



# **Performance Evaluation and Model Checking of Probabilistic Real-time Actors**

**Ali Jafari**

Doctor of Philosophy

April 2016

School of Computer Science

Reykjavík University

**Ph.D. Dissertation**





# Performance Evaluation and Model Checking of Probabilistic Real-time Actors

by

Ali Jafari

Dissertation submitted to the School of Computer Science  
at Reykjavík University in partial fulfillment  
of the requirements for the degree of  
**Doctor of Philosophy**

April 2016

Thesis Committee:

Marjan Sirjani, Supervisor  
Professor, Reykjavík University, Iceland

Holger Hermanns,  
Professor, Saarland University, Germany

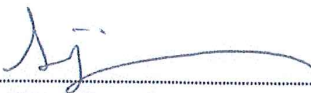
Carolyn Talcott,  
Professor, SRI International, USA


Wan Fokkink, Examiner  
Professor, VU University Amsterdam, Netherlands

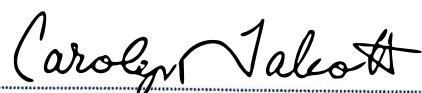
Copyright  
Ali Jafari  
April 2016

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this Dissertation entitled **Performance Evaluation and Model Checking of Probabilistic Real-time Actors** submitted by **Ali Jafari** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy (Ph.D.) in Computer Science**

.....  
date

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

  
.....  
Holger Hermanns,  
Professor, Saarland University, Germany

  
.....  
Carolyn Talcott,  
Professor, SRI International, USA

  
.....  
Wan Fokkink, Examiner  
Professor, VU University Amsterdam, Netherlands



The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this Dissertation entitled **Performance Evaluation and Model Checking of Probabilistic Real-time Actors** and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the Dissertation, and except as herein before provided, neither the Dissertation nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatsoever without the author's prior written permission.

.....  
date

.....  
Ali Jafari  
Doctor of Philosophy



# Performance Evaluation and Model Checking of Probabilistic Real-time Actors

Ali Jafari

April 2016

## Abstract

This dissertation is composed of two parts. In the first part, performance evaluation and verification of safety properties are provided for real-time actors. Recently, the actor-based language, Timed Rebeca, was introduced to model distributed and asynchronous systems with timing constraints and message passing communication. A toolset was developed for automated translation of Timed Rebeca models to Erlang [1]. The translated code can be executed using a timed extension of McErlang for model checking and simulation. In the first part of this dissertation, we induce a new toolset that provides statistical model checking of Timed Rebeca models. Using statistical model checking, we are now able to verify larger models against safety properties comparing to McErlang model checking. We examine the typical case studies of elevators and ticket service to show the efficiency of statistical model checking and applicability of our toolset.

In the second part of this dissertation, we enhance our modeling ability and cover more properties by performance evaluation and model checking of probabilistic real-time actors. Distributed systems exhibit probabilistic and nondeterministic behaviors and may have time constraints. Probabilistic Timed Rebeca (PTRRebeca) is introduced as a timed and probabilistic actor-based language for modeling distributed real-time systems with asynchronous message passing. The semantics of PTRRebeca is a Timed Markov Decision Process (TMDP). We provide SOS rules for PTRRebeca, and develop two toolsets for analyzing PTRRebeca models. The first toolset automatically generates a TMDP model from a PTRRebeca model in the form of the input language of the PRISM model checker. We use PRISM for performance analysis of PTRRebeca models against expected reachability and probabilistic reachability properties. Additionally, we develop another toolset to automatically generate a Markov Automaton from a PTRRebeca model in the form of the input language of the Interactive Markov Chain Analyzer (IMCA). The IMCA can be used as the back-end model checker for performance analysis of PTRRebeca models against expected reachability and probabilistic reachability properties. We present the needed time for the analysis of different case studies using PRISM-based and IMCA-based approaches. The IMCA-based approach needs considerably less time, and so has the ability of analyzing significantly larger models. We show the applicability of both approaches and the efficiency of our tools by analyzing a few case studies and experimental results.

# Mat á frammistöðu og athugun á líkönum í líkindafræðilegum rauntíma leikurum

Ali Jafari

apríl 2016

## Útdráttur

Þessi ritgerð er tvískipt. Í fyrri hlutanum er farið í mat og sannprófun á eiginleikum öryggis í rauntímalíkönum. Fyrir stuttu síðan var leikendabyggða málið, Timed Rebeca, notað við líkana dreifingu og ósamstillt kerfi með tímastillingu og samskipti í skilaboðum. Búið var til verkfærasett fyrir sjálfvirka þýðingu á Timed Rebeca líkөн yfir í Erlang. Hægt er að nota þýdda kóðann með því að nota tímastillta framlengingu af McErlang fyrir líkanaprófun og hermun. Í fyrri hluta þessarar ritgerðar, ætlum við að kynna verkfærasettið sem veitir tölfræðilega prófun á líkөн á Timed Rebeca líkөн. Með því að nota tölfræðileg próf á líkөн er núna hægt að sannreyna stærri líkөн eins og í öryggiskröfum McErlang. Við rannsökum dæmigerðar ferilsathuganir af lyftum og miðasölu til að sýna fram á skilvirkni tölfræðilegra líkana og beitingu verkfærasettsins okkar.

Í seinni hluta þessarar ritgerðar aukum við við getu líkanagerðarinnar og við náum yfir fleiri eiginleika með mati á framkvæmd og prófunum á líkönum á líkinda rauntíma leikara. Dreifð kerfi sýna líkindi og brigðgenga hegðun sem kunna að hafa tímamörk. Probabilistic Timed Rebeca (PTRebeca) er kynnt sem tímastillt og líkinda leikara-byggt mál líkindadreifðra rauntímakerfa með ósamstillta sendingu skilaboða. Merkingarfræði PTRebeca er Timed Markov Decision Process (TMDP). Við verðum með SOS reglur fyrir PTRebeca, og þróum tvö verkfærasett til að greina PTRebeca líkөн.

*I dedicate this to my parents and my wife.*



# Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Prof. Marjan Sirjani for her patience, for getting me interested in the field, and for giving great guidance and useful advice in all the time of my research. Marjan has also helped me with academic issues that I encountered in my PhD studies. Speaking plainly, this thesis would not have been possible without her reliable support.

I wish to thank my thesis committee: Professor Holger Hermanns, Professor Carolyn Talcott, and Professor Wan Fokkink for their patience and valuable comments which improved my thesis a lot.

I would like to specially thank Prof. Holger Hermanns for his technical guidance and support. I am honored to meet Holger in Iran at FSEN conference, and it was the time I got enough interest to continue my research in this field, and learned a lot. I stayed at Saarland University for one week in November 2014, and Holger kindly spent a lot of time to give insightful comments and guidance about my research, especially the part related to probabilistic model checking.

Special thanks to Ehsan Khamespanah for the beneficial collaborations we have had throughout my PhD studies. He put a great deal of time and we discussed many technical details of my work together. I would also like to thank Haukur Kristinnsson who contributed substantially to the ideas in the analysis of TRebeca language. The most part of TRebeca part was performed in the context of his master thesis. We had fruitful collaborations when doing research on TRebeca language. I would also like to thank my friend Alireza Hosseini for useful discussions on mathematical issues that I needed to understand deeply.

Over four years of my stay in Iceland, I have always been most grateful for the support of my entire family: Fahimeh, Abbas, Iman, Sanaz, and particularly my parents. I am always indebted to my parents for more than I could ever express. I must also thank Hanieh, my beloved wife, for her patience, love and support.

This work would not have been possible without the help of many people. They are almost all of my friends, many faculty members, and most of employees at the department of computer science of Reykjavik University. They made four years of my time in Iceland joyful. I'm grateful to all of them because of their friendship, advice, and support. In particular, I would like to thank my friends: Ehsan Pasha, Mahmoud Hassan, Narges Emami, Reza Fazeli, Pedram Ghamisi, Behnoud, Matteo Cimini, Georgiana Caltais, Younes, Aysan, and Koosha Paridel. Koosha, we had great time at ESN feasts and skiing at Akureyri.

The work on this dissertation was supported by the project “Timed Asynchronous Reactive Objects in Distributed Systems: TARO” (nr. 110020021) of the Icelandic Research Fund.

# Contents

<b>Acknowledgements</b>	<b>xiii</b>
<b>Contents</b>	<b>xv</b>
<b>List of Figures</b>	<b>xvii</b>
<b>List of Tables</b>	<b>xix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Actor-based Modeling Languages . . . . .	1
1.2 Different Aspects of System Functionality . . . . .	3
1.3 Motivation and Contributions . . . . .	5
1.4 Related Work . . . . .	7
<b>2 Background</b>	<b>9</b>
<b>3 Performance Evaluation of Timed Rebeca</b>	<b>15</b>
3.1 Mapping Timed Rebeca Models to Erlang Programs . . . . .	17
3.1.1 Handling Time in Erlang . . . . .	17
3.1.2 Timed Semantics of Erlang in McErlang . . . . .	18
3.1.3 Adapting Timed Rebeca with Timed Semantics of McErlang . .	19
3.1.4 New Extensions of Timed Rebeca Language . . . . .	21
3.2 Model Checking of Timed Rebeca Models Using McErlang Monitors . .	22
3.2.1 Checking Safety Properties . . . . .	22
3.2.1.1 Defining Safety Monitors . . . . .	22
3.3 Statistical Model Checking of Timed Rebeca Models . . . . .	25
3.3.1 Optimal Monte Carlo Estimation . . . . .	26
3.3.2 Model Checking Algorithm . . . . .	26
3.4 Performance Evaluation of Timed Rebeca Models . . . . .	29
3.4.1 Performance Evaluation Toolset . . . . .	29
3.4.2 Checkpoints analysis in Simulation . . . . .	30
3.4.2.1 Paired-checkpoint Analysis . . . . .	31
3.4.2.2 Checkpoint Analysis . . . . .	31
3.4.3 Confidence Interval . . . . .	32
3.5 Case Studies and Experimental Results . . . . .	32
3.5.1 Ticket Service System . . . . .	33
3.5.1.1 Model Checking Using McErlang Monitors . . . . .	33
3.5.1.2 Statistical Model Checking . . . . .	34
3.5.1.3 Performance Evaluation . . . . .	36
3.5.2 The Elevator System . . . . .	37

3.5.2.1	Timed Rebeca Model . . . . .	37
3.5.2.2	Model Checking Using McErlang Monitors . . . . .	40
3.5.2.3	Statistical Model Checking . . . . .	40
3.5.2.4	Performance Evaluation . . . . .	42
3.6	Related Work . . . . .	46
<b>4</b>	<b>Probabilistic Timed Rebeca: An Actor-based Modeling Language</b>	<b>49</b>
4.1	Probabilistic Timed Rebeca . . . . .	49
4.2	Semantics of Probabilistic Timed Rebeca . . . . .	51
4.3	Structural Operational Semantics of PRebeca . . . . .	54
<b>5</b>	<b>Performance Analysis of PRebeca Models</b>	<b>61</b>
5.1	Performance Analysis of PRebeca Models Using PRISM . . . . .	63
5.1.1	Analysis of Probabilistic Timed Rebeca based on TMDP . . . . .	64
5.1.1.1	Performance Analysis of Ticket Service . . . . .	65
5.1.1.2	Performance Analysis of Faulty Ticket Service . . . . .	66
5.1.1.3	Performance Analysis of a Toxic Gas Sensing System . . . . .	67
5.1.2	Parallel Composition Approach for Probabilistic Timed Rebeca . . . . .	71
5.1.2.1	Mapping from a PRebeca Model to PTA. . . . .	72
5.1.3	Comparison of Parallel Composition Approach with TMDP Semantics . . . . .	75
5.2	Performance Analysis of PRebeca Models Using IMCA . . . . .	75
5.2.1	Preliminaries . . . . .	76
5.2.2	Expected Time Reachability in TMDP . . . . .	78
5.2.3	Expected Reward Reachability in TMDP . . . . .	79
5.2.4	The Toolset and Case Studies . . . . .	81
5.2.4.1	Performance Analysis of a Toxic Gas Sensing System . . . . .	82
5.2.4.2	Performance Analysis of Network on Chip . . . . .	83
5.3	Comparing the PRISM-based and IMCA-based Approaches . . . . .	89
5.4	Related Work . . . . .	90
<b>6</b>	<b>Conclusions and Future Work</b>	<b>93</b>
	<b>Bibliography</b>	<b>97</b>
<b>A</b>	<b>Pseudocode of Policies</b>	<b>109</b>
<b>B</b>	<b>Proofs of Theorems</b>	<b>113</b>

# List of Figures

2.1	Abstract syntax of Rebeca and Timed Rebeca . . . . .	10
3.1	Stopping Rule Algorithm . . . . .	26
3.2	Model checking algorithm . . . . .	28
3.3	Architecture of the analysis toolset. . . . .	30
3.4	Event graph of the ticket service model. . . . .	33
3.5	The distribution of issued tickets between ticket services for settings 4 and 5. . . . .	36
3.6	The distribution of issued tickets between ticket services for settings 6 and 7. . . . .	37
3.7	Event graph of the centralized elevator system. . . . .	38
4.1	Abstract syntax of PRebeca . . . . .	50
4.2	SOS rules for the execution of statements of PRebeca. . . . .	57
5.1	The min/max probabilities of scientist death when using PRISM-based approach . . . . .	70
5.2	The rebec-behavior PTA of the Customer reactive class . . . . .	73
5.3	The rebec-behavior PTA of the TicketService reactive class . . . . .	73
5.4	The rebec-behavior PTA of the Agent reactive class . . . . .	73
5.5	PTA of rebec-bag for a rebec . . . . .	74
5.6	PTA of After-handler . . . . .	74
5.7	The architectural overview of the analyzer of PRebeca models . . . . .	82
5.8	The min/max probabilities of scientist death when using IMCA-based approach . . . . .	83
5.9	The $4 \times 4$ ASPIN model: The traffic in experiment 1. . . . .	86
5.10	Experiment 1: the min/max expected latency for different scenarios. . . . .	87
5.11	The $4 \times 4$ ASPIN model: The traffic in experiment 2. . . . .	87
5.12	Experiment 2: the min/max expected latency for different scenarios. . . . .	88
5.13	The $4 \times 4$ ASPIN model: The traffic in experiment 3. . . . .	88
5.14	Experiment 3: the min/max expected latency for different scenarios. . . . .	88



# List of Tables

3.1	Abstract mapping of Timed Rebeca extensions to Erlang . . . . .	21
3.2	Verification results for ticket service . . . . .	34
3.3	Statistical MC for ticket service model-setting 4 . . . . .	35
3.4	Statistical MC for ticket service model-setting 1 . . . . .	35
3.5	Paired-checkpoint evaluation for ticket service . . . . .	36
3.6	Safety verification results for the elevator system . . . . .	40
3.7	Statistical MC results of the elevator system- checking elevator1 location $\leq 10$ . . . . .	41
3.8	Statistical MC results of the elevator system- checking elevator2 location $\leq 10$ . . . . .	42
3.9	Statistical MC results of the elevator system- checking elevator1 location $> 0$	42
3.10	Statistical MC results of the elevator system- checking elevator2 location $> 0$	42
3.11	Statistical MC results of the elevator system- checking elevator1 location is $\leq 15$ . . . . .	42
3.12	Statistical MC results of the elevator system- checking elevator1 location is $\leq 20$ . . . . .	43
3.13	Paired-checkpoint analysis of the elevator system-configuration 1 . . . . .	44
3.14	Paired-checkpoint analysis of the elevator system-configuration 2 . . . . .	44
3.15	Paired-checkpoint analysis of the elevator system-configuration 3 . . . . .	45
3.16	Paired-checkpoint analysis of the elevator system-configuration 4 . . . . .	45
3.17	Simulation results for different configurations of the elevators system. . . .	45
5.1	The time and memory needed to evaluate different case studies with PRISM- based and IMCA-based approaches. . . . .	89



# Chapter 1

## Introduction

As the number of distributed systems is growing rapidly, there is an increasing need to develop modeling and verification methods for such systems. Distributed systems are made of cooperating components in which the components are interacting via asynchronous message passing. Modeling and verification of concurrent and distributed systems takes much effort due to the behavioral and structural complexities of these systems. There is thus a need for modeling languages that match well with computational models of such systems, and are supported by tools for analyzing performance and dependability aspects of these systems. A well-established paradigm for modeling the functional behavior of distributed and asynchronous systems is the actor model.

The actor language was originally introduced by Hewitt [2] as an agent-based language for programming distributed systems, and was later developed by Agha [3]–[5] into a concurrent object-based model. Valuable work has been done on formalizing the actor model by Talcott et al. [4], [6]–[8]. In the actor model, *actors* are distributed, autonomous objects that interact via asynchronous messaging. Building on an event-driven and message-based foundation, actors provide scalability and are easy-to-grasp concurrency models.

The actor model and its extensions have been used in several domains, for example, designing embedded systems [9] and wireless sensor networks [10]. With the growth of cloud computing, web services, and multi-core architectures, programming using the actor model has become increasingly relevant [11]–[13]. Compared to mathematical modeling languages, like process algebras, actors are more natural for designers, software engineers, modelers and programmers. Compared to process-oriented models, like Petri nets, the actor model has the advantages of an object-based language, like encapsulation of data and process, and more decoupled modules. Moreover, the formal semantics of actor-based languages builds a firm foundation for formal analysis and verification [14].

### 1.1 Actor-based Modeling Languages

It is common to observe real-time behaviors in distributed systems. Different modeling formalisms have been proposed for design and analysis of real-time systems. Timed automata [15] and TCCS [16] are examples of such formalisms. To analyze real-time systems, model checking of these systems against properties of timed temporal logics, which can refer to the time elapsed along system behaviors, has been studied extensively in, for example, the context of timed automata [17].

UPPAAL [18] and real-time Maude [19] are two well-known modeling languages for the analysis of real-time systems. UPPAAL is an integrated tool for modeling and analysis of systems modeled as networks of timed automata. UPPAAL is the most well-known model checker for real-time systems. Real-time Maude is a language which is based on rewriting logic. The supporting tool provides a wide range of analysis techniques.

Apart from the well-known modeling formalisms, high level modeling languages have been adapted for real-time requirements. Actor-based modeling languages as an example of such languages are extended with timing features to address the functional behaviors of actors and the timing constraints on patterns of actor invocations. In the following paragraphs, we first describe some actor-based languages and their timed extensions. Then the Rebeca language and its timed extension, namely Timed Rebeca, are compared with these languages. The Timed Rebeca language and its probabilistic extension are our focus in this thesis.

The Creol language is an object-oriented language in which asynchronous method calls are taken as the communication primitives for concurrent objects. In the semantics of Creol, asynchronous method calls are encoded using asynchronous message passing. The operational semantics is written in an actor-based style using rewriting logic. Maude has been used as an underlying simulation platform for Creol models [20]. Timed Creol is proposed as a timed extension of Creol in [21] for which the Creol's operational semantics is extended with a notion of discrete time. Time is modeled by a global clock or equivalently, local clocks which evolve with the same rate. In [22], timed Creol has been also extended with a notion of deployment component which is parametric in its concurrent resources per time interval and the operational semantics of object execution on deployment components is formalized. Based on this formalization, Maude is used to validate resource requirements that are needed to maintain the timed behavior of concurrent objects deployed with restricted resources.

The Abstract Behavioral Specification language (ABS) is an object-oriented, concurrent modeling language [23]. The concurrency model of ABS generalizes the concurrency model of Creol from single concurrent objects to concurrent object groups. ABS code is fully executable which is supported by a simulator as well as several code generation back-ends for Java, Haskell, and Erlang. The Real-time ABS language is a timed extension of ABS with a formal semantics and a Java-like syntax [24]. The authors in [25] present a simple and flexible approach to integrating deployment architectures and resource consumption into executable object-oriented models.

RT-synchronizer is proposed as an actor-based and high-level programming language for specifying real-time constraints between objects in a concurrent distributed system [26]. In this language, the specification of an object's functional behavior and the timing constraints imposed on it are separated. This separation can simplify the design, implementation, and reasoning of real-time distributed systems.

*Reactive Objects Language, Rebeca* [27]–[29], is an operational interpretation of the actor model with formal semantics and model checking tools. To the best of our knowledge, Rebeca is the first attempt to provide compositional verification and model checking support for an imperative actor-based language. Sirjani and her research group defined the language Rebeca and its formal semantics, developed its model checking tools, and provided a compositional verification theory and abstraction techniques. They have been actively and successfully investigating specialized reduction techniques for formal verification of Rebeca models, namely, symmetry, partial order, and slicing, that are all based on the formal semantics of the language [30]–[37].

Timed Rebeca [38] was proposed as an extension of Rebeca for modeling actor-based distributed and real-time systems. Timed Rebeca models can be simulated using McErlang as a first implementation of Timed Rebeca. In Timed Rebeca, timing primitives were added to Rebeca to specify both computational and network delay, and assign a deadline for serving a request. Recently, Floating Time Transition Systems (FTTS) were introduced to significantly reduce the state space induced when model checking Timed Rebeca models [39]. Checks for absence of deadlock freedom and schedulability analysis of Timed Rebeca models can be performed using FTTS.

**Comparing Rebeca with Actor-based Modeling Languages** While in Creol and its descendant, ABS, the focus has been on different modeling features, for Rebeca the core of the language is kept simple and adding any complexity is avoided. The focus has been on the analysis and formal verification of Rebeca and its extensions. While RT-synchronizer is an abstraction mechanism for the declarative specification of timing constraints over groups of actors, Timed Rebeca allows us to work at a lower level of abstraction. Using Timed Rebeca, a modeler can easily capture the functional features of a system, together with the timing constraints for both computation and network latencies, and analyze the model from various points of view [40]. In Section 3.6, we compare the Timed Rebeca language and its analysis techniques with more modeling languages and their toolsets.

Since its introduction, Timed Rebeca has been used in different areas. Examples include the analysis of different routing algorithms and scheduling policies in NoC (Network on Chip) designs [41], [42], as well as schedulability analysis of distributed real-time sensor network applications [43], more specifically a real-time continuous sensing application for structural health monitoring in [44]. An ongoing project evaluates different dispatching policies in compute clouds, facing priorities and deadlines in Mapreduce clusters, inspired by [45]. In analyzing all the above mentioned applications, we observed the need for modeling probabilistic behaviors. In an earlier work, pRebeca has been proposed as an extension of Rebeca to model probabilistic systems [46]. However, pRebeca does not support the time features. In this work, we introduce a probabilistic version of Timed Rebeca, for which the model checking approach is used for performance evaluation and functional correctness. The benefits of combining model checking and performance evaluation are elaborated upon in Section 1.2.

## 1.2 Different Aspects of System Functionality

There are two types of questions about computer systems to be answered by engineers and scientists. The first category of questions is related to perceived performance of systems. Consider a major news website; the effect of choosing the number of servers on the waiting time of incoming requests are answered by performance evaluation techniques. One of the prominent techniques to obtain the measures of interest is mathematical analysis in which a mathematical model of the system behavior (at the desired level of detail) is typically constructed in the form of closed-form expressions. The other performance evaluation techniques involve numerical evaluation that heavily relies on methods from linear algebra, and (discrete-event) simulation techniques which are based on statistical methods. The above mentioned techniques are based on the theory of stochastic processes, most notably Markov chains [47].

In the last century, different models have been developed for performance evaluation of systems. In the late 1960s, *queuing networks* were developed for modeling of computer networks and time-sharing computer systems. In early 1980s, *stochastic Petri nets* (SPNs) were developed as a modeling formalism for parallel computers. The above models are mapped to underlying Markovian models, and are analyzed using calculus or numerical analysis.

The second category of questions is whether a system is conforming to the requirements and does not contain any flaws. Formal verification is an important research area in computer science which explores the absence of errors, or finds errors through counter examples (i.e. error traces). The key techniques in this field includes run-time verification, theorem proving, and model checking. The latter is a highly automated model-based technique assessing whether a system model, i.e., the possible system behavior, satisfies a property describing the desirable behavior. In model checking, models are nondeterministic finite-state automata which are usually generated from a high level modeling language such as Petri nets, process algebras, Promela, or Statecharts. Properties are specified in temporal logic such as CTL and LTL. Various extensions of model checking have been developed to treat time and probabilities such as model checking of a timed extension of CTL [17] and probabilistic variants of CTL [48, Ch. 10].

**Combining performance evaluation with model checking.** Developments in performance evaluation are focusing on more complex system behaviors, and the evaluation of more complex measures are needed. On the other hand, timing and probabilistic features are becoming more important in model checking. Performance evaluation and model checking have thus grown in each other's direction. A number of joint efforts has been made, resulting in a quantitative system evaluation known as stochastic model checking. Combining performance evaluation and model checking has important advantages, which are briefly explained in the next paragraphs.

The idea is to provide an integrated model for checking functional correctness and performance evaluation. Using a single model enables the user, i.e. the system architecture, to specify measures of interest in temporal logic. In this way, the properties can be specified at the same abstraction level as the modeling of the system. Mostly, systems are modeled by high-level modeling formalisms such as queuing networks, SPNs, or stochastic process algebra. Temporal logic gives the possibility to specify properties in terms of the high-level models. Additionally, the use of logics provides a high degree of expressiveness and flexibility [47]. To specify complex measures in a concise manner, nested formula can be used. Given the formal semantics of logics, the meaning of complex and nested formulas are precise.

Model checking is used as an algorithmic approach for measure evaluation. The main advantage is the use of a single computational model for evaluation of any possible measure that can be written in temporal logic. This is different from common performance and dependability evaluation where a new algorithm is developed for a new measure. The largest advantage of using model checking for performance evaluation is that all algorithmic details are hidden to the user. In common performance analysis approach, expert knowledge on numerical analysis techniques for the stochastic process is needed. It is worth mentioning that stochastic model checking provides the functional correctness of the model and validation of the measures of interest at the same time.

When using performance evaluation techniques such as simulation or numerical evaluation, modeling formalisms like SPNs and queuing networks are full stochastic models. In the above methods, nondeterministic choices in the model are resolved probabilistically, e.g. by assigning probabilities to the choices, which yields inaccurate results. Nondeterminism is inherent in many applications which implies that the underlying model is not a stochastic process. Stochastic models with nondeterminism are called stochastic decision processes, for which temporal logic and model checking algorithms have been developed with relative ease for CTL [49] and LTL [50], [51]. In these models, nondeterminism is resolved by choosing one probability distribution from a set of them.

### 1.3 Motivation and Contributions

Although actors are attracting more and more attention both in academia and industry, little work has been done on timed actors and even less on analyzing actor-based models. To address the specification and verification of real-time systems, a few timed actor-based modeling languages such as RT-synchronizer [26], Timed Creol [21], Real-time ABS [24] and Timed Rebeca [38] were proposed.

In this work, the first implementation of Timed Rebeca is extended to improve its usability, and also to be able to use the timed version of McErlang which has been recently developed [52]. This version of Timed Rebeca supports performance evaluation and verification of safety properties. For example, we are able to evaluate the following measures: “the max/min/average waiting time in the queue”, and “the max/min/average response time to requests”. To obtain the aforementioned measures of interest, McErlang [53] is used as a simulation tool. To verify safety properties of a Timed Rebeca model, McErlang is used as a back-end model checker. As state space explosion is an inevitable problem in model checking, for large Timed Rebeca models we face the state space explosion using this approach. To deal with this problem, we also provide statistical model checking of Timed Rebeca models, as an alternative approach to avoid an exhaustive exploration of the state space of the model. Therefore, we are able to verify safety properties for larger Timed Rebeca models.

In simulation and statistical model checking, we use the theories and mathematical formulas which are valid for random variables. To make this applicable to Timed Rebeca, we resolve nondeterminism caused by concurrency with uniform distributions to get random executions (traces). Although there is no guarantee that this assumption of uniform distribution is a realistic assumption in the real world, it will provide some insight about the model that can be useful from a practical point of view. Specially if we warn the modeler about our assumption and the possibility that the results may not be realistic.

We are also interested in answering questions about timing and probabilistic aspects of distributed systems. For example, “with probability 0.3, the expected response time is less than 5 time unit”. To derive this kind of probabilistic performance measures, we need to model real-time systems with probabilistic behaviors. Besides performance properties we have correctness properties like “what is the probability to fail within  $d$  time units”. Probabilistic model checking can provide answers to the above questions.

In this work, we enhance our modeling ability and cover more properties by performance evaluation and model checking of probabilistic real-time actors. For modeling a wider range of systems we need to add probability to Timed Rebeca and use

model checking for analyzing the correctness and performance measures of Probabilistic Timed Rebeca (PTRbeca) models. To the best of our knowledge, PTRbeca is the first actor-based modeling language which supports time, probability, and nondeterminism in modeling distributed systems with asynchronous message passing. We propose PTRbeca on the basis of a study of different distributed and asynchronous applications, studied to identify what is needed for modeling and analysis of those applications, relative to different probabilistic and timed probabilistic models (discrete, continuous, stochastic) proposed in the literature. In PTRbeca, time is discrete, and discrete probability distributions are used.

In Timed Rebeca, the modeler may use nondeterministic choices instead of probabilistic ones since there is no way to specify probabilistic choices. In PTRbeca, the modeler can differentiate between a probabilistic choice and a nondeterministic choice, and model checking is used for the performance evaluation of models. Since the whole state space is explored in model checking, the evaluation results are not affected by nondeterminism. In PTRbeca, if the modeler has the lack of knowledge about a choice (nondeterminism exists) and the model is too large to be handled with model checking approaches, simulation will be the only feasible analysis technique and we face the same problems about nondeterminism as we had in the simulation of Timed Rebeca models. In PTRbeca models, nondeterminism always exists because of concurrent execution of actors. At the moment, only model checking is possible for PTRbeca models.

The semantics of PTRbeca is a timed Markov decision process (TMDP). For the analysis of PTRbeca models, we develop two toolsets, each of which uses a different back-end model checker. In the first tool, the TMDP of a PTRbeca model is generated, and is converted to the input language of the PRISM [54] model checker. In the second tool, IMCA (Interactive Markov Chain Analyzer) [55] is used for performance evaluation and model checking of PTRbeca models.

The contributions of this thesis are as follows:

- Analysis of Timed Rebeca models: we present different analysis techniques for Timed Rebeca models including simulation, model checking, and statistical model checking. A corresponding toolset is developed to analyze Timed Rebeca models. We also examine a few case studies to show the applicability of the proposed approach for Timed Rebeca models.
- Probabilistic Timed Rebeca (PTRbeca): the syntax and semantics of the language are defined. We provide Structural Operational Semantics (SOS rules) for the PTRbeca language in the style of Plotkin [56].
- Analyzing PTRbeca: we build two toolsets for analyzing PTRbeca, each of them using an appropriate back-end model checker. Also, the necessary mappings to the input languages of these model checkers are provided. We use PRISM and IMCA model checkers for the analysis of PTRbeca models. The underlying stochastic model of a PTRbeca model can be in the form of a TMDP or a probabilistic timed automaton (PTA) with digital clocks. This causes two different approaches of applying PRISM to PTRbeca models. Moreover, the TMDP of a PTRbeca model can be converted to a Markov Automaton (MA) [57]. The resulting MA can be analyzed using stochastic model checking algorithms implemented in IMCA.

- Modeling and analyzing case studies: a few case studies are selected, and we use PTRebeca for modeling and the supporting tools for performance evaluation and model checking of the case studies. The experimental results show the applicability of the PTRebeca language and the efficiency of the designated toolsets.

The rest of this dissertation is organized as follows. We describe the Rebeca and Timed Rebeca languages and their syntax in Chapter 2. Modeling improvements of Timed Rebeca language and developed analysis techniques for Timed Rebeca models are explained in Chapter 3. The subjects of this chapter were published in [58] and in the Journal of Computer Languages, Systems and Structures [59]. The PTRebeca language, the syntax and the semantics, is introduced in Chapter 4. The analysis of PTRebeca models is explained in Chapter 5. The subjects of the last two chapters were published in [60] and in the Journal of Science of Computer Programming [61].

## 1.4 Related Work

Many systems, such as multimedia equipment, communication protocols, networks and fault-tolerant systems, exhibit probabilistic behavior. To analyze such systems, model checking approaches based on Markov chains or Markov decision processes are used [49]–[51], [62]–[64]. Some systems exhibit both probabilistic and timed behavior, leading to the development of model checking algorithms for such systems [62], [63], [65]–[71].

Probabilistic Timed Automata were proposed as an extension of Timed Automata for modeling and verification of probabilistic real-time systems [66]. There are some works on model checking algorithms for probabilistic timed automata which are applicable to probabilistic temporal logics [66], [72]. These algorithms do not work for checking performance properties such as expected-time or expected-cost. To increase the applicability of PTA for analyzing expected reachability performance measures, digital clocks (integer-valued clocks) were proposed in [73]. PTA with digital clocks are used to verify probabilistic reachability properties which check the probability of reaching a state or a group of states in a specified time bound. In [74], model checking algorithms are considered for subclasses of probabilistic timed automata with one or two clocks, and the time complexity of model checking problems is investigated.

In [73], the digital clocks approach is applied to three probabilistic real-time protocols: the IEEE 1394 root contention protocol, the backoff procedure in the IEEE 802.11 Wireless LANs, and the IPv4 link local address resolution protocol. The IEEE 802.11 Wireless LAN can be modeled in PTA using two clocks [73], and an abstract model of the IEEE 1394 root contention protocol can be modeled with one clock [75].

There are specification languages with formal semantics for describing probabilistic systems. Modest is a high-level compositional modeling language that includes features such as exception handling, dynamic parallelism and recursion [76]. Stochastic real-time systems can be specified by the Modest language. A tool namely mcpta, was developed to support model checking of PTA specified in Modest. The tool supports probabilistic and expected reachability properties by using PRISM as its back-end model checker [77].

There are several model checkers, such as PRISM [78] and CADP [79], that support model checking of probabilistic systems. PRISM provides model checking for several types of probabilistic models such as MDP, PTA, DTMC, and CTMC, as well as a

state-based modeling language to express them. In model checking, a wide range of quantitative properties can be expressed in a language that subsumes the temporal logics PCTL, CSL, LTL and PCTL\*, as well as extensions for quantitative specifications and costs/rewards.

CADP is a toolbox for verifying asynchronous concurrent systems. CADP offers a comprehensive set of functionalities covering the entire design cycle of asynchronous systems: specification, interactive simulation, rapid prototyping, verification, testing, and performance evaluation. To deal with complex systems, CADP implements a wide range of verification techniques (reachability analysis, on-the-fly verification, compositional verification, distributed verification, static analysis) and provides scripting languages for describing elaborated verification scenarios. CADP has been extended with performance evaluation capabilities, based on the Interactive Markov Chain (IMC) theory [80], [81], and Interactive Probabilistic Chain (IPC) theory [82].

In Section 5.4, we compare the PRebeca modeling language, its supporting toolset and the developed analysis techniques with various toolsets.

# Chapter 2

## Background

In this chapter, we first present Rebeca [27], [28], and then we show its extension with timing features to build Timed Rebeca [38].

**Rebeca** Rebeca is an actor-based modelling language with formal semantics that is supported by model checking tools. A Rebeca model consists of the definition of reactive classes and the instantiation part which is called *main*. The *main* part defines instances of reactive classes, called *rebecs*. The reactive class comprises three parts: known rebecs, state variables, and message server definitions.

The known rebecs of a reactive class are the destination rebecs of the messages which may be sent by the instances of the reactive class. Because of the encapsulation of actors, the state variables of an actor cannot be directly accessed by other actors. The behavior of the instances of a reactive class is determined by its message servers. The internal state of a reactive class is represented by the valuation of its state variables. Each message server has a name, a (possibly empty) list of parameters, and the message server body which includes a number of statements. The statements may be assignments, sending of messages, and selections. The syntax of Rebeca is represented in Figure 2.1.

In Rebeca, computation is event-driven, where messages can be seen as events. Each rebec takes a message from its message queue and executes the corresponding message server. Execution of a message server body takes place atomically (non-preemptively). Communication takes place by asynchronous message passing, which is non-blocking for both sender and receiver. The sender rebec sends a message to the receiver rebec and continues its work. The message is put in the message queue of the receiver. The message stays in the queue until the receiver takes and serves it. Although in theory we define no boundary for the queue length, in the supporting tools we always have a queue length that is defined by the user. The operational semantics of Rebeca is introduced in [28], to which we refer for more details.

To show the computational model of Rebeca, we represent an example of a ticket service system. Listing 2.1 shows the Rebeca model. The model consists of three reactive classes: **TicketService**, **Agent**, and **Customer**. The customer *c* sends the `requestTicket` message to the agent *a* and the agent forwards the message to the ticket service *ts*. The ticket service replies to the agent by sending a `ticketIssued` message and the agent responds to the customer by sending the issued ticket identifier.

The behaviour of a Rebeca model is defined as the parallel execution of the released messages of the rebecs. At the initialization state, a number of rebecs—defined in the *main* part—are created statically, and an initial message—specified by the same name

```

Model ::= Class* Main
Main ::= main { InstanceDcl* }
InstanceDcl ::= className rebecName(⟨rebecName⟩*) : (⟨literal⟩*);
Class ::= reactiveclass className { KnownRebecs Vars MsgSrv* }
KnownRebecs ::= knownrebecs { VarDcl* }
Vars ::= statevars { VarDcl* }
VarDcl ::= type ⟨v⟩+;
MsgSrv ::= msgsrv methodName(⟨type v⟩*) { Stmt* }
Stmt ::= v = e; | v = ?(e⟨e⟩+); | Call; | if (e) { Stmt* } [else { Stmt* } ]
Call ::= rebecName.methodName(⟨e⟩*)

```

(a) Abstract Syntax of Rebeca

```

Stmt ::= v = e; | v = ?(e⟨e⟩+); | Call; | if (e) { Stmt* } [else { Stmt* } ] | delay(v);
Call ::= rebecName.MethodName(⟨e⟩*) [after(v)] [deadline(v)]

```

(b) Changes in the syntax of Rebeca to build Timed Rebeca

Figure 2.1: (a) Abstract syntax of Rebeca. Angle brackets ⟨...⟩ are used as meta parentheses, superscript + for repetition at least once, superscript \* for repetition zero or more times, whereas using ⟨...⟩ with repetition denotes a comma separated list. Brackets [...] indicate that the text within the brackets is optional. The symbol ? shows nondet choice. Identifiers *className*, *rebecName*, *methodName*, *v*, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and *e* denotes an (arithmetic, boolean or nondet choice) expression. (b) Changes for Timed Rebeca. The timing primitives are added to *Stmt* and *Call* statements. The value of variable *v* in timing primitives is a natural number.

of its reactive class in the model– is implicitly put in their bags. The release time of the initial messages is zero. The execution of the model continues as rebecs change the values of their state variables and send messages to each other.

```

1 reactiveclass TicketService{
2   knownrebecs{
3     Agent a;
4   }
5   statevars{
6     int issueDelay;
7   }
8   TicketService( ){
9   }
10  msgsrv requestTicket(){
11    a.ticketIssued(1);
12  }
13 }
14 reactiveclass Agent{
15   knownrebecs{
16     TicketService ts;
17     Customer c;
18   }
19   Agent( ){
20   }
21   msgsrv requestTicket(){
22     ts.requestTicket();
23   }
24   msgsrv ticketIssued(byte id ){
25     c.ticketIssued( id );
26   }
27 }
28 reactiveclass Customer{
29   knownrebecs{
30     Agent a;
31   }
32   Customer(){
33     self.try();
34   }
35   msgsrv try(){
36     a.requestTicket();
37   }
38   msgsrv ticketIssued(byte id ){
39     self.try();
40   }
41 }
42 main{
43   Agent a( ts , c):();
44   TicketService ts (a):(3);
45   Customer c(a):();
46 }

```

Listing 2.1: Rebeca model of ticket service example.

**Timed Rebeca** The timing primitives are added to the Rebeca syntax to cover timing features that a modeler might need to address in a message-based, asynchronous and distributed setting. These features (like the computation time, or periodic events) can be common in any setting, and are explained as follows.

- **Computation time:** the time needed for a computation to take place.
- **Message delivery time:** the time needed for a message to travel between two objects, which depends on the network delay (and possibly other parameters).
- **Message expiration:** the time within which a message is still valid. The message can be a request or a reply to a request (a request being served).
- **Periods of occurrences of events:** the time periods for periodic events.

In Timed Rebeca, each actor (also called a rebec) has its own local clock, but there is also a notion of global time based on synchronized distributed clocks of all the rebecs. Instead of a message queue for each rebec, there exists a bag containing all the messages sent for each rebec. Messages that are sent to a rebec are put in its message bag together with their arrival time (called their time tag), and their deadline. Methods are executed atomically, but the passing of time during the execution of methods can be modeled. In addition, communication delay and deadline for execution of messages can be defined in the model. The timing primitives that are added to the Rebeca syntax to support these features are *delay*, *deadline*, and *after*. The descriptions of these constructs are as follows, and their syntax is shown in Figure 2.1.

- **Delay:** *delay(t)*, where *t* is a positive natural number, increases the value of the local clock of the respective rebec by the amount *t*.

- Deadline:  $r.m() \text{ deadline}(t)$ , means that the message  $m$  is sent to the rebec  $r$  and it 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:  $r.m() \text{ after}(t)$ , the message cannot be taken from the bag before  $t$  time units have passed. The *after* primitive is used to model network delays in delivering a message to its destination. Note that *after* primitive can also be used to model periodic events. If we send a message in a loop with *after*( $t$ ), this will cause having the message in the message queue every  $t$  units of time. In Timed Rebeca, loops are modeled by sending a message to itself.

The messages that are sent are put in the message bag together with their time tag and *deadline* tag. The scheduler decides which message is to be executed next, based on the time tags of the messages. The time tag of a message is the value of the local clock of the sender rebec when the message was sent, added to the value of the argument of the *after* if the message is sent with an *after*. The scheduler takes a message from the message bag, executes the corresponding message server atomically, and then takes another message. Every time the scheduler takes a message for execution, it chooses a message with the least time tag. Before the execution of the corresponding method starts, the local time of the receiver rebec is set to the maximum value between its current time and the time tag of the message [38].

```

1  env int requestDeadline, checkIssuedPeriod, 24      } else if (ticketIssued && token <
    retryRequestPeriod;                               maxIssued+1) {
2  env int newRequestPeriod, serviceTime1,           25      ticketIssued = false;
    serviceTime2;                                     26      self.retry() after(newRequestPeriod);
3  env int maxIssued; // maximum number of           27  }
    requests                                           28  }
4  reactiveclass Agent(3) {                           29
5      knownrebecs { TicketService ts1;               30      msgsrv retry() {
        TicketService ts2; }                           31          attemptCount = 0;
6      statevars { int attemptCount; boolean           32          self.findTicket(ts1);
        ticketIssued; int token; }                       33      }
7      msgsrv initial() { self.findTicket(ts1); } 34  }
8
9      msgsrv findTicket(TicketService ts) {           35
10         attemptCount = attemptCount + 1;             36  reactiveclass TicketService(3) {
11         token = token + 1;                             37      knownrebecs { Agent agent; }
12         if(token <= maxIssued) {                       38      statevars { }
13             ts.requestTicket(token)                   39      msgsrv initial() { }
14             deadline(requestDeadline);                 40
15             self.checkTicket()                         41      msgsrv requestTicket(int token) {
16             after(checkIssuedPeriod);}                 42          int wait = ?(serviceTime1,serviceTime2);
17         }                                              43          delay(wait);
18         msgsrv ticketIssued(int tok) { if (token      44          agent.ticketIssued(token);
19             == tok) ticketIssued = true; }              45      }
20         msgsrv checkTicket() {                         46  }
21             if (!ticketIssued && attemptCount == 1      47
22             && token < maxIssued+1) {                   48  main {
23                 self.findTicket(ts2);                 49      Agent agent(ts1, ts2):();
24             } else if (!ticketIssued &&                50      TicketService ts1(agent):();
                attemptCount == 2 && token <              51      TicketService ts2(agent):();
                maxIssued+1) {                             52  }
                self.retry()
                after(retryRequestPeriod);

```

Listing 2.2: Timed Rebeca model - Ticket service system.

An example of a Timed Rebeca model is shown in Listing 2.2. This is a model of a ticket service system. In the main part, the rebecs are instantiated from the reactive

classes. For each rebe, its known rebees are specified as arguments, e.g. rebees **ts1** and **ts2** are the known rebees of rebe **agent** (Line 49). The initial values of the state variables can be specified as arguments in the rebe instantiation (empty parentheses in Line 49 can be used for this purpose, otherwise the default values are used). For example, “Agent agent(ts1, ts2):(10, false, 2)” creates an agent and the values of its state variables **attemptCount**, **ticketIssued** and **token** are initialized to 10, false and 2, respectively. A reactive class has an argument of type integer denoting a user-specified upper bound for its queue size (Agent(3) in Line 4). This is necessary to prevent state space explosion in model checking.

The model in Listing 2.2 consists of two reactive classes: **Agent** and **TicketService**. The agent **a** starts by sending a message to the first ticket service **ts1** and requesting a ticket (Line 13). The message has a deadline of **requestDeadline** time units. When the message is received by the ticket service **ts1**, it issues the ticket after **serviceTime1** or **serviceTime2** units of time (Line 42-44). The issuing process is performed by sending a message back to agent **a**. After requesting a ticket to **ts1**, agent **a** sends a message to itself after **checkIssuedPeriod** time units (Line 14). This message checks whether the ticket has been issued or not. If the ticket is issued, the model continues to the next customer and requests a new ticket after **newRequestPeriod** time units (Line 26). If the ticket was not issued by **ts1**, agent **a** immediately sends a message to the second ticket service **ts2** (Line 21). This scenario is repeated recurrently.



## Chapter 3

# Performance Evaluation of Timed Rebeca

In analyzing real-time systems, performance evaluation is a complementary issue to functional verification. Therefore, analysis techniques should consider both correctness and performance measures to guarantee dependability and efficiency of systems. Different formal timed models have been proposed for modeling and verification of real-time systems. On the other hand, different approaches have been suggested for performance evaluation of real-time systems. Numerical analysis and simulation techniques that are based on statistical methods are two widely used approaches for performance evaluation. In this chapter, we provide analysis techniques and toolset for both verification of correctness and performance evaluation of real-time distributed systems with asynchronous message passing. To address the analysis of real-time distributed systems, we use Timed Rebeca which is a timed actor-based modeling language. The formal semantics of Timed Rebeca was offered using Structural Operational Semantics (SOS) rules [56].

In the first implementation of Timed Rebeca, a toolset was developed to translate Timed Rebeca models to Erlang programs [83] automatically, and McErlang [53] was used to simulate the translated Erlang program [38]. At that time, McErlang, a model checking and simulation tool for Erlang, did not support model checking of Erlang program with timing features. In the untimed version of McErlang, simulation takes place by simply executing the Erlang program, and the reason for using McErlang is the monitors provided by this tool. By using monitors one can stop the execution by observing an erroneous state or unexpected behavior in the program. It is also possible to collect the necessary data during the execution. This tool can be used to run multiple simulations for different settings of parameters in a Timed Rebeca model, and then the results of the executions can be employed to select the most appropriate values for the parameters. This version of McErlang is not efficient for larger models since the progress of time is modeled by the system time; a model with an average size takes a long time to be executed.

We extend the previous version of Timed Rebeca to improve its usability, and also to be able to use the timed version of McErlang which has been recently developed [52]. To improve the usability of Timed Rebeca, the language is extended to support a *list* data structure and the capability of calling custom functions from Erlang. This way the effort for modeling more complicated systems using Timed Rebeca is decreased. Moreover a function named *checkpoint* is added to the language to be able to provide more data to McErlang and hence get more valuable data in the analysis.

Based on the timed version of McErlang, we change the mapping of timing primitives of Timed Rebeca models to Erlang presented in [38], and we adjust the implementation of the tool accordingly. As stated in [52], during the development of McErlang with timed semantics there has been a close collaboration between the two teams. So, the timed semantics of McErlang supports the timing features of Timed Rebeca very well. Now, using the *checkpoint* functions we are able to model check and simulate Timed Rebeca models by McErlang.

The approach employed in the timed version of McErlang is inspired by Lamport's approach to real-time model checking [84]. The McErlang team used the idea of maximum-time-elapse for progress of time. The timer is increased based on the time of the occurrence of the next event, so, we have a jump to the next value for the timer instead of having a tick function to increase the timer by one. Finding the next event is not difficult in Erlang, as all the real-time computations are encountered within *receive* statements where timeouts are defined (in an optional *after* clause). Hence, simulation of Timed Rebeca models is much more efficient compared to the previous work where McErlang basically executed the Erlang programs.

We use checkpoint (user-defined) monitors and predefined monitors of McErlang for verification of safety properties. As state space explosion is an inevitable problem in (especially timed) model checking, for large Timed Rebeca models we face state space explosion using this approach. Therefore, we provide statistical model checking of Timed Rebeca models, as an alternative approach to avoid an exhaustive exploration of the state space of the model. We are thus able to verify larger Timed Rebeca models. To this end, our toolset is extended to support statistical model checking besides the verification and simulation of Timed Rebeca models. In this approach, we run multiple simulations by McErlang, and then an approximation of correctness of the model is calculated for a given safety property.

We use the simulation capability of McErlang for performance evaluation of Timed Rebeca models. The statistical methods are applied to the obtained data from different simulation runs in order to compute performance measures of the model, such as the mean response time for a request to be served. We also calculate the confidence interval to indicate the accuracy of simulation results.

To show the efficiency of our approaches, we examine the elevator case study by applying the proposed analysis techniques. In statistical model checking, we increase the number of floors to get a very large model, for which the model checking of McErlang is not applicable because of the state space explosion problem. As another case study, we consider a ticket service system. The efficiency and applicability of the statistical model checking approach depends only on the size of our models. One of the parameters showing the size of a model is the number of rebecs (actors) and the message passing among them. So, if we increase the number of rebecs greatly, a simple case study like the ticket service can imitate a complicated system.

**Contributions.** The subjects of this chapter were published in [58] and in the Journal of Computer Languages, Systems and Structures [59]. The work in this chapter, except the statistical model checking part, was performed in the context of a master thesis [1] and I was fully involved in that project.

- Modeling: a list data structure, the ability of calling custom functions from Erlang, and checkpoint functions are added to Timed Rebeca language to make the modeling of more complex systems feasible.

- Analysis: McErlang is used as a simulation and model checking tool for the performance evaluation and safety verification of Timed Rebeca models, respectively. To this end, a Timed Rebeca model is mapped to its corresponding Erlang program. Additionally, statistical model checking is provided for Timed Rebeca models to avoid the state space explosion of large Timed Rebeca models.
- Implementation: we develop a toolset to provide performance evaluation, safety verification, and statistical model checking of Timed Rebeca models. In this toolset, McErlang is used as a back-end tool for simulation and model checking of the Erlang program resulting from a Timed Rebeca model.
- Case studies: we examine two typical case studies to show the applicability of our approaches and the efficiency of the developed toolset.

The rest of the chapter is organized as follows. Considering the Timed Rebeca language presented in [38], Section 3.1 defines a new mapping for timing primitives of Timed Rebeca to Erlang while adapting to timed extensions of McErlang. It also includes new features added to the Timed Rebeca language to increase its usability. Section 3.2 explains how safety monitors in McErlang can be used to verify safety properties of Timed Rebeca models. Section 3.3 explains statistical model checking of larger Timed Rebeca models against safety properties. Section 3.4 describes the simulation of Timed Rebeca models using McErlang. The result is a dataset including useful information about system behavior to which different analysis methods can be applied. To show the results precision, we calculate the confidence interval for performance measures under study. In Section 3.5, we apply all methods proposed in the previous sections to the typical examples of the elevator system and the ticket service. Section 3.6 discusses related works.

## 3.1 Mapping Timed Rebeca Models to Erlang Programs

In this section, we explain a new mapping algorithm for Timed Rebeca models to Erlang programs while conforming to the new timed features of McErlang. Since McErlang is used as the back-end model checker and the simulation tool, this mapping is necessary. We also explain new features added to the Timed Rebeca language to make it more convenient to use. New features include checkpoint, calling custom functions, and a list data structure which are explained in more detail in Section 3.1.4.

### 3.1.1 Handling Time in Erlang

Here, we briefly explain the timed Erlang semantics introduced in [52], which will be used in the new mapping of Timed Rebeca models to Erlang. Erlang handles time with the use of **after** as a timeout clause in a **receive** statement, as Listing 3.1 shows. If a message matches any of the patterns, e.g.  $Pattern_j$ , and the corresponding guard,  $Guard_j$ , evaluates to true, the message is removed from the mailbox and evaluation continues with expression  $Expr_j$ .

The oldest message in the process mailbox is evaluated to be matched against the patterns according to the above procedure. If no pattern and guard match this message, the same procedure continues with the second oldest message, and so on. If

no pattern is matched, the process waits for at least `TimeoutValue` milliseconds to receive a matching message. This is the *minimum* amount of time that a timer elapses until the timeout happens. If the timeout occurs, the expression `TimeoutExpression` is evaluated. A zero deadline means, if no matching message is in the mailbox, the timeout can happen immediately. The atom `infinity` may be used as a time deadline to show that the timeout never happens.

```

1 receive
2   Pattern1 when Guard1 -> Expr1;
3   ...
4   PatternN when GuardN -> ExprN;
5 after
6   TimeoutValue -> TimeoutExpression
7 end

```

Listing 3.1: Erlang syntax of a receive with timeout.

### 3.1.2 Timed Semantics of Erlang in McErlang

The main changes made to McErlang to implement a timed semantics of Erlang are to record the current time in the state representation of a running program, and to modify the behaviour of the receive statement in the model checker so that the current time is considered when timeouts are handled [52].

In Listing 3.1, there is no guarantee exactly when the timeout happens after a timer has elapsed `TimeoutValue` milliseconds. In the timed semantics of Erlang, it is possible to specify the urgency of a state with the function `mce_erl:urgent(MaximumWait)`. The parameter `MaximumWait` specifies the *maximum* number of milliseconds the process can remain in the current state, if it has transitions enabled. As an example consider the code in Listing 3.2: a process is spawned and waits between 1000 and 1500 milliseconds for a message to arrive before timing out. In this example, we force the timeout to happen before 1500 milliseconds if the process does not receive a message.

```

1 spawn (fun () ->
2   mce_erl : urgent (1500),
3   receive Msg -> ok
4   after 1000 -> bad
5   end
6 end)

```

Listing 3.2: Erlang code with the urgency construct implemented in McErlang.

In McErlang with the timed Erlang semantics, a new API `mce_erl_time` is introduced to provide the definition and manipulation of timestamps. This new API has the following functions.

- `now()`: returns the current time.
- `nowRef()`: stores the current time in a clock reference.
- `was(Ref)`: returns the time stored in a clock reference.
- `forget(Ref)`: stops a stored clock reference.

Some points should be considered in using this API. The absolute values returned from calls to `now()` can not be used by the program. They can only be compared with

the previously recorded clocks, i.e., relative comparisons are permitted that show how much time has elapsed since an event happened.

### 3.1.3 Adapting Timed Rebeca with Timed Semantics of McErlang

The timed version of McErlang proposed in [52] makes the formal verification of timed programs written in Erlang programming language possible. In the timed semantics, timed actions, i.e. actions with a timeout clause, are ordered based on the timeout value, while untimed actions, i.e. actions without a timeout clause, are executed infinitely fast.

In the Timed Rebeca language, timed behaviors are defined by using the timing primitives *after*, *delay*, and *deadline*. The execution order of messages is specified based on the values of these primitives. In this section, we explain the new mapping of a Timed Rebeca model to an Erlang program according to the timed semantics of Erlang in McErlang. There are two main points to consider regarding the new mapping. Firstly, the mapping algorithm of timing features in Timed Rebeca to Erlang must be changed according to the new timed features of McErlang like timestamps and the urgency construct. Secondly, the new mapping algorithm for Timed Rebeca models should make the correct order of execution of actions possible. In the following paragraphs we explain these two points in more details.

**Mapping timing primitives of Timed Rebeca to Erlang** In the previous Timed Rebeca mapping to Erlang, function `now()` was used to obtain the current time by using the system clock [38]. Timed behaviors like sending messages with **deadline**, **after**, and **delay** statements, were implemented in terms of the system clock. In our new mapping, we use the same concepts as described in [38], but with a few important differences in the implementation. We use clock references accessible from API `mce_erl_time` to map timed actions from Timed Rebeca to Erlang. The main difference is that in the new version we use the simulation/model time and not the real system time (like when a real execution of the program is in order).

An ordinary message send in Timed Rebeca, i.e. a message send without the **after** primitive, is translated to a regular message send in Erlang, as shown in Listing 3.3. Instead of tagging the message with the local time of the sender, as we did in our previous mapping, we utilize a clock reference which is sent as a parameter to the receiver. The clock reference is obtained from calling `nowRef()` and stored in the variable `TT`. The clock can be remembered later for relative comparisons by calling `was(Ref)`. A message send also consists of some other information for the receiver such as deadline, message name, and parameters. The default value for deadline is `inf` (standing for infinity), which denotes no deadline.

```

1 messagesend(Sender, Rebec, Msg, Params, Deadline) ->
2   % Start a clock reference and save it to TT
3   TT = nowRef(),
4   spawn(fun () ->
5     % sending a message to the Rebec
6     Rebec ! {{Sender, TT, Deadline}, Msg, Params}
7   end).
```

Listing 3.3: Pseudo Erlang code for a message send in Timed Rebeca.

After receiving a message, its deadline should be checked by the receiver before processing it. The timestamp of the message is the local time of the sender when sending the message and can be remembered using the function `was(Ref)`. The local time of the receiver when receiving the message can be obtained by the function `nowRef()`. So, if the message has not expired, this condition  $deadline + was(ref) < nowRef()$  is satisfied.

In the Timed Rebeca semantics, a message with the `after(Timeunits)` statement is put in the message bag of the receiver, and it can not be taken from the bag before the specified time, i.e. `Timeunits` milliseconds, has elapsed. In the mapping to Erlang, a function is spawned and waits for `Timeunits` milliseconds before sending the message. The function is an empty receive statement with a timeout clause, and sending the message is placed in the timeout clause, as demonstrated in Listing 3.4.

```

1 messagesend(Sender, After, Rebec, Msg, Params, Deadline) ->
2   TT = nowRef(),
3   spawn(fun () ->
4     % sending the message after Timeunits
5     receive
6       after(Timeunits) ->
7         Rebec ! {{Sender, TT, Deadline}, Msg, Params}
8     end).

```

Listing 3.4: Pseudo Erlang code for a message send with the `after` primitive in Timed Rebeca.

In Timed Rebeca, the `delay(Timeunits)` statement makes the local time of a rebec advance for the specified amount of time (`Timeunits` milliseconds). In Erlang, the delay is translated to the receive statement including just a timeout value, as shown in Listing 3.5. Since there is no pattern in the receive statement, the timeout clause (after clause) will be executed after the specified time (`Timeunits` milliseconds), imitating the delay statement in Timed Rebeca. As stated in [52], the function `mce_erl:urgent(MaximumWait)` can be used to determine the urgency of a state, i.e., how long the process can stay in this state. So, we use the urgent function in the McErlang code to make the delayed process run immediately after the timeout expires.

```

1 timedelay(Timeunits) ->
2   % McErlang Urgent Delay
3   urgent(Timeunits),
4   % Delay by Timeunits
5   receive
6     after (Timeunits) -> ok
7   end.

```

Listing 3.5: Pseudo Erlang code for a delay statement in Timed Rebeca.

**Performing Timed and Untimed Actions in the Correct Sequence** In Timed Rebeca, the execution order of messages is specified with respect to the values of timing primitives `delay` and `after`. In the previous paragraph, we explained how timing primitives in Timed Rebeca are translated to Erlang code. We also explained how a message deadline in Timed Rebeca can be handled using timestamps in McErlang. To execute messages in the correct order in Erlang according to the Timed Rebeca semantics, we should take into account more considerations in Erlang:

- actions without a timeout clause (equivalent to messages without after in Timed Rebeca) should be executed infinitely fast (immediately).
- actions with a timeout clause (equivalent to delays or messages with after in Timed Rebeca) should be executed immediately after the timeout expires. The messages are ordered based on their timeout.

Using the timed extension in McErlang, we can change the way in which timed (with timeout) and untimed (without timeout) actions are treated using the function `mce_erl:urgent` (`MaximumWait`). To execute the untimed actions infinitely *fast*, the `MaximumWait` parameter is set to zero. To execute the timed actions immediately after their timeout expires, the `MaximumWait` parameter is set to the value of timeout.

### 3.1.4 New Extensions of Timed Rebeca Language

We added some capabilities to Timed Rebeca in order to increase the modeling power of the language. These additions include a list data structure, the capability of calling custom functions from the Erlang language, and checkpoints. Table 3.1 shows the syntax of the extensions and their abstract mapping to Erlang.

Checkpoint functions can be used in both simulation and model checking. They are considered as markers in the code that indicate important events. Checkpoints are also used to expose the value of variables in a Timed Rebeca model to McErlang. For simulation, a checkpoint is translated to an Erlang function, and for model checking a checkpoint is translated to a probe in Erlang.

A checkpoint has two mandatory arguments: a *label* and at least one *term*. The label is an arbitrary name which is defined by the modeler and is used to refer to the checkpoint. Note that every piece of data of any type is called a term in Erlang. So, all variables in a Timed Rebeca model are translated to terms. The terms in a checkpoint are variables that are added to the checkpoint function as its arguments. The value of terms can be retrieved during simulation or model checking in McErlang.

Timed Rebeca Syntax	Erlang / McErlang
<b>list</b> $\langle int \rangle N$ ;	→ Erlang list data type as a variable with name $N$ .
<b>erlang.</b> $func(V_1, \dots, V_n)$ ;	→ Call to the function $func$ with parameters $V_1, \dots, V_n$ .
<b>checkpoint</b> $(L, T_1(, T_2, \dots, T_n))$ ;	→ Erlang output function is used for simulation. $L$ and $T_i$ are the arguments.
<b>checkpoint</b> $(L, T_1(, T_2, \dots, T_n))$ ;	→ Erlang probe is used when model checking. $L$ and $T_i$ are its label and term respectively.

Table 3.1: Mapping of Timed Rebeca extensions to Erlang:  $func$  is the name of a function implemented in Erlang,  $L$  is a label for a checkpoint, and  $T_i$  is a term of a checkpoint (a state variable or a local variable). When doing model checking,  $T_i$  is used to define a term of the generated probe.

Another extension in Timed Rebeca language is the ability of calling custom functions in Erlang. A modeler can define a function in Erlang and then call it from the

Timed Rebeca model. For example, in Timed Rebeca there is no function for searching a list. So, this function can be defined in Erlang and be called in a Timed Rebeca model. Using this extension, the Timed Rebeca language has the same programming power as the Erlang language.

This way, the applications in which implementing buffers or queues is essential, like schedulers, can be modeled using the list data structure in the Timed Rebeca language. The elements of a list are of type integer. They can be defined inside message servers as a local variable or as a state variable. In order to facilitate working with the list data structure, the following functions are defined: `remove(intValue)`, `size()`, `first()`, `last()`, `insert(intValue)`. Function `remove(intValue)` removes the integer value of `intValue` from the list and function `insert(intValue)` inserts the value of `intValue` at the end of the list. Functions `first()` and `last()` return the first and the last element of the list, respectively.

## 3.2 Model Checking of Timed Rebeca Models Using McErlang Monitors

McErlang provides two types of model checking facilities for verification of *safety* properties and *Linear Temporal Logic (LTL)* formulas, using *safety monitors* and *Büchi monitors* respectively. In this work safety monitors are used for the corresponding Erlang program of a Timed Rebeca model in order to verify safety properties of the Timed Rebeca model. For a given Erlang program, a safety monitor is defined as a function which is called after creation of each state of the model. If the content of the state is invalid, the safety monitor reports the state as an erroneous state.

### 3.2.1 Checking Safety Properties

McErlang allows safety monitors to access both states of the program and the sequence of actions, as labels of transitions among states, but the values of program variables are not allowed to be accessed. However, the safety properties of a Timed Rebeca model are defined based on the values of its variables. This is why we added the checkpoint construct to the Timed Rebeca language. A checkpoint in a Timed Rebeca model can include the values of specific variables. As we discussed in Section 3.1.4, the values of intended variables are passed as arguments to checkpoints. Also, the occurrence of interesting events can be specified using checkpoints. While doing model checking, in the corresponding Erlang program, checkpoints are translated to probes, which are accessible by safety monitors in McErlang.

#### 3.2.1.1 Defining Safety Monitors

We explain two predefined safety monitors which can be used for Timed Rebeca models, and present a framework for defining safety monitors in McErlang using checkpoints in a Timed Rebeca model.

**Deadlock Monitor.** Detecting deadlock in non-terminating systems is essential. The predefined monitor in Listing 3.6 can be used to investigate the deadlock of Timed Rebeca models. As lines 13 to 20 of Listing 3.6 show, deadlock is detected by checking

the status of processes. If the status of all the processes is marked as *blocked*, deadlock is reported.

**Maximum Queue Length Monitor.** Although in theory message queues are unbounded in Timed Rebeca, in model checking and simulation we need a maximum length for each queue to keep the state space bounded. Trying to put messages beyond the queue size of a rebec results in a queue overflow error. The predefined maximum queue-size monitor in McErlang can be used to monitor the size of a rebec's queue. As lines 7 to 10 of Listing 3.7 show, if a queue of any process exceeds its maximum size, a violation is reported by the monitor. The maximum queue size is specified by the parameter *MaxQueueSize*.

**Checkpoint Monitor (User-defined Monitor).** The purpose of defining checkpoints in a Timed Rebeca model is the verification of safety properties using McErlang. Generally, a safety monitor is a function which is called after the creation of each state of the model. The monitor returns *satisfied* if the state satisfies the specified conditions, otherwise it returns *violation*. If a safety monitor is defined based on the information provided by checkpoints (which is available for McErlang from the translated Erlang program), the monitor is called *checkpoint monitor*. This type of monitor should be implemented by a modeler, while the previously mentioned monitors are available in McErlang.

```

1 monitorType() -> safety.
2
3 init(State) -> {ok, State}.
4
5 stateChange(State, MonState, _) ->
6   case is_deadlocked(State) of
7     true -> deadlock;
8     false -> {ok, MonState}
9   end.
10
11 is_deadlocked(State) ->
12   State#state.ether == [] andalso
13     case mce_ertl:allProcesses(State) of
14       [] -> false;
15       Processes ->
16         case mce_utils:find(fun (P) ->
17           P#process.status /= blocked end,
18           Processes) of
19           {ok, _} -> false;
20           no -> true
21         end
22       end.

```

Listing 3.6: McErlang - Deadlock monitor

```

1 monitorType() -> safety.
2
3 init(MaxQueueSize) -> {ok, MaxQueueSize}.
4
5 stateChange(State, MaxQueueSize, _) ->
6   case mce_utils:find
7     (fun (P) -> length(P#process.queue) > MaxQueueSize end,
8     mce_ertl:allProcesses(State)) of
9     {ok, P} -> {exceeds, P};
10    _ -> {ok, MaxQueueSize}
11  end.

```

Listing 3.7: McErlang - MaxQueue monitor

Listing 3.8 shows a template for checkpoint monitors. Any user-defined function can be used in the template. For example, we define the function `checkLabelCheckPoint` and use it in the monitor (Line 14), in which actions (obtained from the function `actions`) and a checkpoint label are used as arguments. If a checkpoint with the label `CheckpointLabel` occurs in a state, the monitor halts with a `violation`. If the verification terminates without any violation, it is guaranteed that the checkpoint never happens in any paths of the state space.

This template includes two variables *CheckpointLabel* and *CheckpointTerm* that may be used by the user depending on the safety property under study. We also implemented a set of functions to be used in the template. This makes it easier for a modeler to write the safety specification in a monitor. Each of these functions can be replaced by the function *checkLabelCheckPoint* in the template. The signature of each function and a brief explanation are listed below. The implementation of these functions are accessible from [85].

- Checking if a message server is dropped because the deadline is missed. In the following function, the term is equal to the message server name.
  - *checkDropMsgsrv(Actions, CheckpointTerm)*
- Checking if a checkpoint with the specified label occurs.
  - *checkLabelCheckPoint(Actions, CheckPointLabel)*
- Compare the checkpoint term with an integer or boolean. In the following functions, *MaxValue/MinValue* is the maximum/minimum value for the specified term. In the function *checkTermValue*, the value of the specified term is checked to be equal to *value*.
  - *checkTermMaxValue(Actions, CheckPointLabel, CheckpointTerm, MaxValue)*
  - *checkTermMinValue(Actions, CheckPointLabel, CheckpointTerm, MinValue)*
  - *checkTermValue(Actions, CheckPointLabel, CheckpointTerm, value)*

```

1 monitorType() -> safety.
2
3 init(_) -> {ok, satisfied}.
4
5 stateChange(_,satisfied,Stack) ->
6   % Monitor Setup
7   % Usage: checkpoint(Label,Term);
8   CheckpointLabel = checkpoint_label, % Not needed when using function checkDropMsgsrv.
9   CheckpointTerm = checkpoint_term,   % Not applicable when using function
    checkLabelCheckPoint.
10
11   Actions = actions(Stack),
12   % user_defined_function
13   checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 3.8: A template (pseudo code) for checkpoint monitors which is used by McErlang

### 3.3 Statistical Model Checking of Timed Rebeca Models

In the previous section we showed how safety monitors can be defined for the corresponding Erlang program of a Timed Rebeca model, using the checkpoints of the Timed Rebeca model. So, the McErlang can be used as a back-end model checker for the verification of safety properties of the Timed Rebeca model. The major limiting factor in applying model checking for verification of real-world systems is the huge amount of space and time required to store and explore the state space. Alternatively, statistical model checking can be used, which does not have the problem of state explosion.

For statistical model checking, a number of simulation traces are generated. To get a simulation trace, nondeterminism must be resolved in an appropriate way to represent realistic behaviors. According to [86], there are three typical uses for nondeterminism: First, in case of complete absence of knowledge about a certain choice—not even some probabilities are known—that choice can be modeled as a nondeterministic one. Second, in a refinement process where abstract models are progressively refined to more and more concrete implementations, a nondeterministic choice may leave open certain choices. Finally, nondeterminism can allow an unspecified environment to make certain choices in an open model. In the latter two cases, obtaining results for some environment or some implementation is not particularly useful; in fact, if the result happens to be very optimistic (e.g. by not considering some adverse environments or unfortunate implementations), it may lead to unfounded conclusions that may jeopardise the safety of the actual system whose study the model was built for. Although in the first case a uniformly random resolution of the choice seems to make some sense, this is not true. Clearly, three nondeterministic choices like A, B, B should be the same as A, A, B, but the uniform interpretation breaks this.

In Timed Rebeca, nondeterminism appears in two cases: a) concurrently executing actors, and b) nondeterministic assignments in the model which are specified by the modeler. In simulation of Timed Rebeca models using McErlang, both cases are resolved by the scheduler of McErlang, which selects the process that must be executed in the next step based on the uniform distribution. Case (a) is mapped to the third use of nondeterminism in the above paragraph, and case (b) is mapped to the first one. In both cases (a) and (b), the modeler has to be informed that nondeterminism is resolved based on the uniform distribution. The risk is an unrealistic evaluation. In this work, we follow the community that uses simulation-based and statistical model checking approaches for performance evaluation of concurrent systems, and we face the same problems.

In this section, we propose a model checking approach to compute an approximation of the correctness of the system. The main idea is to check a limited number of traces instead of exploring the whole state space. For a given error  $\epsilon$  and confidence value  $\delta$ , we have to provide an upper bound  $N$  on the number of simulation traces which are required to compute an  $(\epsilon, \delta)$ -approximation of the correctness of the system. We explain our approach in the following subsections in more detail.

### 3.3.1 Optimal Monte Carlo Estimation

In many applications, it is necessary to compute the mean value  $\mu_Z$  for the random variable  $Z$  distributed in  $[0, 1]$ . When an exact computation of  $\mu_Z$  is intractable, being for example NP-hard, Monte Carlo methods can be used to compute an  $(\epsilon, \delta)$ -approximation of  $\mu_Z$ . The main idea behind this approach is to use  $N$  different independent random variables (samples)  $Z_1, Z_2, \dots, Z_N$ . If the  $Z_i$  variables are identically distributed with mean  $\mu_Z$ , the value of  $\mu_Z$  is approximated by  $\tilde{\mu}_Z = (Z_1 + Z_2 + \dots + Z_N)/N$ . In the next section, we define the random variable  $Z$  in a way that the mean value of  $Z$  shows the likelihood that a given property is satisfied by a system. Each random variable  $Z_i$  is a random simulation trace (random execution) of a given Timed Rebeca model (system). So, the correctness of the system according to the property is approximated based on random traces.

Based on the *zero-one estimator theorem* [87], if  $N$  is proportional to  $\Upsilon = 4 \ln(2/\delta)/\mu_Z \epsilon^2$  then the value of  $\mu_Z$  is approximated by  $\tilde{\mu}_Z$  with absolute error  $\epsilon$  and with probability  $1 - \delta$ . In other words, we say  $\tilde{\mu}_Z$  is an  $(\epsilon, \delta)$ -approximation of  $\mu_Z$  if  $\Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$ . But, applying the zero-one estimator theorem encounters a difficulty which is the fact that  $N$  depends on  $1/\mu_Z$ , the inverse of the value that one intends to approximate. In addition, the factor of  $1/\mu_Z \epsilon^2$  makes the value of  $N$  unnecessarily large.

A way of computing  $N$  without relying on  $\mu_Z$  is provided by the *Stopping Rule Algorithm* (SRA) in [88]. As Figure 3.1 shows,  $\tilde{\mu}_Z = (1 + (1 + \epsilon)\Upsilon)/N$ , where  $N$  is the number of traces which are needed to be analyzed until at least  $\lfloor 1 + (1 + \epsilon)\Upsilon \rfloor$  of them satisfy the given property. Note that each  $Z_i$  is a Bernoulli trial which takes the value of 1 or 0.

**SRA algorithm**

**input:**  $(\epsilon, \delta)$  with  $0 < \epsilon < 1$  and  $\delta > 0$ .

**input:** Random variables  $Z_i$  with  $i > 0$ , independent and identically distributed.

**output:**  $\tilde{\mu}_Z$  approximation of  $\mu_Z$ .

- (1)  $\Upsilon = 4(e - 2)\ln(2/\delta)/\epsilon^2$ ;  $\Upsilon_1 = 1 + (1 + \epsilon)\Upsilon$ ;
- (2) for ( $N = 0$ ,  $S = 0$ ;  $S \leq \Upsilon_1$ ;  $N++$ )  $S = S + Z_N$ ;
- (3)  $\tilde{\mu}_Z = S/N$ ; return  $\tilde{\mu}_Z$ ;

Figure 3.1: Stopping rule algorithm.

### 3.3.2 Model Checking Algorithm

We present the model checking approach based on the Monte Carlo SRA and the capabilities of McErlang to verify a property using monitors. The samples we are looking for are the cycles reachable from the initial state of the time transition system of a Timed Rebeca model. Our approach is similar to the approach presented in [89], but we use SRA and make it appropriate for our case. The model checking algorithm in [89] is a Monte Carlo algorithm to decide on whether or not a property specified in temporal logic holds for a system specification. Using our proposed approach, we are able to verify properties that can be defined by checkpoint monitors.

**Definition 1 (Timed Rebeca time transition system.)** *A Timed Rebeca time transition system is a labeled transition system  $T_{\mathcal{M}} = (S, s_0, Act, \hookrightarrow)$ , where:*

- $S$  is a set of states in Timed Rebeca,
- $s_0 \in S$  is the initial state,
- $Act$  is a set of actions, containing all possible messages in Timed Rebeca,
- $\hookrightarrow \subseteq S \times Act \times S$  is the transition relation.

□

**Definition 2 (Trace sample space.)** *Given the time transition system  $T_{\mathcal{M}}$  of a given Timed Rebeca model  $\mathcal{M}$ , a finite trace of  $s_0x_0 \dots s_nx_ns_{n+1}$  is included in the sample space if  $s_0 \dots s_n$  are pairwise distinct and  $s_{n+1} = s_i$  for some  $0 \leq i \leq n$ . Note that  $s_i \in S$ ,  $x_i \in Act$  and  $(s_i, x_i, s_{i+1}) \in \hookrightarrow$ . The sample space  $U$  is the set of all these traces.*

□

A trace of  $T_{\mathcal{M}}$  is sampled via a random walk through the transition graph of  $T_{\mathcal{M}}$ , starting from the unique initial state. The trace is generated by exploring the outgoing transitions uniformly.

**Definition 3 (Trace probability.)** *The probability  $Pr[\sigma]$  of a finite trace  $\sigma = s_0x_0 \dots s_nx_ns_{n+1}$  of a time transition system  $T_{\mathcal{M}}$  is defined inductively as follows:  $Pr[s_0] = 1$  and  $Pr[s_0x_0 \dots s_nx_ns_{n+1}] = Pr[s_0x_0 \dots s_n] \cdot \pi[s_nx_ns_{n+1}]$  where  $\pi[s \ x \ t] = 1/m$  if  $(s, x, t) \in \hookrightarrow$  and  $m$  is the number of outgoing transitions from state  $s$ .*

□

**Proposition 1 (Probability space.)** *Given the time transition system  $T_{\mathcal{M}}$ , the pair  $(\mathcal{P}(U), Pr)$  defines a discrete probability space.*

□

The proof first considers the infinite tree  $T$  corresponding to the infinite unfolding of  $Act$ .  $T'$  is the (finite) tree obtained by making a cut in  $T$  at the first repetition of a state along any path in  $T$ . It can be shown by induction on the height of  $T'$  that the sum of the probabilities of the traces associated with the leaves of  $T'$  is 1.

**Definition 4 (Random variable.)** *The Bernoulli random variable  $Z$  associated with the probability space  $(\mathcal{P}(U), Pr)$  of a time transition system  $T_{\mathcal{M}}$  is defined as follows:  $p_z = Pr[Z = 1] = \sum_{\lambda_a \in U} Pr[\lambda_a]$  and  $q_z = Pr[Z = 0] = \sum_{\lambda_n \in U} Pr[\lambda_n]$  where  $\lambda_a$  is a finite accepting trace and  $\lambda_n$  is a finite non-accepting trace.*

□

Accepting traces satisfy the defined property and non-accepting traces do not satisfy the property. The expectation of random variable  $Z$  equals  $p_z$ , because  $\mu_Z = 0 \times q_z + 1 \times p_z$ . So,  $1 - p_z$  shows the number of counterexamples in  $T_{\mathcal{M}}$ , weighted by their probability. Since an exact computation of  $p_z$  is not tractable, we get an  $(\epsilon, \delta)$ -approximation  $\tilde{p}_z$  of  $p_z$  using SRA, as shown in Figure 3.1.

The statistical model checking algorithm is shown in Figure 3.2. The *Random Variable Generator* (RVG) routine uses our developed toolset to generate a simulation trace from  $T_{\mathcal{M}}$ . The routine returns 1 if the trace satisfies the property, otherwise throws the trace as a counterexample. The algorithm works as follows: (1) Our developed

toolset provides independent random samples (simulation traces)  $Z_i$ , each identically distributed according to  $Z$  as required by SRA. (2) If a trace that does not satisfy the property is found, the model checking procedure stops and reports the trace as a counterexample. (2) If all traces satisfy the property, we conclude  $p_Z$  is 1 with error margin  $\epsilon$  and confidence value  $\delta$  such that  $Pr[|p_Z - 1| < \epsilon] \geq 1 - \delta$ .

**Theorem 1 (Algorithm correctness.)** *Given a time transition system  $T_{\mathcal{M}}$ , error margin  $\epsilon$ , and confidence ratio  $\delta$ ,  $\tilde{p}_Z$  is computed by our model checking algorithm as the  $(\epsilon, \delta)$ -approximation of  $p_Z$  such that if  $\tilde{p}_Z < 1$  then  $T_{\mathcal{M}} \not\models \varphi$ , and if  $\tilde{p}_Z = 1$  then the weighted expectation  $p_Z$  that  $T_{\mathcal{M}} \models \varphi$  satisfies  $Pr[|p_Z - 1| < \epsilon] \geq 1 - \delta$ .*  $\square$

*Proof. Independence:* Each call to RVG generates an independent Bernoulli trial. *Distribution:* The probability that a random trace is accepting is the same for all samples and is given by  $p_Z$ . *Correctness:* If a counterexample (non-accepting trace) is found, then  $T_{\mathcal{M}} \not\models \varphi$  by definition, otherwise  $\tilde{p}_Z = 1$  and the result follows from  $Pr[|p_Z - \tilde{p}_Z| < \epsilon] \geq 1 - \delta$ .

There is an estimator mode for our model checking algorithm. In this mode, model checking does not stop upon finding a counterexample (non-accepting trace), but rather continues until the computation of  $\tilde{p}_Z$  is completed. This mode has some advantages, such as we may find more counterexamples which is useful to modify the model, and the algorithm provides an estimation of how false is the judgment  $T_{\mathcal{M}} \models \varphi$ . As our algorithm uses SRA for the computation of  $\tilde{p}_Z$ , the number of accepting traces should be greater than  $\mathcal{Y}_1$ , otherwise model checking may not terminate. To prevent this situation, if  $\mathcal{Y}_1$  non-accepting traces are found, model checking in the estimator mode is terminated.

**Model checking algorithm**

**input:** Time transition system  $T_{\mathcal{M}} = (S, s_0, Act, \hookrightarrow)$ .

**input:**  $(\epsilon, \delta)$  with  $0 < \epsilon < 1$  and  $\delta > 0$ .

**output:** Either counterexample or estimate  $\tilde{p}_Z$  with  $Pr[|p_Z - \tilde{p}_Z| < \epsilon] \geq 1 - \delta$

- (1)  $\mathcal{Y} = 4(e - 2)\ln(2/\delta)/\epsilon^2$ ;  $\mathcal{Y}_1 = 1 + (1 + \epsilon)\mathcal{Y}$ ;
- (2) for ( $N = 0$ ,  $S = 0$ ;  $S \leq \mathcal{Y}_1$ ;  $N++$ ) {
  - try**  $\{Z_N = RVG(T_{\mathcal{M}}); \}$  **catch** (e) {**return** e;}
  - $S = S + Z_N$ ;
- (3)  $\tilde{p}_Z = S/N$ ; **return**  $\tilde{p}_Z$ ;

Figure 3.2: Model Checking Algorithm.

Now, we have to specify the subset of formulas which can be model checked by means of statistical model checking. As shown in [90], formulas with unbounded until operators (and nested until operators) can be model checked using statistical model checking. As a result, our proposed approach can work for formulas with until operators which are both safety and LTL properties. Model checking of LTL properties is beyond the scope of this dissertation, and so Timed Rebeca models can only be

verified against *safety properties*, using predefined monitors like *Deadlock Monitor* and *Maximum Queue Length Monitor*, and checkpoint monitors.

As a final step in developing a statistical model checker, we have to implement the above algorithm to calculate an approximation of correctness of a Timed Rebeca model. As Figure 3.3 shows, the *statistical model checking (SMC)* component works with the present tool, which was developed in [58]. The *simulation wrapper* component is employed to generate needed simulation traces for the SMC component. Figure 3.3 demonstrates the analysis toolset which includes the SMC component and the performance evaluation tool. In the following section, we describe the architecture of the performance evaluation tool.

## 3.4 Performance Evaluation of Timed Rebeca Models

Most discrete-event simulators rely on hidden schedulers to resolve nondeterministic choices, which may influence results in unexpected ways. McErlang is one of the simulators which call an explicit scheduler to resolve nondeterministic choices in a uniformly distributed random manner. McErlang provides facilities for simulation of Erlang programs. In the simulation mode, the next state of an Erlang program is determined randomly, by choosing one of the available transitions from the current state. Therefore, a randomly chosen path of execution is explored in each simulation run. To have an accurate understanding of the model's behavior, data is gathered from different simulation runs, each of them including a different trace. For performance evaluation, statistical methods are applied to the collected data and the results are used to reason about the behavior of the model. Since the resulting information of a performance measurement may be very large, we use the average moving method to reduce the dataset for visualization. This well-known method smooths out short-term fluctuations and highlights long-term trends of the data [91].

### 3.4.1 Performance Evaluation Toolset

We implement a toolset to provide performance evaluation of Timed Rebeca models using McErlang. As shown in Figure 3.3, the toolset contains three components as follows.

- *translator*: for translating Timed Rebeca models to Erlang programs.
- *trace analyzer*: to apply statistical analysis methods to stored information. Different analysis techniques are implemented in this component.
- *simulation wrapper*: it sends required data to other components and stores data of simulation runs. The modeler can define the number of simulations as well as the duration of each simulation run.

Figure 3.3 shows that *simulation wrapper* component sends Timed Rebeca models to the *translator* component to be translated to an Erlang program. The translated Erlang program is sent to McErlang for simulation. The generated data from the simulation is sent to the *simulation wrapper* component at run-time. The *simulation*

*wrapper* component categorizes the simulation data of different simulation runs in a way to be used by *trace analyzer*.

Two different analysis techniques have been implemented in the component *trace analyzer*, called *checkpoint analysis* and *paired-checkpoint analysis*, to provide performance evaluation of Timed Rebeca models. In the next section, we explain how information provided by checkpoints can be used in *trace analyzer* to achieve performance measures of interest.

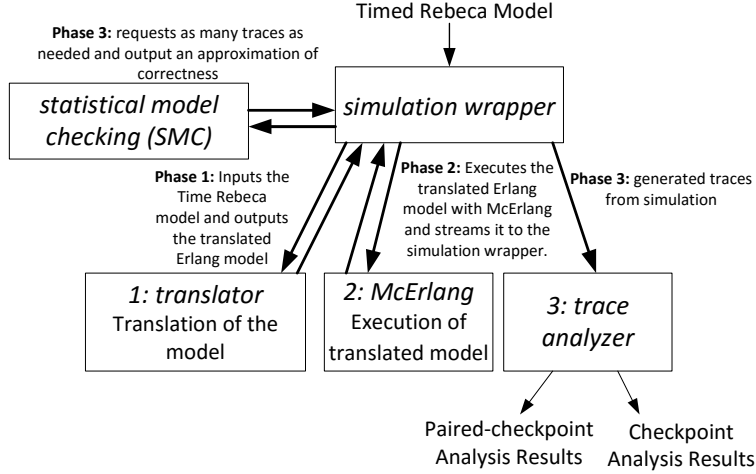


Figure 3.3: Architecture of the analysis toolset.

### 3.4.2 Checkpoints analysis in Simulation

As we discussed in Section 3.1.4, checkpoints were added to Timed Rebeca language to provide needed information for model checking and simulation. Each checkpoint is translated to a function such that McErlang can access the value of variables and be notified of the occurrence of events. We analyze models based on the information provided by checkpoints.

During the simulation, every time a checkpoint is executed the value of terms (variables or any value of available data types), the label, the time of observing the checkpoint and the name of the rebec including the checkpoint are stored for performance evaluation purposes.

```

1  env int requestDeadline, checkIssuedPeriod, 15  checkpoint(ticketIssued,tok);}
   retryRequestPeriod;                        16  else { checkpoint(ticketNotIssued,tok); }
2  env int newRequestPeriod, serviceTime1,    17  }
   serviceTime2;                               18
3  reactiveclass Agent(3) {                   19  msgsrv checkTicket() { ... }
4  ...                                         20  msgsrv retry() { ... }
5  msgsrv findTicket(TicketService ts) {      21  }
6  attemptCount = attemptCount + 1;          22
7  token = token + 1;                         23  reactiveclass TicketService(3) {
8  checkpoint(requestStart,token);            24  ...
9  ts.requestTicket(token)                    25  msgsrv requestTicket(int token) { ... }
   deadline(requestDeadline);                26  }
10 self.checkTicket()                         27  main { ... }
   after(checkIssuedPeriod);
11 }
12
13 msgsrv ticketIssued(int tok) {
14 if (token == tok) { ticketIssued = true;

```

Listing 3.9: Timed Rebeca model - Ticket service system

To understand the role of checkpoints in simulation, we consider a running example and explain the analysis techniques based on it. In the ticket service system shown in Listing 2.2, the average response time to incoming requests shows the efficiency of the system. It can be important to get the distribution of issued tickets between ticket services. The model is examined for different time settings, each of which represents a different behavior of the model. Different time settings are specified by assigning values to timing variables in the model. As an example, there may exist a time setting in which a ticket service issues most of the tickets and the other one issues a few tickets. If this behavior is not expected, we should find the reason. This undesirable behavior can reveal a problem in the system design or we should figure out how timing variables affect each other and cause an overload on a ticket service.

In order to collect the required data for performance evaluation of the model, we add three checkpoints to the ticket service model in Listing 2.2, as shown in Listing 3.9. For the sake of simplicity, we keep the message servers to which checkpoints are added and delete other message servers. These checkpoints store data about when the request is sent to the ticket service (line 8), when the ticket is received by the agent *a*, i.e. the ticket is issued (line 15), and whether the ticket is not issued (line 16). We are able to define as many checkpoints as needed depending on the safety properties and the performance measures we are interested in. In these checkpoints we should provide the values of variables that are needed for the intended analysis.

For each time setting, the model is simulated, and the analysis techniques explained in the following subsections are applied to the obtained data to compute the average response time, and to find the distribution of issued tickets between ticket services.

#### 3.4.2.1 Paired-checkpoint Analysis

The paired-checkpoint method is implemented in the *trace analyzer* tool. In this analysis technique, two checkpoints are grouped together. The modeler specifies paired checkpoints with the use of labels when running the tool. The elapsed time between observing two paired checkpoints is important and can show different performance measures. There is a command in our tool that enables the modeler to specify paired checkpoints. For example, the starting checkpoint in line 8 (labelled by *requestStart*) shows that the request is sent to the ticket service and the ending checkpoint in line 15 (labelled by *ticketIssued*) represents that the ticket was issued. Consequently, the passed time between the occurrence of these two checkpoints is considered as the response time of the issued ticket.

#### 3.4.2.2 Checkpoint Analysis

In checkpoint analysis, instead of pairing checkpoints, a certain checkpoint is provided to expose the changes of a particular variable over time. For example, in the ticket service system, we are interested in knowing how many tickets are issued by ticket service *ts1* and how many of them are issued by ticket service *ts2*. This information is available in the simulation results by defining the checkpoint with label *ticketIssued* in the model. When a ticket is issued at run-time, the time of occurrence and the name of rebeccas including the checkpoint are stored in the simulation results.

### 3.4.3 Confidence Interval

While using statistical methods, there is an important question of how precise the results are. Here, we calculate the confidence interval for simulation results to indicate their accuracy. We presume that our measurements follow a normal distribution. Using this method, an estimated range of values for the mean value of the sample is computed. The confidence interval shows how close our measurement is to the original value if the experiment is repeated. The margin of the error is calculated from the following formula.

$$Z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$$

where  $\sigma$  is the standard deviation of the intended phenomenon (like response time),  $n$  is the sample size,  $\alpha$  is the confidence level and  $Z_{\alpha/2}$  is the confidence coefficient. The most commonly used confidence levels are 90%, 95% and 99%. Suppose the confidence level is 95% ( $\alpha = 0.95$ ), to find the value of  $Z_{\alpha/2}$  the z table is checked for the value  $0.95/2 = 0.475$  [92]. In the z table, the intersection of row 1.9 and the column of 0.06 shows a cell with the value 0.475 (or the closest value to 0.475), so  $Z_{0.475}$  equals 1.96.

The confidence interval is obtained from the following formula, where  $\bar{x}$  is the mean value of the intended phenomenon (like response time).

$$\bar{x} \pm Z_{\alpha/2} \times \frac{\sigma}{\sqrt{n}}$$

## 3.5 Case Studies and Experimental Results

In this section, we present two case studies to illustrate the applicability of the approaches of this work. For each case study, after an intuitive description of the model using an event graph [93], the detailed description of the Timed Rebeca model is presented. We use an event graph to give a highly abstracted view of events and their causality relations. Event graphs are widely used for the explanation of event-based models. In this graph, the vertices represent events in a system and the edges represent the causality relation between events (vertices). Additionally, we add a label below each vertex that shows in which reactive class the event occurs. Edges can be conditional (thick edge), mandatory (thin edge) or marking an initial event (jagged edge). Model checking, statistical model checking and performance evaluation are applied for the case studies. In model checking using McErlang, we have limitations on the size of the models to avoid state space explosion. In statistical model checking, we are able to check larger models, and increment the size of the models greatly.

We should remind that nondeterminism is resolved by uniform distribution. The assumption of uniform distribution may affect the validity of results in statistical model checking and simulation. On the other hand, this assumption can provide useful insights about the model which is important practically. For example, we check a property for different time settings in a model. Like in the ticket service case study we may find a time setting in which no ticket is issued (a property is violated unlike our expectation). So, we find that either the model or the time setting should be changed.

As another example, in elevator case study, if we have 10 floors, it wouldn't be correct to have a value of 11 for elevator location. If we find a trace by chance with a value greater than 10 for elevator location, we can report it as a counter example for the model. It means that the model is not a correct model of the elevator system.

Otherwise we report that we picked “n” consecutive number of traces and couldn’t find any trace to violate the property (elevator location  $\leq 10$ ). So, we can say that the property is satisfied with specified error and confidence interval. Our disclaimer hold again: the property is satisfied with certain confidence when nondeterminism is resolved by uniform distribution.

There are performance properties that give us insight and may have practical merit, again with the same disclaimer. Like in the elevator case study, by simulation, we examine different scheduling policies and report max, min, and mean response time to requests for each floor. This way we report on efficiency of different scheduling policies. The validity of mean values is affected by our assumption.

### 3.5.1 Ticket Service System

Our first case study is the ticket service system, which is shown in Listing 2.2. As we already described the details of this model in Chapter 2, here, we only demonstrate the event graph of the Ticket Service model in Figure 3.4. As shown in Figure 3.4, initially the message server `initial` in the rebec `agent` sends a message to itself that triggers the event (the message server) `findTicket`. Execution of this event causes sending a message to the rebec `TicketService` which raises the event `requestTicket`. After a number of trials (which is modeled by causality relation among `findTicket`, `checkTicket`, and `retry`), the event `ticketIssued` is raised to inform that a ticket is issued.

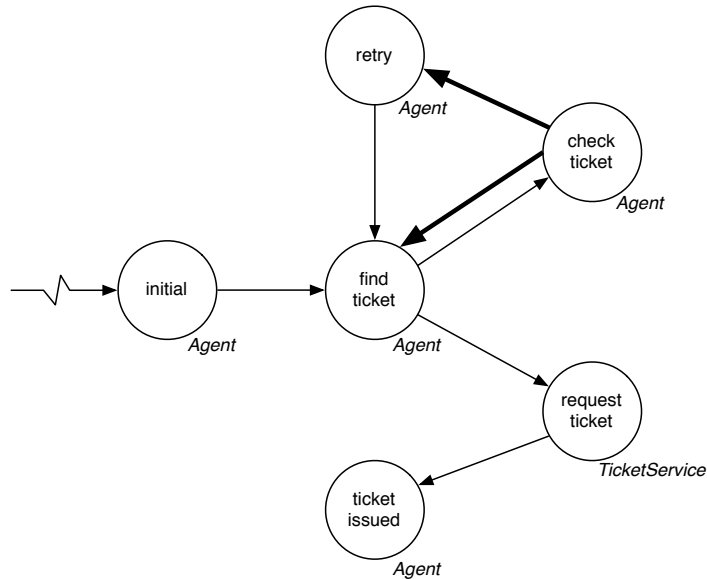


Figure 3.4: Event graph of the ticket service model.

#### 3.5.1.1 Model Checking Using McErlang Monitors

The model in Listing 2.2 is revised to be usable in monitor-based model checking. A variable is added to the model to restrict the number of ticket requests that are sent to `ts1` and `ts2`. The maximum number of ticket requests is set to seven. This modification is necessary to avoid state space explosion. In the ticket service system, the goal is to issue tickets. We aim at checking the property of “at least one ticket is issued” for a set of time settings. If a time setting fails to satisfy this property, it

shows that in this time setting no ticket is issued. To be able to check the property, we add a checkpoint with label `ticketIssued` to the model where a ticket is issued (refer to Listing 3.9). The checkpoint monitor shown in Listing 3.10 is used for safety verification. The property is satisfied if a ticket is issued. This property verification is performed by using the predefined function `checkLabelCheckPoint`, explained in Section 3.2.1.1.

```

1 monitorType() -> safety.
2
3 init(_) -> {ok, satisfied}.
4
5 stateChange(_,satisfied,Stack) ->
6   CheckpointLabel = ticketissued,
7   Actions = actions(Stack),
8
9   checkLabelCheckPoint(Actions, CheckpointLabel).
```

Listing 3.10: The checkpoint monitor for checking whether a ticket is issued.

The results of model checking of the Ticket Service system using McErlang are shown in Table 3.2. We considered different settings for the model each of which has different values for variables. As shown in the table, there is no tickets issued in the first three settings.

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

Table 3.2: Verification results for ticket service. Property is satisfied if at least one ticket is issued.

### 3.5.1.2 Statistical Model Checking

We verify the ticket service model shown in Listing 3.9, with a huge number of ticket requests in the model. We check the property of “at least one ticket is issued” for a set of time settings. For each setting in Table 3.2, we run the statistical model checking (SMC) component with different error values and confidence values. Table 3.3 shows the results for setting 4. The results for settings 5, 6, and 7 are the same for setting 4. Table 3.4 shows the verification results for setting 1. Settings 2 and 3 have the same results as setting 1, because no ticket is issued in these settings.

For a given property, we generate as many simulation traces as needed to get  $N_{ct} = \lfloor 1 + (1 + \epsilon)\Upsilon \rfloor$  number of traces that satisfy the property (refer to Section 3.3 for the  $\Upsilon$  formula). The approximation of correctness of the property is defined as  $\tilde{\mu}_Z = N_{ct}/N'$ , where  $N'$  is the total number of explored traces.

Considering the error value and the confidence value of the first experiment of Table 3.3,  $N_{ct} = 289$ . We generate as many simulation traces as needed to get 289 traces that satisfy the defined property. The total number of simulation traces for this experiment is 289 ( $N' = 289$ ), meaning all traces satisfy the property. So, in this experiment the approximation of correctness is one,  $\tilde{\mu}_Z = 1$ . More accurately, we obtain an  $(\epsilon, \delta)$ -approximation of correctness where  $Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$ ,  $\mu_Z$  is the real approximation of correctness. For the first experiment of Table 3.3,  $Pr[|\mu_Z - 1| < 0.05] \geq 0.95$ .

Experiment#	Number of traces to be satisfied	Total number of traces ( $N'$ )	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness ( $\tilde{\mu}_Z$ )
1	289	289	0.05	0.05	1
2	203	203	0.1	0.01	1

Table 3.3: Statistical model checking results for the ticket service model with parameters equal to setting 4. The approximation of correctness is calculated for the safety property “at least one ticket is issued”.

As described before, we generate as many simulation traces as needed until  $N_{ct}$  traces satisfy the property. When a trace violating the property is found, the trace is reported as a counterexample and we can stop searching the state space, i.e. stop generating simulation traces. We selected another way; we continue generating traces until we get  $N_{ct}$  traces satisfying the property. The advantages of this approach are that we may find more counterexamples, and we can compute which percentage of traces satisfy the property. Both give more insight about the model, and may be used for modification of the model.

If the model never satisfies the property, the trace generation should continue forever to find  $N_{ct}$  satisfied traces. To avoid this situation, in the implementation of the SMC component we stop generating traces if the first  $N_{ct}$  traces do not satisfy the property. This case happens for setting 1, so the approximation of correctness equals zero as presented in Table 3.4.

Experiment#	Number of traces to be satisfied	Total number of traces ( $N'$ )	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness ( $\tilde{\mu}_Z$ )
1	289	289	0.05	0.05	0
2	203	203	0.1	0.01	0

Table 3.4: Statistical model checking results for ticket service model with parameters equal to setting 1. The approximation of correctness is calculated for the safety property “at least one ticket is issued”.

We are also able to verify the model with more actors (rebecs) for which the model checking approach based on McErlang monitors explodes. For example, the number of agents and ticket services is increased to four and nine, respectively. We check the safety property of “at least one ticket is issued” for this model. The approximation of correctness equals one for the following parameters:  $\epsilon = 0.05$ ,  $\delta = 0.05$ . We use a different setting which is not listed in Table 3.2. In this setting, the values of variables

(from left to right in Table 3.2) equal 3, 3, 2, 2, 4, 7. So, large ticket service models can be verified against safety properties using statistical model checking.

### 3.5.1.3 Performance Evaluation

In the simulation, the limitation of the number of ticket requests is removed from the model. Considering the verification results, we know that some tickets are issued in settings 4, 5, 6, and 7. We use the methods introduced in Section 3.4 to evaluate the performance evaluation of different settings of the model. For each setting, the mean response time to ticket requests is calculated using the paired-checkpoint analysis. The simulation results are shown in Table 3.5. Each setting is simulated 5 times, each for 200 seconds. The error margin is calculated for different confidence levels of 99%, 95% and 90%.

Setting	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Starting checkpoints	Checkpoint pairs
4	3.0	3.0	3.0	0	3.0	3.0	3.0	519350	614
5	$2.1 \pm 0.00114$	$2.1 \pm 0.000864$	$2.1 \pm 0.00073$	0.1	2	3.0	2.0	511709	51476
6	4.0	4.0	4.0	0	4.0	4.0	4.0	363891	81585
7	3.0	3.0	3.0	0	3.0	3.0	3.0	573551	286948

Table 3.5: Paired-checkpoint evaluation for Ticket Service. The specification of settings is available in Table 3.2 where all settings guarantee that some tickets are issued. SD, WCT, BCT denote standard deviation, worst-case time and best-case time, respectively.

Figures 3.5 and 3.6 show the distribution of issued tickets between ticket services for settings 4, 5, 6, and 7. The results are obtained by using the checkpoint analysis method. According to these results, in settings 6 and 7, a large number of tickets is issued by ticket services in comparison to settings 4 and 5. This shows that in these settings the system is more responsive. This results may not be used directly for improving the model, but if we are sure that we modeled the real system correctly, we can find a better time setting for the system.

The results in Figure 3.3 show that most requests are responded by the ticket service 2 (ts2) in setting 4 and the total number of issued tickets is very low. By comparing the distribution of issued tickets between ts1 and ts2 in different time settings, the modeler may reason about the relation between this behavior and the values of timing variables. This can be used to improve the design.

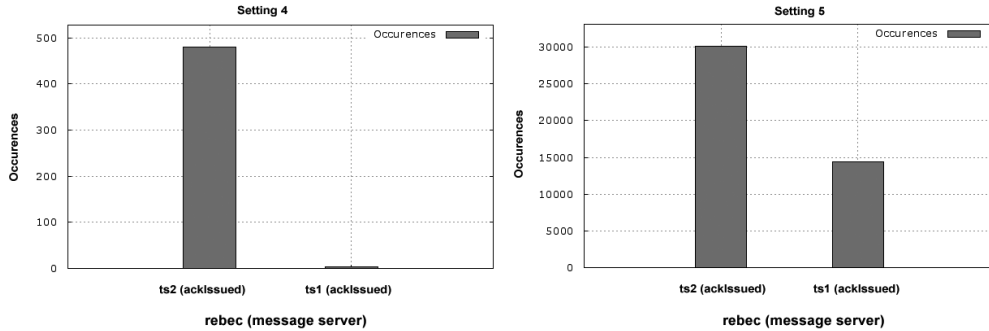


Figure 3.5: The distribution of issued tickets between ticket services for settings 4 and 5.

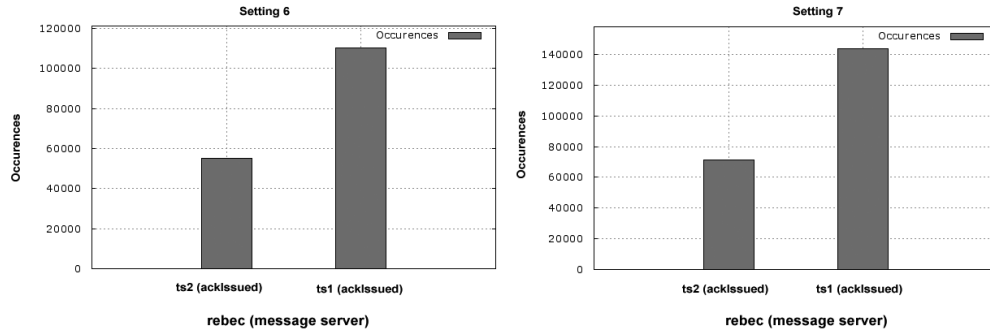


Figure 3.6: The distribution of issued tickets between ticket services for settings 6 and 7.

### 3.5.2 The Elevator System

Our second case study is an elevator system, where a centralized coordinator dispatches the coming requests among the elevators, and also decides on the direction of elevators movement. In this system, the approach of dispatching requests is called the *scheduling policy*, and the decision on the movement of elevators between the floors is called the *movement policy*. Figure 3.7 shows the event graph of the elevator model. As shown in the figure, a person requests to enter one of the elevators by raising the event `callElevator` or he is already in an elevator and presses one button to ask it to stop on one of the floors by raising the event `requestFloor`. Both of these events result in raising the `handleRequest` event which is the event of the centralized coordinator. Based on the current locations of the elevators and received requests, the centralized coordinator schedules movement for elevators by raising `moveUp`, `moveDown`, and `stopOpen` events.

#### 3.5.2.1 Timed Rebeca Model

The Timed Rebeca code of the elevator system is shown in Listing 3.11. The number of rebecs in the main part can be changed in order to make different variants of the elevator system with different sizes (e.g. we increase the number of floors from three to ten in Section 3.5.2.3). There are four reactive classes `Person`, `Floor`, `Elevator`, and `Coordinator` in this model. Rebecs `e11` and `e12` are instantiated from `Elevator` as the two elevators of the system. Also, rebecs `floor1` to `floor3`, rebec `pers`, and rebec `coord` are instantiated from reactive classes `Floor`, `Person`, and `Coordinator` respectively, to show that there are three floors, one person and one coordinator in the model (Lines 113-119).

The rebec `pers` starts the model. In the initialization phase, the message `go` is sent to the `pers` by itself (Line 90). The message server `go` models all behaviors of the `pers`. In this message server, we model two possibilities for the person. The person can be either inside an elevator or outside elevators. Being in floors 1 and 2 are considered as the person is outside elevators. Being in the floor 3 is considered as the person is inside one of the two elevators. At the start point, the person is put in one of the floors nondeterministically (Line 93).

If the person is in the first floor or in the second one, the `callElevator` message is sent to one of the floors nondeterministically (Lines 94 and 96-99). Sending this message shows that the person standing in the specified floor presses the button and

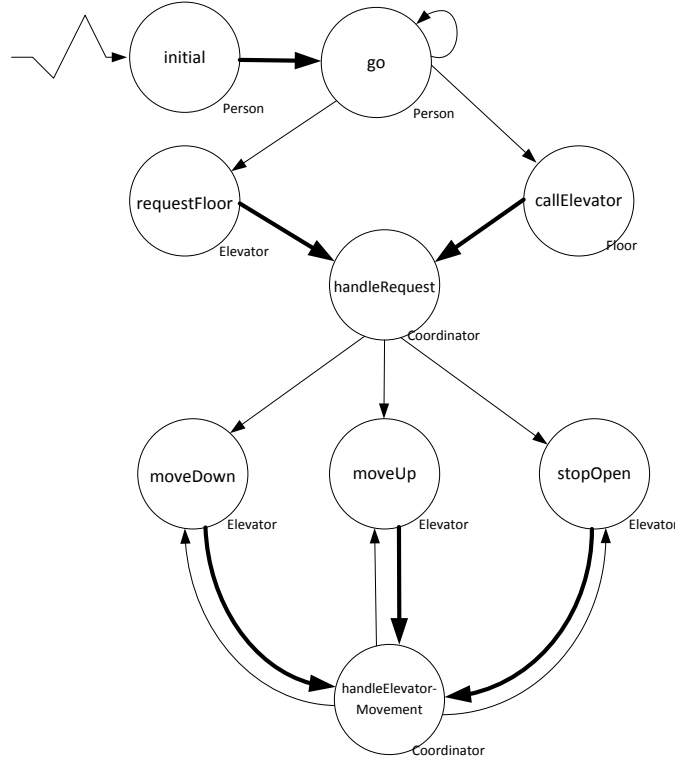


Figure 3.7: Event graph of the centralized elevator system.

asks for an elevator to come. This request has to be forwarded to the appropriate elevator later to be served (in the message server **handleRequest**).

Being in floor 3 ( $fc = 3$  in Line 100) implies that the person is inside one of the two elevators and requests to go to one of the floors specified by **flr** (Lines 95 and 101-102). As in this case the person is inside the elevator, it sends its request directly to the elevator by sending the **requestFloor** message. All the requests are modeled through sending the messages **requestFloor** (Lines 101-102) and **callElevator** (Lines 97-99) by the person, and are forwarded (Lines 12 and 34) to the message server **handleRequest** in the coordinator.

Algorithms which are related to the scheduling and movement policies are implemented in the message servers **handleRequest** and **handleElevatorMovement** of the **Coordinator**. Different types of request are served in the **handleRequest** message server. For example, the conditional statement in line 60 contains the handling mechanism of requests which are sent from floors. Based on the implemented policy, if a floor requests an elevator and one of the elevators is on the requested floor, that elevator is assigned to the floor (Lines 62-65). Otherwise, one of the elevators is selected nondeterministically (Line 61) and the request is assigned to that elevator (Lines 66-70). There are more cases which are eliminated here and can be found in A.

We implemented three different scheduling policies, namely *shortest distance*, *shortest distance with movement priority*, and *shortest distance with load balancing*, and two different movement policies, namely *up priority*, and *maintain movement*. We define four different configurations for the elevator system, each of them including one of the

aforementioned scheduling and movement policies (not all combinations are considered).

```

1 env int simDelay, simItter, elevMoveDelay;
2 env int doorDelay1, ..., doorDelay4;
3
4 reactiveclass Floor(4) {
5   knownrebecs { Coordinator coord; }
6   statevars {
7     int floorIdent;
8     boolean isRestricted;
9   }
10  msgsrv initial(int floorID) { ... }
11  msgsrv callElevator() {
12    coord.handleRequest(floorIdent,true);
13  }
14 }
15
16 reactiveclass Elevator(4) {
17   knownrebecs { Coordinator coord; }
18   statevars {
19     int movementDelay;
20     boolean isRestricted;
21   }
22  msgsrv initial(int mDelay) { ... }
23  msgsrv moveUp(int floor) {
24    delay(movementDelay);
25    coord.handleElevatorMovement(1,1);
26  }
27  msgsrv moveDown(int floor) { ... }
28  msgsrv stopOpen(int floor, int movementwas) {
29    delay(?(doorDelay1, ..., doorDelay4));
30    coord.handleElevatorMovement(0, movementwas);
31  }
32  msgsrv requestFloor(int floor){
33    // Send request to handler
34    coord.handleRequest(floor,false);}
35  msgsrv stopRequest(int floor) { ... }
36 }
37
38 reactiveclass Coordinator(4) {
39   knownrebecs {
40     Floor flr1, flr2, flr3;
41     Elevator el1, el2;
42   }
43   statevars {
44     int el1loc, el1move, el2loc, el2move,
45       scheDelay;
46     list<int> el1Q, el2Q;
47   }
48  msgsrv initial() { ... }
49  msgsrv handleElevatorMovement(int movement,int
50    movementbefore) {
51    if(sender == el1 && movement != 0) {
52      el1move = movement;
53      if(movement == -1) { el1loc -= 1; }
54      else if(movement == 1) { el1loc += 1; }
55      ...
56    } else if(sender == el1) { ... }
57    if(sender == el2 && movement != 0) { ... }
58    else if(sender == el2) { ... }
59  }
60  msgsrv handleRequest(int floor, boolean
61    isFloor){
62    //Requests from floors.
63    if(isFloor == true) {
64      int choice = ?(1,2);
65      if(erlang.contains(el1Q,floor) != 1 &&
66        erlang.contains(el2Q,floor) != 1) {
67        if(el1loc == floor || el2loc == floor) {
68          if(el1loc==floor) el1Q.insert(floor);
69          else if(el2loc==floor)
70            el2Q.insert(floor);
71        } else if(el1loc == el2loc) {
72          if(choice == 1) el1Q.insert(floor);
73          else if(choice == 2) el2Q.insert(floor);
74        }
75        ...
76      } else { ... }
77      if(el1move == 0 && el1Q.size() > 0) { ... }
78      if(el2move == 0 && el2Q.size() > 0) { ... }
79    }
80    // Requests from elevator.
81    else { ... }
82  }
83 }
84
85 reactiveclass Person(4) {
86   knownrebecs {
87     Floor flr1, flr2, flr3;
88     Elevator el1, el2;
89   }
90   statevars {
91     int delayinsec, itterations;
92   }
93  msgsrv initial(int d, int i) {
94    delayinsec = d;
95    itterations = i;
96    self.go(delayinsec,0);
97  }
98  msgsrv go(int delays, int incritt) {
99    int fc = ?(1, 2, 3);
100   int flr = ?(1, 2, 3);
101   int elv = ?(1,2);
102   if(fc == 1 || fc == 2) {
103     if(fl1 == 1) { flr1.callElevator(); }
104     if(fl1 == 2) { flr2.callElevator(); }
105     if(fl1 == 3) { flr3.callElevator(); }
106   } else {
107     if(elv == 1) { el1.requestFloor(fl1); }
108     else if(elv == 2) { el2.requestFloor(fl1); }
109   }
110   delay(delays);
111   if(incritt < itterations) {
112     self.go(delayinsec,incritt+1);
113   }
114 }
115
116 main {
117   Elevator el1(coord):(elevMoveDelay);
118   Elevator el2(coord):(elevMoveDelay);
119   Floor flr1(coord):(1);
120   Floor flr2(coord):(2);
121   Floor flr3(coord):(3);
122   Coordinator coord(fl1, flr2, flr3, el1,
123     el2):();
124   Person pers(fl1, flr2, flr3, el1, el2):( ...
125     );
126 }

```

Listing 3.11: The Timed Rebeca model of the elevator system.

### 3.5.2.2 Model Checking Using McErlang Monitors

The complete Timed Rebeca model for the elevator system can be found at [85] and [1]. To avoid state space explosion, we use the model with three floors for model checking (as shown in Listing 3.11). We use checkpoint monitors as discussed in Section 3.2, to verify the safety properties of the model.

The first safety property which is verified to ensure the correctness of the model is the value of the elevator location. This value must be within the valid range which is one to three. In the Timed Rebeca model, the checkpoint `elevatorLocation` is defined to make the value of elevator locations available for model checking. To check the maximum and minimum value of checkpoint `elevatorLocation`, we use the predefined functions `checkTermMaxValue` and `checkTermMinValue` respectively, as shown in Listing 3.12.

```

1 monitorType() -> safety.
2 init(_) -> {ok, satisfied}.
3 stateChange(_,satisfied,Stack) ->
4   Actions = actions(Stack),
5   checkTermMinValue(Actions,elevatorLocation,0),
6   checkTermMaxValue(Actions,elevatorLocation,3),
7   checkTermValue(Actions,elevator1StopReqInList,-1),
8   checkTermValue(Actions,elevator2StopReqInList,-1),

```

Listing 3.12: Checkpoint monitor for the elevator system with three floors.

We are also interested in checking whether the elevators stop on the floors which are not requested. The predefined function `CheckTermValue` is used to check whether the values of checkpoints `elevator1StopReqInList` and `elevator2StopReqInList` both equal -1, which means the elevator stops at incorrect floors. The results of model checking the Elevator model, using the mentioned properties, are shown in Table 3.6.

Parameter	Condition	Result
Elevator location	Location > 0	Satisfied (40929 states) 112.4 seconds
Elevator location	Location < 3	Satisfied (40929 states) 111.6 seconds
Stop Queue 1	$\neq -1$	Satisfied (40929 states) 110.5 seconds
Stop Queue 2	$\neq -1$	Satisfied (40929 states) 109.5 seconds

Table 3.6: Safety verification results for the elevator system.

### 3.5.2.3 Statistical Model Checking

In the previous section we checked the elevator system moving between three floors. Here, we model check a larger elevator model by increasing the number of floors to ten, for which the monitor-based model checking is not applicable because the state space is very large. To have the elevator system with ten floors, some parts of the code in Listing 3.11 (e.g. the main part) must change. The checkpoints `elevatorLocation1` and `elevatorLocation2` are defined in the model to make the value of elevator locations (the floor numbers from which the elevator passes) available for model verification. We

injected a bug in the model to provide a few situations in which the elevators can go to floors which do not exist. We check whether elevators stop at the correct floors ranging from one to ten. We define two safety properties: the elevator location is greater than zero, and the elevator location is less than or equal to ten.

Here, we use McErlang to simulate the Erlang program obtained from a Timed Rebeca model. Each simulation run generates a trace from the possible set of traces, through selecting the next state of the trace randomly. In this way, all nondeterminisms are resolved using uniform distributions. The chosen trace is investigated for the defined property. To get the needed set of traces for statistical model checking, we use the *simulation wrapper* component to run different simulations, each with 15000 random floor requests with a delay of 2 time units. The delay of the elevator movement is 2 time units and the delay of an elevator door opening and closing is set to a nondeterministic choice of 1, 2, 4 or 6 time units. The nondeterministic assignment is a syntactic sugar for a probabilistic assignment with a uniform distribution among its choices. All these parameters are set by using environment variables in Listing 3.11 (Lines 1-2).

Table 3.7 shows the model checking results for the safety property of “the location of elevator1 is less than or equal to 10”. The approximation of correctness of the property is calculated for different error values ( $\epsilon$ ) and confidence values ( $\delta$ ). To understand the way of computing the approximation of correctness and its meaning, we explain the first experiment of Table 3.7 in more detail.

Considering error value (0.01) and confidence value (0.1),  $N_{ct} = 1189$  traces have to satisfy the defined property (refer to Section 3.5.1.2 for formulas). The total simulation traces to get this number of satisfied traces is  $N' = 1248$ . This means that some traces do not satisfy the property as we expected. So, in this experiment the approximation of correctness is  $\tilde{\mu}_Z = 0.953$ . More accurately, we obtain an  $(\epsilon, \delta)$ -approximation of the approximation of correctness where  $Pr[|\mu_Z - \tilde{\mu}_Z| < \epsilon] \geq 1 - \delta$ ,  $\mu_Z$  is the real approximation of correctness. Therefore, for the first experiment of Table 3.7,  $Pr[|\mu_Z - 0.953| < 0.01] \geq 0.9$ .

Tables 3.8, 3.9, and 3.10 show the model checking results for other safety properties. In this section, for each simulation trace of each experiment, 150 floor requests are sent randomly to the elevators, where requests are sent every 2 units of time. Also, movement between floors takes 2 units of time. The needed time for opening and closing of an elevator door is set nondeterministically to 1, 2, 4 or 6 time units. The scheduling policy is *shortest distance* and the movement policy is *up priority*. The detailed explanations on different policies can be found in Section 3.5.2.4.

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for Elevator1location $\leq 10$
1	1189	1248	0.01	0.1	0.953
2	523	556	0.03	0.03	0.941
3	289	296	0.05	0.05	0.976
4	203	210	0.1	0.01	0.967

Table 3.7: Statistical model checking results for the elevator system. The approximation of correctness is calculated for the safety property “the elevator1 location is less than or equal to 10”.

To show the applicability of our approach for larger models, we increase the number of floors to 15 and 20. For these two extended models, we check the safety property

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for Elevator2location $\leq 10$
1	1189	1256	0.01	0.1	0.947
2	523	548	0.03	0.03	0.954
3	289	306	0.05	0.05	0.945
4	203	214	0.1	0.01	0.949

Table 3.8: Statistical model checking results for the elevator system. The approximation of correctness is calculated for the safety property “the elevator2 location is less than or equal to 10”.

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for Elevator1location $> 0$
1	1189	1189	0.01	0.1	1
2	523	523	0.03	0.03	1
3	289	289	0.05	0.05	1
4	203	203	0.1	0.01	1

Table 3.9: Statistical model checking results for the elevator system. The approximation of correctness is calculated for the safety property “the elevator1 location is greater than zero”.

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for Elevator2location $> 0$
1	1189	1189	0.01	0.1	1
2	523	523	0.03	0.03	1
3	289	289	0.05	0.05	1
4	203	203	0.1	0.01	1

Table 3.10: Statistical model checking results for the elevator system. The approximation of correctness is calculated for the safety property “the elevator2 location is greater than zero”.

that the elevator1 location shouldn’t exceed the number of floors. The obtained results are shown in Tables 3.11 and 3.12.

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for Elevator1location $\leq 15$
1	289	292	0.05	0.05	0.99
2	203	208	0.1	0.01	0.976

Table 3.11: Statistical model checking results for the elevator system with 15 floors. The approximation of correctness is calculated for the safety property “the elevator1 location is less than or equal 15”.

### 3.5.2.4 Performance Evaluation

In this section, we explain different scheduling and movement policies which are implemented in the message servers `handleRequest` and `handleElevatorMovement`, respectively. We consider four different scenarios, each of them with different scheduling and movement policies. The efficiency of the proposed scenarios is revealed by comparing

Experiment#	Number of satisfied traces	Total number of traces	Error value ( $\epsilon$ )	Confidence value ( $\delta$ )	Approximation of correctness for ElevatorLocation $\leq 20$
1	289	295	0.05	0.05	0.98
2	203	209	0.1	0.01	0.971

Table 3.12: Statistical model checking results for the elevator system with 20 floors. The approximation of correctness is calculated for the safety property “the elevator location is less than or equal 20.”.

the mean response time of the scenarios. The simulation of the scenarios take place with the same settings to be able to compare the simulation results.

**Scheduling Policy** *Shortest distance, shortest distance with movement priority, and shortest distance with load balancing* are three different scheduling policies which are studied in the experiments. Listing A.1 in Appendix A shows the message server `handleRequest` in which two different requests are handled. First, the requests sent to a floor are enqueued in the nearest elevator to the floor based on the *shortest distance* scheduling policy. Second, the requests sent to an elevator are enqueued in it.

In the second algorithm which is shown in Listing A.2 in A, both the moving direction of the elevator and shortest distance are taken into account to enqueue the requests in the elevators. In this approach, for assigning a request to an elevator, the moving direction of the elevators has precedence over the distance of the elevators to the floor from which the request is sent. For example, in the case that `e11` is not moving towards the requested floor and `e12` is moving towards it, although the new request is closer to `e11`, it is enqueued in the queue of `e12`.

The third scheduling policy is implemented as shown in Listing A.3 in A. Here the main goal is to balance the number of the requests assigned to the elevators, called *load balancing* policy. We also consider the *shortest distance* approach. The queue size of elevators has preference over the distance of a request from the elevators. For example, if the requested floor is closer to `e12`, and the queue size of `e11` is less than the queue size of `e12`, then the requested floor is enqueued in the queue of `e11`.

**Movement Policy** We implemented two movement policies which are *up priority* and *maintain movement*. Listing A.4 in A shows the message server `handleElevatorMovement`, in which the *up priority* movement policy is implemented. The policy implies that the elevator attempts to go up first and serve the requests at the higher floors. This message server updates the elevator location and simulates its movement between different floors.

Listing A.5 in A represents the pseudo code of *maintain movement* policy. In this policy, if the elevator is moving upward (downward) and there are requests from higher (lower) floors, the elevator will continue the moving direction and serve the requests; otherwise it changes its moving direction. In other words, the elevator responds to all requests on its way.

**Simulation Results** We consider four different configurations in which scheduling and movement policies are different:

- configuration 1: scheduling policy: *shortest distance*, movement policy: *up priority*

- configuration 2: scheduling policy: *shortest distance*, movement policy: *maintain movement*
- configuration 3: scheduling policy: *shortest distance with movement priority*, movement policy: *maintain movement*.
- configuration 4: scheduling policy: *shortest distance with load balancing*, movement policy: *maintain movement*.

For each configuration, we used the *simulation wrapper* component to execute 10 simulations, each with 15000 random floor requests with a delay of 2 time units. The delay of the elevator movement is 2 time units and the delay of an elevator door opening and closing is set to a nondeterministic choice of 1, 2, 4 or 6 time units.

The results of the analysis of four configurations are shown in Tables 3.13, 3.14, 3.15 and 3.16. Each row of the tables represents the mean response time to requests of a specific floor. The margin error is calculated for three different confidence levels of 99%, 95%, and 90%.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	58.5± 2.83	58.5± 2.16	58.5± 1.81	76.2	29.0	683	1	4772
2	44.4± 2.1	44.4± 1.6	44.4± 1.34	61.0	18.0	564	1	5591
3	33.1± 1.46	33.1± 1.12	33.1± 0.93	46.1	14.0	467	1	6568
4	24.5± 0.92	24.5± 0.7	24.5± 0.58	30.6	12.0	317	1	7361
5	20.6± 0.63	20.6± 0.48	20.6± 0.4	21.6	13.0	196	1	7880
6	17.5± 0.4	17.5± 0.31	17.5± 0.26	14.6	13.0	131	1	8182
7	14.6± 0.3	14.6± 0.23	14.6± 0.19	10.9	12.0	85	1	8615
8	13.4± 0.29	13.4± 0.22	13.4± 0.18	10.6	11.0	82	1	8966
9	14.7± 0.34	14.7± 0.26	14.7± 0.22	12.3	11.0	89	1	8777
10	18.0± 0.37	18.0± 0.28	18.0± 0.24	13.3	15.0	99	1	8442

Table 3.13: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement Policy: **Up priority**. SD, WCT, BCT stands for standard deviation, worst-case time and best-case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	21.6± 0.42	21.6± 0.32	21.6± 0.27	15.5	18.0	95	1	9004
2	17.3± 0.37	17.3± 0.28	17.3± 0.24	14.1	12.0	87	1	9508
3	14.8± 0.30	14.8± 0.23	14.8± 0.19	11.7	11.0	68	1	9926
4	14.6± 0.27	14.6± 0.21	14.6± 0.17	10.5	12.0	72	1	9915
5	14.7± 0.247	14.7± 0.188	14.7± 0.158	9.5	12.0	65	1	9762
6	14.6± 0.25	14.6± 0.191	14.6± 0.16	9.7	12.0	62	1	9915
7	14.3± 0.27	14.3± 0.205	14.3± 0.17	10.4	11.0	77	1	9919
8	14.8± 0.3	14.8± 0.23	14.8± 0.19	11.8	11.0	80	1	9930
9	17.1± 0.36	17.1± 0.28	17.1± 0.23	13.9	12.0	81	1	9555
10	21.7± 0.42	21.7± 0.32	21.7± 0.27	15.5	17.0	86	1	9021

Table 3.14: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst-case time and best-case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	28.3± 0.622	28.3± 0.474	28.3± 0.397	19.9	24.0	99	1	6767
2	22.4± 0.534	22.4± 0.407	22.4± 0.341	17.9	17.0	92	1	7420
3	18.7± 0.426	18.7± 0.325	18.7± 0.272	15.0	14.0	90	1	8168
4	16.7± 0.349	16.7± 0.267	16.7± 0.223	12.5	14.0	78	1	8444
5	16.3± 0.307	16.3± 0.234	16.3± 0.196	11.0	14.0	67	1	8457
6	16.2± 0.3	16.2± 0.229	16.2± 0.192	10.9	14.0	63	1	8688
7	16.8± 0.344	16.8± 0.262	16.8± 0.219	12.3	14.0	73	1	8449
8	18.6± 0.427	18.6± 0.326	18.6± 0.273	15.0	14.0	79	1	8142
9	21.6± 0.516	21.6± 0.393	21.6± 0.329	17.6	17.0	92	1	7691
10	28.1± 0.618	28.1± 0.471	28.1± 0.394	19.9	24.0	103	1	6843

Table 3.15: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with movement priority**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst-case time and best-case time, respectively.

Floor	Mean (0.99)	Mean (0.95)	Mean (0.9)	SD	Median	WCT	BCT	Checkpoint Pairs
1	28.1± 0.5	28.1± 0.381	28.1± 0.319	16.4	28.0	79	1	7096
2	22.9± 0.452	22.9± 0.345	22.9± 0.289	15.3	21.0	76	1	7554
3	18.9± 0.36	18.9± 0.282	18.9± 0.236	13.0	16.0	67	1	8161
4	16.8± 0.306	16.8± 0.234	16.8± 0.195	10.9	14.0	64	1	8354
5	15.5± 0.255	15.5± 0.194	15.5± 0.163	9.2	14.0	53	1	8600
6	15.6± 0.262	15.6± 0.2	15.6± 0.167	9.5	14.0	52	1	8695
7	16.5± 0.305	16.5± 0.232	16.5± 0.194	10.9	14.0	63	1	8457
8	19.2± 0.378	19.2± 0.288	19.2± 0.241	13.2	16.0	66	1	8071
9	22.7± 0.447	22.7± 0.341	22.7± 0.285	15.2	21.0	68	1	7627
10	28.4± 0.508	28.4± 0.387	28.4± 0.324	16.7	28.0	85	1	7140

Table 3.16: Paired-checkpoint Analysis - Scheduling policy: **Shortest distance with load balancing**. Movement policy: **Maintain movement**. SD, WCT, BCT stands for standard deviation, worst-case time and best-case time, respectively.

Table 3.17 shows the mean response time to all floor requests of each configuration. It shows that the configuration of shortest distance policy as scheduling policy and maintain movement policy as movement policy results in the optimum solution among the suggested configurations. Although shortest distance with the movement priority policy may seem to have better performance, experimental results show otherwise.

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

Table 3.17: Simulation results for different configurations of the elevators system. Each row contains the results related to all floor requests of each configuration.

### 3.6 Related Work

Comparing to Erlang which is a functional actor-based programming language, Timed Rebeca is an imperative actor-based modeling language. So, by using Timed Rebeca while respecting the actor programming style we can write our code in an imperative style which is more familiar to most of the programmers nowadays. Moreover, by using Timed Rebeca we are using a model-driven development approach. We can start with small models and use model checking and simulation to find possible correctness problems in our core algorithms, and also find how to improve the performance by changing some parameters while the code is still small, understandable, and easily manageable.

The authors in [94] present an approach to verify safety properties of Erlang-like, higher-order concurrent programs automatically. Following the Core Erlang [95],  $\lambda$ Actor is introduced as a prototypical functional language which is augmented with asynchronous message-passing concurrency and dynamic process creation. The authors formalize an abstract model of  $\lambda$ Actor programs, called Actor Communicating System (ACS). A tool is developed to generate an ACS from an annotated Erlang module, for which safety properties like unreachability of error program locations and mutual exclusion can be defined. This approach starts from an implemented code, while using Timed Rebeca we start from a model. The same discussion holds here as the one comparing Erlang and Timed Rebeca.

Two of the mostly used timed modeling languages are UPPAAL [18] and real-time Maude [19]. UPPAAL is an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata [15], extended with data types (bounded integers, arrays etc.). The tool is currently the most well-known model checker for real-time systems. The modeling languages used by Timed Rebeca and UPPAAL differ greatly, while Timed Rebeca has a programming-like syntax, UPPAAL uses automata. UPPAAL is more convenient for modeling systems with synchronous agents while Timed Rebeca focuses on distributed and asynchronous agents. Modeling the message queue can cause state space explosion in UPPAAL very quickly. The verification tools are different in Timed Rebeca and UPPAAL. Timed properties can be checked in UPPAAL while in this work we focus on checking Timed Rebeca safety properties, which is explained in Section 3.3.

Real-time Maude is a language accompanied with a tool for the formal specification and analysis of real-time and hybrid systems. The specification formalism is based on rewriting logic, and emphasizes generality and ease of specification, and is suitable to specify object-oriented real-time systems. The tool offers a wide range of analysis techniques, including timed rewriting for simulation purposes, and time-bounded linear temporal logic model checking. Timed Rebeca and Real-Time Maude are different in the computational paradigms that they naturally support. Timed Rebeca is based on actor model of computation while you are free in your modeling style using real-time Maude. Timed Rebeca benefits from its similarity with other commonly used programming languages and is more susceptible to get used by modelers without intimate knowledge of formal methods.

In [96], authors introduce UPPAAL SMC in which systems are represented via networks of automata. In UPPAAL SMC, each component of the system is modeled with an automaton whose clocks can evolve with various rates. To provide efficient analysis of probabilistic properties, statistical model checking is used as a technique for fully stochastic models. The work supports modeling and performance analysis of systems

with continuous time behaviors and dynamical features. The modeling languages used in Timed Rebeca and UPPAAL SMC are different. In UPPAAL SMC time is continuous, but in Timed Rebeca time is discrete. In this work, timed performance and functional properties are supported, but in UPPAAL SMC probabilistic performance properties are validated.

There are some works on safety critical real-time Java programs [97], [98] and WCET analysis of Java Bytecode-based programs [99], [100]. A new approach is presented in [97] for schedulability analysis of Safety Critical Hard Real-time Java programs. The approach is based on a translation of programs, written in the Safety Critical Java (SCJ) [101], to timed automata models which are verified by the UPPAAL model checker. In this approach, worst case execution time (WCET) calculation and schedulability analysis are performed to verify that deadline misses never occur. The authors in [99] present a tool for statically determining the WCET of Java Bytecode-based programs. In this approach, the Java program, the JVM, and the hardware are modeled as Networks of Timed Automata (NTA) and given to the UPPAAL model checking tool. While the above works only support schedulability analysis of Java programs, verification of any safety property will be possible in Timed Rebeca if the property can be defined by a checkpoint function. Additionally, performance evaluation of Timed Rebeca models is also provided in this work. Moreover, the modeling paradigm is different in Timed Rebeca and Real-time Java.

Regarding other analysis techniques and tools for Timed Rebeca, a new approach was proposed for schedulability and deadlock freedom analysis of Timed Rebeca models in [102]. The authors proposed the notion of Floating Time Transition System (FTTS) for which the formal definition is presented. The authors proved a bisimulation relation between FTTS and the transition system derived from the SOS rules of Timed Rebeca in [38]. They developed a verification tool based on FTTS and integrated it in the Afra toolset [85]. In this work, the verification of Timed Rebeca models is restricted to deadlock freedom and schedulability analysis, and the performance evaluation of Timed Rebeca models is not supported. The direct model checking approach of TCTL properties for Timed Rebeca models in [103] has the same limitation; however, it verifies majority of TCTL formulas in  $O(n^2 \cdot |\Phi|)$  for a given formula  $\Phi$ . This order is the most efficient algorithm for verification of TCTL formulas in discrete time systems which is the same as the order of the verification of CTL formulas.

Another work on verification of Timed Rebeca models is presented in [104]. In this paper, authors defined an executable formal semantics for Timed Rebeca in Real-Time Maude. This enables a wide range of formal analysis methods for Timed Rebeca models, including simulation, reachability analysis, and both timed and untimed temporal logic model checking. The presented semantics executes all deterministic instantaneous statements in a message server in a single “atomic” step. This approach significantly reduces the number of interleavings and drastically improves the performance of model checking analyses. In addition, in this work, dynamic topology and dynamic creation in Timed Rebeca models is supported. Although the proposed approach covers analysis of an extended version of Timed Rebeca, there is no way for using high-level user defined functions in the models. These functions must be defined in the Maude language which requires expertise in rewriting logic.



## Chapter 4

# Probabilistic Timed Rebeca: An Actor-based Modeling Language

In this chapter, we propose Probabilistic Timed Rebeca (PTRRebeca) which benefits from modeling features of Timed Rebeca and pRebeca, combining the syntax of pRebeca and Timed Rebeca languages. This aims at enhancing our modeling ability in order to cover more properties, by performance evaluation of probabilistic real-time actors. Although the syntax of PTRRebeca is a combination of Timed Rebeca and pRebeca, their semantics and supporting tools are not applicable for PTRRebeca. Consequently, we propose a semantics to support timing, probabilistic, and nondeterministic features.

To the best of our knowledge, PTRRebeca is the first actor-based language which supports time, probability, and nondeterminism in modeling distributed systems with asynchronous message passing. We propose PTRRebeca on the basis of a study of different distributed and asynchronous applications, studied to identify what is needed for modeling and analysis of those applications, relative to different probabilistic and timed probabilistic models (discrete, continuous, stochastic) proposed in the literature. In PTRRebeca, time is discrete, and discrete probability distributions are used. Using probabilistic and nondeterministic assignments, the computation outcomes and network delays can become probabilistic or nondeterministic. The syntax of PTRRebeca is presented in Section 4.1. We continue to use the ticket service example to explain the modeling features of PTRRebeca. In Section 4.2, the semantics of a PTRRebeca model is defined in timed Markov decision process (TMDP) (presented in [60]). Finally, we present the structural operational semantics of PTRRebeca in Section 4.3. The subjects of this chapter were published in [60] and in the Journal of Science of Computer Programming [61].

### 4.1 Probabilistic Timed Rebeca

PTRRebeca language supports modeling and verification of real-time systems with probabilistic behaviors. In Figure 4.1, we show the extension made to the syntax of Timed Rebeca to build PTRRebeca [60]. In a probabilistic assignment, a value is assigned to the variable with the specified probability. In the probabilistic assignment, all  $e_p$  are real values between 0 and 1, and sum up to 1. Notably, by using probabilistic assignments, the values of the timing constructs (delay, after, and deadline) can also become probabilistic.

```

Model ::= Class* Main
Main ::= main { InstanceDcl* }
InstanceDcl ::= className rebecName(⟨rebecName⟩*) : (⟨literal⟩*);
Class ::= reactiveclass className { KnownRebecs Vars MsgSrv* }
KnownRebecs ::= knownrebecs { VarDcl* }
Vars ::= statevars { VarDcl* }
VarDcl ::= type ⟨v⟩+;
MsgSrv ::= msgsrv methodName(⟨type v⟩*) { Stmt* }
Stmt ::= v = e; | v = ?(e⟨e⟩+); | Call; | if (e) { Stmt* } [else { Stmt* } ]
Call ::= rebecName.methodName(⟨e⟩*)

```

(a) Abstract Syntax of Rebeca

```

Stmt ::= v = e; | v = ?(e⟨e⟩+); | Call; | if (e) { Stmt* } [else { Stmt* } ] | delay(v);
Call ::= rebecName.MethodName(⟨e⟩*) [after(v)] [deadline(v)]

```

(b) Changes in the syntax of Rebeca to build Timed Rebeca

```

Stmt ::= v = e; | v = ?(e⟨e⟩+); | Call; | if (e) { Stmt* } [else { Stmt* } ] |
    delay(v); | v = ?(ep : e⟨ep : e⟩+);

```

(c) Changes in the syntax of Timed Rebeca to build PRebeca

Figure 4.1: (a) Abstract syntax of Rebeca. Angle brackets  $\langle \dots \rangle$  are used as meta parentheses, superscript  $+$  for repetition at least once, superscript  $*$  for repetition zero or more times, whereas using  $\langle \dots \rangle$  with repetition denotes a comma separated list. Brackets  $[ \dots ]$  indicate that the text within the brackets is optional. The symbol  $?$  shows nondeterministic choice. Identifiers *className*, *rebecName*, *methodName*, *v*, *literal*, and *type* denote class name, rebec name, method name, variable, literal, and type, respectively; and *e* denotes an (arithmetic, boolean or nondeterministic choice) expression.

(b) Changes for Timed Rebeca. The timing primitives are added to *Stmt* and *Call* statements. The value of variable *v* in timing primitives is a natural number.

(c) Changes for Probabilistic Timed Rebeca. The probabilistic assignment is added to *Stmt*. The expression *e<sub>p</sub>* denotes an expression which returns a probability. The symbol  $?$  shows either nondeterministic assignment or probabilistic assignment.

Different probabilistic behaviors can be modeled using the PRebeca language, depending on the system under study. We present a simple ticket service system in

Figure 4.1 to illustrate how PRebeca can be applied. Each entity in the system is mapped to an actor in the PRebeca model. The ticket service model includes a customer, a ticket service, and an agent. The customer *c* sends a ticket request by sending the message `requestTicket()` to the agent *a* (line 39). The agent forwards the request to the ticket service *ts* by sending the message `requestTicket()` (line 24). The message `requestTicket()` has a deadline which is set nondeterministically (line 23). The ticket service issues a ticket and replies to the agent request by sending the message `ticketIssued()` (line 13). The agent sends the message `ticketIssued` to the customer to complete the issuing process (line 27). The customer sends a new request after 10 or 30 units of time with probabilities 0.25 or 0.75, respectively (lines 42 and 43).

```

1 | reactiveclass TicketService {
2 |   knownrebecs {
3 |     Agent a;
4 |   }
5 |   statevars {
6 |     int issueDelay;
7 |   }
8 |   msgsrvv initial(int myDelay) {
9 |     issueDelay = myDelay;
10 |   }
11 |   msgsrvv requestTicket() {
12 |     delay(issueDelay);
13 |     a.ticketIssued(1);
14 |   }
15 | }
16 |
17 | reactiveclass Agent {
18 |   knownrebecs {
19 |     TicketService ts;
20 |     Customer c;
21 |   }
22 |   msgsrvv requestTicket() {
23 |     a = ?(4,5);
24 |     ts.requestTicket() deadline(a);
25 |   }
26 |   msgsrvv ticketIssued(byte id) {
27 |     c.ticketIssued(id);
28 |   }
29 | }
30 |
31 | reactiveclass Customer {
32 |   knownrebecs {
33 |     Agent a;
34 |   }
35 |   msgsrvv initial() {
36 |     self.try();
37 |   }
38 |   msgsrvv try() {
39 |     a.requestTicket();
40 |   }
41 |   msgsrvv ticketIssued(byte id) {
42 |     b = ?(0.75:30,0.25:10);
43 |     self.try() after(b);
44 |   }
45 | }
46 |
47 | main {
48 |   Agent a(ts, c):();
49 |   TicketService ts(a):(3);
50 |   Customer c(a):();
51 | }

```

Listing 4.1: The PRebeca model of the ticket service system.

## 4.2 Semantics of Probabilistic Timed Rebeca

In this section, we define the Timed Markov Decision Process (TMDP) semantics of a PRebeca model. Formally, a TMDP is defined as follows [74].

**Definition 5 (Timed Markov Decision Process)** *A timed Markov decision process is a tuple  $(TMDP)T = (S, s_0, Act, \rightarrow, L)$  that consists of the following components:*

- A set of states  $S$  with an initial state  $s_0 \in S$ ,
- A set of actions  $Act$ ,
- A timed probabilistic, nondeterministic transition relation  $\rightarrow \subseteq S \times Act \times \mathbb{N} \times Dist(S)$  such that, for each state  $s \in S$ , there exists at least one tuple  $(s, -, -, -) \in \rightarrow$ ,
- A labeling function  $L : S \rightarrow 2^{AP}$ , where  $AP$  is the set of atomic propositions.  $\square$

The transitions in a TMDP are performed in two steps: given that the current state is  $s$ , the first step is a nondeterministic selection of  $(s, act, d, \nu) \in \rightarrow$ , where  $act$  denotes a possible action and  $d$  specifies the duration of the transition; in the second step, a probabilistic transition to state  $s'$  is made with probability  $\nu(s')$ . Function  $\nu \in Distr(S)$ , where  $Distr(S)$  denotes the set of discrete probability distribution functions over the countable set  $S$ .

In the following, we define some concepts for PRebeca models before turning to the TMDP semantics of PRebeca.

**Definition 6 (Probabilistic Timed Rebeca Model)** *A Probabilistic Timed Rebeca model  $\mathcal{M}$  is the set of rebecs which are concurrently executing.*  $\square$

A computation of a PRebeca model  $\mathcal{M}$  takes place by execution of all rebecs defined in the model according to the SOS-semantics in [38]. For a Probabilistic Timed Rebeca model  $\mathcal{M}$ , the function  $O(\mathcal{M})$  returns all rebecs in the model  $\mathcal{M}$ .

**Definition 7 (State of a PRebeca model in TMDP)** *A state of a PRebeca model  $\mathcal{M}$  is a tuple  $s = \left( \prod_{r_i \in O(\mathcal{M})} (state(r_i) \times pc \times rt) \right) \times \mathbb{T}$ , where  $state(r_i)$  is the state of rebec  $r_i$ ,  $\mathbb{T} \in \mathbb{N}$  is the current time of state,  $pc \in \mathbb{N}$  is the program counter of rebec  $r_i$ , and  $rt \in \mathbb{N}$  is the resuming time of rebec  $r_i$ .*  $\square$

Each rebec of  $\mathcal{M}$  has a state which consists of the values of its state variables, its local time, and its message bag. Functions  $sv(s, r_i)$ ,  $bag(s, r_i)$ , and  $now(s, r_i)$  return the state variable valuation function, the content of message bag, and the local time of rebec  $r_i$  in state  $s$ , respectively. In the TMDP semantics of a PRebeca model, the local times of rebecs have the same value. We define the function  $now(s)$  to access the time in state  $s$ .

The rebec program counter,  $pc$  of rebec  $r_i$  specifies the statement to be executed, and function  $pc(s, r_i)$  returns the value of the program counter of rebec  $r_i$  in state  $s$ . The rebec resuming time,  $rt$  of rebec  $r_i$  determines the time when the statement of the message server of rebec  $r_i$ , pointed to by  $pc$ , is executed. Function  $rt(s, r_i)$  returns the value of the resuming time of rebec  $r_i$  in state  $s$ .

In the initial state, the local times of all rebecs are set to zero, and the constructor of all rebecs are executed to initialize state variables and queues content. Initially, for all rebecs the value of the program counter and the value of the resuming time are supposed to be *null*.

**Definition 8 (The Content of a Message Bag)** *A tuple  $tmmsg = (msgsig, arrival, deadline)$  is a message where  $msgsig$  is the message content,  $arrival$  is the arrival time of the message, and  $deadline$  is the deadline of the message. The arrival time of the message is computed based on the local time of the sender and the value of “after” of the send message statement. The deadline of the message is also computed based on the local time of the sender.*  $\square$

For  $tmmsg \in bag(s, r_i)$ , the functions  $sig(tmmsg)$ ,  $ar(tmmsg)$ , and  $dl(tmmsg)$  return the  $msgsig$ ,  $arrival$ , and  $deadline$  of the message  $tmmsg$ , respectively. The message content  $msgsig$  consists of the message name, the sender, the receiver, and its actual parameters and is shown as “sender  $\rightarrow$  receiver.msgname(parameters)”.

**Definition 9 (Possible Messages)** *The set of messages  $Tmsg = \{tmsg \mid \exists r_i, r_j \in O(M), \exists ar, dl \in \mathbb{N}, tmsg = (r_i \rightarrow r_j.msgname(), ar, dl)\}$  is the set of all possible messages which can be sent by rebeccas  $r_i$  to rebeccas  $r_j$  at arrival time  $ar$  and deadline  $dl$ .*  $\square$

**Definition 10 (Rebec Enabled Messages)** *Enabled messages of a rebeccas are messages whose arrival time is less than the time of state  $s$ :  $em(s, r_i) = \{tmsg \in bag(s, r_i) \mid ar(tmsg) \leq now(s)\}$ .*  $\square$

**Definition 11 (TMDP semantics of a PRebeca model)** *A TMDP of PRebeca model  $\mathcal{M}$  is a tuple  $(S, s_0, Act, \rightarrow, L)$ , where:*

- $S$  is the set of states according to Definition 7,
- $s_0 \in S$  is the initial state,
- $Act$  is a set of  $Tmsg \cup \{\tau\} \cup \mathbb{T}$ , where  $Tmsg$  is the set of all possible messages which can be sent by any rebeccas to its known rebeccas,  $\tau$  is an internal action and  $\mathbb{T} \in \mathbb{N}$  is the progress of time.
- $\rightarrow \subseteq S \times Act \times \mathbb{N} \times Dist(S)$  is the transition relation, where  $(s, act, d, \nu) \in \rightarrow$  if and only if one of the following conditions holds for  $s$ .

1. **(Taking a message for execution)** *If in state  $s$ , there exists  $r_i \in O(\mathcal{M})$  such that  $pc(s, r_i) = \text{null}$  and  $em(s, r_i) \neq \emptyset$ : The execution of  $tmsg \in em(s, r_i)$  results in  $s'$  with probability  $\nu(s') = 1$  and  $d=0$ . In this case  $act$  is equal to  $tmsg$ ,  $tmsg$  is extracted from the message bag of the rebeccas  $r_i$ ,  $pc(s, r_i)$  is set to the first statement of message server  $tmsg$ , and  $rt(s, r_i)$  is set to  $now(s)$ .*
2. **(Internal action  $\tau$ )** *If in state  $s$ , there exists  $r_i \in O(\mathcal{M})$  such that  $pc(s, r_i) \neq \text{null}$  and  $rt(s, r_i) = now(s)$ : The statement of the message server of  $r_i$  specified by  $pc(s, r_i)$  is executed and one of the following cases may occur based on the statement execution:*
  - a) *The statement is an ordinary statement: the execution of statement may change the value of some state variables of the rebeccas  $r_i$  or may induce sending a message to a rebeccas. Then,  $pc(s, r_i)$  is increased by one, the  $act$  is  $\tau$ ,  $d=0$ , and the execution of  $\tau$  results in  $s'$  with probability  $\nu(s') = 1$ .*
  - b) *The statement is a nondeterministic assignment: the execution of nondeterministic assignment  $a = ?(v_1, \dots, v_n)$  results in  $n$  different transitions from  $s$  to states  $s'_1, s'_2, \dots, s'_n$ , where  $a = v_i$  in state  $s'_i$ . For each transition, the  $act$  is  $\tau$ ,  $d=0$ , and the execution of  $\tau$  results in  $s'_i$  ( $1 \leq i \leq n$ ) with probability  $\nu(s'_i) = 1$ .*
  - c) *The statement is a probabilistic assignment: the execution of probabilistic assignment  $a = ?(p_1 : v_1, \dots, p_n : v_n)$  results in a transition from  $s$  to states  $s'_1, s'_2, \dots, s'_n$ , where  $a = v_i$  in state  $s'_i$ . The  $act$  is  $\tau$ ,  $d=0$ , and the execution of  $\tau$  results in  $s'_i$  ( $1 \leq i \leq n$ ) with probability  $\nu(s'_i) = p_i$ .*

- d) *The statement is a delay statement with parameter  $t \in \mathbb{N}$ : the execution of the delay statement does not change  $pc(s, r_i)$  (because the execution of the delay statement is not yet complete), and  $rt(s, r_i)$  is set to  $now(s) + t$ . (note: the value of  $pc(s, r_i)$  will change to the next statement after completing the execution of the delay, which can be seen in item 3.) The act is  $\tau$ ,  $d=0$ , and the execution of  $\tau$  results in  $s'$  with probability  $\nu(s') = 1$ .*

*When the last statement of the message server of  $r_i$  is executed,  $pc(s, r_i)$  is set to null.*

3. **(Progress of time)** *If in state  $s$ , none of the aforementioned conditions in items 1 and 2 hold: this means  $\nexists r_i \in \mathcal{O}(\mathcal{M}), ((pc(s, r_i) = \text{null} \wedge em(s, r_i) \neq \emptyset) \vee (pc(s, r_i) \neq \text{null} \wedge rt(s, r_i) = now(s)))$ . In this case,  $now(s)$  is increased by the minimum amount of  $t_1 \in \mathbb{N}$  such that one of the aforementioned conditions becomes true. If  $pc(s, r_i) \neq \text{null}$  and  $rt(s, r_i) = now(s)$  (the current value of  $pc(s, r_i)$  points at a delay statement),  $pc(s, r_i)$  is increased by one. The act is set to  $t$ ,  $d = t_1$ , and the execution of action time results in  $s'$  with probability  $\nu(s') = 1$ .*

– A labelling function  $L : S \rightarrow 2^{AP}$ .

*When more than one transition is enabled in state  $s$ , a nondeterministic selection is made.*

□

### 4.3 Structural Operational Semantics of PRebeca

We present the TMDP of a PRebeca model as a tuple  $(S, s_0, Act, \rightarrow, \Rightarrow)$  where  $S$  is a set of states,  $s_0$  is the initial state,  $Act$  is a set of actions which consists of  $\tau$ , signatures of all the messages, and  $\mathbb{N}$ . The union of scheduler and msg-fetcher transitions is  $\rightarrow$  (probabilistic transitions) and the set of time-progress transitions (delay transitions) is  $\Rightarrow$ . Scheduler transitions, msg-fetcher transitions, and time-progress transitions are defined in the following paragraphs.

In this section we provide an SOS semantics for PRebeca in the style of Plotkin [56]. The behaviour of PRebeca programs is described by means of transition relations that govern the step-by-step evolution of the system.

The states of the system are tuples  $(Env, B, T)$ , where  $Env$  is a finite set of environments,  $B$  is a bag of messages and  $T$  is a natural number that represents the current time of the system. For each rebecc  $A$  of the system,  $Env$  contains an environment  $\sigma_A$  that is a function that maps variables to their values. Basically,  $\sigma_A$  is the private store of the rebecc  $A$ . Environments contain four special-purpose variables: *self*, which contains the name of the rebecc, *pc*, which stands for *program counter* and contains the code that is currently being executed, *rt*, which stores the resume time of the rebecc, and *sender*, which stores the name of the rebecc that invoked the method that is currently being executed. Whenever a rebecc  $A$  of a reactive class  $O$  is created, an environment  $\sigma_A$  is assumed to be initialized. In particular, the code of each message server  $m$  of  $O$  is loaded in  $\sigma_A(m)$  as a *null*-terminated list of statements.

The bag contains an unordered collection of messages of the form

$$(A_i, m(\bar{v}), A_j, TT, DL).$$

Intuitively, such a tuple says that at time  $TT$  the sender  $A_j$  sent the message to the rebeccas  $A_i$  asking it to execute its method  $m$  with actual parameters  $\bar{v}$ . Moreover this message expires at time  $DL$ .

We denote by  $Tmsg$  the set of all the possible messages. Given a message  $msg \in Tmsg$ ,  $ar(msg)$  denotes the arrival time of the message  $msg$ , that is,  $TT$  in the tuple above. At each step, the system progresses thanks to one of three transition relations:  $\xrightarrow{\tau}$ ,  $\xrightarrow{msg}$  with  $msg \in Tmsg$ , and  $\xrightarrow{n}$  with  $n \in \mathbb{N}$ . Any of these transitions evolves a state  $(Env, B, T)$  into a probability distribution  $pv$  that assigns probability values to states. For readability, we represent  $pv$  as a set of mappings, for instance the probability distribution  $\{(Env, B, T) \mapsto 1\}$  maps the state  $(Env, B, T)$  to probability 1. Whenever more cases need to be specified for  $pv$ , they will be embraced in a large bracket and the mappings involved in the distribution will be graphically clear. States that are not mentioned in  $pv$  are assumed to be mapped to probability 0.

As a convention, whenever we single out an element from a set, as in the sets  $\sigma_A \cup Env$  and  $msg \cup B$ , we will assume that  $\sigma_A \not\subseteq Env$  and  $msg \not\subseteq B$ . Moreover, we will use the notation  $\sigma[x = e]$  to denote the mapping  $\sigma$  where  $x$  is redefined in order to map  $x$  to  $e$ .

The transitions  $\xrightarrow{\tau}$ ,  $\xrightarrow{msg}$ ,  $\xrightarrow{n}$ , are formally defined by the following rules.

$$\begin{aligned}
(scheduler) \quad & \frac{\sigma_A(pc) = s \quad s \neq null \quad \sigma_A(rt) = T \quad (s, \sigma_A[pc = null], Env, B, T) \xrightarrow{s} pv}{(\{\sigma_A\} \cup Env, B, T) \xrightarrow{\tau} pv} \\
(msg-fetcher) \quad & \frac{\sigma_{A_i}(pc) = null \quad TT \leq T \leq DL \quad \sigma'_{A_i} = \sigma_{A_i}[pc = \sigma_{A_i}(m), \sigma_{A_i}(rt) = T, \overline{arg} = \bar{v}, sender = A_j] \quad msg = \{(A_i, m(\bar{v}), A_j, TT, DL)\}}{(\{\sigma_{A_i}\} \cup Env, msg \cup B, T) \xrightarrow{msg} \{(\{\sigma'_{A_i}\} \cup Env, B, T) \mapsto 1\}} \\
(time-progress) \quad & \frac{\begin{array}{cc} (Env, B, T) \xrightarrow{\tau} & (Env, B, T) \xrightarrow{msg} \\ n_1 = \min_{\sigma \in Env} \{\sigma(rt)\} & n_2 = \min_{msg \in B} \{ar(msg)\} \\ T' = \min\{n_1, n_2\} & n = T' - T \end{array}}{(Env, B, T) \xrightarrow{n} \{(Env, B, T') \mapsto 1\}}
\end{aligned}$$

The *(scheduler)* rule is responsible for picking a rebeccas and executing its pending statements. This rule chooses a rebeccas nondeterministically among those for which the program counter still contains statements to execute (conditions  $\sigma_A(pc) = s$  and  $s \neq null$ ). Moreover, a rebeccas is eligible for being chosen only as long as its resume time coincides with the current time (condition  $\sigma_A(rt) = T$ ). Rebeccas that have previously executed a delay statement might have a resume time ahead of the current time and in that case they would not be chosen. The execution of the statement is performed with the auxiliary transition relation  $\xrightarrow{s}$ , described in detail later. Such a transition is responsible for the execution of one statement from the list of statements  $s$ , the first one. It is to notice that the program counter is consumed immediately before the call to statement execution (indeed, the environment  $\sigma_A[pc = null]$  is passed). However,  $s$  might contain more than one statement and, moreover, statements such as if-then-else might imply the execution of further statements (one of the branches). As we will see later, these scenarios are taken care of by the transition  $\xrightarrow{s}$ . This transition will be

responsible to feed the program counter back with the possible leftover statements to be executed.

In our semantics, the transition  $\xrightarrow{s}$  returns the probability distribution  $pv$  for the next state of the system. The (*scheduler*) rule simply uses  $pv$  for the transition. In order to let the system progress after a step, we implicitly assume picking a state of  $pv$  according to its probability.

The (*msg-fetcher*) rule allows the system to progress by picking up a message from the bag and initializing the rebece receiver of the message for the execution of such message. This rule is applicable for a rebece only as long as the latter is not in the phase of executing any other message (condition  $\sigma_{A_i}(pc) = null$ ). Moreover, the message can be picked only while it is not too soon for fetching it nor too late (condition  $TT \leq T \leq DL$ ). The rule prepares the rebece  $A_i$  for the execution of the message  $m$  in the following way.

- The method body of  $m$  is looked up from the environment of  $A_i$  and loaded in the program counter.
- The resume time for  $A_i$  is set to the current time of the system, stating that is to be executed immediately.
- The variable *sender* is set to the sender of the message.
- In executing the method  $m$ , the formal parameters  $\overline{arg}$  are set to the values of the actual parameters  $\overline{v}$ . Methods of arity  $k$  are indeed supposed to have  $arg_1, arg_2, \dots, arg_k$  as formal parameters. This is without loss of generality since such a change of variable names can be performed in a pre-processing step for any program.

The (*time-progress*) rule is responsible for letting time pass for some units of time. This happens when the system has no eligible statements of rebece to execute and no eligible messages that can be picked from the bag (eligible w.r.t. the conditions of rules (*scheduler*) and (*msg-fetcher*), respectively). In such a scenario, the system lets the time pass for the minimum amount of time necessary to enable the rebece whose resume time is the closest to the current time ( $\min_{\sigma \in Env} \{\sigma(rt)\}$ ) or to enable the fetch of a message whose picking time is the closest to the current time ( $\min_{msg \in B} \{ar(msg)\}$ ).

Figure 4.2 shows the SOS rules for the execution of statements in PTRebece. The transition relation  $\xrightarrow{s}$  defines the execution of statements. The general form of this type of transition is  $(s, \sigma, Env, B, T) \xrightarrow{s} pv$ , where  $s$  is a list of statements or a single statement<sup>1</sup>,  $\sigma$  is the local environment where to evaluate statements, and  $Env$ ,  $B$ , and  $T$  are the components of the system state. The step evolves into a probability distribution  $pv$ . Carrying the global bag  $B$  is important because new messages may be added to it with the execution of a statement. The global set of environments  $Env$  is also required because *new* statements create new rebece and may therefore add new environments to it. In the semantics,  $\sigma$  is separated from  $Env$  and passed as a parameter for the sake of clarity and also because nearly every rule needs to readily affect it. A few statements make use of the current time  $T$  which is therefore promoted as parameter as well.

---

<sup>1</sup>We overload  $\xrightarrow{s}$  for lists of statements in rule (*stmts\**). We prefer this presentation rather than splitting  $\xrightarrow{s}$  into two relations or splitting the scheduler into two parts.

$$\begin{array}{l}
\text{(msg)} \quad \frac{pv = (\sigma \cup Env, \{(\sigma(\text{varname}), m(\text{eval}(\bar{v}, \sigma)), \sigma(\text{self}), T + d, T + DL)\} \cup B, T) \mapsto 1}{(\text{varname}.m(\bar{v}) \text{ after}(d) \text{ deadline}(DL), \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(delay)} \quad \frac{pv = (\sigma[rt = T + d] \cup Env, B, T) \mapsto 1}{(\text{delay}(d), \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(assign)} \quad \frac{pv = (\sigma[x = \text{eval}(e, \sigma)] \cup Env, B, T) \mapsto 1}{(x = e, \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(non-det)} \quad \frac{pv = (\sigma[x = \text{eval}(e_i, \sigma)] \cup Env, B, T) \mapsto 1 \quad (\text{with } 1 \leq i \leq n)}{(x = ? e_1 \oplus e_2 \dots \oplus e_n, \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(prob)} \quad \frac{pv = \begin{cases} (\sigma[x = \text{eval}(e_1, \sigma)] \cup Env, B, T) \mapsto p_1 \\ (\sigma[x = \text{eval}(e_2, \sigma)] \cup Env, B, T) \mapsto p_2 \\ \dots \\ (\sigma[x = \text{eval}(e_n, \sigma)] \cup Env, B, T) \mapsto p_n \end{cases}}{(x = ? p_1 : e_1 \oplus p_2 : e_2 \dots \oplus p_n : e_n, \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(create)} \quad \frac{\begin{array}{l} \sigma_A = \text{initialEnviroment}(O) \quad \text{with } A \text{ fresh in } \sigma \cup Env \\ pv = (\sigma[\text{varname} = A] \cup \{\sigma_A[\text{self} = A, pc = null]\} \cup Env, \\ \{(A, \text{initial}(\text{eval}(\bar{v}, \sigma)), \sigma(\text{self})), T, +\infty\} \cup B, T) \mapsto 1 \end{array}}{(\text{varname} = \text{new } O(\bar{v}), \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(cond}_1\text{)} \quad \frac{\text{eval}(e, \sigma) = \text{true} \quad pv = (\sigma[pc = s_1] \cup Env, B, T) \mapsto 1}{(\text{if } (e) \text{ then } s_1 \text{ else } s_2, \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(cond}_2\text{)} \quad \frac{\text{eval}(e, \sigma) = \text{false} \quad pv = (\sigma[pc = s_2] \cup Env, B, T) \mapsto 1}{(\text{if } (e) \text{ then } s_1 \text{ else } s_2, \sigma, Env, B, T) \xrightarrow{s} pv} \\
\\
\text{(stmts}^*\text{)} \quad \frac{(s_1, \sigma, Env, B, T) \xrightarrow{\tau, n} pv \quad \text{inject}(\text{rest}, \sigma(\text{self}), pv) = pv'}{(s_1 :: \text{rest}, \sigma, Env, B, T) \xrightarrow{s} pv'}
\end{array}$$

where the function  $\text{inject}(\text{rest}, \text{ref}, pv)$  is defined below:

$$\begin{array}{l}
\text{if } pv = \begin{cases} (\sigma \cup Env_1, B_1, T) \mapsto p_1 \\ (\sigma \cup Env_2, B_2, T) \mapsto p_2 \\ \dots \\ (\sigma \cup Env_n, B_n, T) \mapsto p_n \end{cases} \text{ where } \sigma(\text{self}) = \text{ref} \\
\text{then } \text{inject}(\text{rest}, \text{ref}, pv) = \begin{cases} (\sigma[pc = \sigma(pc) :: \text{rest}] \cup Env_1, B_1, T) \mapsto p_1 \\ (\sigma[pc = \sigma(pc) :: \text{rest}] \cup Env_2, B_2, T) \mapsto p_2 \\ \dots \\ (\sigma[pc = \sigma(pc) :: \text{rest}] \cup Env_n, B_n, T) \mapsto p_n \end{cases}
\end{array}$$

We assume that the append operation  $::$  is such that  $null :: \text{rest} = \text{rest}$ .

Figure 4.2: SOS rules for the execution of statements of PTRebeca.

For all rules with the exception of  $(prob)$ , the result of the step  $\xrightarrow{s}$  first creates a new state that has a new environment, a new bag and the current time. Then this state is injected into a probability distribution function where it has probability 1. The simple and nondeterministic assignment statements are handled by rules  $(assign)$  and  $(non-det)$ , respectively, and it is easy to see that they follow the schema just depicted: their semantics coincides indeed with the standard one, modulo our injection to a probability distribution.

Rule  $(prob)$  handles the probabilistic assignment  $x =? p_1 : e_1 \oplus p_2 : e_2 \dots \oplus p_n : e_n$ . In such a case,  $n$  states are created that differ only in the assignment to the variable  $x$  for the local environment being used. These states are injected into a probability distribution  $pv$  that maps them to the probabilities  $p_1, p_2, \dots, p_n$ . The rules for the timing primitives deserve some explanation.

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

The creation of new rebeccs is handled by the rule *create*. Whenever a rebecc must be created out of the reactive class  $O$ , we first pick a fresh name  $A$  that it is used to identify the newly created rebecc. The name  $A$  is assigned to the variable *varname* of the sender. We assume a function  $initialEnvironment(O)$  that returns a new environment  $\sigma_A$  that is initialized depending on the specification of the rebecc  $O$ , i.e. inspecting the body of the specification **reactiveclass**  $O \dots$ . In particular, the code of each message server  $m$  of  $O$  is loaded in  $\sigma_A(m)$  as a *null*-terminated list of statements. Ultimately, a message is put in the bag in order to execute the *initial* method of the newly created rebecc.

The reader should recall that the scheduler sets the program counter to *null* before executing a statement (passing  $\sigma_A[pc = null]$  in rule  $(scheduler)$ ). The statements that we have described so far have no continuation and simply leave the *pc* variable set to *null*.

A conditional statement *if* ( $e$ ) *then*  $s_1$  *else*  $s_2$  is different in this respect because after evaluating the guard  $e$  the continuation is either  $s_1$  or  $s_2$ . Rules  $(cond_1)$  and  $(cond_2)$  handle the execution of conditional statements and they take care of setting *pc* to  $s_1$  or  $s_2$  according to the evaluation of  $e$ .

Another rule that affects the *pc* variable is  $(stmts^*)$ . This rule handles the execution of the first statement of a list. After the first statement has been executed, it might return some continuation statements. We therefore need to put these latter statements in front, before evaluating the rest of the original list of statements (*rest*). However, the execution step  $\xrightarrow{s}$  returns a probability distribution. We therefore use an auxiliary function in order to inject these statements in all of the possible states of such a distribution. Precisely, the function  $inject(rest, ref, pv)$  seeks for the private store of the

rebec *ref* in each of the states of the distribution *pv* and queues the statements *rest* in the program counter of those private stores.



## Chapter 5

# Performance Analysis of PRebeca Models

In Chapter 4, we introduced Timed Markov Decision Processes (TMDP) as the semantics of PRebeca, to support timing, probabilistic, and nondeterministic features. TMDP can be regarded as the discrete-time semantics of probabilistic timed automata (PTA) [73], or as variation of interactive probabilistic chains [82]. In this chapter, we present different techniques and corresponding toolsets (available at Rebeca home-page [85]) for the analysis of PRebeca models. The proposed techniques use either PRISM [54] or IMCA (Interactive Markov Chain Analyzer) [55] as the back-end model checker. We employ probabilistic model checking for both functional verification and performance evaluation. The benefits of combining performance evaluation with functional verification are elaborated upon in [47].

There are three different ways of using PRISM as the back-end model checker: 1) standard PRISM input language, 2) explicit engine of PRISM, and 3) parallel composition. The first two approaches are based on TMDP semantics and the last one is based on PTA. These techniques are developed to overcome shortages that we faced in the analysis of case studies with different sizes and needs.

In the first approach, the TMDP of a PRebeca model is constructed in the form of a single, flat Markov Decision Process (MDP) module with an integer-valued variable for time. The MDP module is input to PRISM for the analysis of the PRebeca model. In the second approach, we used the explicit engine of PRISM which works with an intermediate transition matrix representation. In the parallel composition approach, each component (reactive object) in a PRebeca model is mapped to a PTA. Then the parallel composition of PTA (of all components) represents the behavior of the PRebeca model.

As another analysis technique, we use IMCA as the back-end model checker. IMCA accepts Markov Automaton (MA) [57] and Interactive Markov Chain (IMC) [81] models. In order to use IMCA as the back-end model checker, we need to convert the TMDP of an underlying PRebeca model to its corresponding MA. To this end, we can use our previously developed tools to generate the TMDP of our models automatically. The obtained TMDP is converted to its MA which is then the input to IMCA. Using this approach, we are able to evaluate the performance of our models against probabilistic reachability, expected reward reachability, and expected time reachability properties. In Section 5.2, we mathematically prove that the values of expected time reachability in TMDP and its corresponding MA are identical.

For the analysis of properties based on rewards, the basic idea is that probabilistic models can be augmented with costs or rewards, meaning that real values are associated with certain states or transitions of the model. PRISM supports rewards for MDP and PTA models, and IMCA supports rewards for MA. To give some examples, PRISM or IMCA can be used to compute properties such as “expected time”, “expected number of lost messages” or “expected power consumption”. Since there is no practical distinction between cost and reward, the modeler can interpret the values associated to transitions or states as they want.

**Contributions.** The subjects of this chapter were published in [60] and the Journal of Science of Computer Programming [61]. The following contributions are covered in this chapter:

- **Analysis:** we leverage probabilistic model checking algorithms developed for PTA and MDP for the analysis of probabilistic timed properties. For the analysis, we use PRISM [54] as the back-end model checker, so as to also support expected reachability and probabilistic reachability analysis for PTRebeca models.

In another method, we use IMCA as the back-end model checker for PTRebeca, and we use probabilistic model checking algorithms developed for MA to analyze PTRebeca models.

- **Implementation:** we present a tool developed to generate the TMDP of PTRebeca models automatically. The generated TMDP is in the form of an XML file. The XML file is converted to the standard input language of PRISM. In another toolset, the TMDP of a PTRebeca model is converted to an MA, the input language of IMCA.
- **Case studies:** we present a ticket servicing and a toxic gas sensing system application example to demonstrate the feasibility of the approach using PRISM. We also present the analysis of a Network on Chip (NoC) architecture to show the applicability of our approach, which is using IMCA, for a real-world case study.

The rest of this chapter is organized as follows. In Section 5.1, PRISM is used as the back-end model checker for performance analysis of PTRebeca models. Section 5.1.1 includes three case studies, for which the TMDP is generated. The first two case studies are input to PRISM using the standard input language, and the last one uses the explicit engine of PRISM. An alternative approach for performance analysis of PTRebeca models is introduced in Section 5.1.2, which is based on the parallel composition of PTA rather than the TMDP semantics. In Section 5.1.3, the TMDP-based and parallel composition approaches are compared. In Section 5.2, we use IMCA for the analysis of PTRebeca models. This section explains the conversion of the TMPD of a PTRebeca model to an MA. We also examine a few case studies to show the applicability of the approach and the developed toolset. The PRISM-based and IMCA-based approaches are compared in Section 5.3 in terms of the needed time and memory for the analysis of different case studies with different sizes. Finally, the related works are presented in Section 5.4.

## 5.1 Performance Analysis of PTRebeca Models Using PRISM

In this section, we discuss three approaches of modeling and verification of PTRebeca models in PRISM. The first two approaches are based on the TMDP semantics of the PTRebeca language, and the last one is based on the parallel composition of PTA. We need an approach in which two features are preserved: First, the definition of rewards is possible to be able to verify expected reachability properties. Second, model checking of a large PTRebeca model should take a reasonable amount of time.

To be able to implement PTA with digital clocks in PRISM, there are some restrictions in [73]. It does not allow atomic constraints of the form  $x > c$  or  $x < c$  (closed) or  $x - y \sim c$  (diagonal free), where  $c \in \mathbb{N}$ ,  $\sim \in \{\leq, =, \geq\}$ ,  $x$  and  $y$  are different clocks. In this way, the “digital clocks” engine of PRISM is used, and PTA modules are defined. In another way, we can consider MDP modules with integer-valued variables representing clocks in order to define PTA with digital clocks in PRISM. Using this approach, there is no need to satisfy the above restrictions. In an MDP module, variables can be compared together without any limitations.

In [73], the semantics of probabilistic timed automata is defined in terms of *timed probabilistic systems*, which show timed, nondeterministic, and probabilistic behaviours. They are a variant of Markov decision processes [105] and Segala’s PTA [106], which are defined as follows.

**Definition 12 (PTA)** *A probabilistic timed automaton is a tuple  $(L, \bar{l}, \chi, \sum, I, \text{prob})$  where:  $L$  is a finite set of locations including the initial location  $\bar{l}$ ;  $\chi$  is a set of clocks;  $\sum$  is a finite set of events; the function  $I : L \rightarrow \text{Zones}(\chi)$  is the invariant condition; and the finite set  $\text{prob} \subseteq L \times \text{Zones}(\chi) \times \sum \times \text{Dist}(2^\chi \times L)$  is the probabilistic edge relation.*  $\square$

Let  $\mathbb{T} \in \{\mathbb{R}, \mathbb{N}\}$  be the time domain of either the non-negative reals or naturals. A point  $v \in \mathbb{T}^{|\chi|}$  is referred to as a *clock valuation*. Let  $\mathbf{0} \in \mathbb{T}^{|\chi|}$  be the clock valuation which assigns 0 to all clocks in  $\chi$ . For any  $v \in \mathbb{T}^{|\chi|}$  and  $t \in \mathbb{T}$ , the clock valuation  $v \oplus t$  denotes the *time increment* of values in  $v$  by  $t$ . We use  $v[X := 0]$  to denote the clock valuation obtained from  $v$  by resetting all of the clocks in  $X \subseteq \chi$  to 0. A zone is the solution set of a clock constraint, that is the maximal set of clock assignments satisfying the constraint. Let  $\text{Zones}(\chi)$  be the set of zones over  $\chi$ , which are conjunctions of atomic constraints of the form  $x \sim c$  for  $x \in \chi$ ,  $\sim \in \{\leq, =, \geq\}$ , and  $c \in \mathbb{N}$ . The clock valuation  $v$  satisfies the zone  $\zeta$ , written  $v \models \zeta$ , if and only if  $\zeta$  resolves to true after substituting each clock  $x \in \chi$  with the corresponding clock value from  $v$ . A state of a PTA is a pair  $(l, v)$  where  $l \in L$  and  $v \in \mathbb{T}^{|\chi|}$  are such that  $v \models I(l)$ .

**Standard PRISM input language (based on the TMDP semantics).** In this approach, the TMDP semantics of a PTRebeca model is input in the form of the standard input language of PRISM. In Section 5.1.1, we examine different case studies with different sizes. When the PTRebeca model is small, like the ticket service example, the model checking is fast and takes a few seconds. When the model has a medium size, like the toxic gas sensing system example, its corresponding MDP module (with an integer-valued variable for time) includes many states and transitions and its analysis takes a few minutes. Obviously, model checking a large case study will take more time. To support this claim, in Section 5.3 we report the time and memory needed

to evaluate different PRebeca models with PRISM as the back-end model checker. Although this approach supports the definition of rewards, model checking of large PRebeca models takes a significant amount of time.

**Explicit engine of PRISM (based on the TMDP semantics).** To tackle the problem mentioned above, instead of using the standard PRISM input language, we decided to use the possibility of constructing models in PRISM through direct specification of transition and state matrices. In Section 5.1.1, we used this method for the toxic gas sensing system case study. We provided the corresponding MDP (with an integer-valued variable for time) in the form of its transition and state matrices, and input the matrices into PRISM for model checking. This allows us to analyze larger models, but PRISM does not provide full support for this format, specifically rewards are not supported. Therefore, we were only able to use it for the analysis of probabilistic reachability properties, but not for the expected reachability ones.

**Parallel composition approach (based on PTA).** To provide faster model checking and support rewards for PRebeca models, we introduce the parallel composition approach for PRebeca models which can be verified using PRISM. In this approach, rewards can be defined and so the evaluation of both expected reachability and probabilistic reachability properties is possible. In Section 5.1.2, we first introduce the approach and then we investigate the efficiency of our approach for medium-size and large PRebeca models in terms of the state space size.

### 5.1.1 Analysis of Probabilistic Timed Rebeca based on TMDP

We have developed a tool set [85] in order to generate the TMDP semantics from a PRebeca model. This TMDP semantics can be exported to PRISM as a single MDP module with one integer-valued variable modeling the passage of time. In another way, instead of defining a variable for time, we use dedicated actions starting with the word *time* for time transitions. The passage of time is modeled by assigning an integer value, equal to the intended amount of time, as reward to the time transition. This way, we can analyze expected-time reachability and time-bounded probabilistic reachability properties.

In PRebeca models, the capacity of message bags is bounded. The number of states in a PRebeca model can be finitely represented when the system shows recurrent behavior. We also use the time-shift equivalence approach proposed in [39] to avoid state space explosion otherwise induced by time progress. In this approach, two TMDP states  $s$  and  $t$  (in the sense of Definition 7) are time-shift equivalent if the values of all variables except timing variables, i.e. local time, arrival time, deadline, in states  $s$  and  $t$  are identical. Therefore, the two states can be identified by shifting time.

The PRISM modeling language is a state-based language while the PRebeca language benefits from high-level data structures and constructs which arguably make modeling easier. PRISM models are thus closer to the underlying probabilistic models and therefore we bridge to PRISM at the semantics level. Listing 5.1 displays PRISM code equivalent to the ticket service example presented in Listing 4.1. The timing features of the PRebeca model like after, delay, and message deadline are implemented

by progressing of time, and according to the TMDP semantics. Time transitions with appropriate labels are generated automatically by the toolset (lines 7, 9, 14, and 15). In the “rewards” part, the appropriate rewards are defined for time transitions (lines 19-21). When a time transition is traversed, time progresses with the same value as the transition reward. For example, when the transition labeled with *time\_3* is traversed, time is increased by three. The reason is that we defined a reward of three for this transition (Line 19).

<pre> 1 mdp 2 module PTRebecaSS 3   s : [0..32767] init 1; 4   [c_TRY]s=1 -&gt; (s'=2); 5   [a_SENDRREQUEST]s=2 -&gt; (s'=3); 6   [a_SENDRREQUEST]s=2 -&gt; (s'=4); 7   [time_3]s=5 -&gt; (s'=6); 8   [ts_REQUESTTTICKET]s=3 -&gt; (s'=5); 9   [time_3]s=7 -&gt; (s'=8); 10  [ts_REQUESTTTICKET]s=4 -&gt; (s'=7); 11  [ts_tau_REQUESTTTICKET]s=6 -&gt; (s'=9); 12  [ts_tau_REQUESTTTICKET]s=8 -&gt; (s'=9); 13  [a_SENDRREQUEST]s=9 -&gt; (s'=10); 14  [time_30]s=11 -&gt; (s'=1); </pre>	<pre> 15 [time_10]s=12 -&gt; (s'=1); 16 [c_GETTTICKET]s=10 -&gt; 0.75 : (s'=11) +     0.25 : (s'=12); 17 endmodule 18 rewards 19 [time_3] true: 3; 20 [time_10] true: 10; 21 [time_30] true: 30; 22 endrewards </pre>
---	---

Listing 5.1: PRISM code generated from the ticket service example shown in Listing 4.1.

In the following sections, we present three different case studies demonstrating the applicability of the proposed approaches for performance analysis of asynchronous distributed systems. The first two case studies are different versions of the ticket service model shown in Listing 4.1, for which the standard PRISM input language is used. The last case study is a toxic gas sensing system for which the explicit engine of PRISM is used. Since the state space of a toxic gas sensing system is rather large, modeling its MDP module with the use of transition and state matrices is more efficient.

### 5.1.1.1 Performance Analysis of Ticket Service

We extend the simple ticket service model shown in Listing 4.1 to a more complicated scenario detailed in Listing 5.2. We select this simple case study for two purposes: 1) we are able to model check expected reachability properties, which is provided by PRISM for models presented in its standard input language, and 2) we can get an insight about modeling in the PTRebeca language and the analysis in PRISM.

In this case study, there are two customers, two ticket services, and one agent. Each customer sends a ticket issue request to the agent and the agent forwards the request to the first ticket service with probability 0.6, and to the other one with probability 0.4. The ticket service issues a ticket and replies to the agent request. The agent sends the message to the customer to complete the issuing process.

It is essential to ensure that customers get tickets after a number of requests to the system. This shows that the system meets the primitive goal of issuing tickets. There are also some performance measures that show the efficiency of the system, and can be used to understand the system behavior. As an example, if the expected number of requests until a ticket is issued is unacceptably large, we should figure out why some requests are not responded by the ticket services. This may happen because of timing variables. If customers send requests fast (the value of *after* in Line 16 becomes small), the deadline of requests becomes small (Lines 28 and 30), and issuing tickets takes considerable time (the value of *issueDelay* becomes large in Line 44), some requests will be expired before being responded by one of the ticket services. We

should say that this scenario cannot happen in this model as a new request will be sent only if the previous one has been responded.

We analyzed the following probabilistic reachability and expected reachability properties for the ticket service model:

- We check whether eventually tickets are issued for both customers. The property is specified as:  $P \geq 1 [F (c1\_issued = true) \& (c2\_issued = true)]$ . The property is satisfied which is to be expected according to the model.
- We check the maximum expected time until tickets are issued for both customers. Since we define rewards on transitions representing the passage of time, we are able to check such an expected time property. The property is specified by:  $R\{“time”\}max = ?[F (c1\_issued = true) \& (c2\_issued = true)]$ . The result is 3.76 units of time.
- We check the maximum expected number of requests until tickets are issued for both customers. To model check this property, we define rewards on transitions corresponding to a requesting ticket. The property is formulated by:  $R\{“request”\}max = ?[F (c1\_issued = true) \& (c2\_issued = true)]$ . The result is two requests, meaning that tickets are issued for customers after at most two requests. This shows that all requests are responded by the ticket services, because each customer needs to send a request to get a ticket issued.

```

1  reactiveclass Customer(3) {
2    knownrebecs {Agent a;}
3    statevars {byte id;
4      boolean issued;
5    }
6    Customer(byte myId) {
7      id = myId;
8      self.try();
9    }
10   msgsrv try() {
11     issued = false;
12     a.sendRequest(id);
13   }
14   msgsrv getTicket() {
15     issued = true;
16     self.try() after(29);
17   }
18 }
19 reactiveclass Agent(10) {
20   knownrebecs {
21     TicketService ts1, ts2;
22     Customer c1, c2;
23   }
24   statevars { }
25   Agent() { }
26   msgsrv sendRequest(byte id) {
27     if (? (0.6 : true, 0.4 : false))
28       ts1.requestTicket(id) deadline(24);
29   }
30   else
31     ts2.requestTicket(id) deadline(24);
32   }
33   msgsrv sendTicket(byte id) {
34     if (id == 1) c1.getTicket();
35     else if (id == 2) c2.getTicket();
36   }
37 }
38 reactiveclass TicketService(10) {
39   knownrebecs {Agent a;}
40   statevars {int issueDelay;}
41   TicketService(int myIssueDelay) {
42     issueDelay = myIssueDelay;
43   }
44   msgsrv requestTicket(byte id) {
45     delay(issueDelay);
46     a.sendTicket(id);
47   }
48 }
49 main {
50   Agent a(ts1, ts2, c1, c2):();
51   TicketService ts1(a):(2), ts2(a):(3);
52   Customer c1(a):(1), c2(a):(2);
53 }

```

Listing 5.2: The PTRebeca model of ticket service example.

### 5.1.1.2 Performance Analysis of Faulty Ticket Service

We also examine a variation of the ticket service model in which a system fault is injected to the model. The ticket service model presented in Listing 5.3 is similar to the model in Listing 4.1 except that the ticket service only responds to the requests of customer *c1* (line 53). We examine this case study to show that in the presence

of a fault in the system, i.e. not issuing ticket for customer *c2*, it is detected through model checking. We analyzed the following probabilistic properties:

- We check whether eventually the ticket is issued for customer *c2*. The PCTL formula is:  $P \geq 1 [F (c2\_ticketIssued = true)]$ . The property is not satisfied, which is expected.
- We check the maximum expected time until the ticket is issued for customer *c2*. This property is specified by:  $R\{“time”\}max = ?[F (c2\_ticketIssued = true)]$ . The result is *infinity* since the ticket service never responds to the requests of customer *c2*.
- We check the minimum expected time until the ticket is issued for customer *c2*. This property is specified by:  $R\{“time”\}min = ?[F (c2\_ticketIssued = true)]$ . The result is *infinity* because of the existence of a fault in the system.
- We check the maximum expected number of requests until the ticket is issued for customer *c2*. The property is formulated by:  $R\{“request”\}max = ?[F (c2\_ticketIssued = true)]$ . The result is *infinity*.

```

1 reactiveclass Customer(3) {
2   knownrebecs {
3     Agent a;
4   }
5   statevars {
6     byte id;
7     boolean ticketIssued;
8   }
9   Customer(byte myId) {
10    id = myId;
11    ticketIssued = false;
12    self.try();
13  }
14  msgsrv try() {
15    ticketIssued = false;
16    a.sendRequest(id);
17  }
18  msgsrv getTicket() {
19    ticketIssued = true;
20    int prob = ?(0.3:5,0.7:30);
21    self.try() after(prob);
22  }
23 }
24 reactiveclass Agent(10) {
25   knownrebecs {
26     TicketService ts;
27     Customer c1;
28     Customer c2;
29   }
30   statevars {
31   }
32   Agent() {
33   }
34   msgsrv sendRequest(byte id) {
35     ts.requestTicket(id) deadline(24);
36   }
37   msgsrv sendTicket(byte id) {
38     if (id == 1)
39       c1.getTicket();
40     else if (id == 2)
41       c2.getTicket();
42   }
43 }
44 reactiveclass TicketService(10) {
45   knownrebecs {
46     Agent a;
47   }
48   statevars {
49   }
50   TicketService() {
51   }
52   msgsrv requestTicket(byte id) {
53     if(id ==1){
54       int issueDelay = ?(0.2:40,0.8:10);
55       delay(issueDelay);
56       a.sendTicket(id);
57     }
58   }
59 }
60 main {
61   Agent a(ts, c1, c2):();
62   TicketService ts(a):();
63   Customer c1(a):(1);
64   Customer c2(a):(2);
65 }

```

Listing 5.3: The PTRebeca model of ticket service example with an injected fault.

### 5.1.1.3 Performance Analysis of a Toxic Gas Sensing System

This case study can be considered as a simple example of a wireless sensor and actuator networks (WSANs) application. In WSAN, a number of sensors is spread out in an

environment to sense and potentially also control the environment [107]. The sensors communicate the information through wireless links enabling interaction between people or computers and the surrounding environment. The data gathered by different sensors is sent to a sink which can be used to make an effect on the environment, through for example actuators.

In this case study, we consider a sensor instead of a network of sensors. The system consists of a lab environment in which the level of a toxic gas changes over time. There is one scientist in this lab. If the toxic gas level raises above a certain threshold, the scientist's life is in danger. A sensor in the lab periodically measures the amount of toxicity in the air, and sends the measurements to a central controller. The central controller periodically checks whether the scientist is in danger. If so, it notifies the scientist about the danger. The scientist should acknowledge the notification; if the scientist fails to do so in a timely manner, the central controller notifies a rescue team. When the team reaches the lab, it notifies the controller that the scientist has been rescued. If the controller does not receive this notification, it means that the scientist has lost his life.

**PTRebeca Model.** The PTRebeca model of this system is shown in Listing 5.4, containing four different reactive classes: **Environment**, **Controller**, **Sensor**, and **Scientist**. The toxic level of the environment changes periodically by a probabilistic assignment of line 23. The sensor periodically measures the level of toxic gas by sending a `giveGas` message to the environment (which is modeled in line 44). After sensing, the sensor reports the measured data to the controller. The sensor may fail to report the measured data as shown by the probabilistic assignment of line 46. Upon receiving the measured data from a sensor (in the `report` message server), the controller stores the value in `sensorValue0` (line 82).

Periodically, in the `checkSensors` message server, the controller checks if the reported value is above the normal amount. In case of detecting high toxicity, the controller informs the scientist by sending an `abortPlan` message (line 93), and checks the scientist's acknowledgment after a specified amount of time (line 95). If the controller does not receive an `ack` message from the scientist, the rescue team is informed about the situation. If this process takes more than the value of `scientistDeadline` units of time, the scientist will die and this is modeled by sending a message `die` to the scientist by the environment. This message is scheduled immediately after changing the gas level to the dangerous level (line 25).

In this model, there are different timing variables that can affect the probability of the scientist's death. The values of these variables are specified at the first lines of the model (Lines 1-5). For example, the network delay is assumed to be one time unit (`netDelay`), and the period of checking the sensor's data by controller is set to 5 time units (`controllerCheckDelay`). Note that the value of `checkingPeriod` ranges from 1 to 25 in different experiments.

**Experimental Results.** The main goal of this system is to gather information about the environment and react to a dangerous situation appropriately. Obviously, saving the scientist's life is the ultimate goal. In this case study, we are interested in finding the optimum value of the timing variable `checkingPeriod`. This variable shows the period in which the sensor measures the toxic level of the lab. We run different experiments to find the value of `checkingPeriod` for which the probability of the scientist's death is minimum. In different experiments, we keep the value of all timing variables except

```

1  env byte scientistDeadline = 10;
2  env byte rescueDeadline = 5;
3  env byte netDelay = 1;
4  env byte controllerCheckDelay = 5;
5  env byte sciAckDeadline = 5;
6
7  reactiveclass Environment(10){
8    knownrebecs{
9      Sensor sensor0;
10     Sensor sensor1;
11     Scientist scientist;
12   }
13   statevars{
14     byte gasLevel, changingPeriod;
15     boolean meetDangerousLevel;
16   }
17   Environment(){
18     changingPeriod = 5;
19     gasLevel = 2; // 2 = safe, 4 = dangerous
20     self.changeGasLevel() after(changingPeriod);
21   }
22   msgsrv changeGasLevel(){
23     if(gasLevel == 2) gasLevel=? (0.98:2, 0.02:4);
24     if(gasLevel > 2 && !meetDangerousLevel) {
25       scientist.die() after(scientistDeadline);
26       meetDangerousLevel = true;
27     }
28     self.changeGasLevel() after(changingPeriod);
29   }
30   msgsrv giveGas(){
31     if(sender==sensor0)
32       sensor0.doReport(gasLevel);
33     if(sender==sensor1)
34       sensor1.doReport(gasLevel);
35   }
36   reactiveclass Sensor(7) {
37     knownrebecs {
38       Controller controller;
39       Environment environment;
40     }
41     statevars { int checkingPeriod; }
42     Sensor(int myPeriod) {
43       checkingPeriod = myPeriod;
44       self.checkGasLevel();
45     }
46     msgsrv checkGasLevel() {environment.giveGas();}
47     msgsrv doReport(byte value) {
48       boolean working = ?(0.01:false,0.99:true);
49       if(working){
50         controller.report(value) after(netDelay);
51         self.checkGasLevel() after(checkingPeriod);
52       }
53   }
54   reactiveclass Scientist(7) {
55     knownrebecs { Controller controller; }
56     statevars {boolean isDead, isOutEnv, ackSent;}
57     msgsrv die(){ if(!isOutEnv) isDead = true; }
58     msgsrv abortPlan() {
59       isOutEnv = true;
60       if(!ackSent) controller.ack()
61         after(netDelay);
62       ackSent = true;
63     }
64   }
65   msgsrv leftEnv(){isOutEnv = true;}
66 }
67
68 reactiveclass Rescue(7) {
69   knownrebecs {Controller controller;}
70   msgsrv go() {
71     delay(2); //unexpected obstacle
72     controller.rescuereach() after(netdelay)
73       deadline(rescuedeadline-netdelay);
74   }
75 }
76
77 reactiveclass Controller(13) {
78   knownrebecs {
79     Sensor sensor0;
80     Sensor sensor1;
81     Scientist scientist;
82   }
83   statevars {
84     int sensorValue0;
85     int sensorValue1;
86     boolean scientistAck,scientistDead,ackIsSent;
87   }
88   Controller() { self.checkSensors(); }
89   msgsrv report(int value) {
90     if (sender == sensor0) sensorValue0 = vale;
91     if (sender == sensor1) sensorValue1 = vale;
92   }
93   msgsrv rescueReach() {
94     scientistReached = true;
95     scientist.leftEnv();
96   }
97   msgsrv checkSensors() {
98     boolean danger = false;
99     if (sensorValue0 > 3) danger = true;
100    if (sensorValue1 > 3) danger = true;
101    if(!scientistAck){
102      if (danger) {
103        scientist.abortPlan() after(netDelay);
104        if(!ackIsSent)
105          self.checkScientistAck()
106            after(sciAckDeadline);
107        ackIsSent = true;
108      }
109      self.checkSensors()
110        after(controllerCheckDelay);
111    }
112   }
113   msgsrv ack() {scientistAck = true;}
114   msgsrv checkScientistAck() {
115     if (!scientistAck)
116       rescue.go() after(netDelay);
117     scientistAck = false;
118   }
119 }
120
121 main {
122   Environment environment(sensor0, sensor1,
123     scientist):();
124   Sensor sensor0(controller,environment):(10);
125   Sensor sensor1(controller,environment):(10);
126   Scientist scientist(controller):();
127   Controller controller(sensor0, sensor1,
128     scientist, rescue):();
129   Rescue rescue(controller):();
130 }

```

Listing 5.4: The model of a toxic gas sensing system.

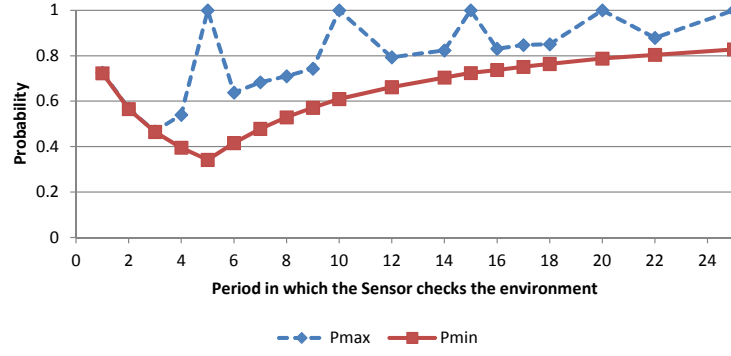
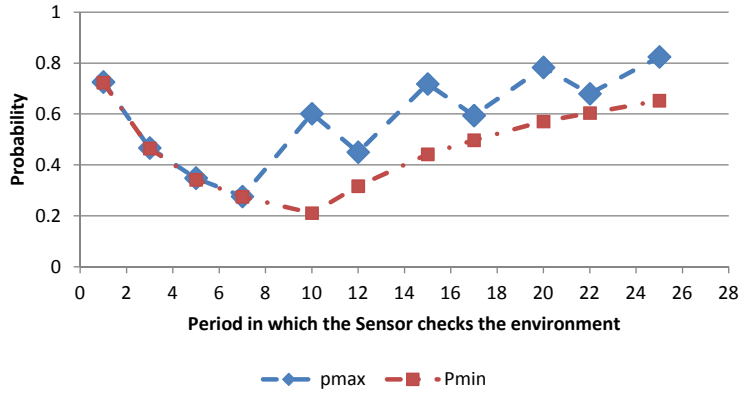
(a) The value of variable *scientistDeadline* is 10.(b) The value of variable *scientistDeadline* is 12.

Figure 5.1: The maximum and minimum probabilities that the scientist eventually dies, when the sensor frequency changes.

*checkingPeriod* fixed, and calculate the probability of the scientist's death. Since the model includes nondeterministic behaviors, the model checker computes the maximum and the minimum probabilities over all paths in the generated state space.

Figure 5.1a shows the maximum and the minimum probabilities of the scientist's death when the value of variable *checkingPeriod* of the sensor changes. If the sensor checks the environment with a high frequency (i.e. the value of variable *checkingPeriod* is low) the probability of sensor failure will increase, resulting in a high probability of the scientist's death. For example, when the sensor checks the environment once every unit of time, the environment is checked five times before the first change in the environment. Therefore, the cost of the sensor use and consequently the probability of sensor failure increases. When the sensor frequency is low, the environment changes cannot be detected on time; resulting in a high probability of the scientist's death. The optimal value for the variable *checkingPeriod* (i.e. sensor frequency) is five according to the obtained results reported in Figure 5.1a for the minimum probability of the scientist's death.

As the results show, the trend of changes in the value of the maximum probability of the scientist's death is almost the same as the minimum probability trend, but at times 5, 10, 15, 20, and 25 it jumps to one. At these times, because of concurrency between time related behaviors in the system, there is a scenario in which the dangerous level is reported too late to the administrator and the scientist will die. At these times, the execution sequence of the following messages is important and causes the special

behavior: (1) checking the sensor value by the controller (it is repeated periodically after 5 units of time), (2) changing the toxic level of the environment to a dangerous level (period is 5 units of time), (3) checking the environment by the sensor (Figure 5.1a shows the probability of the scientist’s death for a different value of this period), and (4) sending a message *die* to the scientist (after 10 units of time) when the environment is dangerous.

In Figure 5.1b, the value of variable *scientistDeadline* equals 12; the scientist has more time to be saved before being killed by the toxic environment. The maximum probability of the scientist’s death is not equal to one at times 5, 10, 15, 20, and 25, but because of concurrency between time related behaviors, there is a scenario in which the dangerous level is reported too late and consequently the maximum probability of the scientist’s death increases. Similar to the previous case, there is an optimum value for the variable *checkingPeriod*, i.e. sensor frequency, which is ten in this experiment.

### 5.1.2 Parallel Composition Approach for Probabilistic Timed Rebeca

Probabilistic timed automata (PTA) are one of the most widely used modeling languages for modeling of real-time probabilistic systems. They are supported by the Modest toolset [108] and by PRISM. An alternative approach for performance analysis of a PRebeca model is a component-wise mapping of the PRebeca model to a number of PTA. The parallel composition of these modules (PTAs) represents the PRebeca model. We optimized the mapping to achieve the smallest possible state space, similar to what we did for the mapping from Timed Rebeca to timed automata in [39]. In the proposed mapping, each rebe is mapped into two timed automata, called *rebec-behavior* automaton and *rebec-bag* automaton. Additionally, one time automaton is defined to handle the behavior of *after* primitive for all rebecs, called *after-handler* automaton.

The *rebec-behavior* automaton models the behavior of a rebe according to the statements of its message servers and valuations of state variables. The state variables of each rebe are mapped into variables of its corresponding *rebec-behavior* automaton and its statements are mapped to transitions of the automaton. The *rebec-bag* automaton handles the behavior of the message bag of each rebe using an internal buffer. The *rebec-bag* accepts messages which are sent to its corresponding rebe asynchronously, regardless of the state of the corresponding *rebec-behavior* automaton. The *after-handler* automaton handles the messages which should be delivered to the *rebec-bag* automaton in the future (messages which are sent by *after* primitive). The *after-handler* automaton accepts messages and puts them into its buffer until the release time of the messages arrives. When a message in the buffer of *after-handler* is released, the message is sent to its corresponding *rebec-bag* automaton. Each PTA can be implemented in PRISM in the form of an MDP module with integer-valued variables representing digital clocks.

In the Rebeca language and its extensions, the execution of message servers is atomic, making the coarse-grain execution of a Rebeca model possible [28]. The coarse-grain execution of message servers reduces the state space size significantly. We use the same approach in the TMDP semantics of a PRebeca model to reach a smaller state space. In the parallel composition approach, we implemented coarse-grain execution by combining statements of different transitions; however, because of synchronization

points among automata, there is a poor chance for combining statements. It is the main obstacle against using a network of PTAs as an ideal approach for PTRebeca models. Different automata need to be synchronized on different points: when a message is sent; when a message is taken from the message bag to start its execution; when a transition modeling a *delay* statement is reached; when it is the time for a sent message to be delivered to its receiver. The mapping from a PTRebeca model to PTA is discussed in more details in the following subsection.

### 5.1.2.1 Mapping from a PTRebeca Model to PTA.

To show the mapping procedure, the PTRebeca model of Figure 4.1 in Section 4.1 is considered and the resulting PTAs are explained. In these PTAs, the condition on a state should be satisfied on its outgoing transitions. It means that, the condition on a state can be considered as a guard for all the outgoing transitions. The transition guard is the same as the one in PTA, meaning that a transition can be enabled if its guard is satisfied. The mapping is not straightforward because in PTRebeca message passing is asynchronous, while in PTA message passing occurs synchronously.

**Rebec-behavior.** The rebec-behavior automaton models the behavior of a rebec according to the statements of its message servers and valuations of variables. To construct the rebec-behavior automaton of a rebec, a corresponding PTA is generated for each message server, and then PTAs (of message servers) are connected together in a way to describe the overall behavior of the rebec. Figures 5.2, 5.3, and 5.4 show the rebec-behavior PTA for the reactive classes of Customer, TicketService, and Agent, respectively.

Here, we explain the mapping of different statements and valuation of state variables.

- **State Variables:** State variables are mapped to variables of the PTA.
- **Ordinary Statements:** The mapping for statements like conditionals, loops, assignments, etc., is straightforward.
- **nondeterministic Assignment:** A nondeterministic statement is mapped to a number of states and transitions. The number of states depends on the number of different possible values for the variable. Line 23 of Figure 4.1 is mapped to transitions from “S2” to “S5” and to “S6” in Figure 5.4. The transitions are chosen nondeterministically.
- **Probabilistic Assignment:** In PTA, a probabilistic assignment statement is mapped to a number of states, each of them assigning a different value to the variable, and a probabilistic transition into the states. The mapping of line 42 of Figure 4.1 is shown in Figure 5.2 as the transitions from “S2” to “S4” and “S5”.
- **Delay Statement:** Delays are mapped by the use of one clock, a location and transition guards. Mapping for delay statement of line 12 of Figure 4.1 is depicted in Figure 5.3, specified as transitions from “S1” to “S3”. The required clock is extracted from a pool of clocks using the function *selectClock*.
- **Sending Message Statement:** In PTRebeca, each rebec has an internal clock, which shows the time elapsed since the creation of the rebec. This specifies an absolute model of time, which cannot be implemented in PTA, because it makes

clock values grow unboundedly. To solve the problem, for message sending, a clock is dedicated to the message. The clock of a message is used for checking its deadline and enabling time. The clock is returned to the pool, when the message is delivered to the rebec-behavior automaton for execution. For example, message sending of Line 24 of Figure 4.1 is mapped to the transitions from “S1” to “S3”, and from “S3” to “S7” in Figure 5.4.

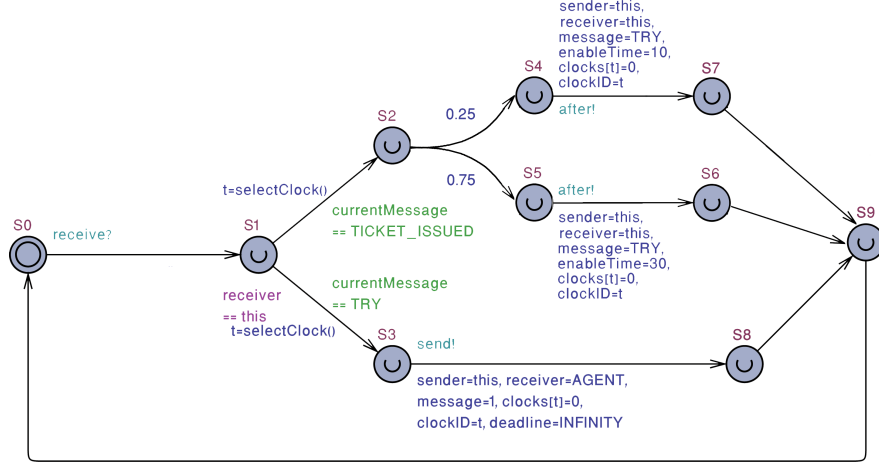


Figure 5.2: The rebec-behavior PTA of the Customer reactive class

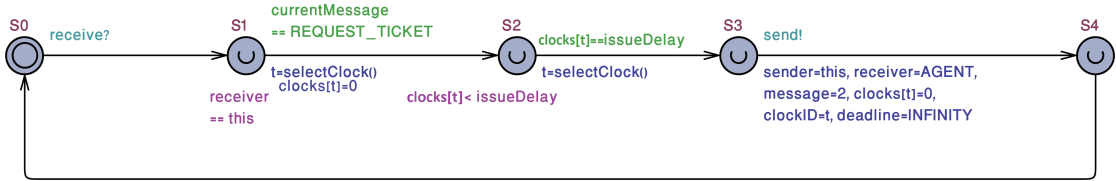


Figure 5.3: The rebec-behavior PTA of the TicketService reactive class

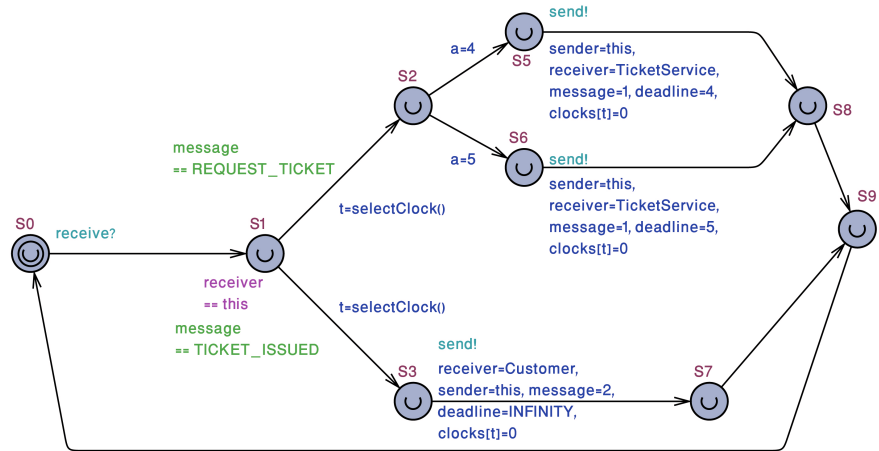


Figure 5.4: The rebec-behavior PTA of the Agent reactive class

Message sending is synchronized with either the rebec-bag automaton or the after-handler automaton. We use channel “send” if the message is sent immediately and

channel “after” if the sent message has the “after” value. Messages which are sent via the send channel are directly put in the rebec-bag of their receivers. Messages which are sent via the after channel are put in a buffer in the after-handler automaton. A message will be delivered to the rebec-bag of the receiver when the value of the clock which is dedicated to the message reaches the value “after”.

**Rebec-bag.** The rebec-bag PTA always accepts messages asynchronously, regardless of the state of the corresponding rebec-behavior, and then delivers them, upon the rebec-behavior automaton’s request. The rebec-bag is responsible to handle activation time and deadlines of messages. As depicted in Figure 5.5, the rebec-bag PTA inserts the incoming messages of the owner rebec (transition from “S1” to “S3”), discards the messages with passed deadlines (self loop transition in “S1”), and extracts the messages from its buffer and delivers them (transition from “S1” to “S2”). Extracting the message from the buffer is done by the *shift* function which is used as the update function of the transition from “S2” to “S1”.

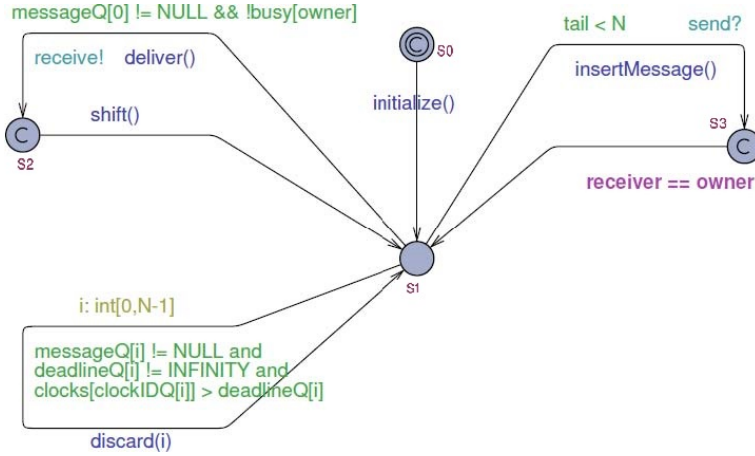


Figure 5.5: PTA of rebec-bag for a rebec

**After-handler.** The after-handler PTA always accepts messages asynchronously and puts them in a buffer until their enabling time. Figure 5.6 shows the PTA of the after-handler. As depicted in Figure 5.6, it inserts the incoming messages (transition from “S1” to “S2”) and extracts the messages from its buffer and delivers them if the clock of any of them reaches the value of its corresponding enable time, i.e. the value of its after (self loop transition of “S1”).

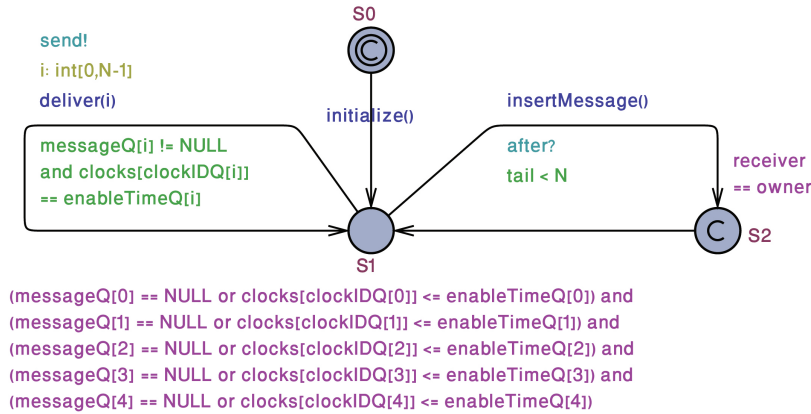


Figure 5.6: PTA of After-handler

### 5.1.3 Comparison of Parallel Composition Approach with TMDP Semantics

In this section, we discuss which of the proposed PRISM-based analysis techniques is appropriate for performance analysis of PTRebeca models. In [39] we reported the size of the state space for the timed semantics of Timed Rebeca and parallel composition approach of Timed Rebeca. In the parallel composition approach, each rebec is converted to a timed automaton and parallel composition of timed automata represents the behavior of the Timed Rebeca model. There, UPPAAL was used, a well-known model checker for timed systems, for the parallel composition of timed automata. We developed a tool to generate the state space based on the timed semantics of Timed Rebeca.

Experimental results show that the parallel composition of timed automata generates too many states in comparison to timed semantics of Timed Rebeca. The main reason of this difference lies in the modeling of asynchronous message passing between actors using synchronous communication between timed automata. This increases the number of states. This problem is also mentioned in [84] on modeling distributed systems using timed automata. Additionally, the number of clocks grows linearly by the number of rebecs. When the number of clock increases, the state space grows exponentially. We have the same results for the comparison of the parallel composition of PTAs and the TMDP semantics of a PTRebeca model. The parallel composition of PTAs generates too many states.

We explained the parallel composition approach in Section 5.1.2 without considering the details of the implementation. We chose PRISM, a well-established model checker, for modeling PTAs and verifying probabilistic properties. Since the input language of PRISM is a state-based language and lacks array, conditional and loops statements, implementing these statements in PTAs increases the number of generated states significantly. For example, implementation of rebec-queue PTA and related functions (like insert, shift, and discard shown in Figure 5.5) adds many states to the corresponding PTA. So, the proposed approach becomes more complicated and creates a large state space.

Although the parallel composition approach supports the definition of rewards, it generates more states comparing to the TMDP semantics of a PTRebeca model, and it cannot be the suitable approach for large PTRebeca models. In Section 5.2, we provide a new approach in which the TMDP model of a large case study is converted into one Markov automaton. Then, the IMCA is used as the back-end model checker to analyze the PTRebeca model against expected reachability and probabilistic reachability properties.

## 5.2 Performance Analysis of PTRebeca Models Using IMCA

As we concluded in Section 5.1.3, the parallel composition approach is not efficient for performance analysis of large PTRebeca models. To provide a practical approach, we convert the TMDP underlying a PTRebeca model to a Markov automaton (MA) [109] to be able to use the IMCA model checker for performance analysis of PTRebeca models. In this section, we mathematically prove that expectation properties are preserved by this conversion. The proofs are presented for minimum expected time

reachability and minimum expected reward reachability properties. Maximum values of expected time reachability and expected reward reachability can be proved similarly.

### 5.2.1 Preliminaries

Prior to our proof, we have to prepare the following definitions and notations for TMDP. We also define how a TMDP is converted to MA.

**Definition 13 (Timed Markov Decision Process)** *A timed Markov decision process  $\mathcal{T} = (S, s_0, Act, \hookrightarrow, L)$  consists of a set  $S$  of states, an initial state  $s_0 \in S$ , a set  $Act$  of actions, and a timed probabilistic, nondeterministic transition relation  $\hookrightarrow \subseteq S \times Act \times \mathbb{N} \times Dist(S)$  such that, for each state  $s \in S$ , there exists at least one tuple  $(s, a, d, \mu) \in \hookrightarrow$ .  $\square$*

The transitions in a TMDP are performed in two steps: given that the current state is  $s$ , the first step is a nondeterministic selection of  $(s, act, d, \mu) \in \hookrightarrow$ , where  $act$  denotes a possible action and  $d$  specifies the duration of the transition; in the second step, a probabilistic transition to state  $s'$  is made with probability  $\mu(s')$ . Function  $\mu \in Distr(S)$ ,  $Distr(S)$  denotes the set of discrete probability distribution functions over the countable set  $S$ .

We present the TMDP  $\mathcal{T}_{\mathcal{M}}$  of a given PRebeca model  $\mathcal{M}$  as a tuple  $(S, s_0, Act, \rightarrow, \Rightarrow)$  where  $S$  is a set of states,  $s_0$  is the initial state,  $Act$  is a set of actions which consists of  $\tau$ , signatures of all the messages, and  $\mathbb{N}$ . Considering Section 4.3, the union of scheduler and msg-fetcher transitions is  $\rightarrow$  (probabilistic transitions) and the set of time-progress transitions (delay transitions) is  $\Rightarrow$ .

In the TMDP of a PRebeca model, because of the maximal progress assumption, probabilistic transitions have a higher priority than delay transitions in the execution as their execution time is zero. According to the maximal progress assumption, transitions with execution time of zero, i.e. probabilistic transitions, must be executed before any time progress which is caused by the execution of delay transitions. Therefore, in states with enabled probabilistic transitions, delay transitions are disabled. Here, states with some enabled probabilistic transitions are called probabilistic states (PS) and states with delay transitions are called delay states (DS). For a given delay state  $s$  the value of its unique outgoing delay transition is shown by  $d_s$ .

**Definition 14 (Paths)** *A path in a TMDP is an infinite sequence  $\pi = s_0 \xrightarrow{\sigma_0, \mu_0, t_0} s_1 \xrightarrow{\sigma_1, \mu_1, t_1} \dots$  where  $s_i \in S$ ,  $\sigma_i \in Act \cup \{\perp\}$ , and  $t_i \in \mathbb{N}$ . In case of  $\sigma_i \in Act$  the value of  $t_i$  is zero, which means that the TMDP moves from  $s_i$  to  $s_{i+1}$  using a probabilistic transition with probability  $\mu_i = \mu_{\sigma_i}^{s_i}(s_{i+1})$ . In case of  $\sigma_i = \perp$  the value of  $t_i$  is larger than zero and the TMDP moves from  $s_i$  to  $s_{i+1}$  after residing  $t_i$  units of time with probability  $\mu_i = 1$ . For any given  $t \in \mathbb{N}_{>0}$ ,  $\pi @ t$  denotes the sequence of states that  $\pi$  occupies at time  $t$ .  $\square$*

Due to the instantaneous probabilistic transitions, a TMDP may occupy various states at the same time instance. The time elapsed along the path  $\pi$  is computed by  $\sum_{i=0}^{\infty} t_i$ . Path  $\pi$  is Zeno whenever this summation converges to a number and its corresponding TMDP has Zeno behavior. A TMDP has Zeno behavior if and only if it has a strongly connected component with only probabilistic transitions. In the rest of this chapter we assume that TMDPs do not have Zeno behavior.

**Definition 15 (Policies)** *Policies are used to resolve nondeterministic choices in states. To define a probability space, nondeterminism should be resolved. A policy is a measurable function (ranged over  $D$ ) which provides for each finite path ending in state  $s$ , a probability distribution over the set of enabled transitions in  $s$ . A stationary deterministic policy is a special type of policy which always takes the same decision in a state  $s$ .*  $\square$

**Definition 16 (Stochastic Shortest Path (SSP) Problem)** *A tuple  $(S, s_0, G, Act, \mathbf{P}, c, g)$  is an SSP problem (non-negative) such that  $(S, s_0, Act, \mathbf{P})$  is a MDP,  $G \subseteq S$  is a set of goal states,  $c : S \setminus G \times Act \rightarrow \mathbb{R}_{\geq 0}$  is a cost function for non-goal states, and  $g : G \times Act \rightarrow \mathbb{R}_{\geq 0}$  is a cost function for goal states.*  $\square$

As described in [110], the minimum expected cost reachability of one of the goal states in  $G$  from state  $s$ , shown by  $eR^{min}(s, \Diamond G)$ , can be obtained by solving a linear programming (LP) problem. To compute the minimum expected cost reachability, we reduce the analysis of a TMDP to the analysis of a non-negative SSP problem to be able to use an LP problem.

In addition to the above definitions on TMDPs, we have to formally define MAs.

**Definition 17 (A Markov Automaton)** *An MA is a transition system with two types of transitions, called probabilistic and Markovian transitions, shown by the tuple  $(S', s'_0, Act', \rightarrow', \Rightarrow')$ . Here,  $S'$  is a set of states,  $s'_0 \in S'$  is an initial state,  $Act'$  is a set of actions,  $\rightarrow'$  is a set of probabilistic transitions, and  $\Rightarrow'$  is a set of Markovian transitions. Probabilistic transitions are instantaneous transitions which are defined as  $\rightarrow' \subseteq S' \times Act' \times Distr(S')$  (where  $Distr(S')$  denotes the set of discrete probability distribution functions over the countable set  $S'$ ) and Markovian transitions are defined as  $\Rightarrow' \subseteq S' \times \mathbb{R}_{\geq 0} \times S'$  [109].*  $\square$

Here, transition  $(s', \alpha, \mu) \in \rightarrow'$  is abbreviated to  $s' \xrightarrow{\alpha} \mu$  and  $(s', \lambda, t') \in \Rightarrow'$  by  $s' \xRightarrow{\lambda} t'$ . An MA can evolve via its probabilistic and Markovian transitions. In case of  $s' \xrightarrow{\alpha} \mu$ , it leaves state  $s'$  by executing action  $\alpha$  and state  $t'$  is its destination with the probability of  $\mu(t')$ . Here,  $s'$  is called a probabilistic state (PS). In case of  $s' \xRightarrow{\lambda} t'$ , state  $s'$  is left after waiting for exponentially distributed units of time with rate  $\lambda$  and the target state is  $t'$ . It means that the expected delay from  $s'$  to  $t'$  is  $1/\lambda$ . Here, state  $s'$  is called Markovian state (MS).

In the rest of this chapter we use the primed version of alphabet and arrows to address MAs and the normal ones to address TMDPs.

**Definition 18 (Conversion of the TMDP of PTRebeca model  $\mathcal{M}$ )** *A given TMDP  $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$  is converted to MA  $\mathcal{A}_{\mathcal{M}} = (S', s'_0, Act', \rightarrow', \Rightarrow')$  such that  $S = S'$ ,  $s_0 = s'_0$ ,  $Act = Act'$ , and  $\rightarrow = \rightarrow'$ . In addition,  $(s, d, t) \in \Rightarrow$  implies that  $(s', 1/d, t') \in \Rightarrow'$ . In other words,  $\mathcal{T}_{\mathcal{M}}$  and  $\mathcal{A}_{\mathcal{M}}$  are the same except that the delay transitions in  $\mathcal{T}_{\mathcal{M}}$  are converted to Markovian transitions in  $\mathcal{A}_{\mathcal{M}}$ . In this conversion, if a given state  $s \in S$  is a delay state in  $\mathcal{T}_{\mathcal{M}}$ , its corresponding state  $s' \in S'$  is a Markovian state in  $\mathcal{A}_{\mathcal{M}}$ , and if  $s$  is a probabilistic state in  $\mathcal{T}_{\mathcal{M}}$ , its corresponding state  $s'$  is a probabilistic state in  $\mathcal{A}_{\mathcal{M}}$ .*  $\square$

### 5.2.2 Expected Time Reachability in TMDP

Assume that for a given PRebeca model  $\mathcal{M}$ , its TMDP  $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$  and the set of goal states  $G \subseteq S$  are defined. Here, we want to find the minimum expected time for reaching one of the states in  $G$ . We define random variable  $V_G : Paths \rightarrow \mathbb{N}$  as the elapsed time in paths from the start state to one of the goal states of  $G$ . So, the minimum expected time reachability for a given state  $s \in S$  to one of the goal states is defined by

$$eT^{min}(s, \Diamond G) = \inf_D \mathbb{E}_{s,D}(V_G) = \inf_D \sum_{\pi \in Paths} V_G(\pi) \cdot Pr_{s,D}(\pi)$$

where  $D$  is a generic policy on  $\mathcal{M}$ . To compute the value of  $eT^{min}(s, \Diamond G)$  we have to reformulate the above equation into a linear equation system, as shown in the following theorem. Note that the proofs of theorems are given in Appendix B.

**Theorem 2** *The function  $eT^{min}$  is a fix point of the Bellman operator*

$$[L(v)](s) = \begin{cases} d_s + v(t) & s \in DS \setminus G \\ \min_{a \in Act(s)} \left\{ \sum_{t \in S} \mu_a^s(t) \cdot v(t) \right\} & s \in PS \setminus G \\ 0 & s \in G \end{cases} \quad (5.1)$$

where  $Act(s) = \{\alpha | s \xrightarrow{\alpha} \mu\}$  and  $\mu_a^s \in Distr(S)$  such that  $s \xrightarrow{\alpha} \mu_a^s$ .  $\square$

The above result is explained as follows. For a goal state, the expected time is zero. For a delay state  $s \notin G$ , the minimal expected time to  $G$  is the summation of the sojourn time in  $s$  (which is  $d_s$ ) and the expected time to reach  $G$  from one of its successor states  $t \in S$ . For a probabilistic state  $s \notin G$ , an action is selected that minimizes the expected time according to the distribution  $\mu_a^s$ .

The characterization of  $eT^{min}(s, \Diamond G)$  in Theorem 2 allows us to reduce computing the minimum expected time reachability problem in a TMDP to the minimum expected time reachability in a non-negative SSP problem, denoted by  $ssp_{et}$ .

**Definition 19 (SSP for minimum expected time reachability)** *The SSP of a given TMDP  $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$  for the expected time reachability to a set of goal states  $G \subseteq S$  is a tuple  $ssp_{et}(\mathcal{M}) = (S, s_0, Act \cup \{\perp\}, G, c, g)$  where:*

- $S, s_0$ , and  $Act$  in TMDP and  $ssp_{et}$  are the same,

$$\bullet \mathbf{P}(s, \alpha, t) = \begin{cases} 1 & s \in DS \setminus G \\ \mu_{\alpha}^s(t) & s \in PS \setminus G, \\ 0 & s \in G \end{cases}$$

$$\bullet c(s, \alpha) = \begin{cases} d_s & s \in DS \setminus G \wedge \alpha = \perp, \\ 0 & otherwise \end{cases},$$

- $g(s) = 0$ .

$\square$

As shown in [111], the minimum expected cost problem of an SSP has a unique fixed point; which enables us using standard solution techniques like value iteration and linear programming to compute the minimum expected cost of the SSP.

**Theorem 3** *For a TMDP  $\mathcal{T}_M$  the value of  $eT^{min}(s, \Diamond G)$  equals  $cR^{min}(s, \Diamond G)$  in  $ssp_{et}(\mathcal{M})$ .*  $\square$

This way, we showed how the minimum expected time reachability for a TMDP is computed. As we want to use the IMCA for computing the expected time reachability of the TMDP, we present the conversion of a TMDP to its corresponding MA (which can be analyzed by the IMCA). Then, we prove that expected time reachability in the TMDP and its conversion in the form of MA are equal.

As shown in [109], for a given MA  $\mathcal{A}_M = (S', s'_0, Act', \rightarrow', \Rightarrow')$  the following Bellman operator is used for finding the expected time reachability.

$$[L(v)](s') = \begin{cases} \frac{1}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t') & s' \in MS \setminus G' \\ \min_{a \in Act(s')} \left\{ \sum_{t' \in S'} \mu_a(t') \cdot v(t') \right\} & s' \in PS \setminus G' \\ 0 & s' \in G' \end{cases} \quad (5.2)$$

The TMDP conversion to its corresponding MA preserves the expected time reachability properties. As depicted in Equations 5.1 and 5.2, for a given state  $s \in S$  in  $\mathcal{T}_M$  where  $s$  is a probabilistic (or goal) state, its corresponding state  $s' \in S'$  in  $\mathcal{A}_M$  is a probabilistic (or goal) state, and the equations for finding  $[L(v)](s)$  are the same as for  $[L(v)](s')$ . In the case that  $s$  is a delay state, based on the semantics of PTRebeca, delay states have only one outgoing delay transition. So, in its corresponding state  $s'$  in  $\mathcal{A}_M$  there is only one outgoing transition with probability one, which results in changing the formula of computing the expected time reachability from  $\frac{1}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t')$  to  $\frac{1}{E(s')} + v(t')$ . As during conversion from  $\mathcal{T}_M$  to  $\mathcal{A}_M$  a delay value  $d_s$  is changed to  $1/d_s$ , we have  $\frac{1}{E(s')} + v(t') = d_s + v(t)$ . Here, we assumed

that there are states  $t \in S$  and  $t' \in S'$  such that  $s \xrightarrow{d_s} t$  and  $s' \xrightarrow{1/d_s} t'$ . In a nutshell, the minimum expected time reachability in all three cases of Equation 5.1 for state  $s$  is the same as the minimum expected time reachability in all three cases of Equation 5.2 for state  $s'$ .

### 5.2.3 Expected Reward Reachability in TMDP

We want to compute expected reward reachability in TMPDs where the rewards are associated to delay states and probabilistic transitions. This is similar to what we did for computing expected time reachability in TMDPs. Assume that there are two functions  $\rho$  and  $r$  for accessing to the associated rewards to states and transitions respectively. For a given state  $s$ , function  $\rho(s)$  returns the reward which is associated to  $s$ . For a given transition from  $s$  to  $t$  with action  $\alpha$ , function  $r(s, \alpha)$  returns the reward which is associated to the transition.

Now, assume that TMDP  $\mathcal{T}_M = (S, s_0, Act, \rightarrow, \Rightarrow)$  is given and the set of goal states is defined as  $G \subseteq S$ . Here, we want to find the minimum expected reward which is gained from each state  $s \in S$  to one of the states in  $G$ . So, we need to define a random

variable on the total reward which is gained in paths from  $s$  to one of the goal states of  $G$ . Assume that random variable  $R_G : Paths \rightarrow \mathbb{N}$  is this random variable. So, the minimum expected reward reachability from  $s$  to one of the goal states is defined by

$$eR^{min}(s, \Diamond G) = \inf_D \mathbb{E}_{s,D}(R_G) = \inf_D \sum_{\pi \in Paths} R_G(\pi) \cdot Pr_{s,D}(\pi)$$

where  $D$  is a generic policy on  $\mathcal{M}$ . To compute the value of  $eR^{min}(s, \Diamond G)$ , we have to reformulate the above equation into a linear equation system, as shown in Theorem 4.

**Theorem 4** *The function  $eR^{min}$  is a fix point of the Bellman operator*

$$[L(v)](s) = \begin{cases} d_s \times \rho(s) + v(t) & s \in DS \setminus G \\ \min_{a \in Act(s)} \left\{ r(s, a) + \sum_{t \in S} \mu_a^s(t) \cdot v(t) \right\} & s \in PS \setminus G \\ 0 & s \in G \end{cases} \quad (5.3)$$

Let us explain the above result. For a goal state, the expected reward is zero. For a delay state  $s \notin G$ , the minimal expected reward to  $G$  is the multiplication of the sojourn time in state  $s$  and its associated reward plus the expected reward to  $G$  from one of its successor states  $t \in S$ . For a probabilistic state  $s \notin G$ , an action is chosen which minimizes the expected reward to  $G$  according to  $\mu_a^s$  plus the reward associated to the transition.

The characterization of  $eR^{min}(s, \Diamond G)$  in Theorem 4 allows us to reduce computing the minimum expected reward reachability problem in TMDPs to the minimum expected cost in non-negative SSP problems, shown by  $ssp_{er}$ .

**Definition 20 (SSP for minimum expected reward reachability)** *The SSP of a given TMDP  $\mathcal{T}_{\mathcal{M}} = (S, s_0, Act, \rightarrow, \Rightarrow)$  for the expected reward reachability to a set of goal states  $G \subseteq S$  is a tuple  $ssp_{er}(\mathcal{M}) = (S, s_0, Act \cup \{\perp\}, G, c, g)$  where:*

- $S, s_0$ , and  $Act$  in TMDP and  $ssp_{er}$  are the same,

- $\mathbf{P}(s, \alpha, t) = \begin{cases} 1 & s \in DS \setminus G \\ \mu_\alpha^s(t) & s \in PS \setminus G, \\ 0 & s \in G \end{cases}$

- $c(s, \alpha) = \begin{cases} r(s, \alpha) & s \in MS \setminus G \\ 0 & otherwise \end{cases},$

- $g(s) = \rho(s)$ .

□

As the problem is reduced to the minimum expected cost problem of an SSP, we conclude that there is only one fixed point in TMDPs as discussed before.

**Theorem 5** *For a TMDP  $\mathcal{T}_{\mathcal{M}}$  the value of  $eR^{min}(s, \Diamond G)$  equals  $cR^{min}(s, \Diamond G)$  in  $ssp_{er}(\mathcal{M})$ .*

□

This way, we showed how the minimum expected reward reachability for a TMDP is computed. As we want to use the IMCA for computing the expected reward reachability of TMDPs, we present the conversion of a TMDP to its corresponding MA. Then, we prove that expected reward reachability in a TMDP and its conversion in the form of an MA are equal.

A given TMDP  $\mathcal{T}_M = (S, s_0, Act, \rightarrow, \Rightarrow)$  with reward functions  $\rho$  and  $r$  is converted to MA  $\mathcal{A}_M = (S', s'_0, Act', \rightarrow', \Rightarrow')$  with reward functions  $\rho'$  and  $r'$  such that  $S = S'$ ,  $s_0 = s'_0$ ,  $Act = Act'$ ,  $\rightarrow = \rightarrow'$ ,  $r(s, a) = r'(s', a)$ , and  $\rho(s) = \rho'(s')$ . The properties of this conversion is the same as the properties of conversion which is described in Section 5.2.2.

As shown in [112], for a given MA  $\mathcal{A}_M = (S', s'_0, Act', \rightarrow', \Rightarrow')$  with reward functions  $\rho'$  and  $r'$  the following Bellman operator is used for finding the expected reward reachability.

$$[L(v)](s') = \begin{cases} \frac{\rho'(s')}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t') & s' \in MS \setminus G' \\ \min_{a \in Act(s')} \left\{ r'(s', a) + \sum_{t' \in S'} \mu_a(t') \cdot v(t') \right\} & s' \in PS \setminus G' \\ 0 & s' \in G' \end{cases} \quad (5.4)$$

As depicted in Equations 5.3 and 5.4, for a given state  $s \in S$  in  $\mathcal{T}_M$  where  $s$  is a probabilistic (or goal) state, its corresponding state  $s' \in S'$  in  $\mathcal{A}_M$  is a probabilistic (or goal) state, and the equations for finding  $[L(v)](s)$  are the same as for  $[L(v)](s')$ . In the case that  $s$  is a delay state, based on the semantics of PTRebeca, delay states have only one outgoing delay transition. So, in its corresponding state  $s'$  in  $\mathcal{A}_M$  there is only one outgoing transition with probability one, which results in changing the formula of computing the expected reward reachability from  $\frac{\rho'(s')}{E(s')} + \sum_{t' \in S'} P(s', t') \cdot v(t')$  to  $\frac{\rho'(s')}{E(s')} + v(t')$ . As during conversion from  $\mathcal{T}_M$  to  $\mathcal{A}_M$  a delay value  $d_s$  is changed to  $1/d_s$ , we have  $\frac{\rho'(s')}{E(s')} + v(t') = d_s \times \rho(s) + v(t)$ . Here, we assumed that there are states  $t \in S$  and  $t' \in S'$  such that  $s \xRightarrow{d_s} t$  and  $s' \xRightarrow{1/d_s} t'$ . In a nutshell, the minimum expected reward reachability in all three cases of Equation 5.3 for state  $s$  is the same as the minimum expected reward reachability in all three cases of Equation 5.4 for state  $s'$ .

### 5.2.4 The Toolset and Case Studies

To illustrate the applicability of the IMCA-based approach, we developed a toolset and analyzed two different case studies, which are accessible from the Rebeca home page [85]. The architectural overview of the toolset is depicted in Figure 5.7. As shown in the figure, Afra IDE serves as the front-end of the toolset and IMCA [55] is the back-end model checking engine of the toolset.

Using the Afra IDE, a number of C++ files are generated for a PTRebeca model. These C++ files are compiled and linked by a g++ compiler, which results in an executable file. Running the executable file generates the TMDP of the model (i.e. the state space of the model). In PTRebeca models, the size of message bags is bounded. The state space of a PTRebeca model is finite when the model shows recurrent behavior. We used the time-shift equivalence approach, proposed in [39], to make the state space finite.

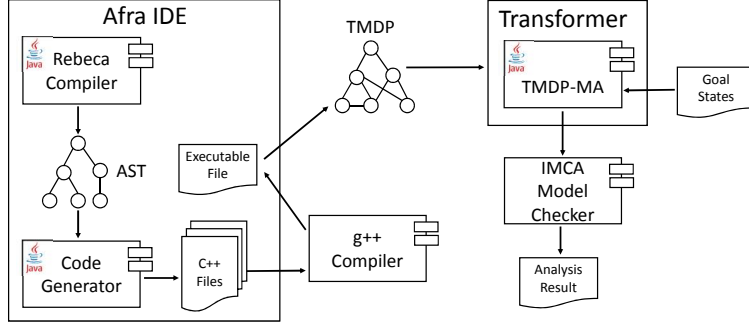


Figure 5.7: The architectural overview of the analyzer of PTRebeca models

The *TMDP-MA* tool is developed to convert the TMDP of the model to the input language of IMCA model checker. To perform the conversion, the generated TMDP and the specification of the *goal states* of the model are input to *TMDP-MA* and one Markov automaton is generated. The obtained MA is imported to the IMCA for model checking.

**Evaluation of The Toolset** IMCA provides algorithms for expected time and expected reward reachability analysis, long-run average analysis, time-bounded probabilistic reachability and probabilistic reachability analysis of MA. Since IMCA is used as the back-end model checker for PTRebeca models, we investigate which properties are preserved by the conversion (refer to Definition 18), and so can be evaluated by our developed toolset.

In Section 5.2, we proved that expected time reachability and expected reward reachability properties are preserved by the conversion. By using a dedicated time action in the TMDP (equivalently to its corresponding MA) and because of the ability of assigning rewards to the transitions in IMCA, expected reachability properties can be computed for a PTRebeca model. Therefore, our toolset can be used for the evaluation of expected reachability properties of PTRebeca models.

According to Definition 18, probabilistic transitions in the TMDP are directly converted to probabilistic transitions in the MA. Obviously, probabilistic reachability properties are preserved in this conversion, and so can be checked for PTRebeca models. The rate of a Markovian transition in the MA is estimated by the inverse of the integer value of a corresponding delay transition in the TMDP. Because of this estimation, time-bounded probabilistic reachability properties are not preserved by the conversion. So, we are not able to evaluate this type of properties for PTRebeca models. We believe that long-run average properties are preserved by the conversion, but its mathematical proof remains as a future work.

In the following sections, we choose two case studies to cover the evaluation of two types of properties: expected reachability and probabilistic reachability. The probabilistic reachability property is checked for toxic gas sensing system, and the expected time reachability is calculated for a network on chip case study. These case studies show the applicability of the toolset for the performance evaluation of systems.

#### 5.2.4.1 Performance Analysis of a Toxic Gas Sensing System

In Section 5.1.1.3, we examined toxic gas sensing case study and used PRISM as the back-end model checker. Here, we perform the same experiments, but we use IMCA

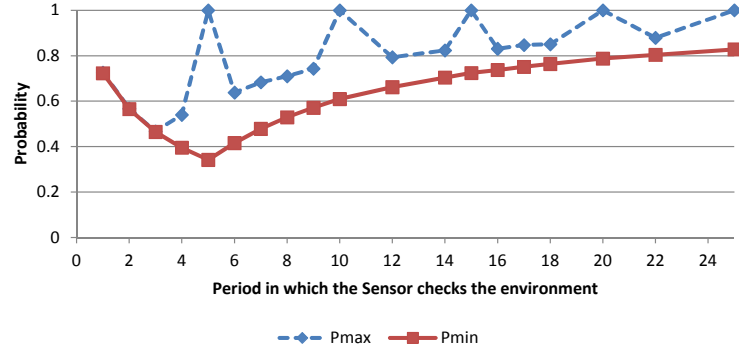
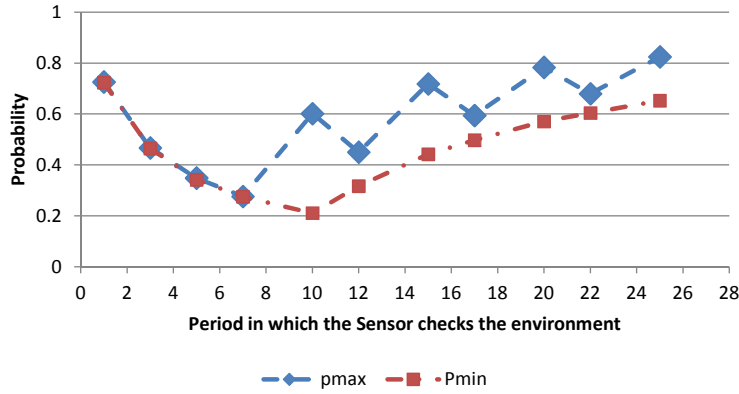
(a) The value of variable *scientistDeadline* is 10.(b) The value of variable *scientistDeadline* is 12.

Figure 5.8: The maximum and minimum probabilities that the scientist eventually dies, when the sensor frequency changes.

for the performance evaluation of the model. As we discussed earlier, the ultimate goal is to find a time setting, specifically a value for the variable *checkingPeriod*, in which the min/max probabilities of the scientist's death have the least values. Figure 5.8 shows the maximum and the minimum probabilities of the scientist's death when IMCA is used as the back-end model checker. Comparing to the results in Figure 5.1, we obtain the same results via PRISM and IMCA.

#### 5.2.4.2 Performance Analysis of Network on Chip

Our second example is a model of a network on chip (NoC). NoC has emerged as a promising architecture paradigm for many-core systems. As complexity grows in NoCs, functional verification and performance evaluation in the early stages of the design process are suggested as ways to reduce the fabrication cost. Globally Asynchronous Locally Synchronous (GALS) NoC [113] has gained much attention in designing such systems. As an example of a NoC, we model and analyze ASPIN (Asynchronous Scalable Packet switching Integrated Network), which is a fully asynchronous two-dimensional GALS NoC design using XY routing algorithm. Using this algorithm, packets can only move along the X direction first, and then along the Y direction to reach their destination. In ASPIN, packets are transferred through channels, using a four-phase handshake communication protocol. The protocol uses two signals, namely *Req* and *Ack*, to implement the four-phasing protocol. To transfer a packet, first, the sender sends a request by rising the *Req* signal, and waits for an acknowl-

edgment which is raising *Ack* from the receiver. All the signals return to zero after a successful communication. There are four adjacent routers to each router and also four internal buffer for storing the incoming packets of different neighbors.

The timed version of ASPIN was investigated in [41] using simulation and model checking. The Timed Rebeca language was used for modeling of ASPIN, and the Afra tool-set [85] was applied to the model for estimating the maximum end-to-end latency through model checking. Here, we add faulty routers to ASPIN, and examine the model for different traffic patterns and faulty routers. In the PTRebeca model, all nodes are working correctly with probability one except a few of them which are specified in their constructors. For example, the node with *Xid*=1 and *Yid*=0 is supposed to be faulty (Line 15 of Figure 5.5). The fault probability is determined in the message server *coreIsFaulty* (Line 18) for all faulty nodes. A faulty node fails to send received packets with the specified probability  $p$  and sends packets with probability  $1 - p$ . In other words, a faulty node works most of the time, and the node is not broken. The probabilistic version of the case study is similar to the timed version presented in [41]. The way we model channels, the topology of the communication, routing algorithm, buffer status, and communication protocol in the model is the same as in [41].

**PTRebeca Model.** The simplified version of the PTRebeca model of ASPIN is shown in Figure 5.5, which contains two different reactive classes: **Manager** and **Router**. The **Manager** does not exist in real NoCs. Here, it is used as the starter of the model. It sends an *init* message to routers to ask them to generate packets. This way, different traffic patterns are created by modifying only **Manager**. The **Router** is the model of a router in an ASPIN. So, its definition contains four known rebecs which are its neighbor routers (line 7), its id in XY manner (*Xid* and *Yid* in line 8), its buffer variables which show that the buffer is enabled or busy (line 10), a variable which shows whether it works properly or not (line 11), and variables that show whether its neighbors are faulty or not (line 12). The communication channel functionalities among neighbors are modeled by message passing in Rebeca. The four-phase handshake protocol is modeled using three message servers: *reqSend*, *giveAck*, and *getAck*. A router calls its *reqSend* message server to send a request to its neighbor. The XY-routing algorithm is implemented inside *reqSend* (lines 25-56) and determines to which neighbor router the packet is sent. If the neighbor router is faulty, a dynamic XY-routing algorithm presented in [114] is used to reroute the packet. The congestion links are not considered in our algorithm. The packet is rerouted to an operative neighbor by calling function *reRoute* (e.g. line 31). In lines 40-46 of Figure 5.5, the details of routing a packet with  $X_{target} > X_{id}$  and  $Y_{target} > Y_{id}$  are shown. If the packet must be sent to the router's east neighbor and the east neighbor is not faulty (line 41), the function *routeToEast* is called (line 42). In this function message *giveAck* is sent to the east neighbor and the internal state of the sender router is changed. The *giveAck* message server first checks the address of the destination of the newly received packet. If the address is the same as the current router, then the packet is consumed (line 85). Otherwise, if the router's buffer is not full (line 75), the packet will be stored and an acknowledgment is sent to the sender router by calling its *getAck* message server (line 80). If the incoming buffer of the neighbor is full (line 73), the router must wait for some amount of time and try sending later, which is modeled by sending to itself (line 74).

```

1  env byte bufSize = 2;
2  reactiveclass Manager(10){
3    knownrebecs {Router r00, r10, ... , r33;}
4    msgsrv reset(){ r00.init(); r23.init();}
5  }
6  reactiveclass Router(10) {
7    knownrebecs {Router N, E, S, W;}
8    statevars { byte Xid, Yid;
9      byte[4] bufNum;
10     boolean[4] full, enable, outMutex;
11     boolean recieved, isWorking;
12     boolean [4] neighborIsWorking; // 0=N,
        1=E, 2=S, 3=W
13  }
14  Router(byte X, byte Y){ Xid = X; Yid = Y;
        recieved = false; //specifying
        faulty nodes
15    if(Xid == 1 && Yid == 0)
        self.coreIsFaulty();
16    ...
17  }
18  msgsrv coreIsFaulty (){ isWorking = ?
        (0.95:true,0.05:false);}
19  msgsrv init(){ ... }
20  void routeToSouth() {...}
21  void routeToNorth() {...}
22  void routeToWest() {...}
23  void routeToEast() {...}
24  void reRoute() {...}
25  msgsrv reqSend(byte Xtarget, byte
        Ytarget, int dirS, int packId,
        boolean routing) {
26    if (enable[directionS]){
27      boolean sent = false; int
        hardwareDelay; hardwareDelay = 26;
28      if (Xid == Xtarget){
29        if(Ytarget > Yid){
30          if(neighborIsWorking[2] && senderR
            != 2){routeToSouth();}
31          else{reRoute();}
32        }
33        else if (Ytarget < Yid ){...}
34      }
35      else if (Yid == Ytarget){
36        if(Xtarget > Xid){...}
37        else if (Xtarget < Xid){...}
38      }
39      else{ //first move through horizontal
        channels
40        if(Xtarget > Xid && Ytarget > Yid){
41          if(neighborIsWorking[1] && senderR
            != 1){
42            routeToEast();}
43          else if(neighborIsWorking[2] &&
            senderR != 2){
44            routeToSouth();}
45          else{
46            reRoute();}
47        } else if(Xtarget < Xid && Ytarget >
            Yid){...}
48        else if(Ytarget < Yid && Xtarget >
            Xid){...}
49      }
50      else if(Ytarget < Yid && Xtarget <
            Xid){...}
51    }
52  }
53  else
54    self.reqSend(Xtarget,Ytarget,directionS,
        packId,senderR) after(1);
55  }
56  msgsrv isFaulty(int senderCore,byte
        Xtarget, byte Ytarget, int
        directionS, int packId){...}
57  msgsrv getAck(int dirS){ ... }
58  msgsrv giveAck(byte Xtarget, byte
        Ytarget,int dirS, int dirD, int
        msgSender, int packId, boolean
        routing) {
59    int MSGSender;
60    if(sender == N) MSGSender = 0;
61    else if (sender == E) MSGSender = 1;
62    ...
63    else MSGSender = msgSender;
64    if(!isWorking){
65      if(MSGSender == 0)
66        N.isFaulty(2,Xtarget,
        Ytarget,directionS, packId);
67      else if(MSGSender == 1)
68        E.isFaulty(3,Xtarget,
        Ytarget,directionS, packId);
69      ...
70    }
71  }
72  if(!(Xtarget == Xid && Ytarget == Yid))
    {
73    if (full[dirD])
74      self.giveAck(Xtarget,
        Ytarget,dirS,dirD, MSGSender,
        packId, routing) after(2)
        deadline(5);
75    else {
76      bufNum[dirD] = (byte)bufNum[dirD] +
        1;
77      if (bufNum[dirD] == bufSize)
78        full[dirD] = true;
79      self.reqSend(Xtarget, Ytarget, dirD,
        packId,routing) after(1);
80      if(MSGSender == 0) N.getAck(dirS);
81      else if(MSGSender == 1)
        E.getAck(dirS);
82      ...
83    }
84  }
85  else if((Xtarget == Xid && Ytarget ==
        Yid)) { ... if (packId == 1)
        recieved = true; ...}
86  }
87  }
88  main {
89    Manager m(r00,r10, ... ,r33):();
90    Router r00(m,r03,r10,r01,r30):(0,0);
91    Router r10(m,r13,r20,r11,r00):(1,0);
92    ...
93    Router r33(m,r32,r03,r30,r23):(3,3);
94  }

```

Listing 5.5: The model of the ASPIN network.

To model the behavior of router buffers, we use the rebec's queue to store all packets received by a router and only keep track of the length of north, south, east and west buffers to have the buffer status at all time. The variable `bufSize` specifies the buffer

size in each direction of the routers. In the experiments, `bufSize` equals two (Line 1). Each router has an array `bufNum` which keeps the number of sent packets in each direction for which their `ack` signals haven't been received yet. When a message is sent to a direction, the number of sent messages to that direction is increase by one (Line 76). When an `ack` signal is received from a direction, the number of sent messages to that direction is decreased by one. The complete PRebeca model of ASPIN is accessible from the Rebeca home page [85].

**Experimental Results** The dynamic XY-routing algorithm implemented in each node reroutes a packet if the node's neighbor is faulty. All faulty nodes do not work with the specified probability  $p$ . So, two scenarios happen when a packet face a faulty node: 1) the packet is rerouted to a different path to round the faulty node with probability  $p$ , and 2) the faulty node can route the packet correctly with probability  $1 - p$ . The main goal of our experiments is to understand the relation between the value of the fault probability and the expected latency for packet (1). This relation is important if we notice that rerouting may increase the expected latency.

We performed three experiments in a  $4 \times 4$  ASPIN model. In each experiment we consider different scenarios, each of which include different faulty nodes. The traffic pattern of all scenarios of each experiment is identical. In each experiment, the minimum and maximum expected latencies of packet (1) are reported. The expected latency shows the needed time for delivery of packet (1) to its destination. The minimum and maximum probabilities of packet (1) reaching its destination are equal to one for all scenarios. In other words, there is no scenario in which packet (1) does not reach its destination. In the following we explain each experiment and the obtained results in more detail.

As shown in Figure 5.9, in the first experiment, packet (1) is sent from R00 to R23 and there is no other packet in the network. In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty, and in scenario 3, routers R10, R11 and R12 are faulty. The results are presented in Figure 5.10, where the minimum expected latency is the same as the maximum expected latency for all scenarios. Scenario 3 has the highest expected latency since there are more faulty routers and the packet is rerouted more times in comparison to other scenarios. Also, scenario 1 has the least expected latency as there are fewer faulty nodes in the network.

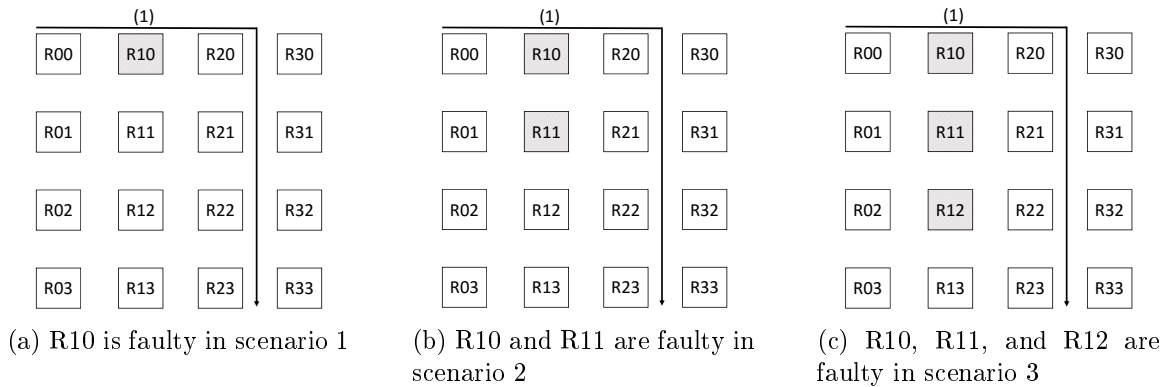


Figure 5.9: The  $4 \times 4$  ASPIN model: The traffic in experiment 1.

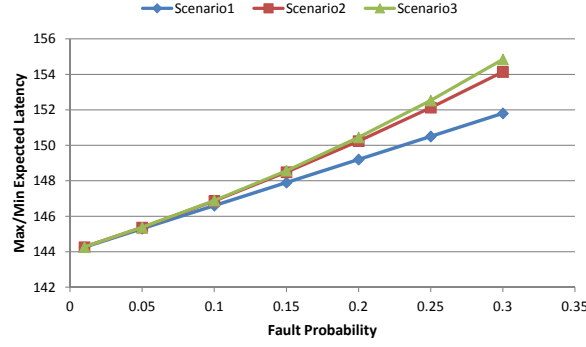


Figure 5.10: Experiment 1: the min/max expected latency for different scenarios.

In the second experiment as shown in Figure 5.11, router R10 generates packet (2) as soon as it receives packet (1), thus packet (2) may cause disruption to packet (1). On the other hand, R02 produces packet (3) in a way that it reaches R22 at the same time as packet (1), so packet (1) may be delayed by packet (3) too. In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty, and in scenario 3 routers R10, R11 and R12 are faulty. The results are presented in Figure 5.12. Scenario 3 has the highest and scenario 1 has the least expected latency. The reason is the same as the one explained for the first experiment.

In the third experiment, packet (1) is disrupted by packet (2), and packet (2) is itself disrupted because of congestion in R21. On the other hand, congestion in R23 leads packet (5) to be blocked until packet (4) leaves the input port of R22. This may result in disruption of packet (1) by packet (5), if they reach R32 at the same time (Figure 5.13). In scenario 1, router R10 is faulty. In scenario 2, routers R10 and R11 are faulty. The results are presented in Figure 5.14. In contrast to experiments 1 and 2, increasing the probability of fault decreases the maximum expected latency. The reason is that rerouting packet (1) from the normal path (i.e.  $R00 \rightarrow R10 \cdots \rightarrow R30 \rightarrow R31 \cdots \rightarrow R33$ ) to an alternative path (i.e.  $R00 \rightarrow R01 \rightarrow R02 \rightarrow R12 \cdots \rightarrow R32 \rightarrow R33$  in case of in scenario 2), avoids the congestion caused by packets (3), (5), and (7). So, the total latency decreases.

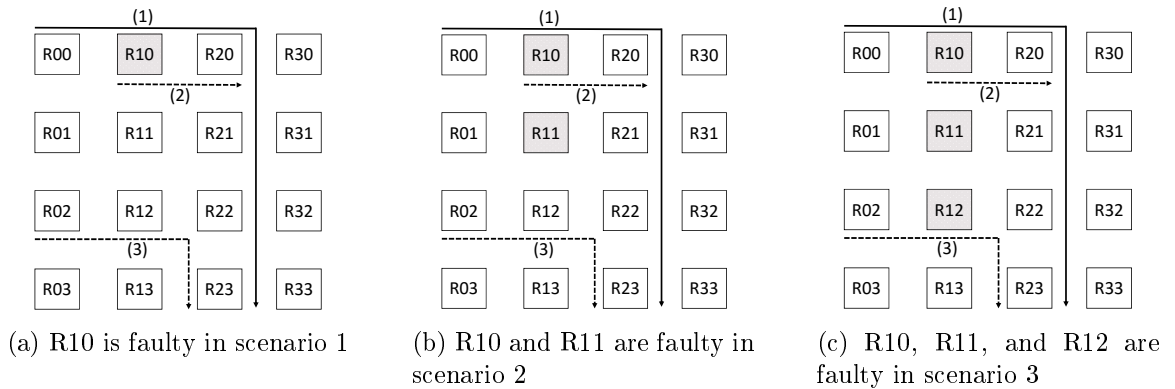
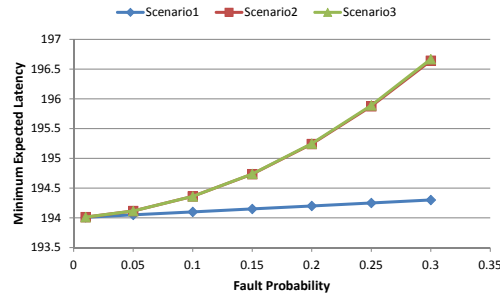
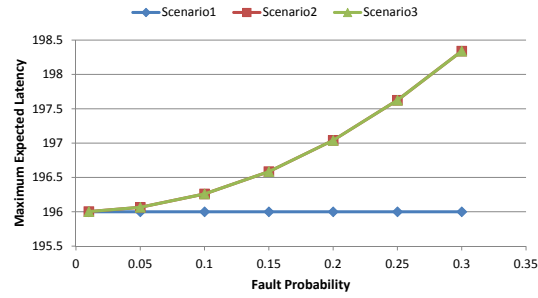


Figure 5.11: The  $4 \times 4$  ASPIN model: The traffic in experiment 2.

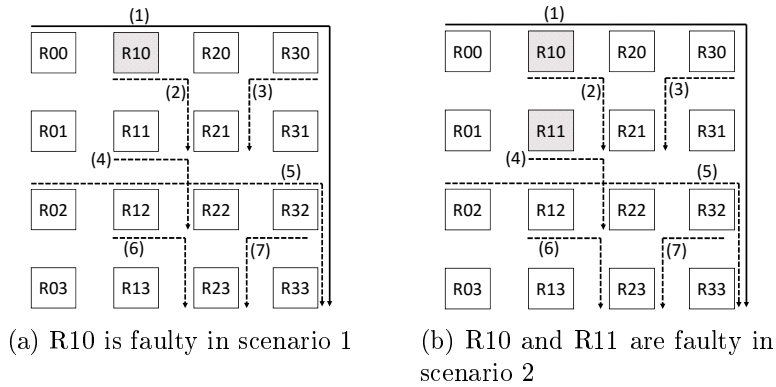
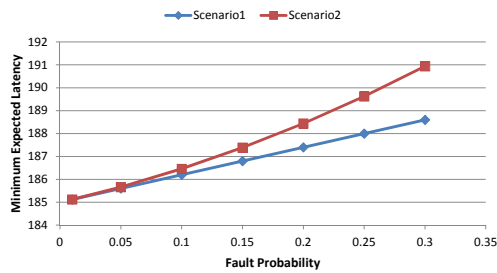


(a) The minimum expected latency

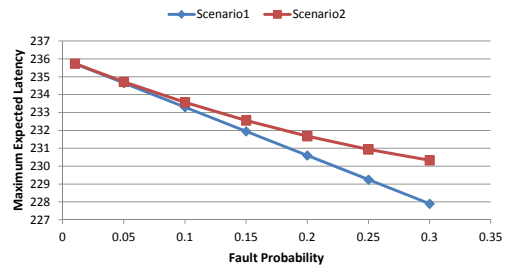


(b) The maximum expected latency

Figure 5.12: Experiment 2: the min/max expected latency for different scenarios.

Figure 5.13: The  $4 \times 4$  ASPIN model: The traffic in experiment 3.

(a) The minimum expected latency



(b) The maximum expected latency

Figure 5.14: Experiment 3: the min/max expected latency for different scenarios.

### 5.3 Comparing the PRISM-based and IMCA-based Approaches

In this section, we investigate the time and memory necessary to run experiments using the two approaches based on PRISM and IMCA. Table 5.1 presents the results for different case studies with different sizes. The experiments of the PRISM-based approach are run on a laptop with Windows 7, 4 GB RAM, and Intel Core i5-2430M CPU 2.4 GHz. To run the experiments of the IMCA-based approach, Ubuntu 12.04.5 LTS is installed on the same laptop, but RAM is restricted to 1 GB. In the PRISM-based approach, the TMDP of a PRebeca model is input to PRISM as a single MDP module. The MDP module is defined using the standard PRISM input language. The IMCA-based approach was explained in the previous section.

Problem		#states	#trans	Using PRISM		Using IMCA	
				time (sec)	memory	time (sec)	memory
Toxic Gas Sensing System	<b>1 sensor</b>	506	1170	222.787 + 0.01	NA	0.032514	~ 87.383 KB
NoC	<b>Exp 1-Scenario 1</b>	84	109	23.52	NA	0.000711	~ 12.945 KB
	<b>Exp 1-Scenario 2</b>	484	909	167.392 + 0.02	NA	0.017655	~ 79.07 KB
	<b>Exp 1-Scenario 3</b>	507	666	307.439 + 0.03	NA	0.02387	~ 77.172 KB
	<b>Exp 2-Scenario 1</b>	342	379	161.835 + 0.03	NA	0.033238	~ 51.187 KB
	<b>Exp 2-Scenario 2</b>	2184	3045	1031.11 + 0.17 (~17 min)	NA	0.509681	~ 337.226 KB
	<b>Exp 2-Scenario 3</b>	5220	9922	2955.611 + 0.94 (~49 min)	NA	1.768932	~ 857.344 KB
	<b>Exp 3-Scenario 1</b>	10032	15915	3228.434 + 0.94 (~54 min)	NA	10.631929	~ 1.561 MB
	<b>Exp 3-Scenario 2</b>	43290	71106	<b>crashed</b>	NA	136.916137	~ 4.842 MB

Table 5.1: The time and memory needed to evaluate different case studies with the PRISM-based and IMCA-based approaches. NA means not available.

As Table 5.1 shows, two numbers are reported for time when using the PRISM-based approach. The first one is the time needed via PRISM to construct the model, and the second one is the computation time to model check the model. The time for model construction is considerable, and this makes the approach inefficient even for small case studies like scenario 1 of experiment 2 in NoC case study (refer to **Exp 2-Scenario 1** in Table 5.1). In the IMCA-based approach, the Markov automaton of a PRebeca model is input using a state-based language. So, the time for model construction is negligible, and the approach is efficient for large PRebeca models like scenario 2 of experiment 3 in NoC case study (refer to **Exp 3-Scenario 2** in Table 5.1). PRISM crashed when trying to construct the model for **Exp 3-Scenario 2**. The needed memory for model checking is not reported by PRISM, so it's not available to be compared with the IMCA-based approach.

In this research, we started with PRISM as the back-end model checker. After applying the toolset to different case studies, we found that we should turn to another model checker because of what we explained in this section. Model checking of PRebeca models using PRISM is efficient only if we need to check probabilistic reachability properties. In this case, we can use the explicit engine of PRISM, and model construction takes little time because the obtained MDP module is input to PRISM in the form of matrices. Table 5.1 confirms what we explained in Section 5.1 about the efficiency of different PRISM-based approaches, especially about the standard input language of PRISM. This approach is not suitable for large PRebeca models as the model construction takes a lot of time.

We tried to define a PTRebeca model in terms of PTA modules in PRISM. The idea is to construct the probabilistic model corresponding to the PTRebeca model as the parallel composition of PTA modules. As we explained in Section 5.1.3, this approach is not applicable to PTRebeca models.

## 5.4 Related Work

**IMCA** IMCA is a powerful model checker for analyzing interactive Markov chains (IMCs) and Markov automata (MA). IMCA has a state-based input language and lacks high-level programming constructs. Expected time and long-run average objectives, time-bounded probabilistic reachability and probabilistic reachability properties are supported for MA and IMC models [55], [115].

In contrast to IMCA, PTRebeca provides high-level programming constructs and primary data structures, which makes modeling easier. In modeling we use the capabilities of PTRebeca, and in analysis we use IMCA for the evaluation of probabilistic timed properties. The semantics of PTRebeca is defined in TMDP. To be able to use IMCA, the TMDP of a PTRebeca model is converted to an MA. This conversion preserves all the above mentioned objectives except time-bounded reachability.

**UPPAAL SMC** In [96], authors introduce UPPAAL SMC in which systems are represented via networks of automata. In UPPAAL SMC, each component of the system is modeled with an automaton whose clocks can evolve with various rates. To provide efficient analysis of probabilistic properties, statistical model checking is used as a technique for fully stochastic models. The work supports modeling and performance analysis of systems with continuous time behaviors and dynamical features.

PTRebeca has a Java-like syntax which makes the language easy to use for practitioners. In PTRebeca time is discrete and discrete probability distributions are used to model probabilistic behaviors. In this work, we use the stochastic model checking algorithms for performance evaluation of systems via the IMCA model checker.

**PRISM** PRISM is a well-established and powerful model checker with a state-based input language. An input model of PRISM is composed of a number of modules which can share variables and interact with each other. PRISM is well equipped with theories and reduction techniques [54], but lacks high-level programming constructs like loops, and primary data structures like arrays, which makes modeling hard.

In contrast, PTRebeca provides high-level object-based programming features and asynchronous message passing, which makes modeling easier. In modeling we benefit from capabilities of PTRebeca, and in analysis we use the capabilities of the PRISM and the IMCA model checkers. As we showed earlier, using IMCA, we are able to model check larger PTRebeca models comparing to PRISM as the back-end model checker.

**Modest** Modest [108] is a high-level and convenient language for describing stochastic timed and hybrid systems. It supports loop constructs, structs and arrays, exception handling, and other advanced programming constructs. It also supports various model checking approaches. For the probabilistic timed fragment of Modest, model checking can be performed using a digital time semantics [116] or by a direct mapping to PTA. Both approaches use PRISM as the back-end model checker.

In contrast to Modest, PRebeca supports object-based programming features, and follows the asynchronous message passing paradigm of actors, while Modest relies on synchronous message passing. We used PRISM and IMCA for the analysis of systems, which are similar to Modest with respect to the analysis. The Modest toolset supports all formalisms in modeling and verification which are provided by PRISM and IMCA.

**ProbMela** ProbMela is a probabilistic version of Promela [117]. The operational semantics of ProbMela is defined as an MDP [118]. In [119], ProbMela is used as input language for the MDP model checker LiQuor which provides qualitative and quantitative analysis of LTL properties. There is also a mapping from ProbMela to the PRISM language, which makes probabilistic analysis possible [120].

PRebeca is an event-driven and actor-based language whereas ProbMela is process-based. Both languages are asynchronous in spirit. We proposed a semantics of PRebeca as TMDP (or PTA with digital clocks), enabling the analysis of timing and probabilistic behaviors of asynchronous systems via PRISM. Additionally, the TMDP obtained from a PRebeca model can be converted to an MA and the IMCA model checker can be used for the performance evaluation analysis.

**PMaude** PMaude extends standard rewriting theories of Maude with probability [121]. There is an actor extension of probabilistic rewriting theories for PMaude which removes nondeterminism. A statistical technique is provided to analyze quantitative aspects of systems using discrete-event simulation. In comparison with PMaude, modeling asynchronous systems is more straightforward in PRebeca language as it is an actor-based language. Also PRebeca supports nondeterminism in the model and there is no need to resolve it by assuming distribution on different choices of nondeterminism. It is because of the probabilistic model checking facilities which are provided by PRISM and IMCA.

**Actor Languages** Some work has been done on the development of actor frameworks based on familiar languages such as C/C++, Smalltalk, Python, Ruby, .NET and Java. To mention a few examples, Scala Actors library [122], Kilim [123], and ActorFoundry [124] are Java implementations of the actor model. More examples of actor frameworks for the above languages can be found in [125].

Comparing to the above actor-based programming languages, we are using a model-driven development approach in PRebeca language. We can start with small models and use model checking and simulation to find possible correctness problems in our core algorithms, and also find how to improve the performance by changing some parameters while the code is still small, understandable, and easily manageable.

**PCreol** Creol is an object-oriented modeling language based on concurrent objects, communicating by asynchronous message passing [20]. PCreol is the probabilistic extension of Creol, oriented towards quantitative analysis [126]. PCreol is integrated with VeStA [127] which enables the statistical model checking and quantitative analysis of PCreol models. Using VeStA, the full state-space exploration is replaced by Monte Carlo simulation, controlled by means of statistical hypothesis testing.

To have more accurate results, probabilistic model checking is provided for PRebeca models via PRISM and IMCA model checkers, which allows functional correctness

and performance evaluation of PTRebeca models. Both languages are similar with respect to asynchronous message passing among concurrent objects.

**Summary** In PMAude, probability distribution functions (rates and stochastic functions) are provided for modeling probabilistic behaviors. Also, PMAude implements stochastic continuous-time. In ProbMela, probabilities are drawn from discrete probability distributions, and passage of time can be modeled using a timer process. Modest enables a direct high-level modelling of PTA and more complex models. In all aforementioned languages, nondeterministic behavior can be modeled. In analysis, PMAude resolves nondeterminism, and uses statistical model checking to verify properties which results in inaccurate results. In the analysis of ProbMela and Modest, nondeterminism is not resolved. Modest also provides the option of a digital clock semantics, which, just like we did in Section 5.1.1, is handed over to PRISM for model checking.

Our focus in designing PTRebeca has been on ease of modeling and efficiency of analysis mainly for asynchronous applications. To this end, we use discrete time model and discrete probability distributions. These decisions showed to be effective in modeling different applications that we have targeted. Moreover, resolving nondeterminism by a discrete probability distribution generates inaccurate estimations, so, we avoided that by choosing TMDP as the semantics of PTRebeca. We were able to formalize the advance of time in our model using a single integer-valued variable. The language design of PTRebeca and its analysis approach when using PRISM, is closest to the Modest approach, apart from the latter not being object-oriented and not being asynchronous by design.

We also converted the TMDP resulted from a PTRebeca model to an MA. This way, we are able to use the IMCA model checker for large PTRebeca models. In MA, delays are governed by exponential distributions while in PTRebeca time is discrete. In the conversion, the rate in a Markovian transition of the MA is approximated by the integer value of the corresponding delay transition in the TMDP. To ensure the approximations are correct, we mathematically proved that expectation properties are preserved by this conversion.

## Chapter 6

# Conclusions and Future Work

This dissertation contains two parts. In the first part presented in Chapter 3, we developed techniques and extensions for making modeling and analysis of Timed Rebeca models easier. From the modeling point of view, we proposed an extension to the Timed Rebeca language [38] which provides the ability of calling Erlang functions. This way, the modeler may define functions and modules using all the programming features of Erlang which makes modeling easier than before. We also added the list data structure to Timed Rebeca, which is useful in modeling queues and buffers.

From the analysis point of view, a notable extension in the language is adding *checkpoint* functions to Timed Rebeca models. Our extensions in the language as well as timed extensions in McErlang provide us with model checking and performance evaluation of timed models. We developed a toolset to translate the Timed Rebeca models to Erlang. The mapping rules of the translation from Timed Rebeca to Erlang have been modified, compared to [38], to support timed extensions in McErlang. While model checking, safety monitors in McErlang can be defined to verify the correctness of models with respect to safety properties. In addition to these analysis facilities, we developed a statistical model checking tool for Timed Rebeca models. Using statistical model checking, we are able to verify safety properties of larger models for which the McErlang model checking suffers from the state space explosion problem.

McErlang is used to generate simulation traces of Timed Rebeca models. The traces are used for performance evaluation and statistical model checking of Timed Rebeca models. In simulation, the statistical methods are applied to simulation traces to reveal the system performance. In this work, two kinds of performance analysis are provided, which are paired-checkpoint analysis and checkpoint analysis. In checkpoint analysis, our focus is on the evolution of a particular parameter during time. In paired-checkpoint analysis, we study the difference between two values, like the duration of waiting, or service. This way, we provide the performance analysis of the system.

We evaluated the developed toolset and the proposed approaches using two case studies. In the elevator example, for different configurations we measure the response time of the requests arriving from each floor. Each configuration includes a different scheduling algorithm and a movement policy, which are responsible for assigning the requests to the elevators and determining how the elevators move between the floors, respectively. We also checked safety properties using both McErlang as the back-end model checker and the statistical model checking approach. In the ticket service example, for different settings the mean response times to ticket requests are calculated. Also, the safety property of “at least one ticket is issued” is checked using safety monitors and the statistical model checking method.

In the second part of this dissertation presented in Chapters 4 and 5, we introduced the PTRebeca language, an actor-based modeling language, and we also developed appropriate techniques and a supporting toolset for performance evaluation and model checking of distributed real-time systems with probabilistic behaviors.

In Chapter 4, we introduced the syntax and semantics of PTRebeca for modeling and verification of probabilistic real-time actor systems. The semantics of PTRebeca is presented in SOS rules. As the model of time in PTRebeca is discrete, we decided to use discrete-time MDP with an integer-valued time variable for the semantics of PTRebeca. PTRebeca models can thus be analyzed against PCTL, expected reachability, and probabilistic reachability properties.

In Chapter 5, we used PRISM as the back-end model checker for performance evaluation of PTRebeca models. As the TMDP of a PTRebeca model is input as one MDP module to PRISM, only small models like the ticket service can be input via the PRISM input language. To support the modeling of larger PTRebeca models, we used the explicit engine of PRISM which works with an intermediate transition matrix representation. Using this method, we could analyze larger models like the toxic gas sensing system, but PRISM does not support all the features for this format. So, we could analyze models only against probabilistic reachability properties. To overcome this shortage, we examined a parallel composition approach in which each PTRebeca component is converted to a PTA. The parallel composition of all PTAs represents the model behavior. The resulting PTA can be input to PRISM for performance analysis. We showed that this approach creates a larger state space comparing to the TMDP semantics. So, it is not efficient for performance analysis of PTRebeca models.

To provide probabilistic reachability and expected reachability properties for larger models, we proposed an approach in which the TMDP of a PTRebeca model is converted to one MA. The MA is input to the IMCA model checker for performance evaluation. We developed a toolset for automatic mapping of the TMDP to a single MA. We examined two case studies to show the applicability of our approach. The toxic gas sensing system was examined using the explicit engine of PRISM. Here, we obtained the identical results via mapping the TMDP to one MA and using the IMCA model checker, but in less amount of time. We also modeled a case study of a NoC network using PTRebeca, and evaluated the expected time properties by using the developed toolset.

In addition to the benefits of using the TMDP semantics for analysis of PTRebeca models, our technique is based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. Hence, the proposed semantics is general enough to be applied to similar computation models where there are message-driven communication and autonomous objects as units of concurrency, and there exists discrete probabilistic behavior in the model as is the case with agent-based systems.

**Future work.** The state space generated from the TMDP interpretation of PTRebeca models suffers from the state space explosion problem. In our semantics, the executions of statements of message servers are interleaved from various actors that are concurrently being executed in the real-time system. The semantics also includes a discrete global time and probabilistic information which make the state space explosion problem even more serious. In the semantics, the local time of all actors progresses in a lock step manner with the global time.

In [39], the authors proposed a floating time transition system (FTTS) as a solution of the state space explosion problem in model checking of Timed Rebeca models. In FTTS, actors proceed with their own rates with independent local clocks instead of synchronizing with the global time. In the Timed Rebeca language, and consequently in PRebeca language, actors can request a service from other actors by sending a message to them; each actor has a bag of messages which stores the received messages. The receiver actor takes a message from its bag and executes its corresponding message server to provide the requested service. In FTTS, by taking a transition, all statements of a message server of an actor are executed and the execution result is available in the next state. The execution of statements of the message server does not interleave with the execution of statements of other message servers from other actors. Since the message server may include timed statements, the local time of actors can have different values in a state. Relaxing the synchronization of progress of time among actors and the complete execution of a message server in a step avoid many interleaves and result in a significant state space reduction in FTTS.

As a future work, we can propose probabilistic floating time transition system (PFTTS) as an alternative semantics for the PRebeca language. PFTTS is a probabilistic extension of FTTS proposed in [39]. Similar to FTTS, the proposed semantics reduces the state space significantly in comparison to the TMDP semantics. Our intuitive understanding is that for a given PRebeca model, its TMDP interpretation and its PFTTS interpretation are probabilistic trace-distribution equivalence, but a formal proof is essential. If probabilistic trace-distribution equivalency holds, there is no LTL-without-Next formula which distinguishes two semantics in the sense that the min/max probabilities are the same for whatever formula is picked. As actions are not logged in the traces of a TMDP, internal actions are not logged in the traces, so LTL-without-Next properties are preserved. Therefore, the model checking algorithms proposed for LTL properties can be applied to PFTTS instead of TMDP. Hence, we should also develop a supporting toolset to generate the PFTTS semantics of a PRebeca model and to verify LTL properties.

In this work, we used PRISM and IMCA for performance evaluation and model checking of PRebeca models. As a future work, we can use Modest as the back-end toolset for the analysis of PRebeca models. The Modest toolset supports the modeling and analysis of hybrid, real-time, distributed and stochastic systems. Modest is a modular framework centered around the stochastic hybrid automata (SHA) formalism [128], and provides a variety of input languages and analysis back-end tools. A wide range of well-known and extensively studied formalisms such as PTA, timed automata (TA), MDP, MA, IMC can be seen as special cases of SHA. The Modest toolset uses well-established tools such as UPPAAL and PRISM for model checking of TA and PTA, respectively. Therefore, Modest can provide what we did for the analysis of PRebeca models, and can be chosen as an alternative back-end tool.

In this work, we provided model checking for performance analysis of PRebeca models. If a model is too large to be analyzed by model checking, a simulation technique is the only way of analyzing the model. As a future work, we should provide simulation techniques for PRebeca models. We can propose a mapping from PRebeca models to appropriate underlying models in Modest, and use the modes tool for simulation. The modes tool, a discrete-event simulator in Modest toolset, uses methods based on partial order reduction to decide, on-the-fly whether any nondeterminism it encounters can be safely resolved in an arbitrary way, or whether doing so could skew the simulation results [129].

In partial order reduction, only nondeterminism that results from interleaving due to parallel composition can be identified as spurious. The presence of a spurious nondeterministic choice in the model does not actually affect the simulation results. In modes, simulation proceeds as usual until a nondeterministic choice is encountered. Whenever that is the case, the partial order method is invoked to check which of the alternatives can safely be eliminated; if all can but one, simulation proceeds with that one [86].

# Bibliography

- [1] H. Kristinsson, *Event-based analysis of real-time actor models - Master Thesis*, Reykjavik University, Iceland, 2012.
- [2] C. Hewitt, “Description and theoretical analysis (using schemata) of PLAN-NER: A language for proving theorems and manipulating models in a robot”, Department of Computer Science, MIT, MIT Artificial Intelligence Technical Report 258, Apr. 1972.
- [3] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA, USA: MIT Press, 1990.
- [4] G. Agha, I. Mason, S. Smith, and C. Talcott, “A foundation for actor computation”, *Journal of Functional Programming*, vol. 7, pp. 1–72, 1997.
- [5] G. Agha, “The structure and semantics of actor languages”, in *REX Workshop*, 1990, pp. 1–59.
- [6] I. A. Mason and C. L. Talcott, “Actor languages: Their syntax, semantics, translation, and equivalence”, *Theoretical Computer Science*, vol. 220, no. 2, pp. 409–467, Jun. 1999, ISSN: 0304-3975. [Online]. Available: <http://www.elsevier.com/cas/tree/store/tcs/sub/1999/220/2/3170.pdf>.
- [7] C. Talcott, “Composable semantic models for actor theories”, *Higher-Order and Symbolic Computation*, vol. 11, no. 3, pp. 281–343, Dec. 1998, ISSN: 1388-3690. [Online]. Available: <http://www.wkap.nl/oasis.htm/188360>.
- [8] C. Talcott, “Actor theories in rewriting logic”, *Theoretical Computer Science*, vol. 285, no. 2, pp. 441–485, Aug. 2002, ISSN: 0304-3975.
- [9] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin, “Actor-oriented design of embedded hardware and software systems”, *Journal of Circuits, Systems, and Computers*, vol. 12, no. 3, pp. 231–260, 2003.
- [10] E. Cheong, E. A. Lee, and Y. Zhao, “Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks”, in *Proceedings of the 3<sup>rd</sup> international conference on Embedded networked sensor systems, SenSys 2005*, 2005, pp. 302–302.
- [11] P.-H. Chang and G. Agha, “Supporting reconfigurable object distribution for customized web applications”, in *The 22nd Annual ACM Symposium on Applied Computing (SAC’07)*, 2007, pp. 1286–1292.
- [12] —, “Towards context-aware web applications”, in *7th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS)*, 2007, pp. 239–252.

- [13] C. Hewitt, “What is commitment? Physical, organizational, and social (revised)”, in *Proceedings of Coordination, Organizations, Institutions, and Norms in Agent Systems II*, ser. Lecture Notes in Computer Science, Springer, 2007, pp. 293–307. DOI: 10.1007/978-3-540-74459-7\_19.
- [14] M. Sirjani and M. M. Jaghoori, “Ten years of analyzing actors: rebecca experience”, in *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, G. Agha, O. Danvy, and J. Meseguer, Eds., ser. Lecture Notes in Computer Science, vol. 7000, Springer, 2011, pp. 20–56, ISBN: 978-3-642-24932-7. DOI: 10.1007/978-3-642-24933-4\_3. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-24933-4\\_3](http://dx.doi.org/10.1007/978-3-642-24933-4_3).
- [15] R. Alur and D. Dill, “A theory of timed automata”, *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994. DOI: 10.1016/0304-3975(94)90010-8.
- [16] W. Yi, “Automata, languages and programming: 18th international colloquium madrid, spain, july 8–12, 1991 proceedings”, in J. L. Albert, B. Monien, and M. R. Artalejo, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, ch. CCS + time = an interleaving model for real time systems, pp. 217–228, ISBN: 978-3-540-47516-3. DOI: 10.1007/3-540-54233-7\_136. [Online]. Available: [http://dx.doi.org/10.1007/3-540-54233-7\\_136](http://dx.doi.org/10.1007/3-540-54233-7_136).
- [17] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking in dense real-time”, *Information and Computation*, vol. 104, pp. 2–34, 1993.
- [18] UPPAAL, *Uppaal homepage*, <http://uppaal.com>.
- [19] P. C. Ölveczky and J. Meseguer, “Semantics and pragmatics of real-time maude”, *Higher Order Symbol. Comput.*, vol. 20, no. 1-2, pp. 161–196, Jun. 2007, ISSN: 1388-3690. DOI: 10.1007/s10990-007-9001-5. [Online]. Available: <http://dx.doi.org/10.1007/s10990-007-9001-5>.
- [20] E. B. Johnsen and O. Owe, “An asynchronous communication model for distributed concurrent objects”, *Software & Systems Modeling*, vol. 6, no. 1, pp. 39–58, 2007.
- [21] J. Bjørk, E. B. Johnsen, O. Owe, and R. Schlatte, “Lightweight time modeling in timed creol”, in *Proceedings First International Workshop on Rewriting Techniques for Real-Time Systems, RTRTS 2010, Longyearbyen, Norway, April 6-9, 2010.*, P. C. Ölveczky, Ed., ser. Electronic Proceedings in Theoretical Computer Science, vol. 36, 2010, pp. 67–81. DOI: 10.4204/EPTCS.36.4. [Online]. Available: <http://dx.doi.org/10.4204/EPTCS.36.4>.
- [22] E. Broch Johnsen, O. Owe, R. Schlatte, and S. L. Tapia Tarifa, “Formal verification of object-oriented software: International conference, foveos 2010, paris, france, june 28-30, 2010, revised selected papers”, in B. Beckert and C. Marché, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ch. Validating Timed Models of Deployment Components with Parametric Concurrency, pp. 46–60, ISBN: 978-3-642-18070-5. DOI: 10.1007/978-3-642-18070-5\_4. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-18070-5\\_4](http://dx.doi.org/10.1007/978-3-642-18070-5_4).

- [23] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, “Formal methods for components and objects: 9th international symposium, fmco 2010, graz, austria, november 29 - december 1, 2010. revised papers”, in, B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. ABS: A Core Language for Abstract Behavioral Specification, pp. 142–164, ISBN: 978-3-642-25271-6. DOI: 10.1007/978-3-642-25271-6\_8. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-25271-6\\_8](http://dx.doi.org/10.1007/978-3-642-25271-6_8).
- [24] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa, “User-defined schedulers for real-time concurrent objects”, *Innovations in Systems and Software Engineering*, vol. 9, no. 1, pp. 29–43, 2013, ISSN: 1614-5054. DOI: 10.1007/s11334-012-0184-5. [Online]. Available: <http://dx.doi.org/10.1007/s11334-012-0184-5>.
- [25] E. B. Johnsen, R. Schlatte, and S. L. T. Tarifa, “Integrating deployment architectures and resource consumption in timed object-oriented models”, *J. Log. Algebr. Meth. Program.*, vol. 84, no. 1, pp. 67–91, 2015. DOI: 10.1016/j.jlamp.2014.07.001. [Online]. Available: <http://dx.doi.org/10.1016/j.jlamp.2014.07.001>.
- [26] S. Ren and G. Agha, “RT-synchronizer: language support for real-time specifications in distributed systems”, in *Workshop on Languages, Compilers and Tools for Real-Time Systems*, 1995, pp. 50–59.
- [27] M. Sirjani and A. Movaghar, “An actor-based model for formal modelling of reactive systems: Rebeca”, Sharif University of Technology, Department of Computer Engineering, Tehran, Iran, Tech. Rep. CS-TR-80-01, 2001.
- [28] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer, “Modeling and verification of reactive systems using Rebeca”, *Fundamenta Informatica*, vol. 63, no. 4, pp. 385–410, Dec. 2004.
- [29] M. Sirjani, A. Movaghar, and M. Mousavi, “Compositional verification of an object-based reactive system”, in *Proceedings of the Workshop on Automated Verification of Critical Systems (AVoCS’01)*, Oxford, UK, Apr. 2001, pp. 114–118. [Online]. Available: <http://mehr.sharif.edu/%5Csim%5Csirjani>.
- [30] M. Sirjani, A. Shali, M. Jaghoori, H. Irvanchi, and A. Movaghar, “A front-end tool for automated abstraction and modular verification of actor-based models”, in *Proceedings of Fourth International Conference on Application of Concurrency to System Design (ACSD’04)*, IEEE Computer Society, 2004, pp. 145–148.
- [31] M. Sirjani, A. Movaghar, A. Shali, and F. de Boer, “Model checking, automated abstraction, and compositional verification of Rebeca models”, *Journal of Universal Computer Science*, vol. 11, no. 6, pp. 1054–1082, 2005.
- [32] M. Sirjani, F. S. de Boer, and A. Movaghar, “Modular verification of a component-based actor language”, *Journal of Universal Computer Science*, vol. 11, no. 10, pp. 1695–1717, 2005.
- [33] M. M. Jaghoori, M. Sirjani, M. R. Mousavi, and A. Movaghar, “Efficient symmetry reduction for an actor-based model”, in *2nd International Conference on Distributed Computing and Internet Technology*, ser. Lecture Notes in Computer Science, vol. 3816, 2005, pp. 494–507.

- [34] M. M. Jaghoori, A. Movaghar, and M. Sirjani, “Modere: The model-checking engine of Rebeca”, in *Proceedings of the 21st Annual ACM Symposium on Applied Computing (SAC’06), Software Verificatin Track*, Apr. 2006, pp. 1810–1815.
- [35] H. Hojjat, M. Sirjani, M. R. Mousavi, and J. F. Groote, “Sarir: A Rebeca to mCRL2 translator”, *ACSD’07*, pp. 216–222, 2007.
- [36] H. Sabouri and M. Sirjani, “Slicing-based reductions for rebeca”, vol. 260, 2010, pp. 209–224.
- [37] —, “Actor-based slicing techniques for efficient reduction of Rebeca models”, *Sci. Comput. Program.*, vol. 75, no. 10, pp. 811–827, 2010.
- [38] L. Aceto, M. Cimini, A. Ingólfssdóttir, A. H. Reynisson, S. H. Sigurdarson, and M. Sirjani, “Modelling and simulation of asynchronous real-time systems using timed rebeca”, in *FOCLASA’11*, 2011, pp. 1–19.
- [39] E. Khamespanah, Z. Sabahi Kaviani, M. Sirjani, R. Khosravi, and M.-J. Izadi, “Timed rebeca schedulability and deadlock freedom analysis using bounded floating-time transition system”, in *Journal of Science of Computer Programming*, 2014.
- [40] M. Sirjani and E. Khamespanah, “Theory and practice of formal methods: Essays dedicated to frank de boer on the occasion of his 60th birthday”, in, E. Ábrahám, M. Bonsangue, and B. E. Johnsen, Eds. Cham: Springer International Publishing, 2016, ch. On Time Actors, pp. 373–392, ISBN: 978-3-319-30734-3. DOI: 10.1007/978-3-319-30734-3\_25. [Online]. Available: [http://dx.doi.org/10.1007/978-3-319-30734-3\\_25](http://dx.doi.org/10.1007/978-3-319-30734-3_25).
- [41] Z. Sharifi, M. Mosaffa, S. Mohammadi, and M. Sirjani, “Functional and Performance Analysis of Network-on-Chips Using Actor-based Modeling and Formal Verification”, ser. In proceedings of AVOCs’13, 2013.
- [42] Z. Sharifi, S. Mohammadi, and M. Sirjani, “Comparison of noc routing algorithms using formal methods”, ser. In proceedings of PDPTA’13, 2013.
- [43] E. Khamespanah, K. A. Mehitov, M. Sirjani, and G. Agha, “Schedulability analysis of distributed real-time sensor network applications using actor-based model checking”, ser. In Proceedings of the 23rd International SPIN symposium on Model Checking of Software, 2016.
- [44] L. Linderman, K. Mehitov, and B. F. Spencer, “Tinyos-based real-time wireless data acquisition framework for structural health monitoring and control”, *Structural Control and Health Monitoring*, vol. 20, no. 6, pp. 1007–1020, June 2013.
- [45] I. Gupta, B. Cho, M. R. Rahman, T. Chajed, C. L. Abad, N. Roberts, and P. Lin, “Natjam: eviction policies for supporting priorities and deadlines in mapreduce clusters”, 2009.
- [46] M. Varshosaz and R. Khosravi, “Modeling and verification of probabilistic actor systems using prebeca”, in *Proceedings of the 14th international conference on Formal Engineering Methods (Kyoto, Japan)*, ser. ICFEM’12, Berlin, Heidelberg: Springer-Verlag, 2012, pp. 135–150, ISBN: 978-3-642-34280-6.

- [47] C. Baier, B. R. Haverkort, H. Hermanns, and J.-P. Katoen, “Performance evaluation and model checking join forces”, *Commun. ACM*, vol. 53, no. 9, pp. 76–85, Sep. 2010, ISSN: 0001-0782.
- [48] C. Baier and J.-P. Katoen, *Principles of Model Checking*. The MIT Press, May 2008.
- [49] A. Bianco and L. D. Alfaro, “Model checking of probabilistic and nondeterministic systems”, in *Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag, 1995, pp. 499–513.
- [50] C. Courcoubetis and M. Yannakakis, “The complexity of probabilistic verification”, *J. ACM*, vol. 42, no. 4, pp. 857–907, Jul. 1995, ISSN: 0004-5411.
- [51] M. Y. Vardi, “Automatic verification of probabilistic concurrent finite state programs”, in *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, ser. SFCS ’85, Washington, DC, USA: IEEE Computer Society, 1985, pp. 327–338. DOI: 10.1109/SFCS.1985.12. [Online]. Available: <http://dx.doi.org/10.1109/SFCS.1985.12>.
- [52] C. B. Earle and L. Fredlund, “Verification of timed Erlang programs using McErlang”, in *Proceedings of the 14th joint IFIP WG 6.1 international conference and Proceedings of the 32nd IFIP WG 6.1 international conference on Formal Techniques for Distributed Systems (Stockholm, Sweden)*, ser. FMOODS’12/FORTE’12, Berlin, Heidelberg: Springer-Verlag, 2012, pp. 251–267, ISBN: 978-3-642-30792-8. [Online]. Available: [http://dx.doi.org/10.1007/978-3-642-30793-5\\_16](http://dx.doi.org/10.1007/978-3-642-30793-5_16).
- [53] L.-Å. Fredlund and H. Svensson, “McErlang: a model checker for a distributed functional programming language”, *SIGPLAN Not.*, vol. 42, no. 9, pp. 125–136, 2007, ISSN: 0362-1340. DOI: <http://doi.acm.org/10.1145/1291220.1291171>.
- [54] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker, “Prism: A tool for automatic verification of probabilistic systems”, in *Proceedings of 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’06)*, ser. Lecture Notes in Computer Science, Springer-Verlag, 2006, pp. 441–444.
- [55] D. Guck, T. Han, J.-P. Katoen, and M. R. Neuhäuser, “Quantitative timed analysis of interactive markov chains”, in *Proceedings of the 4th international conference on NASA Formal Methods (Norfolk, VA)*, ser. NFM’12, Berlin, Heidelberg: Springer-Verlag, 2012, pp. 8–23, ISBN: 978-3-642-28890-6.
- [56] G. D. Plotkin, “A structural approach to operational semantics”, Computer Science Department, Aarhus University, Aarhus, Denmark, Tech. Rep. DAIMI FN-19, Sep. 1981.
- [57] C. Eisentraut, H. Hermanns, and L. Zhang, “On probabilistic automata in continuous time”, in *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*, 2010, pp. 342–351.
- [58] H. Kristinsson, A. Jafari, E. Khamespanah, B. Magnusson, and M. Sirjani, “Analysing timed rebeca using mcerlang”, in *Proceedings of the 2013 Workshop on Programming Based on Actors, Agents, and Decentralized Control (Indianapolis, Indiana, USA)*, ser. AGERE! 2013, New York, NY, USA: ACM, 2013, pp. 25–36, ISBN: 978-1-4503-2602-5.

- [59] A. Jafari, E. Khamespanah, H. Kristinsson, M. Sirjani, and B. Magnusson, “Statistical model checking of Timed Rebeca models”, *Journal of Computer Languages, Systems and Structures (in press)*, 2015. DOI: <http://dx.doi.org/10.1016/j.cl.2016.01.004>.
- [60] A. Jafari, E. Khamespanah, M. Sirjani, and H. Hermanns, “Performance analysis of distributed and asynchronous systems using probabilistic timed actors”, ser. In proceedings of AVoCS’14, 2014.
- [61] A. Jafari, E. Khamespanah, M. Sirjani, H. Hermanns, and M. Cimini, “PTRebeca: modeling and analysis of distributed and asynchronous systems”, *Journal of Science of Computer Programming*, 2015.
- [62] H. Hansson and B. Jonsson, “A logic for reasoning about time and reliability”, *Formal Aspects of Computing*, vol. 6, pp. 102–111, 1994.
- [63] L. De Alfaro, “Formal verification of probabilistic systems”, PhD thesis, Stanford, CA, USA, 1998, ISBN: 0-591-90826-3.
- [64] C. Baier and M. Kwiatkowska, “Model checking for a probabilistic branching time logic with fairness”, *Distributed Computing*, vol. 11, pp. 125–155, 1998.
- [65] R. Alur, C. Courcoubetis, and D. Dill, “Model-checking for probabilistic real-time systems”, in *Automata, Languages and Programming: Proceedings of the 18th ICALP*, ser. Lecture Notes in Computer Science 510, 1991.
- [66] M. Kwiatkowska, G. Norman, R. Segala, and J. Sproston, “Automatic verification of real-time systems with discrete probability distributions”, *Theor. Comput. Sci.*, vol. 282, no. 1, pp. 101–150, Jun. 2002, ISSN: 0304-3975.
- [67] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen, “Model-checking algorithms for continuous-time markov chains”, *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 29, no. 7, p. 2003, 2003.
- [68] F. Laroussinie and J. Sproston, “Model checking durational probabilistic systems”, in *Proceedings of the 8th international conference on Foundations of Software Science and Computation Structures (Edinburgh, UK)*, ser. FOSSACS’05, Berlin, Heidelberg: Springer-Verlag, 2005, pp. 140–154, ISBN: 3-540-25388-2. DOI: 10.1007/978-3-540-31982-5\_9. [Online]. Available: [http://dx.doi.org/10.1007/978-3-540-31982-5\\_9](http://dx.doi.org/10.1007/978-3-540-31982-5_9).
- [69] R. Alur and M. Bernadsky, “Bounded model checking for GSMP models of stochastic real-time systems”, in *Proceedings of the 9th international conference on Hybrid Systems: Computation and control (Santa Barbara, CA)*, ser. HSCC’06, Berlin, Heidelberg: Springer-Verlag, 2006, pp. 19–33, ISBN: 3-540-33170-0. DOI: 10.1007/11730637\_5. [Online]. Available: [http://dx.doi.org/10.1007/11730637\\_5](http://dx.doi.org/10.1007/11730637_5).
- [70] C. Baier, L. Cloth, B. Haverkort, M. Kuntz, and M. Siegle, “Model checking action- and state-labelled markov chains”, in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, ser. DSN ’04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 701–, ISBN: 0-7695-2052-9. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1009382.1009786>.

- [71] S. Donatelli, S. Haddad, and J. Sproston, “Cslta: An expressive logic for continuous-time markov chains”, in *Proceedings of the Fourth International Conference on Quantitative Evaluation of Systems*, ser. QEST '07, Washington, DC, USA: IEEE Computer Society, 2007, pp. 31–40, ISBN: 0-7695-2883-X. DOI: 10.1109/QEST.2007.14. [Online]. Available: <http://dx.doi.org/10.1109/QEST.2007.14>.
- [72] H. E. Jensen, “Model checking probabilistic real time systems”, in *Chalmers Institute of Technology*, 1996, pp. 247–261.
- [73] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, “Performance analysis of probabilistic timed automata using digital clocks”, *Formal Methods in System Design*, vol. 29, pp. 33–78, 2006.
- [74] M. Jurdziński, F. Laroussinie, and J. Sproston, “Model checking probabilistic timed automata with one or two clocks”, in *Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems (Braga, Portugal)*, ser. TACAS'07, Berlin, Heidelberg: Springer-Verlag, 2007, pp. 170–184, ISBN: 978-3-540-71208-4.
- [75] M. Stoelinga, “Alea jacta est: Verification of probabilistic, real-time and parametric systems”, PhD thesis, University of Nijmegen, Netherlands, 2002.
- [76] H. Bohnenkamp, P. R. D’Argenio, H. Hermanns, and J.-P. Katoen, “Modest: A compositional modeling formalism for hard and softly timed systems”, *IEEE Transactions on Software Engineering*, vol. 32, no. 10, pp. 812–830, 2006, ISSN: 0098-5589. DOI: <http://doi.ieeecomputersociety.org/10.1109/TSE.2006.104>.
- [77] A. Hartmanns and H. Hermanns, “A MoDeST approach to checking probabilistic timed automata”, in *Proceedings of the 2009 Sixth International Conference on the Quantitative Evaluation of Systems*, ser. QEST '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 187–196, ISBN: 978-0-7695-3808-2. DOI: 10.1109/QEST.2009.41. [Online]. Available: <http://dx.doi.org/10.1109/QEST.2009.41>.
- [78] M. Kwiatkowska, G. Norman, and D. Parker, “Prism: Probabilistic model checking for performance and reliability analysis”, *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 40–45, 2009.
- [79] N. Coste, H. Garavel, H. Hermanns, F. Lang, R. Mateescu, and W. Serwe, “Ten years of performance evaluation for concurrent systems using CADP”, in *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II (Heraklion, Crete, Greece)*, ser. ISoLA'10, Berlin, Heidelberg: Springer-Verlag, 2010, pp. 128–142, ISBN: 3-642-16560-5. [Online]. Available: <http://dl.acm.org/citation.cfm?id=1939345.1939366>.
- [80] H. Hermanns and J.-P. Katoen, “Automated compositional markov chain generation for a plain-old telephone system”, in *SCIENCE OF COMPUTER PROGRAMMING*, 1999, pp. 97–127.
- [81] H. Hermanns, *Interactive Markov chains: And the quest for quantified quality*. Berlin, Heidelberg: Springer-Verlag, 2002, ISBN: 3-540-44261-8.

- [82] N. Coste, H. Hermanns, E. Lantreibeccq, and W. Serwe, “Towards performance prediction of compositional models in industrial gals designs”, in *Proceedings of the 21st International Conference on Computer Aided Verification (Grenoble, France)*, ser. CAV ’09, Berlin, Heidelberg: Springer-Verlag, 2009, pp. 204–218, ISBN: 978-3-642-02657-7.
- [83] Erlang, *Erlang programming language homepage*, <http://www.erlang.org>.
- [84] L. Lamport, “Real-time model checking is really simple”, in *Proceedings of the 13 IFIP WG 10.5 international conference on Correct Hardware Design and Verification Methods (Saarbrücken, Germany)*, ser. CHARME’05, Berlin, Heidelberg: Springer-Verlag, 2005, pp. 162–175, ISBN: 3-540-29105-9.
- [85] Rebeca, *Rebeca homepage*, <http://www.rebeca-lang.org>.
- [86] A. Hartmanns, “Model-checking and simulation for stochastic timed systems”, in *FMCO*, B. K. Aichernig, F. S. de Boer, and M. M. Bonsange, Eds., ser. Lecture Notes in Computer Science, vol. 6957, Springer, Dec. 2010, pp. 372–391, ISBN: 978-3-642-25270-9. eprint: <http://www.modestchecker.net/Link.aspx?id=pub:H10>.
- [87] R. M. Karp, M. Luby, and N. Madras, “Monte-carlo approximation algorithms for enumeration problems”, *J. Algorithms*, vol. 10, no. 3, pp. 429–448, 1989.
- [88] P. Dagum, R. M. Karp, M. Luby, and S. M. Ross, “An optimal algorithm for monte carlo estimation”, *SIAM J. Comput.*, vol. 29, no. 5, pp. 1484–1496, 2000.
- [89] R. Grosu and S. A. Smolka, “Quantitative model checking”, in *International Symposium on Leveraging Applications of Formal Methods, ISoLA 2004, October 30 - November 2, 2004, Paphos, Cyprus. Preliminary proceedings*, 2004, pp. 165–174.
- [90] A. Legay, B. Delahaye, and S. Bensalem, “Statistical model checking: an overview”, in *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, 2010, pp. 122–135.
- [91] J. S. Simonoff, *Smoothing Methods in Statistics*. Springer, 1998, pp. I–XII, 1–338, ISBN: 978-0-387-94716-7.
- [92] *Z table site*, <http://www.statisticshowto.com/tables/z-table/>.
- [93] A. H. Buss, “Modeling with event graphs”, ser. In Proceedings of the 28th conference on winter simulation, Washington, DC, USA, 1996, pp. 153–160.
- [94] E. D’Osualdo, J. Kochems, and C.-H. Ong, “Automatic verification of erlang-style concurrency”, in *Static Analysis*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds., vol. 7935, Springer Berlin Heidelberg, 2013, pp. 454–476, ISBN: 978-3-642-38855-2.
- [95] R. Carlsson, “An introduction to core erlang”, in *In Proceedings of the PLI’01 Erlang Workshop*, 2001.
- [96] A. David, K. Larsen, A. Legay, M. Mikučionis, and D. Poulsen, “Uppaal smc tutorial”, *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015, ISSN: 1433-2779.

- [97] T. Bøgholm, H. Kragh-Hansen, P. Olsen, B. Thomsen, and K. G. Larsen, “Model-based schedulability analysis of safety critical hard real-time java programs”, in *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (Santa Clara, California, USA)*, ser. JTRES '08, New York, NY, USA: ACM, 2008, pp. 106–114, ISBN: 978-1-60558-337-2.
- [98] T. Henties, J. J. Hunt, D. Locke, K. Nilsen, M. Schoeberl, and J. Vitek, “Java for safety-critical applications”, in *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.
- [99] C. Frost, C. S. Jensen, K. S. Luckow, and B. Thomsen, “Wcet analysis of java bytecode featuring common execution environments”, in *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (York, United Kingdom)*, ser. JTRES '11, New York, NY, USA: ACM, 2011, pp. 30–39, ISBN: 978-1-4503-0731-4.
- [100] M. Schoeberl, W. Puffitsch, R. U. Pedersen, and B. Huber, “Worst-case execution time analysis for a java processor”, *Softw. Pract. Exper.*, vol. 40, no. 6, pp. 507–542, May 2010, ISSN: 0038-0644.
- [101] M. Schoeberl, H. Søndergaard, B. Thomsen, and A. P. Ravn, “A profile for safety critical java”, in *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*, 2007, pp. 94–101.
- [102] E. Khamespanah, M. Sirjani, Z. Sabahi-Kaviani, R. Khosravi, and M. Izadi, “Timed rebecca schedulability and deadlock freedom analysis using bounded floating time transition system”, *Sci. Comput. Program.*, vol. 98, pp. 184–204, 2015. DOI: 10.1016/j.scico.2014.07.005. [Online]. Available: <http://dx.doi.org/10.1016/j.scico.2014.07.005>.
- [103] E. Khamespanah, R. Khosravi, and M. Sirjani, “Efficient tctl model checking algorithm for timed actors”, in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control (Portland, Oregon, USA)*, ser. AGERE! '14, New York, NY, USA: ACM, 2014, pp. 55–66, ISBN: 978-1-4503-2189-1. DOI: 10.1145/2687357.2687366. [Online]. Available: <http://doi.acm.org/10.1145/2687357.2687366>.
- [104] Z. Sabahi-Kaviani, R. Khosravi, M. Sirjani, P. C. Ölveczky, and E. Khamespanah, “Formal semantics and analysis of timed rebecca in real-time maude”, in *FTSCS*, 2013, pp. 178–194.
- [105] C. Derman, *Finite State Markovian Decision Processes*. Orlando, FL, USA: Academic Press, Inc., 1970, ISBN: 0122092503.
- [106] R. Segala, “Modeling and verification of randomized distributed real-time systems”, PhD thesis, Cambridge, MA, USA, 1995.
- [107] R. Verdone, D. Dardari, G. Mazzini, and A. Conti, *Wireless Sensor and Actuator Networks*, ser. Elsevier. Academic Press, 2008.
- [108] E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen, “A compositional modelling and analysis framework for stochastic hybrid systems”, *Formal Methods in System Design*, vol. 43, no. 2, pp. 191–232, 2013.

- [109] D. Guck, H. Hatefi, H. Hermanns, J. Katoen, and M. Timmer, “Analysis of timed and long-run objectives for markov automata”, *Logical Methods in Computer Science*, vol. 10, no. 3, 2014.
- [110] D. P. Bertsekas and J. N. Tsitsiklis, “An analysis of stochastic shortest path problems”, *Math. Oper. Res.*, vol. 16, no. 3, pp. 580–595, Aug. 1991, ISSN: 0364-765X.
- [111] L. de Alfaro, “Computing minimum and maximum reachability times in probabilistic systems”, in *CONCUR '99: Concurrency Theory, 10th International Conference, Eindhoven, The Netherlands, August 24-27, 1999, Proceedings, 1999*, pp. 66–81.
- [112] D. Guck, M. Timmer, H. Hatefi, E. Ruijters, and M. Stoelinga, “Modelling and analysis of markov reward automata”, in *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings, 2014*, pp. 168–184.
- [113] M. Krstić, E. Grass, F. K. Gürkaynak, and P. Vivet, “Globally asynchronous, locally synchronous circuits: Overview and outlook”, *IEEE Design & Test of Computers*, no. 5, pp. 430–441, 2007.
- [114] A. Hosseini, T. Ragheb, and Y. Massoud, “A fault-aware dynamic routing algorithm for on-chip networks”, in *International Symposium on Circuits and Systems (ISCAS 2008), 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA, 2008*, pp. 2653–2656.
- [115] D. Guck, H. Hatefi, H. Hermanns, J. Katoen, and M. Timmer, “Modelling, reduction and analysis of markov automata”, in *Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings, 2013*, pp. 55–71.
- [116] A. Hartmanns and H. Hermanns, “A modest approach to checking probabilistic timed automata”, in *QEST*, IEEE Computer Society, 2009, pp. 187–196, ISBN: 978-0-7695-3808-2.
- [117] G. J. Holzmann, “The model checker SPIN”, *Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [118] C. Baier, F. Ciesinski, and M. Groesser, *Probmela: A modeling language for communicating probabilistic processes*, 2004.
- [119] F. Ciesinski and C. Baier, “Liquor: a tool for qualitative and quantitative linear time analysis of reactive systems”, in *Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST'06)*, IEEE CS Press, 2006, pp. 131–132.
- [120] F. Ciesinski, C. Baier, M. Groesser, and D. Parker, “Generating compact mtbdd-representations from probmela specifications”, in *Proceedings of the 15th international workshop on Model Checking Software (Los Angeles, CA, USA)*, ser. SPIN '08, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 60–76, ISBN: 978-3-540-85113-4.
- [121] G. Agha, J. Meseguer, and K. Sen, “Pmaude: Rewrite-based specification language for probabilistic object systems”, *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 153, no. 2, pp. 213–239, May 2006, ISSN: 1571-0661.

- [122] P. Haller and M. Odersky, “Actors that unify threads and events”, in *Coordination Models and Languages*, Springer, 2007, pp. 171–190.
- [123] S. Srinivasan and A. Mycroft, “Kilim: Isolation-typed actors for java”, in *ECOOOP 2008–Object-Oriented Programming*, Springer, 2008, pp. 104–128.
- [124] M. Astley, “The actor foundry: a java-based actor programming environment”, *University of Illinois at Urbana-Champaign: Open Systems Laboratory*, 1998.
- [125] R. K. Karmani, A. Shali, and G. Agha, “Actor frameworks for the jvm platform: a comparative analysis”, in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ACM, 2009, pp. 11–20.
- [126] L. Bentea and O. Owe, “A probabilistic framework for object-oriented modeling and analysis of distributed systems”, in *Formal Verification of Object-Oriented Software*, Springer, 2012, pp. 105–122.
- [127] K. Sen, M. Viswanathan, and G. A. Agha, “Vesta: A statistical model-checker and analyzer for probabilistic systems”, in *Second International Conference on the Quantitative Evaluation of Systems (QEST 2005), 19-22 September 2005, Torino, Italy*, IEEE Computer Society, 2005, pp. 251–252.
- [128] E. M. Hahn, A. Hartmanns, H. Hermanns, and J.-P. Katoen, “A compositional modelling and analysis framework for stochastic hybrid systems”, *Formal Methods in System Design*, 2012, ISSN: 0925-9856. eprint: <http://www.modestchecker.net/Link.aspx?id=pub:HHHK12>.
- [129] J. Bogdoll, A. Hartmanns, and H. Hermanns, “Simulation and statistical model checking for modestly nondeterministic models”, in *MMB/DFT*, ser. Lecture Notes in Computer Science, vol. 7201, Springer, Mar. 2012, pp. 249–252, ISBN: 978-3-642-28539-4. eprint: <http://www.modestchecker.net/Link.aspx?id=pub:BHH12>.



# Appendix A

## Pseudocode of Policies

```

1 msgsrv handleRequest(Floor f)
2 {
3   if (sender is instance of Floor) {
4     if Contains(ElvQueue1,f) || Contains(ElvQueue2,f)
5       donothing;
6     else {
7       if (ElvLoc1 == f)
8         Add(ElvQueue1,f);
9       else if (ElvLoc2 == f)
10        Add(ElvQueue2,f);
11       else if (ElvLoc1 == ElvLoc2){
12         RandQueue = chooseRand(ElvLoc1,ElvQueue2);
13         Add(RandQueue,f);
14       }
15       else if (abs(f-ElvLoc1) > abs(f-ElvLoc2))
16         Add(ElvQueue2,f);
17       else
18         Add(ElvQueue1,f);
19     }
20   }
21   else if (sender is instance of Elevator){
22     if (sender == Elevator1 && ElvLoc1 != f && !Contains(ElvQueue1,f))
23       Add(ElvQueue1,f);
24     else if (sender == Elevator2 && ElvLoc2 != f && !Contains(ElvQueue2,f))
25       Add(ElvQueue2,f);
26     else if (ElvLoc1 == f || ElvLoc2 == f)
27       SendMessage(sender,StopAndOpen);
28   }
29   // any idle elevators should be started
30   ...
31 }

```

Listing A.1: Pseudo code of message server *HandleRequest* where the scheduling policy is shortest distance policy.

```

1  /* Scheduling policy: Shortest distance policy with movement priority. */
2  ...
3  /* Check if any elevators are already located on the requested floor */
4  ...
5  else if (abs(floor-Elv1Loc) > abs(floor-Elv2Loc)){
6      if (floor > Elv2Location && Elv2Movment==1)
7          Add(Elv2Queue,floor);
8      else if (floor < Elv2Location && Elv2Movment==-1)
9          Add(Elv2Queue,floor);
10     else if (floor > Elv1Location && Elv1Movement==1)
11         Add(ElvQueue1,floor);
12     else if (floor < Elv1Location && Elv1Movement==-1)
13         Add(ElvQueue1,floor);
14     else
15         Add(ElvQueue2,floor);
16 }
17 else{
18     if (floor > Elv1Location && Elv1Movment==1)
19         Add(Elv1Queue,floor);
20     else if (floor < Elv1Location && Elv1Movment==-1)
21         Add(Elv1Queue,floor);
22     else if (floor > Elv2Location && Elv2Movement==1)
23         Add(ElvQueue2,floor);
24     else if (floor < Elv2Location && Elv2Movement==-1)
25         Add(ElvQueue2,floor);
26     else
27         Add(ElvQueue1,floor);
28 }
29 ...

```

Listing A.2: Timed Rebeca pseudo code for scheduling policy *shortest distance with movement priority*. [...] denotes the deleted code which has been already shown in Listing A.1. The variable *floor* is the requested floor number sent by the *rebec pers*.

```

1  /* Scheduling policy: Shortest distance policy with load balancing. */
2  ...
3  /* Check if any elevators are already located on the requested floor */
4  ...
5  else if (abs(floor-Elv1Loc) > abs(floor-Elv2Loc)){
6      if (Size(Elv2Queue) < Size(Elv1Queue) || Size(Elv2Queue) = Size(Elv1Queue))
7          Add(Elv2Queue,floor);
8      else
9          Add(Elv1Queue,floor);
10 }
11 else {
12     if (Size(Elv1Queue) < Size(Elv2Queue) || Size(Elv1Queue) = Size(Elv2Queue))
13         Add(Elv1Queue,floor);
14     else
15         Add(Elv2Queue,floor);
16 }
17 ...

```

Listing A.3: Timed Rebeca pseudo code for scheduling policy *shortest distance with load balancing*. [...] denotes deleted code which has been already shown in Listing A.1. The variable *floor* is the requested floor number sent by the *pers* rebec.

```

1 msgsrv handleElevatorMovement(int movement)
2 {
3     // movement=0 means elevator stopped,
4     // movement=1 means elevator is going up
5     // movement=-1 means elevator is going down
6     if (sender == Elevator1 && movement != 0){ //moving elevator
7         Elv1Movement = movement;
8         if (movement == -1)
9             Elv1Location-=1;
10        else if (movement == 1)
11            Elv1Location+=1;
12        if (Size(Elv1Queue) > 0){
13            if (Contains(Elv1Queue, Elv1Location)){
14                Elv1Queue.Remove(Elv1Location);
15                SendMessage(Elevator1,StopOpen);
16            }
17            else{
18                if (Next(Elv1Queue,Elv1Location,1) != -1){
19                    Elv1Movement = 1;
20                    SendMessage(Elevator1,MoveUp);
21                }
22                else{
23                    Elv1Movement = -1;
24                    SendMessage(Elevator1,MoveDown);
25                }
26            }
27        }
28    }
29    else if (sender == Elevator1){ // Stopped Elevator
30        Elv1Movement = movement;
31        if (Elv1Movement == 0 && Size(Elv1Queue1) > 0){
32            if (Next(Elv1Queue,Elv1Location,1) != -1){
33                Elv1Movement = 1;
34                SendMessage(Elevator1,MoveUp);
35            }
36            else{
37                Elv1Movement = -1;
38                SendMessage(Elevator1,MoveDown);
39            }
40        }
41    }
42 }
43 // movement for elevator 2
44 ....
45 }

```

Listing A.4: Timed Rebeca pseudo code of message server *handleElevatorMovement* where the movement policy is *up priority* policy. Contains and Next are custom functions. Pseudo code presented is for Elevator 1 in the model.

```

1  /* Movement policy: Maintain movement Policy. */
2  ...
3  /* Check if elevators are on the requested floor before moving */
4  ...
5  /* If elevator queue is not empty: */
6  /* If movement is UP and there is a request higher than the current floor then go up.
   Otherwise go down. */
7  if (Movement==1){
8      if (Next(Elv1Queue,Elv1Location,1) != -1){
9          Elv1Movement=1;
10         SendMessage(Elevator1,MoveUp);
11     }
12     else{
13         Elv1Movement=-1;
14         SendMessage(Elevator1,MoveDown);
15     }
16     /* ElseIf movement is DOWN and there is a request lower than the current floor then go
       down. Otherwise go up. */
17 else{
18     if (Next(Elv1Queue,Elv1Location,-1) != -1){
19         Elv1Movement=-1;
20         SendMessage(Elevator1,MoveDown);
21     }
22     else{
23         Elv1Movement=1;
24         SendMessage(Elevator1,MoveUp);
25     }
26     ...

```

Listing A.5: Timed Rebeca pseudo code for movement policy *Maintain movement*. The pseudo code shows the algorithm for elevator 1.

# Appendix B

## Proofs of Theorems

**Proof of Theorem 2.** We show that

$$L(eT^{min}(s, \Diamond G)) = eT^{min}(s, \Diamond G) \quad (\text{B.1})$$

for all  $s \in S$ . To this aim, we distinguish three cases which are  $s \in DS \setminus G$ ,  $s \in PS \setminus G$ , and  $s \in G$ .

- in case of  $s \in DS \setminus G$ , the left-hand side of (B.1) is:

$$L(eT^{min}(s, \Diamond G)) = d_s + eT^{min}(t, \Diamond G), \quad (\text{B.2})$$

where  $d_s$  is delay time reaching state  $t$  from state  $s$ . This delay time is deterministic. On the other hand,

$$eT^{min}(t, \Diamond G) = \inf_D \mathbb{E}_{t,D}(V_G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{t,D}(\pi) \quad (\text{B.3})$$

Combining (B.2) and (B.3), we have

$$\begin{aligned} L(eT^{min}(s, \Diamond G)) &= d_s + \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (V_G(\pi) + d_s) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (V_G(\pi)) \cdot Pr_{s,D}(\pi) \\ &= eT^{min}(s, \Diamond G) \end{aligned} \quad (\text{B.4})$$

Note that in (B.4), in the second line, paths, start from  $t$ , whereas, in the third line, paths start from  $s$ .

- in case of  $s \in PS \setminus G$  there is:

$$\begin{aligned}
eT^{min}(s, \Diamond G) &= \inf_D \mathbb{E}_{s,D}(V_G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{s,D}(\pi) \\
&= \inf_D \sum_{s \xrightarrow{\alpha, \mu, 0} t} D(s)(\alpha) \cdot \mathbb{E}_{t,D(s) \xrightarrow{\alpha, \mu, 0} \cdot} (V_G) \\
&= \inf_D \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D(s) \xrightarrow{\alpha, \mu, 0} \cdot} (V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \inf_D \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D(s) \xrightarrow{\alpha, \mu, 0} \cdot} (V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \inf_D \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D}(V_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \sum_{t \in S} \mu_\alpha^s(t) \cdot eT^{min}(s, \Diamond G) \\
&= \min_{\alpha \in Act(s)} \sum_{t \in S} \mu_\alpha^s(t) \cdot eT^{min}(s, \Diamond G) \\
&= L(eT^{min}(s, \Diamond G))
\end{aligned}$$

- in case of  $s \in G$ , based on the definition there is  $eT^{min}(s, \Diamond G) = \inf_D \sum_{Paths} V_G(\pi) \cdot Pr_{s,D}(\pi) = 0$ , which is the same as the value of the Bellman operator for goal states.

**Proof of Theorem 3.** From [110],  $cR^{min}(s, \Diamond G)$  is the unique fixpoint of the bellman operator  $L'$  defined as

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s, \alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s, \alpha, s') \cdot v(s') + \sum_{s' \in G} \mathbf{P}(s, \alpha, s') \cdot g(s') \right\}. \quad (\text{B.5})$$

Now we show that the Bellman operator  $L$  defined in Theorem 2, and the Bellman operator  $L'$  defined in (B.5) for  $sset(\mathcal{M})$  are the same. By definition 19, for each  $s \in S$ ,  $g(s) = 0$ , therefore

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s, \alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s, \alpha, s') \cdot v(s') \right\}. \quad (\text{B.6})$$

Consider three cases,  $s \in DS \setminus G$ ,  $s \in PS \setminus G$  and  $s \in G$ .

- *Case (I):* Assume  $s \in DS \setminus G$ , by definition 19,  $c(s, \alpha) = d_s$  and

$$\mathbf{P}(s, \alpha, s') = \begin{cases} 1 & s' \text{ is reaching state from } s \text{ by delay } d_s \text{ and } \alpha = \perp, \\ 0 & \text{otherwise} \end{cases}$$

Only action belongs to  $Act(s)$  is  $\perp$ , furthermore only state after  $s$  is state  $s'$ , thus, from (B.6),

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s, \alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s, \alpha, s') \cdot v(s') \right\} = d_s + v(s'),$$

where  $s'$  is deterministic reaching state from  $s$  by delay time  $d_s$ . However from Theorem 2,  $d_s + v(s') = [L(v)](s)$ , for each  $s \in DS \setminus G$ , so in this case, theorem is proved.

- *Case (II)*: Assume  $s \in PS \setminus G$ . By Definition 19,  $\mathbf{P}(s, \alpha, s') = \mu_\alpha^s(s')$  and  $c(s, \alpha) = 0$ . Therefore,

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s, \alpha) + \sum_{s' \in S} \mathbf{P}(s, \alpha, s') \cdot v(s') \right\} = \min \left\{ \sum_{s' \in S} \mu_\alpha^s(s') \cdot v(s') \right\}$$

However in this case

$$[L(v)](s) = \min \left\{ \sum_{s' \in S} \mu_\alpha^s(s') \cdot v(s') \right\},$$

Therefore  $[L'(v)](s) = [L(v)](s)$ , and the proof of the theorem in this case is complete.

- *Case (III)*: Assume  $s \in G$ . Here in addition to  $g(s) = 0$ , we have  $c(s, \alpha) = 0$ , for each action  $\alpha$ , and  $\mathbf{P}(s, \alpha, s') = 0$ , for each  $\alpha \in Act(s)$  and  $s' \in S$ . Therefore

$$[L'(v)](s) = \min_{\alpha \in Act(s)} \left\{ c(s, \alpha) + \sum_{s' \in S \setminus G} \mathbf{P}(s, \alpha, s') \cdot v(s') \right\} = 0 = [L(v)](s).$$

Now the proof is complete.

**Proof of Theorem 4.** We show that

$$L(eR^{min}(s, \Diamond G)) = eR^{min}(s, \Diamond G) \quad (\text{B.7})$$

for all  $s \in S$ . To this aim, we distinguish three cases which are  $s \in DS \setminus G$ ,  $s \in PS \setminus G$ , and  $s \in G$ .

- in case of  $s \in DS \setminus G$ , the left-hand side of (B.7) is:

$$L(eR^{min}(s, \Diamond G)) = \rho(s) \times d_s + eR^{min}(t, \Diamond G), \quad (\text{B.8})$$

where  $\rho(s)$  is the reward of staying in  $s$ . This reward is deterministic. On the other hand,

$$eR^{min}(t, \Diamond G) = \inf_D \mathbb{E}_{t,D}(R_G) = \inf_D \sum_{Paths} R_G(\pi) \cdot Pr_{t,D}(\pi) \quad (\text{B.9})$$

Combining (B.8) and (B.9), we have

$$\begin{aligned} L(eR^{min}(s, \Diamond G)) &= \rho(s) \times d_s + \inf_D \sum_{Paths} R_G(\pi) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (R_G(\pi) + \rho(s) \times d_s) \cdot Pr_{t,D}(\pi) \\ &= \inf_D \sum_{Paths} (R_G(\pi)) \cdot Pr_{s,D}(\pi) \\ &= eR^{min}(s, \Diamond G) \end{aligned} \quad (\text{B.10})$$

Note that in (B.10), in the second line, paths, start from  $t$ , whereas, in the third line, paths start from  $s$ .

- in case of  $s \in PS \setminus G$  there is:

$$\begin{aligned}
eR^{min}(s, \diamond G) &= \inf_D \mathbb{E}_{s,D}(R_G) = \inf_D \sum_{Paths} R_G(\pi) \cdot Pr_{s,D}(\pi) \\
&= \inf_D \sum_{s \xrightarrow{\alpha, \mu, 0} t} D(s)(\alpha) \cdot \mathbb{E}_{t,D(s \xrightarrow{\alpha, \mu, 0} \cdot)}(R_G) + r(s, \alpha) \\
&= \inf_D \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D(s \xrightarrow{\alpha, \mu, 0} \cdot)}(R_G) + r(s, \alpha) \\
&= \inf_D \min_{s \xrightarrow{\alpha} \mu_\alpha^s} r(s, \alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D(s \xrightarrow{\alpha, \mu, 0} \cdot)}(R_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \inf_D r(s, \alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D(s \xrightarrow{\alpha, \mu, 0} \cdot)}(R_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} \inf_D r(s, \alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot \mathbb{E}_{t,D}(R_G) \\
&= \min_{s \xrightarrow{\alpha} \mu_\alpha^s} r(s, \alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot eR^{min}(s, \diamond G) \\
&= \min_{\alpha \in Act(s)} r(s, \alpha) + \sum_{t \in S} \mu_\alpha^s(t) \cdot eR^{min}(s, \diamond G) \\
&= L(eR^{min}(s, \diamond G))
\end{aligned}$$

- in case of  $s \in G$ , based on the definition there is  $eR^{min}(s, \diamond G) = \inf_D \sum_{Paths} R_G(\pi) \cdot$

$Pr_{s,D}(\pi) = 0$ , which is the same as the value of the Bellman operator for goal states.





School of Computer Science  
Reykjavík University  
Menntavegur 1  
101 Reykjavík, Iceland  
Tel. +354 599 6200  
Fax +354 599 6201  
[www.ru.is](http://www.ru.is)