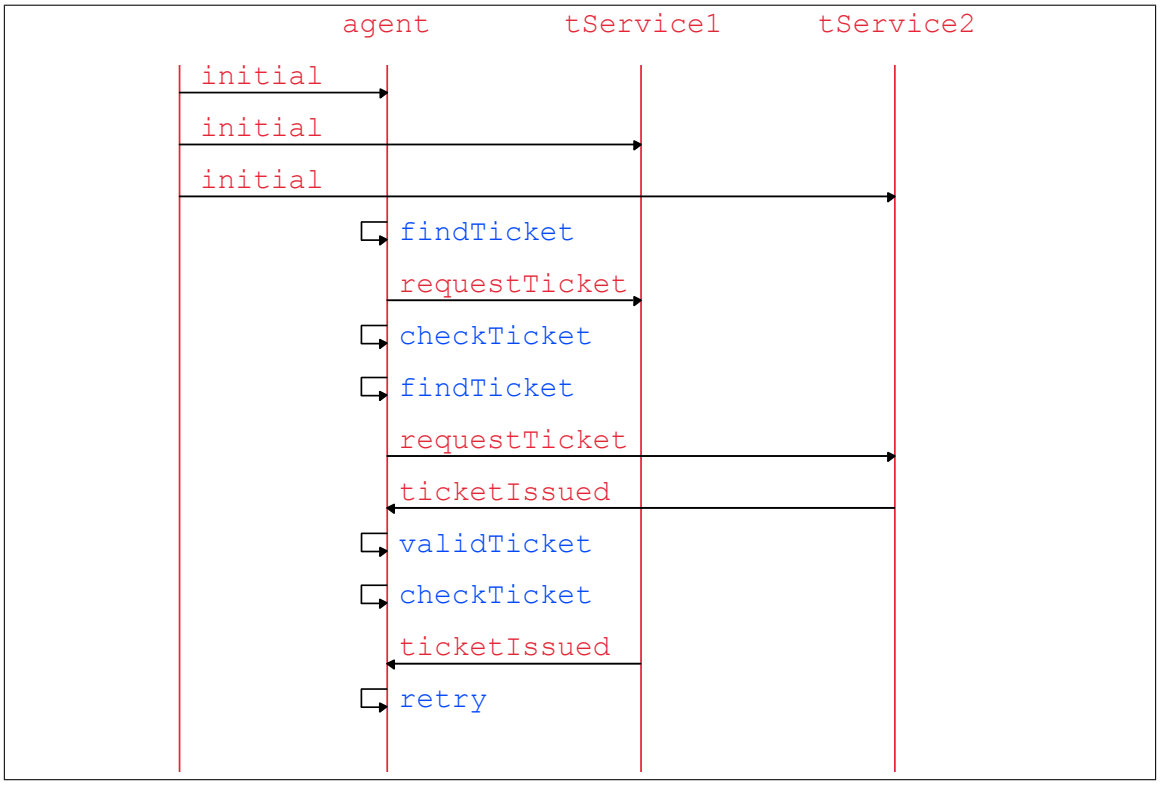


Figure A.5: Sequence diagram of a run of the ticket service where the ticket was not issued by the first ticket service before the agent checked. So the agent tried the second ticket service and got response before checking again. Notice that a ticket was issued by the first ticket service near the end. Note: this is only the beginning of the simulation run.

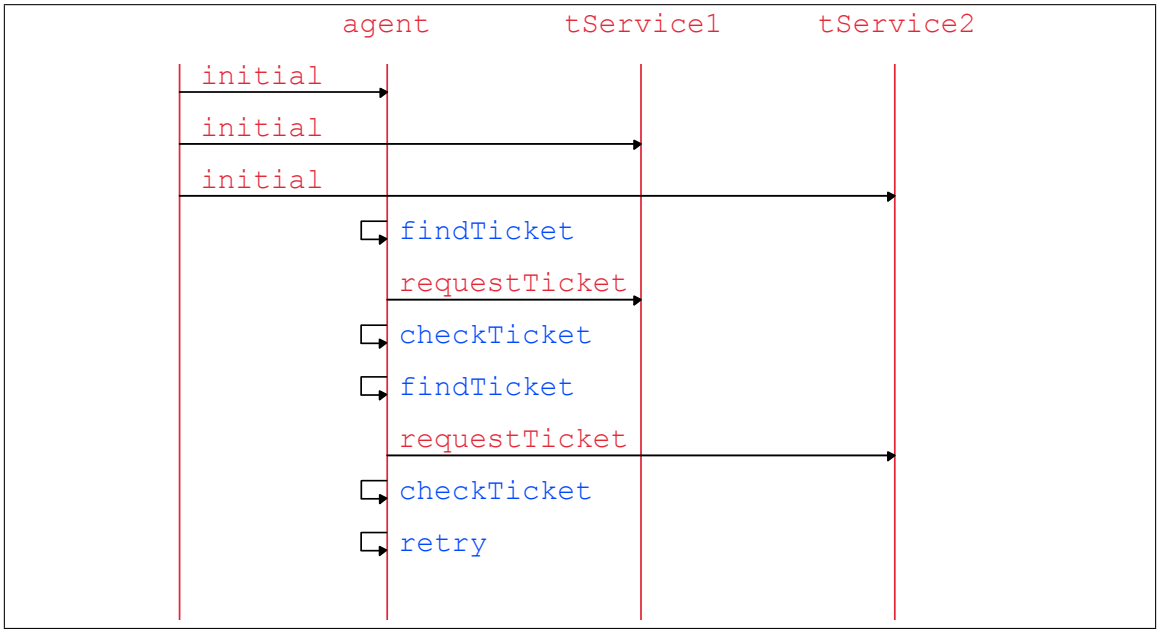


Figure A.6: Sequence diagram of a run of the ticket service where no ticket was issued by either ticket service in the first try. Note: this is only the beginning of the simulation run.

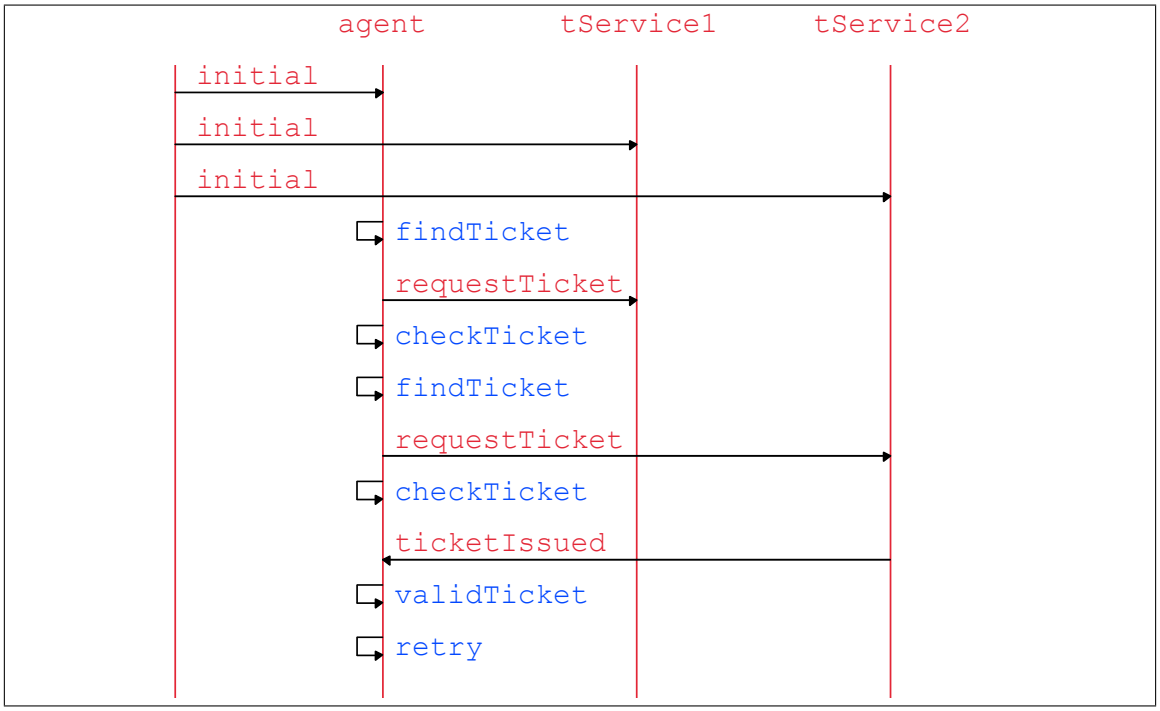


Figure A.7: Sequence diagram of a run of the ticket service where we can see the flaw in the model, the ticket issued is accepted even though the agent has already checked for the ticket. Note: this is only the beginning of the simulation run.

A.3 Sensor Network

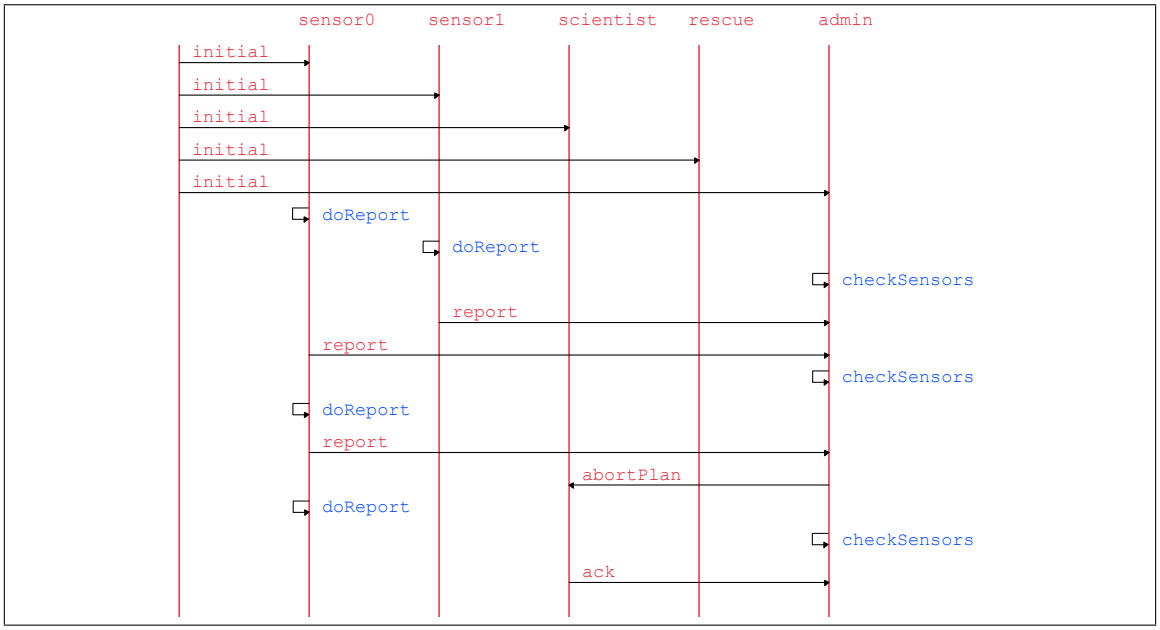


Figure A.8: Sequence diagram of a run of the sensor network where the scientist acknowledges the message. Note: this is only the beginning of the simulation run.

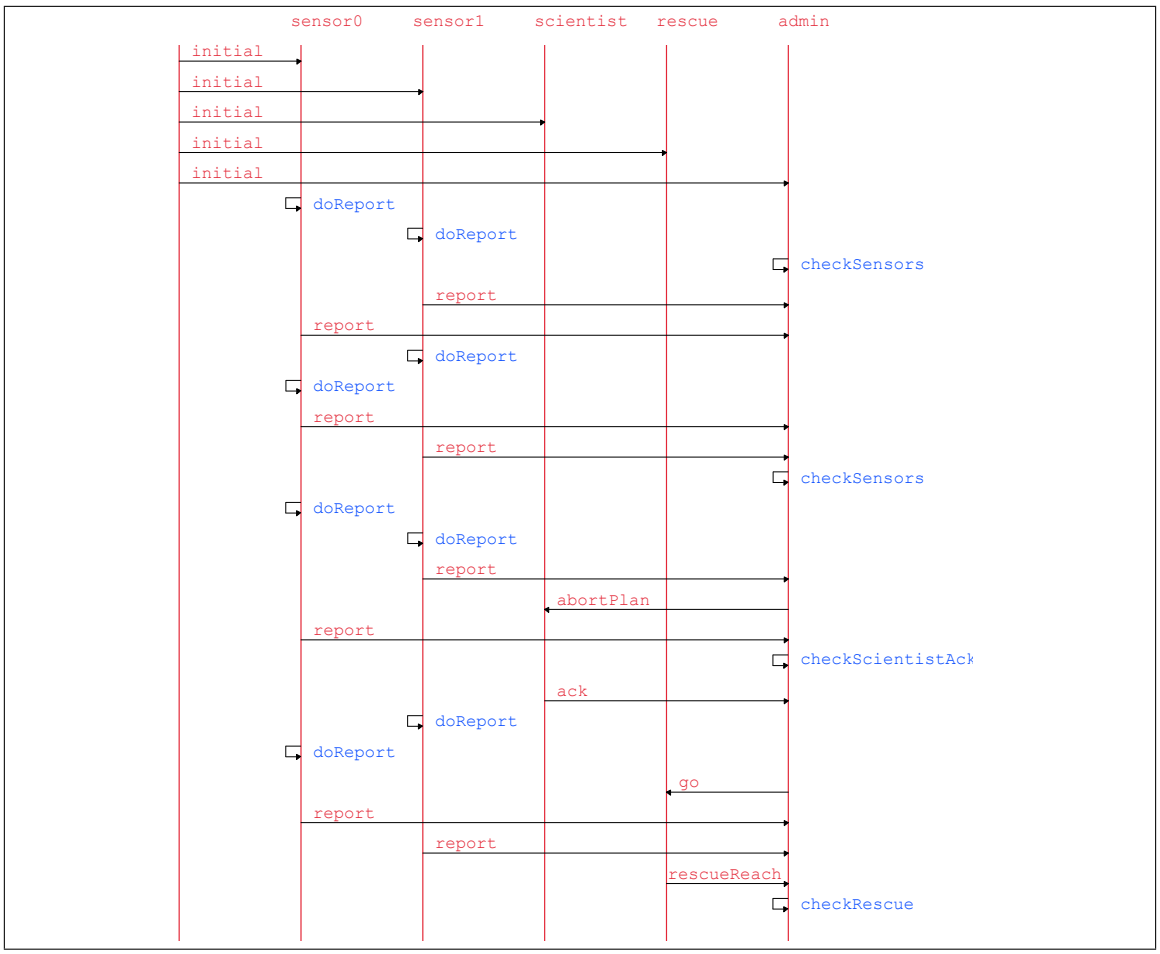


Figure A.9: Sequence diagram of a run of the sensor network where the scientist is rescued in time. Note: this is only the beginning of the simulation run.

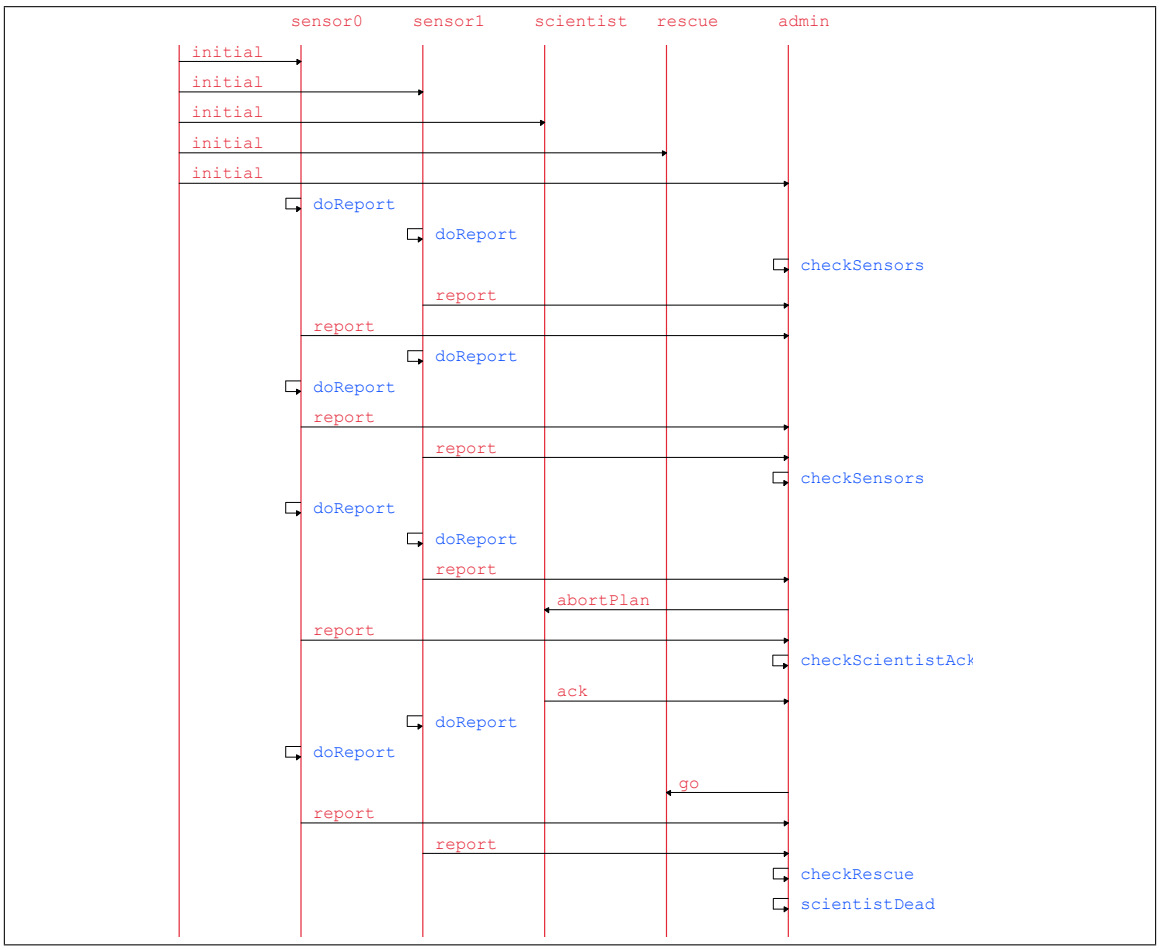


Figure A.10: Sequence diagram of a run of the sensor network where the scientist dies. Note: this is only the beginning of the simulation run.

A.4 Multi Flight Booking

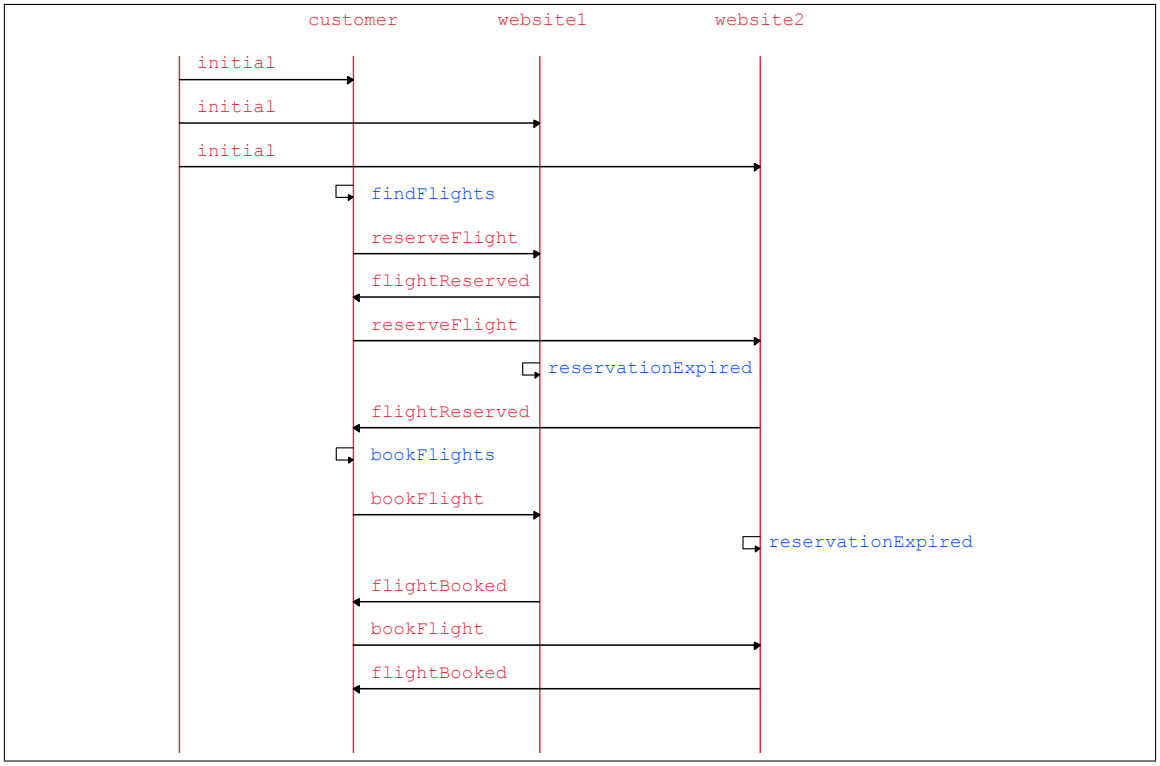


Figure A.11: Sequence diagram of a run of the multi flight booking where no flight is successfully booked.

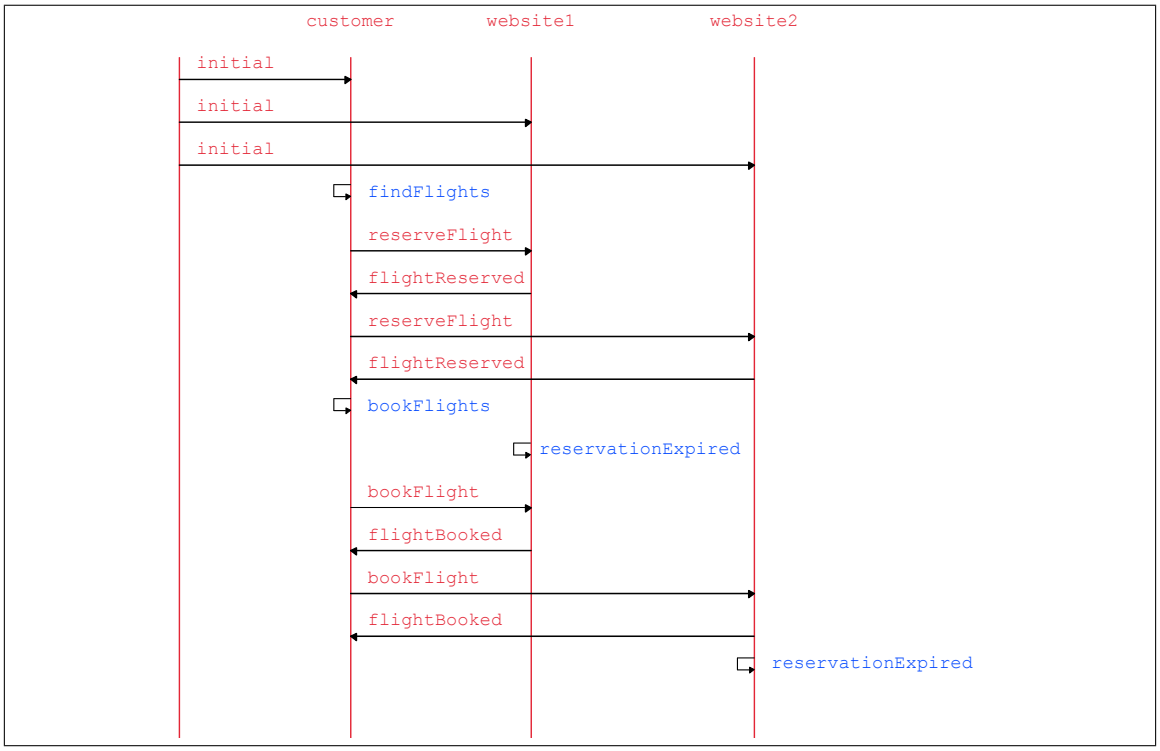


Figure A.12: Sequence diagram of a run of the multi flight booking where one flight is successfully booked.

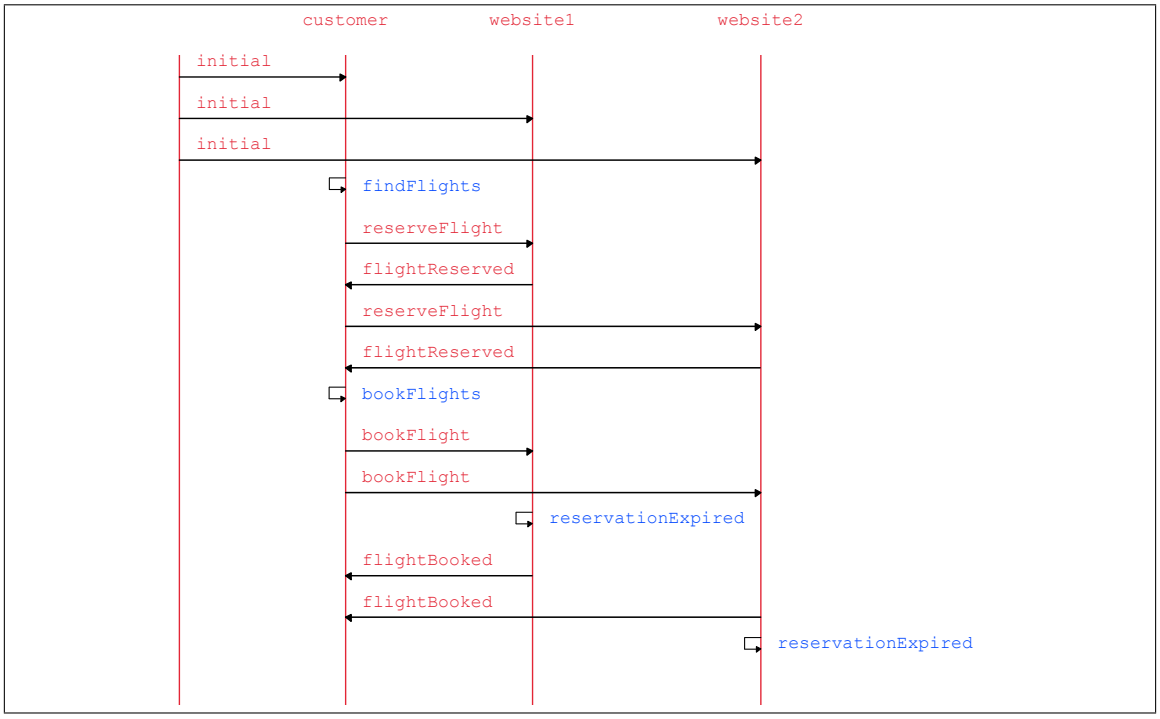


Figure A.13: Sequence diagram of a run of the multi flight booking where both flights are successfully booked.

Appendix B

Revised Models

B.1 Ticket Service

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

reactiveclass Agent {
  knownrebecs { TicketService ts1; TicketService ts2; }

  statevars { int attemptCount; boolean ticketIssued; int token; }

  msgsrv initial() { self.findTicket(ts1); } // initialize system, check 1st ticket service

  msgsrv findTicket(TicketService ts) {
    attemptCount += 1;
    ts.requestTicket(token) deadline(requestDeadline); // send request to the TicketService
    self.checkTicket() after(checkIssuedPeriod); // check for issued ticket
  }

  msgsrv validTicket(int tok) {
    //Event for TeProp
  }

  msgsrv ticketIssued(int tok) {
    if (token == tok) {
        ticketIssued = true;
        self.validTicket(tok);
    }
  }

  msgsrv checkTicket() {
    token += 1;
    if (!ticketIssued && attemptCount == 1) {      // no ticket from 1st service,
      self.findTicket(ts2);                        // try the second TicketService
    } else if (!ticketIssued && attemptCount == 2) { // no ticket from 2nd service,
      self.retry() after(retryRequestPeriod);      // restart from the first TicketService
    } else if (ticketIssued) {                     // the second TicketService replied,
      ticketIssued = false;
      self.retry() after(newRequestPeriod);        // new request for a customer
    }
  }
  msgsrv retry() {
    attemptCount = 0;
    self.findTicket(ts1);                          // restart from the first TicketService
  }
```

```
42 }
43
44 reactiveclass TicketService {
45   knownrebecs { Agent a; }
46   msgsrv initial() { }
47   msgsrv requestTicket(int token) {
48     int wait = ?(serviceTime1,serviceTime2);  // the ticket service sends the reply
49     delay(wait);                              // after a non-determinstic delay of
50     a.ticketIssued(token);                    // either serviceTime1 or serviceTime2
51   }
52 }
53
54 main {
55   Agent a(ts1, ts2):();              // instantiate agent, with two known rebecs
56   TicketService ts1(a):();           // instantiate 1st and 2nd ticket services, with
57   TicketService ts2(a):();           // the agent as their known rebecs
58 }
```

Listing B.1: Timed Rebeca code for the revised ticket service model. This version fixes the flaw discussed in Section 5.5 by incrementing the *token* in the *checkTicket* message server.

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