Technische Universiteit Eindhoven
University of Technology
Where innovation starts

A theory of executability : with a focus on the expressivity of process calculi

Yang, F.

Accepted/In press: 11/06/2018

Document Version
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):
Yang, F. (2018). A theory of executability : with a focus on the expressivity of process calcului Eindhoven: Technische Universiteit Eindhoven

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 07. Jun. 2018

A Theory of Executability
with a Focus on the Expressivity of Process Calculi

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 11 juni 2018 om 11:00 uur

door

Fei Yang

geboren te Jiangsu, China

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	prof.dr. J.J. Lukkien
1^{e} promotor:	prof.dr. J.C.M. Baeten
2^{e} promotor:	prof.dr.ir. J.F. Groote
copromotor:	dr. S.P. Luttik
leden:	prof.dr. J.A. Bergstra (Universiteit van Amsterdam)
	prof.dr.hab. M. Bojańczyk (Warsaw University)
	prof.dr. W.J. Fokkink

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

A Theory of Executability
with a Focus on the Expressivity of Process Calculi

Fei Yang

Copyright © 2018 by Fei Yang. All Rights Reserved.

A catalogue record is available from the Eindhoven University of Technology Library.

ISBN 978-90-386-4532-2

IPA Dissertation Series 2018-11

Cover design by Fei Yang

Printed by Gildeprint

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The author was employed at the Eindhoven University of Technology and sponsored by CSC (Chinese Scholarship Council).

Contents

Contents i

Figures v

Tables vii

Abstract ix

Preface xi

1 Introduction 1
- 1.1 Background . . . 1
  - 1.1.1 Computability Theory . . . 1
  - 1.1.2 Concurrency Theory . . . 3
- 1.2 A Theory of Executability . . . 4
- 1.3 Thesis Outline . . . 6

2 Preliminaries 9
- 2.1 A Characterisation of Discrete-event Behaviour . . . 9
  - 2.1.1 Labelled Transition Systems . . . 9
  - 2.1.2 Behavioural Equivalences . . . 10
  - 2.1.3 Congruence . . . 12
  - 2.1.4 Bisimulation up to . . . 13
- 2.2 Reactive Turing Machines . . . 15
  - 2.2.1 Definition and Semantics . . . 15
  - 2.2.2 Parallel Composition . . . 17
- 2.3 Executable Behaviours . . . 20
- 2.4 Divergence . . . 22

2.4.1 Enumeration with Divergence 22
2.4.2 Unbounded Branching 24
2.5 A Framework of Expressivity 26

3 Interactive Computation 29
3.1 Interactive Turing Machines and ω-Translation 30
3.2 Executability of Interactive Turing Machines 33
3.3 Executable ω-Translations 35
3.4 Advice 41
3.5 Remarks 46

4 Sequential Composition and Intermediate Termination 49
4.1 TCP and Variations of TCP 52
4.1.1 TSP 52
4.1.2 TCP 54
4.1.3 TCP with Non-regular Iterators 55
4.2 Transparency 56
4.3 A Revised Semantics of the Sequential Composition Operator 60
4.4 Context-free Processes and Pushdown Process 68
4.5 Executability in the Context of Termination 76
4.6 Remarks 87

5 RTM and the π-Calculus 89
5.1 The π-Calculus 91
5.1.1 Syntax 91
5.1.2 Structural Operational Semantics 92
5.1.3 Compatibility 94
5.2 Reactively Turing Powerfulness of the π-Calculus 96
5.2.1 Tape 97
5.2.2 Finite Control 101
5.3 Executability of Finite π-Calculus 105
5.3.1 A Gap Between RTMs and the π-Calculus 105
5.3.2 Restricting the π-Calculus 106
5.4 Remarks 110

6 Nominal Executability 113
6.1 Infinitary Reactive Turing Machines 114
6.1.1 Infinitely Many States and Data Symbols 115
6.1.2 Infinitary Reactive Turing Machines 116

6.2 Sets with Atoms 119
6.2.1 Equality Atoms 119
6.2.2 Legality and Orbit-finiteness 120
6.2.3 Definability 122
6.3 Reactive Turing Machines with Atoms 123
6.4 Nominal Executability of the π-Calculus 125
6.5 Negative Result on mCRL2 132
6.5.1 LTSs with Atoms 133
6.5.2 mCRL2 133
6.6 Remarks 134

7 Conclusion **139**
7.1 Robustness 139
7.2 Comparison 140
7.3 Expressivity 140
7.4 Future Work 141

Bibliography **145**

Index **153**

Summary **157**

Curriculum Vitae **161**

IPA Dissertation Series **162**

Figures

2.1 The proof of Lemma 2.9 14
2.2 An example of RTM 18
2.3 An RTM that enumerates and sends the string 1♯11♯111♯... 19
2.4 The transition system T_1 23
2.5 A framework of expressivity 28

3.1 A model of interactive computation 31
3.2 Interactivess is necessary for ω-translation 38

4.1 A transition system with unbounded branching behaviour 57
4.2 The transition system of a half counter 58
4.3 An implementation of the half counter in TCP$^\sharp$ 59
4.4 The transition system of an always terminating half counter 59
4.5 A failed implementation of the always terminating half counter in TCP$^\sharp$ 60
4.6 A transition system in the revised semantics 63
4.7 A PDA to simulate the process in Figure 4.6 71
4.8 An implementation of the always terminating half counter in TCP$^\sharp$ . . 77

5.1 An example to illustrate the link mobility of the π-calculus 95
5.2 Specification of an RTM utilizing the linking structure of the π-calculus 97
5.3 Bisimulation relation between $M_{s,\delta_L,\check{d}\delta_R}$ and $(s, \delta_L, \check{d}\delta_R)$ 104
5.4 A π-calculus process with unbounded branching 109

6.1 A transition system with infinitely many distinct labels 115
6.2 Bisimulation relation in the proof of Theorem 6.5 118
6.3 Enumerating odd numbers with an RTM$^\infty$ 124
6.4 A labelled transition system with infinitely many orbits 125

6.5 Simulation of the transitions from a π-term 131
6.6 A hierarchy of executability 135

Tables

4.1 The operational semantics of TSP 53
4.2 The operational semantics of parallel composition in TCP 54
4.3 The operational semantics of nesting and iteration 56
4.4 The revised semantics of sequential composition 61
4.5 The revised semantics of iteration and nesting 69

5.1 Structural operational semantics of the π-calculus 93

Abstract

Computability and concurrency are two fundamental research areas in theoretical computer science. Computability theory focuses on the aspect of computing algorithmically and concurrency theory focuses on modeling systems in a concurrent setting. Classical computability theory lacks a facility to model a system with interaction, whereas such a facility is presented in most process calculi from concurrency theory. An integration of the two theories could provide a suitable solution to model and analyze systems with interaction. Executability theory is proposed to make such an integration on the basis of Reactive Turing Machines. However, many major differences between the two theories lead to a lot of difficulties in the process of integration. This thesis attempts to overcome some of the difficulties.

This thesis provides some evidence for the robustness of executability theory by comparing Reactive Turing Machines to another interactive computation model. Moreover, this thesis also makes a comparison between a pushdown process, which is a notion induced from computability theory, and a context-free process, which is a notion from concurrency theory, by analysing and revising the operational semantics of process calculi. In addition, this thesis applies executability theory to establish a framework to evaluate the expressivity of process calculi. Several case studies are presented on this issue, in which some variants of the process calculus TCP, the π-calculus and mCRL2 are mentioned. Finally, this thesis proposes a nominal extension of executability theory, which allows an application of the theory to a more general setting.

Preface

I started to study concurrency theory in 2010 at Shanghai Jiao Tong University, benefiting from an application for the BASICS lab for my summer internship programme. In that summer, I read the book "Communication and Concurrency" of Robin Milner. I struggled a lot since I was reading it all by myself. Fortunately, after two months, I was able to answer some of the basic questions in concurrency theory. After my undergraduate programme, I decided to start my master programme in the same group. I was supervised by Prof. Yuxi Fu on a project to study algorithms to check equivalences for process calculi. During the same period, I worked as a teaching assistant on three courses: computability, discrete mathematics and algorithm. Those experiences as a teaching assistant helped me to gain some basic knowledge about computability.

In 2013, I received an email from Dr. Bas Luttik and Prof. Jos Baeten about the project named "A Theory of Executability", which aims at an integrated theory to bring together concurrency and computability. This email eventually brought me to the Netherlands to work on this project at Eindhoven University of Technology since I felt that it perfectly fits my background. I started this PhD project in April, 2014, and spent four years on it. During the initial phase, I was given a copy of the PhD thesis of Paul van Tilburg on the integration of process theory and automata theory. I learned most of the basic knowledge about process theory and Reactive Turing Machines from that thesis, which strengthened my confidence in working on this project.

After I acquired some knowledge on the subject of executability, I started to develop my own research questions with Bas. We looked for many different models on interactive computation and process calculi. Some comparisons had been conducted between those models and ours, but we were not yet satisfied since the π-calculus cannot be perfectly fitted into our theory. We worked on an extension of executability theory based on a nominal variant of Reactive Turing Machines. This storyline of research brought us to a so called theory of nominal executability.

Meanwhile, I also received some research questions from Jos Baeten. I assisted him in a course about models of computation at University of Amsterdam. We dis-

cussed several questions on the integration of process calculi and pushdown automata. We could not reach desired results because of a phenomenon called transparency. We resolved the dilemma by an idea of a revision of the semantics of the sequential composition operator. This work finally became a part of this thesis.

I wrote this thesis out of my research results in executability theory. Although there are still many unanswered questions, unproved theorems and unestablished results left, I hope this thesis provides an overview of our theory of executability and inspires us to deliver further results on our theory of executability.

Acknowledgements

Every PhD candidate writes this part to thank people and organizations who helped them to finish the thesis; so do I.

As a beginning of my list, I want to thank my home country, China. She gave me a chance to receive my education and provided me a peaceful land to grow up. My PhD project was supported by the Chinese Scholarship Council.

I want to thank my supervisor, my promotor, Jos Baeten for accepting me as PhD student and allowing me to work on executability theory. He provided me a lot of useful instructions throughout the past four years. The discussions with Jos were always very pleasant. Jos also gave me a chance to assist his course in Amsterdam, which broader my sights in concurrency theory. He is a wonderful director both in my research and in my career.

I also want to thank my second promotor Jan Friso Groote, who is the group leader of the FSA section. Jan Friso is very enthusiastic about solving difficult problems, and such an enthusiasm encouraged me to make attempts on many difficult problems. As a group leader, Jan Friso asked me to organize the FSA colloquium on every Thursday, which provided me a lot of experience in organizing academic events. Finally, he gave me a lot of comments and suggestions on my thesis.

I would like to give my special thanks to my copromotor, daily supervisor, Bas Luttik. Bas started my PhD project. Many people told me that Bas is one of the best supervisors for a PhD student, and I agree with that. Bas is always so nice to me. I enjoy the discussions with him a lot. He is always interested in listening to every detail of a proof. He encouraged me when I was laking motivation and helped me to organise myself. He spent so much effort to improve my English. I appreciate every comment he made. Moreover, he is also a very good friend in life. I shall never forget his help.

My thanks also goes to Paul van Tilburg, who worked with Jos and Bas previously. His work became a foundation of mine, and eventually became the current version of our theory of executability.

My thanks also goes to the reading committee: Jan Bergstra, Wan Fokkink and Mikołaj Bojańczyk. I give sincere gratitude to them for their efforts in reading, checking and responding on my thesis. Jan Bergstra is an expert in process algebra, his comments are extremely useful in the part about the sequential composition operator. Wan Fokkink gave me a very detailed feedback which helped me to make a lot of improvements. Moreover, he is a great audience member in our FSA colloquium. Mikołaj Bojańczyk is an expert on nominal sets. He already gave me some suggestions on this subject when we met at a conference in 2016. This time, he also checked the correctness of my application on nominal sets.

I had a very pleasant time working in the MDSE group. I want to thank all my colleagues for their supports, collaborations and joyful discussions. I have very nice officemates during the past four years. I would like to thank Mohamoud Talebi for sitting beside me for four years, Omar Alzuhaibi for the interesting discussions about Dutch, Rodin Aarssen for having beer together with me, Sarmen Keshishzadeh for his instructions during the initial period of my PhD, and Sjoerd Cranen for showing me his website. As a Chinese student, it was fantastic to have some Chinese colleagues in the group. Danna Zhang, Yaping Luo and Yuexu Chen are always very nice to me. I really appreciate their instructions on my research, on how to live in Eindhoven and on how to find a girlfriend. I also want to give my thanks especially to our secretaries. I would like to thank Margje Mommers-Lenders for ordering a room and a lunch for the colloquium every week, and Tineke van den Bosch-Zaat for initializing me in the group. I also appreciate all the other previous and current colleagues: Mark van den Brand, Erik de Vink, Julien Schmaltz, Hans Zantema, Tim Willemse, Anton Wijs, Wieger Wesselink, Dragan Bosnacki, Herman Geuvers, Alexander Serebrenik, Loek Cleophas, Thomas Neele, Sander de Putter, Alexandar Fedotov, Ana-Maria Şutîi, Neda Noroozi, Yanja Dajsuren, Ulyana Tikhonova, Weslley Silva Torres, Felipe Ebert, Önder Babur, Miguel Botto Tobar, Mauricio Verano Merino, Kousar Aslam, Priyanka Karkhanis, Sangeeth Kochanthara, Raquel Alvarez Ramirez, Arash Khabbaz Saberi, Josh Mengerink, Bogdan Vasilescu, Maciej Gazda, Sebastiaan Joosten, Gerard Zwaan, Tom Verhoeff, Ruurd Kuiper, Kees Huizing and others.

During my PhD life, I also enjoyed a lot to discuss with people from other institutes. I would like to thank Rob van Glabbeek for his instructions on operational semantics, Petr Jancar for sharing his knowledge on algorithms on pushdown automata, Matias David Lee for talking with me about his project and his life in Argentina, Joshua Moerman for the discussion about nominal sets, Soichiro Hidaka and Tao Zan for the discussions on bidirectional transformations, Qiang Yin, Mingzhang Huang, Hao Huang and Jianxin Xue for the discussion on my research articles.

I am also very grateful for the people in the BASICS lab at Shanghai Jiao Tong University, who taught me a lot during my master programme, and gave me suggestions

after I came to the Netherlands. Especially, I would like to thank Yuxi Fu for his instructions on my research topic. My thanks also goes to Xiaoju Dong, Huan Long, Yijia Chen, Yuxin Deng, Guoqiang Li and Xiaojuan Cai for their helpful suggestions in my life and in my career.

I spent four years in the city of Eindhoven. I met a lot of friends here. I joined the Association of Chinese Students and Scholars and worked in this association for four years. I would like to express my appreciation to all the people in this association who gave me a wonderful experience. My thanks goes to Yawen Wang, Mei Zhang, Jiakun Gong, Bojia He, Cong Wang, Xiong Deng, Hesheng Bao, Wei Huang, Yixiao Qiao, Yixiao Wang, Zhihao Wu and others. I lived in Pisanostraat for four years, and I had a lot of amazing neighbors. I want to thank Huatian Wang, Bitao Pan, Baisong Liu, Jiachun Chai, Qi Zhang, Zhong Li and Lihua Zhang for your help in my life. I met many good friends in the mathematics and computer science department, I would like to thank Guangming Li, Jianpeng Zhang, Cong Liu, Long Cheng, Qingzhi Liu, Jingyue Cao and Yulong Pei for the discussions, lunches and other events. I would like to give my special thanks to Liangliang Lin for watching movies with me, Bao Li and Feng Hong for travelling together, Hefeng Zhou, Haiyu Wang, Qiang Pan for hot pots, Wenjie Bai for haircuts, and Xiaotong Qu for basketball events.

I reserved the last paragraph for my beloved family members. I would like to thank my parents, Guoxian Yang and Zhi Yin. I can never finish my PhD project without their supports. I am also very gratitude for my girlfriend, Ying Huang, her encouragement helped me to move on until the final period of my PhD life.

Fei Yang
Eindhoven, May 2018

Chapter 1

Introduction

The theory of executability that we shall discuss in this thesis is an integration of computability theory and concurrency theory. Based on a reactive version of the Turing Machine, our theory of executability aims to take advantage of both computability theory and complexity theory and yields a method to evaluate the expressivity of process calculi. This thesis addresses four aspects relevant for the theory of executability based on Reactive Turing Machines: the robustness of the theory of executability based on Reactive Turing Machines, a revision on the semantics of sequential composition, the expressivity of the π-calculus, and an extension of a theory of executability.

1.1 Background

Our theory of executability is based on the research in two areas of theoretical computer science, computability theory and concurrency theory. In this section, we briefly introduce computability theory and concurrency theory in order to give an intuition where our study stands in a bigger picture.

1.1.1 Computability Theory

The study of *computability theory* started in the 1930s and became the foundation of theoretical computer science. The basic question in computability theory asks which functions are computable. In classical computability theory, the notion of a computable function on natural numbers characterizes the capability of a computing system. The *Turing Machine* [80] was introduced by Turing and was accepted as one of the models

for computable functions. Many other models of computation, for instance, Church's *λ-calculus* [31], Kleene's *recursive functions* [56], Shepherdson and Sturgis' *Random Access Machines* [78], etc., have turned out to be equivalent to Turing machines. The above equivalences between models of computations give evidence for the famous *Church-Turing thesis*, which rephrases the above equivalence as: every algorithmically calculable function is a computable function.

A Turing machine mathematically models a machine that uses a tape to a compute function. The machine has a finite set of *control states*. A finite set of *tape symbols* is defined which the machine can read and write. The symbol on a tape cell is manipulated using a *tape head*. The machine contains a program represented as a finite automaton, formally defined by a *transition*. Informally, a Turing machine consists of:

1. A finite alphabet of tape symbols which contains a special symbol for the *blank tape cell*.

2. A *tape* divided into consecutive cells. Each tape cell contains a tape symbol. The tape is assumed to be extendable infinitely to the left and to the right.

3. A *tape head* that can read and write symbols in a certain tape cell, and can move to the left or to the right.

4. A finite set of *control states*.

5. A finite set of *instructions* or *transitions*, which tell the machine in a certain state and with a certain tape symbol under the tape head, which symbol it should write in the tape cell, whether it should move the tape head one position to the left or to the right, and what the next control state should be.

6. An *initial state* and a set of *final states* or *accepting states*.

A Turing machine can be used to model the procedure of computing a function on natural numbers with a machine. We use the notion of a configuration, which consists of a control state and a tape instance, to characterise the state of a Turing machine. A computation of a Turing machine usually refers to a sequence of transitions made by the Turing machine after its initial configuration in which a sequence of symbols is written on the tape as the input in the initial state; and the sequence ends up in a final state with a sequence of symbols written on the tape as the output. The Turing machine divides functions into two classes, namely, *computable functions* and *uncomputable functions*. The above distinction inspires the study of computability theory, in which many subjects were developed. For instance, in *complexity theory*, computable functions are categorised into different classes of complexity; and in the *arithmetical hierarchy*, uncomputable functions are categorised into different levels of hardness.

1.1.2 Concurrency Theory

The classical theory of computability lacks a facility to analyze systems in a concurrent setting. For example, an aeroplane has an autopilot system consisting of computers, sensors, radars, engines, cables, etc. It is difficult to analyze such a system by computability theory, e.g., by modeling it as a Turing machine. The system not only uses computers to compute, it also uses sensors and radars to gather information, and sends commands to engines through cables to fly the aeroplane. Not only computation, but also interaction happens during the execution of such a system.

Concurrency theory studies models for concurrency and communication that may specify and analyze a system such as an aeroplane. The study of concurrency theory started from Dijkstra's paper about a mutual exclusion problem [33]. Then Petri proposed his *Petri Net* [74] as a mathematical modeling language for discrete event systems. Later, some process calculi were invented to specify the behaviour of concurrent systems, for example, Milner's *CCS* [67] Bergstra and Klop's *ACP* [19] and Hoare's *CSP* [53]. With the development of concurrency theory, many process calculi were defined for various purposes, such as the π*-calculus* [69], *TCP* [4] and *mCRL2* [50].

The behaviour of a process specified in a certain process calculus can be characterised by a labelled transition system. Structural operational semantics was introduced as an approach to associate with every process a labelled transition system [1]. Various types of behavioural equivalences were also introduced to analyse the systems specified in labelled transition system semantics, for instance, Park's strong bisimulation [72], Milner's weak bisimulation [67] and van Glabbeek and Weijland's branching bisimulation [43]. An inventory of behavioural equivalences is made in van Glabbeek's paper culminating in the so-called linear time - branching time spectrum II [40]. In this thesis, we pay special attention to the finest one in van Glabbeek's article, which is referred to as divergence-preserving branching bisimulation.

There are some major differences between computability theory and concurrency theory. Firstly, computability theory considers functions on natural numbers and uses language equivalence to measure the equivalence between Turing machines and between automata, whereas concurrency theory considers labelled transition systems and uses other behavioural equivalences such as bisimulation. Secondly, concurrency theory involves interaction which is not modeled in computability theory. Thirdly, the result of the computation of a function is deterministic with respect to an input, but the behavior of a concurrent system could be non-deterministic. Finally, from the perspective of concurrency theory, there is a notion of termination in computability theory, which is often absent in models of concurrency theory.

1.2 A Theory of Executability

Both computability theory and concurrency theory have limitations. Computability theory lacks a facility to model interaction whereas interaction is a fundamental element of almost every computing system. Concurrency theory would benefit from a concurrent version of the Church-Turing thesis to evaluate the expressivity of process calculi. The reason is that we could use the thesis to tell whether a process calculus is expressive enough to specify every behaviour that could be executed by a computing system or whether every behaviour specified in a process calculus could be executed by a computing system. An integration of the two theories could be a good solution to the questions above. Moreover, it would give evidence of a concurrent version of the Church-Turing thesis. The thesis could be phrased as "every effectively executable behaviour is an executable transition system", where we use effective executable behaviour to refer to the behaviour that could be executed by some machine with a facility to interact, and we use an executable transition system to refer to a mathematical characterisation of transition systems that could be executed by an interactive variant of the Turing machine.

The concurrent version of the Church-Turing thesis should be based on an interactive model of computation. There are several interactive variants of Turing machines in the literature.

1. In [45], Goldin, Smolka, Attie and Sonderegger presented their *Persistent Turing Machines (PTMs)* and the associated notion of an interactive transition system. The notion of PTMs aims to capture the intuitive notion of *sequential interactive computation* by embedding a facility of interaction in every step of a computation.

2. In [60], van Leeuwen and Wiedermann modeled interactive computations as stream translations, and they introduced *Interactive Turing Machines (ITMs)* to characterise the class of interactively computable stream translations. Moreover, they introduce a notion of interactive Turing machines with advice. ITMs with advice use an advice function as an external computation resource and such a utility enables the characterisation of an evolving system such as the Internet.

3. In [13], Baeten, Luttik and van Tilburg introduced a notion of executability based on *Reactive Turing Machines (RTMs)* as an extension of the Turing machine in a concurrent setting [13], see also [11, 12, 79].

In this thesis, we focus on the third variant. The purpose of Reactive Turing machines in executability theory is the same as that of Turing machines in computability

theory. Every Turing machine computes a function whereas every Reactive Turing Machine defines a transition system. A transition system associated with a Reactive Turing Machine is called executable. Since the semantics of transition systems is parameterised by the choice of behavioural equivalences in concurrency theory, the notion of executability is also parameterised by the choice of behavioural equivalences.

The idea to study a theory of executability was initialized in [7]. In [13], the notion of executable transition systems was introduced, the existence of a universal Turing machine was proved, the expressivity of a variant of TCP was studied with respect to executable transition systems, and the relationship between Reactive Turing Machines and persistent Turing machines was investigated. Following the work of Baeten, Luttik and van Tilburg, we continue the research of a theory of executability on the basis of Reactive Turing Machines.

In this thesis, we address four topics.

1. We study the *robustness* of Reactive Turing Machines as a means to define which behaviour is executable. Robustness serves to validate the choice of a computational model. For instance, the robustness of the Turing Machines is based on the equivalence between Turing Machines and other models such as λ-calculus and recursive functions. We provide evidence to show that a Reactive Turing Machine is a proper model as a basis for the formalism of a theory of executability by comparing it with other models that characterise interactive computation. In the context of robustness, it was already shown in [13] that RTMs can simulate PTMs. In this thesis, we consider van Leeuwen and Wiedermann's theory of interactive computation. They propose a notion of an Interactive Turing Machine (ITM), and characterise interactive computation in terms of stream translations that can be achieved by some ITMs. To establish the robustness of RTMs with respect to ITMs, on the one hand, we show that ITMs naturally give rise to transition systems that can be executed by RTMs, and on the other hand, we need to show that RTMs can be used to compute at least the same set of stream translations as ITMs.

2. We study the *integration* of computability theory and concurrency theory. Two questions are considered for this topic. One is the comparison between the notions in computability theory and the notions in concurrency theory. We are interested in finding correspondences between the notions from each theory. For instance, context-free grammars and pushdown automata are important notions in computability theory or formal language theory [54]. In concurrency theory, the corresponding concepts are usually referred to as context-free processes and pushdown processes [79]. The study of the relationship between the above two notions is then naturally a research question in a theory of executability. The

other one is to find a reactively Turing powerful process calculus. Normally, a reactively Turing powerful process calculus uses recursive specifications, but sometimes recursive specifications are too complicated. We seek for some alternative way. A nesting operator introduced in [18] is considered as a suitable replacement of the recursive specification. The current semantics of the sequential composition operator has a phenomenon called transparency, which is a major obstacle to solve the two questions above. This thesis gives a revision to the semantics of the sequential composition operator which helps us to solve the two questions above.

3. We study the evaluation of the *expressivity* of process calculi using Reactive Turing Machines. We say a calculus is executable iff every transition system associated with the expression of the calculus is executable, and we say it is reactively Turing powerful iff every transition system associated with some Reactive Turing Machine is equivalent to a transition system specified by an expression in the calculus. The π-calculus [68, 77] is a popular process calculus in concurrency theory. Its expressivity with respect to RTMs is an interesting case study for the theory of executability.

4. We study the *extension* of executability theory. Many process calculi, such as the π-calculus and mCRL2 [50], use infinitely many labels in their transition system specifications. Such infinite sets of labels are not facilitated in a theory of executability. We extend the theory of executability to infinite alphabets.

1.3 Thesis Outline

The main contributions of this thesis are as follows:

1. Chapter 3 compares the notion of an interactive Turing machine of van Leeuwen and Wiedermann with our notion of a Reactive Turing Machine. We give semantics to associate with every interactive Turing machine a labelled transition system, and we show that it is executable modulo divergence-preserving branching bisimilarity. We also characterize the class of stream translations that can be done by Reactive Turing Machines, and show that it coincides with the class of stream translations done by interactive Turing machines. By analogy to interactive Turing machines with advice, we also investigate the effect of adding an advice process to Reactive Turing Machines.

   This chapter provides some evidence for the robustness of our theory of executability. This chapter is based on the following publication:

[63] B. Luttik and F. Yang. *On the Executability of Interactive Computation.* In Proceedings of 12th Conference on Computability in Europe (CiE 2016), LNCS 9709, 2016, pp.312-322, Springer, 2016.

2. Chapter 4 addresses some problems in a process calculus that combines sequential composition and successful termination. We discuss the phenomenon of transparency, which is caused by the current operational semantics of the sequential composition operator in the presence of intermediate termination. We revise the semantics and solve two problems. One is to show that every context-free process is equivalent to a pushdown process modulo strong bisimilarity; the other one is to show that a process calculus with the nesting operator is reactively Turing powerful modulo divergence-preserving branching bisimilarity without using recursion.

   This chapter compares the notions in computability theory and concurrency theory; moreover, it gives an application of the a theory of executability in evaluating the expressivity of process calculi. This chapter is based on the following publication:

   [14] J.C.M. Baeten, B. Luttik and F. Yang. *Sequential composition in the presence of intermediate termination (extended abstract).* Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, (EXPRESS/SOS 2017), EPTCS 255, 2017, pp. 1-17.

3. Chapter 5 extensively studies the π-calculus in the context of a theory of executability. We show that the π-calculus is reactively Turing powerful modulo divergence-preserving branching bisimilarity. We also prove that a restricted version of the π-calculus is executable modulo divergence-insensitive variant of branching bisimilarity.

   This chapter addresses a case study of evaluating the expressivity using the a theory of executability. This chapter is based on the following publication:

   [62] B. Luttik and F. Yang. *Executable behaviour and the π-calculus (extended abstract).* In Proceedings 8th Interaction and Concurrency Experience (ICE 2015), EPTCS 189, 2015, pp. 37-52.

4. Chapter 6 proposes a theory of nominal executability. The standard theory of executability assumes a finite alphabet, but we also consider an extension that allows an infinite alphabet. We introduce the notion of a Reactive Turing Machine with atoms. We establish a theory of nominal executability introducing the notion of nominal executable transition systems using Reactive Turing Machines

with atoms. We give a property of the class of nominally executable transition systems independent of Reactive Turing Machines with atoms. We show that every π-calculus process is nominally executable modulo branching bisimilarity, and we show that it is not the case for the calculus mCRL2. Together with the more general notion of an infinitary Reactive Turing Machine, we establish a part of the hierarchy of executability with respect to the sets we allow in the definition of Reactive Turing Machines.

This chapter extends a theory of executability to fit the study of expressivity in a more general setting. This chapter is based on the following report:

[64] B. Luttik and F. Yang. *Reactive Turing Machines with Infinite Alphabets.* CoRR,abs/1610.06552, 2016.

The thesis is concluded with some remarks and open problems in the theory of executability.

Chapter 2

Preliminaries

In this chapter, we first briefly recap the basic definitions of labelled transition systems and behavioural equivalences. Then, we introduce the notion of Reactive Turing Machines and the induced notion of executable transition systems. Several basic results and proof techniques regarding to the theory of executability are introduced. Moreover, we propose a general framework to evaluate the absolute expressivity of process calculi and other models of interactive computation.

2.1 A Characterisation of Discrete-event Behaviour

2.1.1 Labelled Transition Systems

We use transition systems to represent the behaviour of discrete-event systems. We assume a countably infinite set of *action symbols*. A transition system is parameterised by a subset $\mathcal{A}$ of the set of action symbols, denoting the observable events of a system. We shall later impose extra restrictions on $\mathcal{A}$, e.g., requiring that it be finite or have a particular structure, but for now we let $\mathcal{A}$ be just an arbitrary abstract set. We extend $\mathcal{A}$ with a special symbol τ, which intuitively denotes the event that the system performs an unobservable action. We shall abbreviate $\mathcal{A} \cup \{\tau\}$ by $\mathcal{A}_\tau$. We let a, b, c range over $\mathcal{A}_\tau$.

Definition 2.1 (labelled transition systems)**.** An $\mathcal{A}_\tau$*-labelled transition system* (LTS) is a 4-tuple $(\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$, where

1. $\mathcal{S}$ is a set of *states*;

2. $\longrightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ is an $\mathcal{A}_\tau$-labelled *transition relation*;

3. $\uparrow \in \mathcal{S}$ is the initial state; and

4. $\downarrow \subseteq \mathcal{S}$ is the set of *terminating (final) states*.

We write $s \xrightarrow{a} t$ for $(s, a, t) \in \longrightarrow$, we write $s \downarrow$ if $s \in \downarrow$, and we write $s \not\xrightarrow{a}$ if there is no a-labelled transition at all from s and $s \not\longrightarrow$ if there is no transition from s. Let $\longrightarrow$ be an $\mathcal{A}_\tau$-labelled transition relation on a set $\mathcal{S}$, and let $a \in \mathcal{A}_\tau$; we write $s \xrightarrow{(a)} t$ for the formula "$s \xrightarrow{a} t \vee (a = \tau \wedge s = t)$". Furthermore, we denote the transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow^+$ and the reflexive-transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow^*$.

In later chapters, we also use the version of transition systems that does not distinguish terminating states, that is, the $\downarrow$ component is omitted from the notion of labeled transition system as defined in 2.1 ($\downarrow = \emptyset$).

Definition 2.2 (reachability). Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a labelled transition system and let $s, t \in \mathcal{S}$. We define a *word* $w = a_1 \ldots a_n \in \mathcal{A}^*$ as a sequence of actions. We write $s \xrightarrow{w}{}^* t$ iff there exist states $s_0, \ldots, s_n \in \mathcal{S}$ such that

$$s = s_0 \longrightarrow^* \xrightarrow{a_1} \longrightarrow^* s_1 \ldots \longrightarrow^* \xrightarrow{a_n} \longrightarrow^* s_n = t \ .$$

If $s \xrightarrow{w}{}^* t$ for some $w \in \mathcal{A}^*$, then we say that t is *reachable* from s in T.

We denote the set of reachable states from a state s as follows:

$$Reach(s) = \{s' \in \mathcal{S} \mid \exists w \in \mathcal{A}^*. \, s \xrightarrow{w}{}^* s'\} \ .$$

2.1.2 Behavioural Equivalences

In concurrency theory, language equivalence is arguably too coarse as a behavioural equivalence for reactive systems since it abstracts from all moments of choice [4]. A bunch of behaviourial equivalences is used extensively in concurrency theory. We refer to [40] for a spectrum of behavioural equivalences proposed in the literature.

We first introduce the notion of strong bisimulation, originally proposed by Park in [72], extended with termination conditions. This equivalence relation does not distinguish τ-transitions from other labelled transitions.

Definition 2.3 (strong bisimilarity). Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A *strong bisimulation* is a relation $R \subseteq \mathcal{S} \times \mathcal{S}$ such that for all states $s, t \in \mathcal{S}$, $s\mathcal{R}t$ implies:

1. if $s \xrightarrow{a} s'$, then there exists $t' \in \mathcal{S}$, such that $t \xrightarrow{a} t'$, and $s'\mathcal{R}t'$;

2. if $t \xrightarrow{a} t'$, then there exists $s' \in \mathcal{S}$, such that $s \xrightarrow{a} s'$, and $s' \mathcal{R} t'$;

3. if $s \downarrow$, then $t \downarrow$; and

4. if $t \downarrow$, then $s \downarrow$.

The states s and t are *strongly bisimilar* (notation: $s \leftrightarroweq t$) iff there exists a strong bisimulation $\mathcal{R}$ such that $s \mathcal{R} t$.

Strong bisimilarity is the largest strong bisimulation on labeled transition systems, and it is denoted by $\leftrightarroweq$.

Strong bisimilarity does not take into account the intuition associated with the label τ that it stands for an unobservable internal activity. Therefore, it is too strong for some purposes. To this end, we proceed to introduce the notion of (divergence-preserving) branching bisimilarity, proposed by van Glabbeek and Weijland in [43], which does treat τ-transitions as unobservable events. Another reason for adopting (divergence-preserving) branching bisimilarity is that it is the finest behavioural equivalence in van Glabbeek's linear time - branching time spectrum [40]. We now proceed to introduce the definition of (the divergence-insensitive variant of) branching bisimilarity.

Definition 2.4 (branching bisimilarity)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A branching bisimulation is a relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ such that for all states $s, t \in \mathcal{S}$, $s \mathcal{R} t$ implies:

1. if $s \xrightarrow{a} s'$, then there exist $t', t'' \in \mathcal{S}$, such that $t \longrightarrow^* t'' \xrightarrow{(a)} t'$, $s \mathcal{R} t''$ and $s' \mathcal{R} t'$;

2. if $t \xrightarrow{a} t'$, then there exist $s', s'' \in \mathcal{S}$, such that $s \longrightarrow^* s'' \xrightarrow{(a)} s'$, $s'' \mathcal{R} t$ and $s' \mathcal{R} t'$;

3. if $s \downarrow$, then there exists t' such that $t \longrightarrow^* t'$, $s \mathcal{R} t'$ and $t' \downarrow$; and

4. if $t \downarrow$, then there exists s' such that $s \longrightarrow^* s'$, $s' \mathcal{R} t$ and $s' \downarrow$.

The states s and t are *branching bisimilar* (notation: $s \leftrightarroweq_{\mathrm{b}} t$) iff there exists a branching bisimulation $\mathcal{R}$, s.t. $s \mathcal{R} t$.

Note that branching bisimilarity is the largest branching bisimulation on labelled transition systems, and it is denoted by $\leftrightarroweq_{\mathrm{b}}$. It is also referred to as the divergence-insensitive variant of branching bisimilarity.

We shall consider both the divergence-insensitive and the divergence-preserving variants of branching bisimilarity. The divergence-preserving variant is also referred to as branching bisimilarity with explicit divergence in [40, 43]. In this thesis, we use the term *divergence-preserving branching bisimilarity*. The divergence preservation condition is defined as follows.

Definition 2.5 (divergence preservation condition)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A branching bisimulation $\mathcal{R}$ on T is *divergence-preserving* iff, for all states s and t, $s\mathcal{R}t$ implies:

1. if there exists an infinite sequence $(s_i)_{i\in\mathbb{N}}$ such that $s = s_0$, $s_i \xrightarrow{\tau} s_{i+1}$ and $s_i\mathcal{R}t$ for all $i \in \mathbb{N}$, then there exists a state t' such that $t \longrightarrow^{+} t'$ and $s_j\mathcal{R}t'$ for some $j \in \mathbb{N}$; and

2. if there exists an infinite sequence $(t_i)_{i\in\mathbb{N}}$ such that $t = t_0$, $t_i \xrightarrow{\tau} t_{i+1}$ and $s\mathcal{R}t_i$ for all $i \in \mathbb{N}$, then there exists a state s' such that $s \longrightarrow^{+} s'$ and $s'\mathcal{R}t_j$ for some $j \in \mathbb{N}$.

The states s and t are *divergence-preserving branching bisimilar* (notation: $s \leftrightarroweq_{\mathrm{b}}^{\Delta} t$) iff there exists a divergence-preserving branching bisimulation $\mathcal{R}$ such that $s\mathcal{R}t$.

The largest divergence-preserving branching bisimulation relation on labelled transition systems is divergence-preserving branching bisimilarity, denoted by $\leftrightarroweq_{\mathrm{b}}^{\Delta}$. It has been proved that branching bisimilarity is an equivalence relation on labelled transition systems [15], and so is divergence-preserving branching bisimilarity [42].

Sometimes, we leave out the termination condition for some models that do not use explicit termination, i.e., the condition is void for transition systems with $\downarrow = \emptyset$, as we will do in Chapters 3, 5 and 6.

2.1.3 Congruence

Congruence is an important property for equivalence relations on process calculi. It will be used to establish proofs of bisimilarity in Chapter 4 and Chapter 5. Moreover, in equational theory [4], a behavioural equivalence can only be axiomatized if it is a congruence.

Definition 2.6 (congruence)**.** An equivalence relation $\mathcal{R}$ on a process calculus C is called a congruence iff $s_i\mathcal{R}t_i$ for $i = 1, ..., ar(f)$ implies $f(s_1, \ldots, s_{ar(f)})\mathcal{R}f(t_1, \ldots, t_{ar(f)})$, where f is an operator of C, $ar(f)$ is the arity of f, and s_i, t_i are processes defined in C.

(Divergence-preserving) branching bisimilarity is not a congruence with respect to most process calculi. For instance, it is not a congruence for the process calculus TCP since it is not compatible with the nondeterministic choice operator [4]. In order to obtain a congruence relation, a rootedness condition needs to be introduced.

Definition 2.7 (rootedness condition)**.** A (divergence-preserving) branching bisimulation $\mathcal{R}$ on a transition system $(\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ satisfies the *rootedness* condition for a pair of states $s, t \in \mathcal{S}$, if $s\mathcal{R}t$ and

1. if $s \xrightarrow{a} s'$, then $t \xrightarrow{a} t'$ for some t' such that $s'\mathcal{R}t'$;

2. if $t \xrightarrow{a} t'$, then $s \xrightarrow{a} s'$ for some t' such that $s'\mathcal{R}t'$;

3. if $s \downarrow$, then $t \downarrow$; and

4. if $t \downarrow$, then $s \downarrow$.

States s and t are *rooted (divergence-preserving) branching bisimilar* (notation: $s \leftrightarroweq_{\text{rb}}^{(\Delta)} t$) iff there exists a divergence-preserving branching bisimulation $\mathcal{R}$ that satisfies the rootedness condition for s and t.

We can extend the above relations ($\leftrightarroweq$, $\leftrightarroweq_{\text{b}}$, $\leftrightarroweq_{\text{b}}^{\Delta}$, and $\leftrightarroweq_{\text{rb}}^{\Delta}$) to relations over two transition systems by taking the disjoint union of the two transition systems, and two transition systems are bisimilar iff their initial states are bisimilar in the union. Namely, for two transition systems $T_1 = (\mathcal{S}_1, \longrightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \longrightarrow_2, \uparrow_2, \downarrow_2)$, whenever the sets of states of T_1 and T_2 are disjoint, then one can just take the union of the two transition systems. If the sets of states of T_1 and T_2 are not disjoint, then one should first make them disjoint. We present the following pairing trick as a way to make them disjoint. We pair every state $s \in \mathcal{S}_1$ with 1 and every state $s \in \mathcal{S}_2$ with 2. We have $T_i' = (\mathcal{S}_i', \longrightarrow_i', \uparrow_i', \downarrow_i')$ for $i = 1, 2$ where $\mathcal{S}_i' = \{(s, i) \mid s \in \mathcal{S}_i\}$, $\longrightarrow_i' = \{((s, i), a, (t, i)) \mid (s, a, t) \in \longrightarrow_i\}$, $\uparrow_i' = (\uparrow_i, i)$, and $\downarrow_i' = \{(s, i) \mid s \in \downarrow_i\}$. We say $T_1 \equiv T_2$ iff there exists a behavioural equivalence $\equiv$ on $T = (\mathcal{S}_1' \cup \mathcal{S}_2', \longrightarrow_1' \cup \longrightarrow_2', \uparrow_1', \downarrow_1' \cup \downarrow_2')$ such that $\uparrow_1' \equiv \uparrow_2'$.

2.1.4 Bisimulation up to

We also introduce the notion of bisimulation up to $\leftrightarroweq_{\text{b}}$, which will be a useful tool in Chapter 4 and Chapter 5. Note that we adopt a non-symmetric bisimulation up to relation.

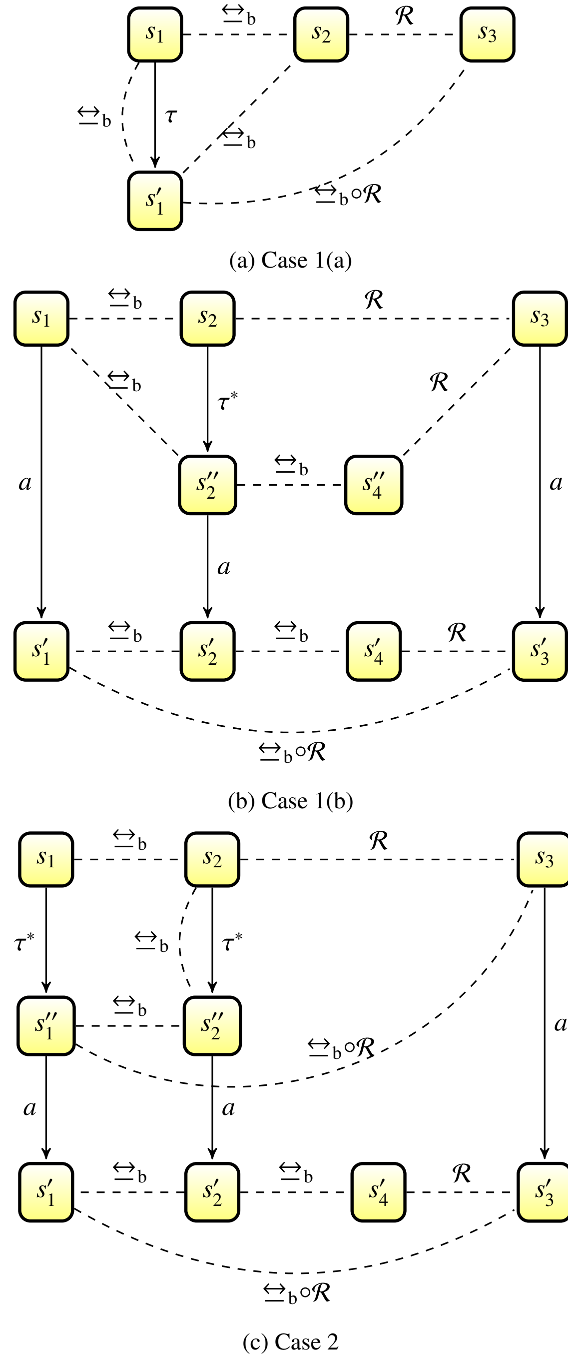
Definition 2.8 (bisimulation up to $\leftrightarroweq_{\text{b}}$)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. A relation $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is a *bisimulation up to* $\leftrightarroweq_b$ iff, whenever $s\mathcal{R}t$, then for all $a \in \mathcal{A}_\tau$:

1. if $s \longrightarrow^* s'' \xrightarrow{a} s'$, with $s \leftrightarroweq_{\text{b}} s''$ and $a \neq \tau \vee s'' \not\leftrightarroweq_{\text{b}} s'$, then there exists t' such that $t \xrightarrow{a} t'$, $s'' \leftrightarroweq_{\text{b}} \circ \mathcal{R}\ t$ and $s' \leftrightarroweq_{\text{b}} \circ \mathcal{R}\ t'$;

2. if $t \xrightarrow{a} t'$, then there exist s', s'' such that $s \longrightarrow^* s'' \xrightarrow{a} s'$, $s'' \leftrightarroweq_{\text{b}} s$ and $s' \leftrightarroweq_{\text{b}} \circ \mathcal{R}\ t'$;

3. if $s \downarrow$, then there exists t' such that $t \longrightarrow^* t'$, $s\mathcal{R}t'$ and $t' \downarrow$; and

4. if $t \downarrow$, then there exists s' such that $s \longrightarrow^* s'$, $s'\mathcal{R}t$ and $s' \downarrow$.



(a) Case 1(a)

(b) Case 1(b)

(c) Case 2

Figure 2.1: The proof of Lemma 2.9

We prove that a bisimulation up to $\leftrightarrow_{\mathrm{b}}$ is a branching bisimulation relation.

Lemma 2.9 (bisimulation up to $\leftrightarrow_{\mathrm{b}}$). *If $\mathcal{R}$ is a bisimulation up to $\leftrightarrow_b$, then $\mathcal{R} \subseteq \leftrightarrow_b$.*

Proof. It is sufficient to prove that $\leftrightarrow_{\mathrm{b}} \circ \mathcal{R}$ is a branching bisimulation, for $\leftrightarrow_{\mathrm{b}}$ is reflexive. Let $s_1, s_2, s_3 \in \mathcal{S}$ and $s_1 \leftrightarrow_{\mathrm{b}} s_2 \; \mathcal{R} \; s_3$. We establish the proof as illustrated in Figure 2.1.

1. Suppose $s_1 \xrightarrow{a} s_1'$. We distinguish two cases to show that condition 1 of Definition 2.4 is satisfied:
   (a) If $a = \tau$ and $s_1 \leftrightarrow_{\mathrm{b}} s_1'$, then $s_1' \leftrightarrow_{\mathrm{b}} s_1 \leftrightarrow_{\mathrm{b}} s_2$, so $s_1' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3$. So condition 1 is satisfied.
   (b) Otherwise, we have $a \neq \tau \vee s_1 \not\leftrightarrow_{\mathrm{b}} s_1'$. Then, since $s_1 \leftrightarrow_{\mathrm{b}} s_2$, according to Definition 2.4, there exist s_2'' and s_2' such that $s_2 \longrightarrow^* s_2'' \xrightarrow{a} s_2'$, $s_1 \leftrightarrow_{\mathrm{b}} s_2''$ and $s_1' \leftrightarrow_{\mathrm{b}} s_2'$. Note that $s_2 \leftrightarrow_{\mathrm{b}} s_1 \leftrightarrow_{\mathrm{b}} s_2''$, so we can apply condition 1 of Definition 2.8. Then we have there exist s_4'', s_4' and s_3' such that $s_3 \xrightarrow{a} s_3'$ and $s_2'' \leftrightarrow_{\mathrm{b}} s_4'' \; \mathcal{R} \; s_3$ and $s_2' \leftrightarrow_{\mathrm{b}} s_4' \; \mathcal{R} \; s_3'$. Since $s_1' \leftrightarrow_{\mathrm{b}} s_2' \leftrightarrow_{\mathrm{b}} s_4'$ and $s_4' \; \mathcal{R} \; s_3'$, it follows that $s_1' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3'$. So condition 1 is satisfied.

2. If $s_3 \xrightarrow{a} s_3'$, we show that condition 2 of Definition 2.4 is satisfied. According to Definition 2.8, there exist s_2'', s_2' and s_4' such that $s_2 \longrightarrow^* s_2'' \xrightarrow{a} s_2'$, $s_2'' \leftrightarrow_{\mathrm{b}} s_2$, $s_2' \leftrightarrow_{\mathrm{b}} s_4'$ and $s_4' \mathcal{R} s_3'$. Thus, we have $s_2' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3'$. since $s_1 \leftrightarrow_{\mathrm{b}} s_2 \leftrightarrow_{\mathrm{b}} s_2''$ and $s_2'' \xrightarrow{a} s_2'$, by Definition 2.4, there exist s_1'' and s_1' such that $s_1 \longrightarrow^* s_1'' \xrightarrow{(a)} s_1'$ with $s_1'' \leftrightarrow_{\mathrm{b}} s_2''$ and $s_1' \leftrightarrow_{\mathrm{b}} s_2'$. Since $s_2'' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3$ and $s_2' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3'$, it follows that $s_1'' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3$ and $s_1' \leftrightarrow_{\mathrm{b}} \circ \mathcal{R} \; s_3'$. So condition 2 is satisfied.

Moreover, the termination conditions of Definition 2.4 are also satisfied by Definition 2.8.

Therefore, a branching bisimulation up to $\leftrightarrow_{\mathrm{b}}$ is included in $\leftrightarrow_{\mathrm{b}}$. □

We can also modify the definition of bisimulation up to make it a relation on two transition systems. Not surprisingly, the above lemma still holds.

2.2 Reactive Turing Machines

2.2.1 Definition and Semantics

The notion of Reactive Turing Machines (RTM) [13] was introduced as an extension of Turing machines to define which behaviour is executable by a computing system

in terms of labelled transition systems. The definition of RTMs is parameterised with the set $\mathcal{A}_\tau$, which we now assume to be a finite set. Furthermore, the definition is parameterised with another finite set $\mathcal{D}$ of *data symbols*. We extend $\mathcal{D}$ with a special symbol $\Box \notin \mathcal{D}$ to denote a *blank tape cell*, and denote the set $\mathcal{D} \cup \{\Box\}$ of *tape symbols* by $\mathcal{D}_\Box$.

Definition 2.10 (Reactive Turing Machine)**.** A *Reactive Turing Machine* (RTM) is a quadruple $(Q, \mapsto, Ini, Fin)$, where

1. Q is a finite set of *states*;
2. $\mapsto \subseteq Q \times \mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\} \times Q$ is a finite collection of $(\mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\})$-labelled *transitions* (we write $s \xmapsto{a[d/e]M} t$ for $(s, d, a, e, M, t) \in \mapsto$);
3. $Ini \in Q$ is a distinguished *initial state*; and
4. $Fin \subseteq Q$ is a finite set of *final states*.

We shall use the symbol $\mathfrak{R}$ to represent the set of all RTMs. Intuitively, the meaning of a transition $s \xmapsto{a[d/e]M} t$ is that whenever the RTM is in state s, and d is the symbol currently read by the tape head, then it may execute the action a, write symbol e on the tape (replacing d), move the read/write head one position to the left or the right on the tape (depending on whether $M = L$ or $M = R$), and then end up in state t.

To formalise the intuitive understanding of the operational behaviour of RTMs, we associate with every RTM $\mathcal{M}$ an $\mathcal{A}_\tau$-labelled transition system $\mathcal{T}(\mathcal{M})$. The states of $\mathcal{T}(\mathcal{M})$ are the configurations of $\mathcal{M}$, which consist of a state from $\mathcal{S}$, its tape contents, and the position of the read/write head. We denote by $\check{\mathcal{D}}_\Box = \{\check{d} \mid d \in \mathcal{D}_\Box\}$ the set of *marked* symbols; a *tape instance* is a sequence $\delta \in (\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ such that δ contains exactly one element of the set of marked symbols $\check{\mathcal{D}}_\Box$, indicating the position of the read/write head. We adopt a convention to concisely denote an update of the placement of the tape head marker. Let δ be an element of $\mathcal{D}_\Box^*$. Then by $\delta^{\check{}}$ we denote the element of $(\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ obtained by placing the tape head marker on the right-most symbol of δ (if it exists), and $\check{\Box}$ otherwise. Similarly ${}^{\check{}}\delta$ is obtained by placing the tape head marker on the left-most symbol of δ (if it exists), and $\check{\Box}$ otherwise.

Definition 2.11 (LTSs associated with RTMs)**.** Let $\mathcal{M} = (Q, \mapsto, Ini, Fin)$ be an RTM. The *transition system* $\mathcal{T}(\mathcal{M})$ *associated with* $\mathcal{M}$ is defined as follows:

1. its set of states is the set $Conf_{\mathcal{M}} = \{(s, \delta) \mid s \in Q,\ \delta \text{ a tape instance}\}$ of all configurations of $\mathcal{M}$;

2. its transition relation $\longrightarrow \subseteq Conf_{\mathcal{M}} \times \mathcal{A}_\tau \times Conf_{\mathcal{M}}$ is a relation satisfying, for all $a \in \mathcal{A}_\tau$, $d, e \in \mathcal{D}_\Box$ and $\delta_L, \delta_R \in \mathcal{D}_\Box^*$:

   - $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L^{<} e \delta_R)$ iff $s \xmapsto{a[d/e]L} t$,
   - $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L e^{>} \delta_R)$ iff $s \xmapsto{a[d/e]R} t$;

3. its initial state is the configuration $(Ini, \check{\Box})$; and

4. its set of final states is the set $\{(s, \delta) \mid \delta \text{ a tape instance}, \ s \in Fin\}$.

We use an example from Van Tilburg [79] to illustrate the semantics of Reactive Turing Machines.

Example 2.12 (an example of RTM). Assume that $\mathcal{A} = \{c!d, c?d \mid c = \{i, o\}, d \in \mathcal{D}_\Box\}$. We use i and o to represent the input and output channel of the RTM, respectively. The label $c!d$ denotes an output of symbol d at the channel c, and $c?d$ denotes an input of symbol d at the channel c. We define an RTM $\mathcal{M} = (\mathcal{S}, \mapsto, Ini, Fin)$ as follows:

1. $Q = \{1, 2, 3, 4, 5, 6\}$;

2. $\mapsto = \{(1, \Box, i?1, 1, R, 1), (1, \Box, i?\sharp, \sharp, L, 2), (2, 1, \tau, 1, L, 2), (2, \Box, \tau, \Box, R, 3), (3, 1, \tau, 1, R, 4), (3, \sharp, \tau, \Box, L, 5), (4, 1, \tau, 1, R, 3), (4, \sharp, \tau, \Box, L, 6), (5, 1, o!1, \Box, L, 5), (5, \Box, o!\sharp, \Box, R, 1), (6, 1, \tau, \Box, L, 6), (6, \Box, \tau, \Box, R, 1)\}$;

3. $Ini = 1$; and

4. $Fin = \{1\}$.

The transitions of $\mathcal{M}$ are represented in Figure 2.2. We use a double square to mark a state as final. The RTM first inputs a string, consisting of an arbitrary number of 1s followed by a symbol $\sharp$. It stores the string and returns to the beginning of the string. Then, it starts a loop to verify whether the number of 1s is odd or even. If it is odd, then the RTM simply removes the string from the tape and returns to the initial state; otherwise, it outputs the string, removes the string from the tape, and returns to the initial state.

2.2.2 Parallel Composition

The RTM are used to model *interaction*. We equip RTMs with a simple form of communication, and define parallel composition on RTMs. We let C be a finite set of *channels* and we let RTMs communicate through the channels with a set of symbols

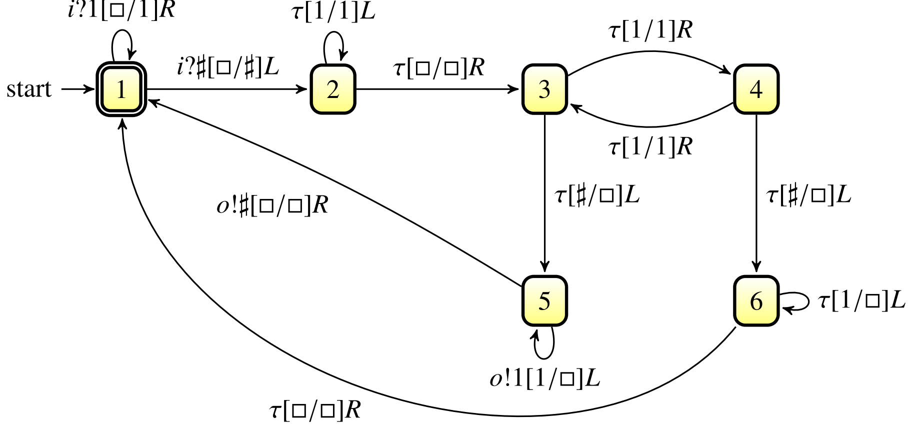


Figure 2.2: An example of RTM

$\mathcal{A}' = \{c!d, c?d \mid c \in C, d \in \mathcal{D}_\Box\}$, where $\mathcal{A}' \subseteq \mathcal{A}$. We use $c!d$ to denote an event that outputs a symbol d through the channel c, and $c?d$ to denote an event that inputs a symbol d through the channel c, respectively.

We first define a notion of parallel composition on transition systems.

Definition 2.13 (parallel composition on LTSs)**.** Let $T_1 = (\mathcal{S}_1, \longrightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \longrightarrow_2, \uparrow_2, \downarrow_2)$ be transition systems, and let $C' \subseteq C$. The *parallel composition* of T_1 and T_2 is the transition system $[T_1 \parallel T_2]_{C'} = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$, with:

1. $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$;
2. $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ iff $a \in \mathcal{A}_\tau - \{c!d, c?d \mid c \in C', d \in \mathcal{D}_\Box\}$ and either
   (a) $s \xrightarrow{a} s_1'$ and $s_2 = s_2'$, or $s_1 = s_1'$ and $s_2 \xrightarrow{a} s_2'$,
   (b) $a = \tau$ and either $s_1 \xrightarrow{c!d} s_1'$ and $s_2 \xrightarrow{c?d} s_2'$, or $s_1 \xrightarrow{c?d} s_1'$ and $s_2 \xrightarrow{c!d} s_2'$ for some $c \in C'$ and $d \in \mathcal{D}_\Box$;
3. $\uparrow = (\uparrow_1, \uparrow_2)$; and

4. $\downarrow = \{(s_1, s_2) \mid s_1 \downarrow_1 \wedge s_2 \downarrow_2\}$.

We define a similar notion of parallel composition on the transition systems associated with RTMs.

Definition 2.14 (parallel composition on RTMs)**.** Let $\mathcal{M}_1 = (Q_1, \mapsto_1, Ini_1, Fin_1)$ and $\mathcal{M}_2 = (Q_2, \mapsto_2, Ini_2, Fin_2)$ be RTMs, and let $C' \subseteq C$. The *parallel composition* of $\mathcal{M}_1$ and $\mathcal{M}_2$ is denoted by $\mathcal{M} = [\mathcal{M}_1 \parallel \mathcal{M}_2]_{C'}$. The transition system $\mathcal{T}([\mathcal{M}_1 \parallel \mathcal{M}_2]_{C'})$ associated with $\mathcal{M}$ is the parallel composition of the transition systems associated with $\mathcal{M}_1$ and $\mathcal{M}_2$, i.e., $\mathcal{T}([\mathcal{M}_1 \parallel \mathcal{M}_2]_{C'}) = [\mathcal{T}(\mathcal{M}_1) \parallel \mathcal{T}(\mathcal{M}_2)]_{C'}$.

Example 2.15 (an example of parallel composition)**.** We let $\mathcal{M}$ denote the RTM in Figure 2.2. Let $\mathcal{A}$ as in Example 2.12 and let $\mathcal{E} = (\mathcal{S}_\mathcal{E}, \mapsto_\mathcal{E}, Ini_\mathcal{E}, Fin_\mathcal{E})$ denote the RTM with the transitions in Figure 2.3 that simulates an environment defined as follows:

1. $\mathcal{S}_\mathcal{E} = \{1, 2, 3, 4\}$;

2. $\mapsto_\mathcal{E} = \{(1, \Box, \tau, 1, R, 2), (2, \Box, \tau, \Box, L, 3), (3, 1, \tau, 1, L, 3),$
   $(3, \Box, \tau, \Box, R, 4), (4, 1, i!1, 1, R, 4), (4, \Box, i!\sharp, 1, R, 2)\}$;

3. $Ini_\mathcal{E} = 1$; and

4. $Fin_\mathcal{E} = \{1\}$.

Then, the parallel composition $[\mathcal{M} \parallel \mathcal{E}]_{\{i\}}$ has the behaviour of outputting the string $11\sharp1111\sharp\ldots\sharp1^n\sharp\ldots$ ($n \geq 2$ and n is an even natural number).

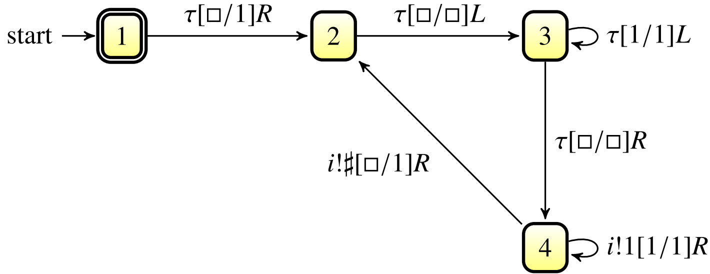


Figure 2.3: An RTM that enumerates and sends the string $1\sharp11\sharp111\sharp\ldots$

2.3 Executable Behaviours

Turing machines were introduced to define the notion of an *effectively computable function* [80]. By analogy, the notion of RTMs can be used to define a notion of *effectively executable behaviour* in terms of transition systems.

Definition 2.16 (executable LTSs)**.** A transition system T is *executable* modulo a behavioural equivalence $\equiv$ iff there exists an RTM $\mathcal{M}$ such that $T \equiv \mathcal{T}(\mathcal{M})$.

Remark 2.17. The notion of executable transition systems is parameterized by the choice of behavioural equivalence. We refer to van Glabbeek's linear time - branching time spectrum [40] for a collection of known behavioural equivalences; we only use a small fraction of the behavioural equivalences from the spectrum in this thesis. The spectrum of behavioural equivalences gives rise to a spectrum of executability. For instance, executability modulo the divergence-insensitive variant of branching bisimilarity is different from executability modulo divergence-preserving branching bisimilarity. We provide evidence (e.g., in Section 5.3) that the transition systems associated with some process calculus (e.g., the finite π-calculus) are executable modulo a coarser notion of behavioural equivalence (e.g., the divergence-insensitive variant of branching bisimilarity), but not modulo a finer notion (e.g., divergence-preserving branching bisimilarity).

Next, we introduce a characterisation of executable labelled transition systems modulo (divergence-preserving) branching bisimilarity that is independent of the notion of RTM. In order to recapitulate some results from Baeten, Luttik and van Tilburg [13], we need the following definitions, pertaining to the recursive complexity and branching degree of transition systems.

Definition 2.18 (effective LTSs)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. We say that T is *effective* iff $\mathcal{S}$ and $\mathcal{A}_\tau$ have a suitable encoding (Gödel numbering); $\longrightarrow$ and $\downarrow$ are recursively enumerable sets with respect to that encoding .

The mapping $out : \mathcal{S} \to 2^{\mathcal{A}_\tau \times \mathcal{S}}$ associates with every state its set of outgoing transitions, i.e., for all $s \in \mathcal{S}$, $out(s) = \{(a, t) \mid s \xrightarrow{a} t\}$.

Definition 2.19 (computable LTSs)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system. We say that T is *computable* iff *out* is a recursive function and $\downarrow$ is a recursive set, again with respect to some suitable encoding.

Definition 2.20 (branching degree)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system, and let $B \in \mathbb{N}$ be a natural number. We say that T has a *branching degree* bounded by B if $|out(s)| \leq B$ for all $s \in \mathcal{S}$. We call it *finitely branching* if $|out(s)|$ is finite for every state

$s \in \mathcal{S}$, and *bounded branching* if there exists a $B \in \mathbb{N}$ such that the branching degree of T is bounded by B.

The following results were established in [13].

Proposition 2.21 (LTSs associated with RTMs)**.** *The transition system $\mathcal{T}(\mathcal{M})$ associated with an RTM $\mathcal{M}$ is computable and boundedly branching.*

Theorem 2.22 (boundedly branching computable LTSs)**.** *For every finite set $\mathcal{A}_\tau$ and every boundedly branching computable $\mathcal{A}_\tau$-labelled transition system T, there exists an RTM $\mathcal{M}$ such that $T \leftrightarrow_b^\Delta \mathcal{T}(\mathcal{M})$.*

Theorem 2.23 (effective LTSs)**.** *For every finite set $\mathcal{A}_\tau$ and every effective $\mathcal{A}_\tau$-labelled transition system T there exists an RTM $\mathcal{M}$ such that $T \leftrightarrow_b \mathcal{T}(\mathcal{M})$.*

Remark 2.24. Note that in the above two theorems, we require the set of labels $\mathcal{A}_\tau$ to be finite. This is not the case for some process calculi, such as the π-calculus [69], mCRL2 [49], and the value-passing calculus [36], etc. The transition systems associated with the above process calculi sometimes use infinite sets of action labels, so they are trivially not executable. We impose some infinite sets in the definition of Reactive Turing Machines in Chapter 6 to adapt the notion of executable transition systems to infinite alphabets. Furthermore, we establish variants of the above theorems with infinite sets.

At this point, we focus on finite alphabets. We proceed to make a connection between the two theorems above. In [75], Phillips associates with every effective transition system a branching bisimilar computable transition system of which, moreover, every state has a branching degree of at most 2.

Proposition 2.25 (effective and boundedly branching LTSs)**.** *For every effective transition system T there exists a boundedly branching computable transition system T' such that $T \leftrightarrow_b T'$.*

However, we shall see in Section 2.4 that this result is no longer valid modulo divergence-preserving branching bisimilarity. Introducing divergence is unavoidable. In other words, effective transition systems and boundedly branching computable transition systems coincide modulo the divergence-insensitive variant of branching bisimilarity; but the notion of boundedly branching computable transition system is strictly included in the notion of effective transition system modulo divergence-preserving branching bisimilarity.

2.4 Divergence

In this section, we discuss some phenomena related to divergence in a transition system. In particular, we shall notice the role played by divergence in enumeration and simulation of unbounded branching behaviour. We first give the definition of divergence.

Definition 2.26 (divergence)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system, and let $s \in \mathcal{S}$. We say that T has a *divergence* if there exists an infinite sequence $(s_i)_{i \in \mathbb{N}}$ in $\mathcal{S}$ such that $s = s_0$, and $s_i \xrightarrow{\tau} s_{i+1}$ for all $i \in \mathbb{N}$.

2.4.1 Enumeration with Divergence

Divergence can be introduced to enumerate the elements of a recursively enumerable set. Concerning Proposition 2.25, every boundedly branching computable transition system is effective, and there exist effective transition systems that are neither computable nor boundedly branching. A crucial insight from Phillips' proof is that a divergence can be exploited to simulate a state with a recursively enumerable set of outgoing transitions. We use an example from [13] which is inspired by [32] to show that divergence is unavoidable.

Example 2.27 (divergence is unavoidable)**.** Assume that $\mathcal{A} = \{a, b\}$ and consider a transition system $T_1 = (\mathcal{S}_1, \longrightarrow_1, \uparrow_1, \downarrow_1)$ defined as follows:

1. $\mathcal{S} = \{s_{1,x}, t_{1,x} \mid x \in \mathbb{N}\}$;
2. $\longrightarrow_1 = \{(s_{1,x}, a, s_{1,x+1}) \mid x \in \mathbb{N}\} \cup \{(s_{1,x}, b, t_{1,x} \mid x \in \mathbb{N}\}$;
3. $\uparrow_1 = s_{1,0}$; and
4. $\downarrow_1 = \{t_{1,x} \mid \phi_x(x) \text{ converges}\}$.

The transition system is depicted in Figure 2.4. Note that ϕ_x is the x-th partial computable function under the standard enumeration of Gödel [44], see also [76].

We rephrase the analysis from [13] to show that there does not exist a computable transition system to simulate T_1 modulo divergence-preserving branching bisimilarity.

We suppose that $T_2 = (\mathcal{S}_2, \longrightarrow_2, \uparrow_2, \downarrow_2)$ is a transition system such that $T_1 \mathrel{\underline{\leftrightarrow}_b^{\Delta}} T_2$, witnessed by some divergence-preserving branching bisimulation relation $\mathcal{R}$. Now we argue that T_2 is not computable by deriving a contradiction from the assumption that it is.

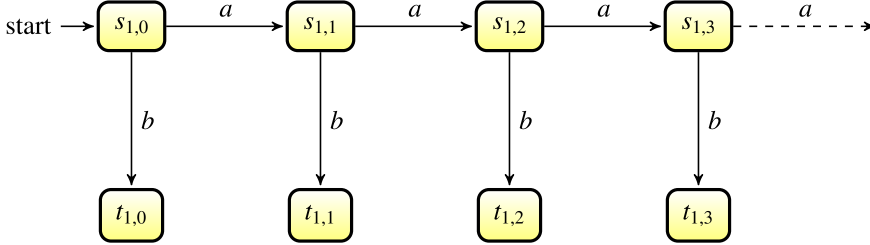


Figure 2.4: The transition system T_1

Clearly, since T_1 does not admit infinite sequences of τ-transitions, if $\mathcal{R}$ satisfies the divergence preservation condition, then T_2 does not admit infinite sequences of τ-transitions either. It follows that if $s_1 \mathcal{R} s_2$, then there exists a state s_2' in T_2 such that $s_2 \longrightarrow_2^* s_2'$, $s_1 \mathcal{R} s_2'$ and there does not exist any τ-transition from s_2'. Moreover, since T_2 is computable and does not admit infinite sequences of consecutive τ-transitions, a state s_2' satisfying the aforementioned properties is produced by the algorithm that, given a state of T_2, selects an enabled τ-transition and recurses on the target of the transition until it reaches a state in which no τ-transitions are enabled. But then we also have an algorithm that determines if $\phi_x(x)$ converges:

1. it starts from $\uparrow_2$;

2. it runs the algorithm to find a state without outgoing τ-transitions, and then it repeats the following steps x times:

   (a) it executes an a-transition to reach the resulting state;

   (b) then it runs the algorithm to find a state without outgoing τ-transitions again;

   since $\uparrow_1 \mathcal{R} \uparrow_2$, it is not hard to see that the above procedure yields a state $s_{2,x}$ in T_2 such that $s_{1,x} \mathcal{R} s_{2,x}$;

3. it executes the b-transition from the state $s_{2,x}$, and then again runs the algorithm to find a state without outgoing τ-transitions; the resulting state is $t_{2,x}$, without any outgoing transitions such that $t_{1,x} \mathcal{R} t_{2,x}$.

It follows from the definition of $\downarrow_1$ and $t_{1,x} \mathcal{R} t_{2,x}$ that $t_{2,x} \downarrow$ iff $\phi_x(x)$ converges. Therefore, we have reduced the problem of deciding whether $\phi_x(x)$ converges to deciding whether $t_{2,x} \downarrow$. Since it is undecidable if $\phi_x(x)$ converges, it follows that $\downarrow_2$ is not recursive, which contradicts the assumption that T_2 is computable.

Hence, for executable transition systems, it is unavoidable to introduce a divergence to simulate the procedure of enumeration. This also gives us some further insight in the result of Theorem 2.23 that only the divergence-insensitive variant of branching bisimilarity can be achieved.

2.4.2 Unbounded Branching

Divergence can be used to simulate the behaviour in a state with a high branching degree using states with lower branching degrees; the idea stems from [5] and is generalised in [75] to prove the statement of Proposition 2.25. We proceed to discuss a criterion to decide whether a transition system is not executable up to divergence-preserving branching bisimilarity, which is based on the notion of branching degree up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$. We will use this criterion in Section 5.3, where we are going to establish that not every finite π-calculus process is executable up to divergence-preserving branching bisimilarity.

Definition 2.28 (branching degree up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$)**.** Let $T = (\mathcal{S}, \longrightarrow, \uparrow, \downarrow)$ be a transition system and let us denote by $[s]_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}$ the equivalence class of $s \in \mathcal{S}$ modulo $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$, i.e.,

$$[s]_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}} = \{s' \in \mathcal{S} \mid s \ \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} \ s'\} \ .$$

The *branching degree* up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$ of s, denoted by $deg_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}(s)$, is defined as the cardinality of the set

$$\left\{(a, [s']_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}) \mid \exists s''.\, s \longrightarrow^{*} s'' \xrightarrow{a} s' \ \&\ s \ \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} \ s'' \ \&\ (a = \tau \implies s'' \not\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} \ s')\right\} \ .$$

The branching degree up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$ of T is the supremum of the branching degrees up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$ of all reachable states, which is defined as

$$deg_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}(T) = \sup(\{deg_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}(s) \mid s \in Reach(\uparrow)\}) \ .$$

We say that T is *boundedly branching* up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$ if there exists $B \in \mathbb{N}$, such that $deg_{\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}}(T) \leq B$, otherwise it is *unboundedly branching* up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$.

Lemma 2.29 (branching degree up to $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$)**.** *Let $T_1 = (\mathcal{S}_1, \longrightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \longrightarrow_2, \uparrow_2, \downarrow_2)$ be $\mathcal{A}_\tau$-labelled transition systems, let $s \in \mathcal{S}_1$, and $t \in \mathcal{S}_2$. If $s \ \underline{\leftrightarrow}_{b}^{\Delta} \ t$, then $deg_{\underline{\leftrightarrow}_{b}^{\Delta}}(s) = deg_{\underline{\leftrightarrow}_{b}^{\Delta}}(t)$.*

Proof. Clearly, if there exist s'', $s' \in \mathcal{S}_1$ and $a \in \mathcal{A}_\tau$ such that $s \longrightarrow_1^{*} s'' \xrightarrow{a}_1 s'$, then it is straightforward to derive from the definition of $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$ that there also exist t'', $t' \in \mathcal{S}_2$

such that $t \longrightarrow_2^* t'' \stackrel{a}{\longrightarrow}_2 t'$, and $s' \leftrightarrow_b^\Delta t'$. Conversely, if there exist $t'', t' \in \mathcal{S}_2$ and $a \in \mathcal{A}_\tau$ such that $t \longrightarrow_2^* t'' \stackrel{a}{\longrightarrow}_2 t'$, then there also exist $s'', s' \in \mathcal{S}_1$ such that $s \longrightarrow_1^* s'' \stackrel{a}{\longrightarrow}_1 s'$, and $s' \leftrightarrow_b^\Delta t'$. In the first case, if $a = \tau$, then it is required that $s'' \not\leftrightarrow_b^\Delta s'$; and in the second case, if $a = \tau$, then it is required that $t'' \not\leftrightarrow_b^\Delta t'$. Hence, there is a bijective correspondence between the sets

$$\left\{(a, [s']_{\leftrightarrow_b^\Delta}) \;\middle|\; \exists s''.\, s \longrightarrow_1^* s'' \stackrel{a}{\longrightarrow}_1 s' \;\&\; s \leftrightarrow_b^\Delta s'' \;\&\; (a = \tau \implies s'' \not\leftrightarrow_b^\Delta s')\right\}$$

and

$$\left\{(a, [t']_{\leftrightarrow_b^\Delta}) \;\middle|\; \exists t''.\, t \longrightarrow_2^* t'' \stackrel{a}{\longrightarrow}_2 t' \;\&\; t \leftrightarrow_b^\Delta t'' \;\&\; (a = \tau \implies t'' \not\leftrightarrow_b^\Delta t')\right\} .$$

It follows that $deg_{\leftrightarrow_b^\Delta}(s) = deg_{\leftrightarrow_b^\Delta}(t)$. □

By analogy to Definition 2.26, we also introduce the notion of divergence up to $\leftrightarrow_b^\Delta$.

Definition 2.30 (divergence up to $\leftrightarrow_b^\Delta$). A *divergence up to* $\leftrightarrow_b^\Delta$ in a transition system is an infinite sequence of reachable states $s_1, s_2, \ldots$ such that $s_1 \stackrel{\tau}{\longrightarrow} s_2 \stackrel{\tau}{\longrightarrow} \cdots$ and $s_i \leftrightarrow_b^\Delta s_j$ for all $i, j \in \mathbb{N}$.

The following lemma shows that, in the absence of a divergence up to $\leftrightarrow_b^\Delta$, boundedly branching transition systems are boundedly branching up to $\leftrightarrow_b^\Delta$.

Lemma 2.31 (boundedly branching up to $\leftrightarrow_b^\Delta$). *If a transition system is boundedly branching and does not have a divergence up to* $\leftrightarrow_b^\Delta$*, then it is boundedly branching up to* $\leftrightarrow_b^\Delta$.

Proof. Let T be a boundedly branching transition system and suppose that T does not have a divergence up to $\leftrightarrow_b^\Delta$. Then there exists $B \in \mathbb{N}$ such that $|out(s)| \leq B$ for all states s of T. It suffices to prove that $deg_{\leftrightarrow_b^\Delta}(s) \leq B$ for all states s of T. To this end, let s be a state of T. Since T does not have a divergence up to $\leftrightarrow_b^\Delta$, there exists t such that $s \longrightarrow^* t$, $s \leftrightarrow_b^\Delta t$ and there does not exist t' such that $t \stackrel{\tau}{\longrightarrow} t'$ and $t \leftrightarrow_b^\Delta t'$. Then

$$\begin{aligned} &\left\{\{(a, [t']_{\leftrightarrow_b^\Delta}) \;\middle|\; \exists t''.\, t \longrightarrow^* t'' \stackrel{a}{\longrightarrow} t' \;\&\; t \leftrightarrow_b^\Delta t'' \;\&\; (a = \tau \implies t'' \not\leftrightarrow_b^\Delta t')\right\} \\ = &\left\{(a, [t']_{\leftrightarrow_b^\Delta}) \;\middle|\; t \stackrel{a}{\longrightarrow} t'\right\} . \end{aligned}$$

From $s \leftrightarroweq_b^\Delta t$ it follows by Lemma 2.29 that $deg_{\leftrightarroweq_b^\Delta}(s) = deg_{\leftrightarroweq_b^\Delta}(t)$, so

$$deg_{\leftrightarroweq_b^\Delta}(s) = deg_{\leftrightarroweq_b^\Delta}(t) = |\{(a, [t']_{\leftrightarroweq_b^\Delta}) \mid t \xrightarrow{a} t'\}| \leq |\{(a, t') \mid t \xrightarrow{a} t'\}| = |out(t)| \leq B \ .$$

We conclude that T is boundedly branching up to $\leftrightarroweq_b^\Delta$. □

Thus we conclude the following theorem from Proposition 2.21, Lemma 2.29 and Lemma 2.31.

Theorem 2.32 (unboundedly branching up to $\leftrightarroweq_b^\Delta$)**.** *If a transition system T has no divergence up to $\leftrightarroweq_b^\Delta$ and is unboundedly branching up to $\leftrightarroweq_b^\Delta$, then it is not executable modulo $\leftrightarroweq_b^\Delta$.*

Proof. We suppose that T is executable modulo $\leftrightarroweq_b^\Delta$, then it follows that there exists an RTM $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarroweq_b^\Delta T$. By Proposition 2.21, $\mathcal{T}(\mathcal{M})$ is boundedly branching. As T has no divergence up to $\leftrightarroweq_b^\Delta$ and $\mathcal{T}(\mathcal{M}) \leftrightarroweq_b^\Delta$, it follows that $\mathcal{T}(\mathcal{M})$ has no divergence up to $\leftrightarroweq_b^\Delta$. By Lemma 2.31, $\mathcal{T}(\mathcal{M})$ is boundedly branching up to $\leftrightarroweq_b^\Delta$. Then from Lemma 2.29, we have that T is also boundedly branching, which contradicts our assumption. □

2.5 A Framework of Expressivity

The expressivity of process calculi has been addressed extensively in the recent decades. There are, roughly, two approaches studying the expressivity of a process calculus. One is to provide an encoding of one process calculus into another, which implies that the expressivity of the latter is at least as great as the former. Gorla presented a unified approach on the encodability of process calculi in [47], which allows us to establish separation results for process calculi. Many results on relative expressivity have been established by providing an encoding from a process calculus into another (see, e.g., [71, 79, 55], etc.).

Another approach is to compare the expressivity of a model with a standard model, i.e., a model that has a power that is equivalent to Turing machines. Milner established in [68] that the π-calculus is Turing powerful, by exhibiting an encoding of the λ-calculus [31] in the π-calculus by which every reduction in the λ-calculus is simulated by a sequence of reductions in the π-calculus. Similarly, in [28] several expressivity results for variants of CCS are obtained via an encoding of Random Access Machines, and also those results only make claims about the computational expressivity of the calculi. In [37], Fu studied the integration of computation and interaction and proposed a

minimal model of communication to evaluate the absolute expressivity of other models. Moreover, the relative expressivity of two models is compared using a notion of subbisimilarity (see, e.g., [38]).

We exploit the theory of executability as a tool to measure the absolute expressivity of models in concurrency theory. We use $\mathfrak{M}$ to denote a collection of models in concurrency theory. $\mathfrak{M}$ contains:

1. a miscellaneous collection of process calculus, e.g., CCS [67], CSP [53], ACP [19], the Theory of Communicating Processes (TCP) [4], the π-calculus [69, 77], mCRL2 [49] and the value-passing calculus [36], etc.;

2. variants of Turing machines, e.g., the persistent Turing machine [45] and the interactive Turing machine [60], etc.;

3. other Turing complete models that include a facility to model interaction, e.g. Abstract State Machines [51].

We use the notion of executable transition system to characterise the expressive power of practical computing systems. It is naturally a criterion on the expressivity of models from $\mathfrak{M}$. In the remainder of this thesis, we shall compare the expressivity of the models from $\mathfrak{M}$ and Reactive Turing Machines in terms of the transition systems associated with them. Moreover, we shall use behavioural equivalences. As we have explained in Section 2.1, an ideal result on expressivity is established modulo divergence-preserving branching bisimilarity, since it is the finest behavioural equivalence we know that abstracts from τ-transitions, and captures the moment of choices, divergence and termination. Moreover, we consider a result as a better one if it is modulo a finer notion of behavioural equivalence comparing to a result that is modulo a coarser notion of behavioural equivalence.

Sometimes, a model in the literature (e.g., the interactive Turing machines, see more details in Chapter 3) does not involve a semantics that allows us to associate with it a transition system. Then, we shall give it a proper semantics to enable the investigation of expressivity with respect to Reactive Turing Machines.

We exhibit our framework of absolute expressivity in Figure 2.5. Given a model $\mathfrak{C} \in \mathfrak{M}$, we naturally raise two questions with respect to an arbitrary behavioural equivalence $\equiv$.

1. Is $\mathfrak{C}$ *reactively Turing powerful* modulo $\equiv$?

2. Is $\mathfrak{C}$ *executable* modulo $\equiv$?

Let $\mathfrak{C} \in \mathfrak{M}$ be an arbitrary model, and let $\equiv$ be an arbitrary behavioural equivalence relation. We associate with every process $P \in \mathfrak{C}$ a transition system $\mathcal{T}(P)$. We use $\mathcal{T}(\mathfrak{R})$

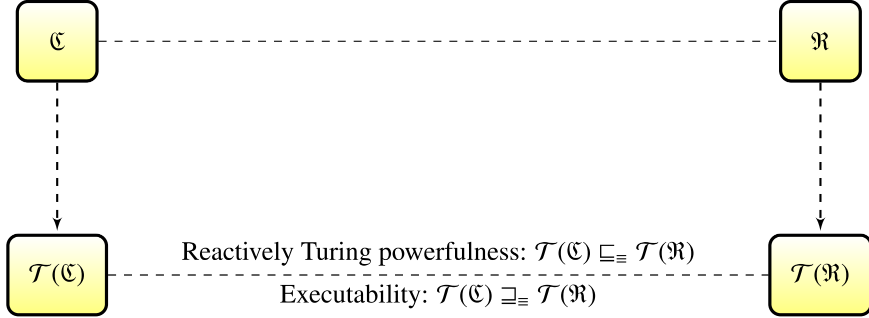


Figure 2.5: A framework of expressivity

to denote that the set of transition systems associated with all Reactive Turing Machines and use $\mathcal{T}(\mathfrak{C})$ to denote the set of transition systems associated with all processes in $\mathfrak{C}$. We give formal definitions of the above two properties. We use $\mathcal{T}(\mathfrak{C}_1) \sqsubseteq_{\equiv} \mathcal{T}(\mathfrak{C}_2)$ to denote the set of transition systems associated with the processes from $\mathfrak{C}_1 \in \mathfrak{M}$ is a subset of the set of transition systems associated with the processes from $\mathfrak{C}_2 \in \mathfrak{M}$ modulo $\equiv$.

Definition 2.33 (reactively Turing powerfulness)**.** If for every Reactive Turing Machine $\mathcal{M}$, there exists a process $P \in \mathfrak{C}$ such that $\mathcal{T}(P) \equiv \mathcal{T}(\mathcal{M})$, then we say $\mathfrak{C}$ is reactively Turing powerful modulo $\equiv$, i.e., $\mathcal{T}(\mathfrak{R}) \sqsubseteq_{\equiv} \mathcal{T}(\mathfrak{C})$.

Definition 2.34 (executability)**.** If for every process $P \in \mathfrak{C}$, there exists a Reactive Turing Machine $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \equiv \mathcal{T}(P)$, then we say $\mathfrak{C}$ is executable modulo $\equiv$, i.e., $\mathcal{T}(\mathfrak{C}) \sqsubseteq_{\equiv} \mathcal{T}(\mathfrak{R})$.

Remark 2.35. Note that apart from labelled transition systems, we also consider other semantics associated with a model $\mathfrak{C}$. For instance, in Section 3.3, we shall consider ω-translations of associated interactive Turing machines. We compromise by finding a subset of Reactive Turing Machines that is suitable for ω-translations. Therefore, we can compare the expressivity of interactive Turing machines and Reactive Turing Machines in two ways, both with respect to labelled transition systems, and with respect to ω-translations.

Chapter 3

Interactive Computation

According to the Church-Turing thesis, the classical Turing machine adequately formalises which functions from natural numbers to natural numbers are effectively computable. There is, however, a considerable semantic gap between computing the result of a function applied to a natural number and the way computing systems operate nowadays. Modern computing systems are reactive, they are in continuous interaction with their environment, and their operation is not supposed to terminate. Quite a number of extended models of computation have been proposed in recent decades to study the combination of computation and interaction (see, e.g., the collection in [46]). In this chapter we compare *Interactive Turing Machines* and *Reactive Turing Machines*.

Van Leeuwen and Wiedermann have developed a theory of interactive computation from the stance that an interactive computation can be viewed as a never-ending exchange of symbols between a component and its unpredictable interactive environment [57]. Semantically, this amounts to studying the recognition, generation and translation of infinite streams of symbols. In [58], the notion of interactive Turing machine (ITM) is put forward as a tool to formally characterise which stream translations are interactively computable.

An ITM is subsequently extended with an (uncomputable) advice mechanism in order to obtain a non-uniform machine model. An advice could be considered as an oracle that tells the machine how to evolve as time goes by. Many systems evolve, and a typical example is the Internet. The Internet changes every moment as a result of the upgrading of the programmes, the replacement of hardware, the modification of network structures, etc. Such an evolving mechanism cannot be predicted by a predefined "function", therefore, an advice that characterises the evolving of the system was introduced as a compliment to an ITM. Van Leeuwen and Wiedermann argue that

the resulting model of *interactive Turing machines with advice* is as powerful as their model of evolving finite automata, and they conclude from this, on intuitive grounds, that ITMs with advice are adequate to model evolving systems such as the Internet [82]. Moreover, in a recent article by Cabessa and Villa it is shown that ITMs with advice are as powerful as the model of interactive evolving recurrent neural networks for computing stream translations [29].

The model of interactive Turing machines focusses on capturing the computational content of sequential interactive behaviour. The included mechanism of interaction is therefore limited to achieving this goal, and does not easily generalise to more than one distributed component, nor does it allow for more fine-grained considerations of the behaviour of reactive systems. The behavioural theory of reactive systems, on the other hand, has focussed on aspects of modelling, specification and verification (see, e.g., [3]).

The aim of this chapter is to make a connection between the theory of interactive computability and the theory of reactive systems, providing a comparison of the models of ITMs and RTMs in both their semantic domains. We shall first, in Section 3.1, recapitulate the theory of interactive computability by introducing the notion of ITMs and ω-translations. Then, in Section 3.2 we present a transition-system semantics for ITMs; the transition system associated with an ITM is executable up to a fine notion of behavioural equivalence. In Section 3.3 we shall identify a subclass of RTMs that can be considered suitable for stream translation, and prove that the stream translation associated with an RTM in this subclass is interactively computable. In Section 3.4 we consider an extension of RTMs with an advice mechanism adapted from the advice mechanism considered for ITMs. RTMs with advice can execute every countable transition system, at the cost of introducing divergence in the computation. The chapter concludes by some remarks and a discussion of future work in Section 3.5.

3.1 Interactive Turing Machines and ω-Translation

In this section, we briefly introduce the theory of interactive computation proposed by van Leeuwen and Wiedermann.

In [60], van Leeuwen and Wiedermann present an analysis of interactive computation on the basis of a *component* C (thought to behave according to a deterministic program) interacting with an unpredictable *environment* E. As exhibited in Figure 3.1, they discuss the consequences of a few general postulates pertaining to the behaviour and interaction of C and E for interactive recognition, interactive generation and interactive translation. In their analysis, the component C acts as a stream transducer, transforming an infinite input stream of data symbols from $\Sigma = \{0, 1\}$ presented by E

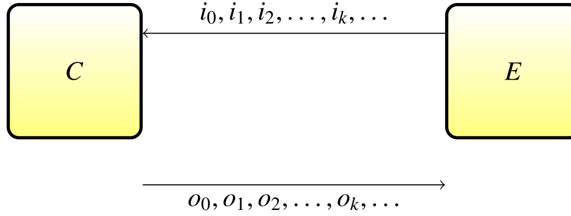


Figure 3.1: A model of interactive computation

at its input port into an infinite output stream of symbols from Σ produced at its output port. Henceforth, by an *ω-translation* we mean a mapping $\phi : \Sigma^\omega \rightarrow \Sigma^\omega$ (with Σ^ω denoting the set of streams, i.e., infinite sequences, over Σ).

Interactive computation is a step-wise process. It is not required that the environment offers a symbol in every step, nor that the component produces a symbol in every step. For the purpose of modelling components, however, it is convenient to record that nothing is offered or produced. The symbol λ is used to indicate the situation that no symbol is offered at the input port or produced at the output port, and we let $\Sigma_\lambda = \Sigma \cup \{\lambda\}$. It is assumed that when E offers a non-λ symbol in some step, then the component C produces a non-λ symbol at its output port within finitely many steps, and vice versa; this assumption is referred to as the *interactiveness* (or *finite delay*) condition in the work of van Leeuwen and Wiedermann.

In order to formally define which ω-translations are interactively computable by a computational device, van Leeuwen and Wiedermann proposed the notion of *interactive Turing machine* [58, 59]. It extends the classical notion of Turing machine with an input port and an output port, through which it exchanges an infinite, never ending stream of data symbols with its environment. Interactive Turing machines use a two-way infinite tape as memory on which they can write symbols from some presupposed set $\mathcal{D}_\square$ of *tape symbols*, not necessarily disjoint from Σ and including the special $\square$ symbol to denote an empty tape cell. Our formal definition below is adapted from [81] (but we leave out the distinction between internal and external states). Here we safely restrict the machines to work with a single tape, since multi-tape machines cannot calculate any more functions than single-tape machines [52, 66], which is also valid for interactive Turing machines.

Definition 3.1 (interactive Turing machines)**.** A *(deterministic) interactive Turing machine* (ITM) with a single work tape is a triple $\mathcal{I} = (Q, \longrightarrow_{\mathcal{I}}, q_{in})$, where

1. Q is its set of *states*;
2. $\longrightarrow_{\mathcal{I}}: Q \times \mathcal{D}_{\Box} \times \Sigma_{\lambda} \to Q \times \mathcal{D}_{\Box} \times \{L, R\} \times \Sigma_{\lambda}$ is a *transition function*; and
3. $q_{in} \in Q$ is its *initial state*.

The contents of the tape of an ITM may be represented by an element of $(\mathcal{D}_{\Box})^*$ (the set of finite sequences of symbols in $\mathcal{D}_{\Box}$). We denote by $\check{\mathcal{D}}_{\Box} = \{\check{d} \mid d \in \mathcal{D}_{\Box}\}$ the set of *marked* symbols; a *tape instance* is a sequence $\delta \in (\mathcal{D}_{\Box} \cup \check{\mathcal{D}}_{\Box})^*$ such that δ contains exactly one element of $\check{\mathcal{D}}_{\Box}$. The marker indicates the position of the tape head.

Intuitively, a transition $(q, d) \xrightarrow{i/o}_{\mathcal{I}} (q', e)$ from the transition function means that whenever the ITM is in state q, its tape head reads the symbol d, and input symbol i is offered at its input port, then it replaces the symbol d by the symbol e on its tape, moves the tape head one position in the direction of M, produces the output symbol o at its output port and then continues in state q'.

A *computation* of an ITM $\mathcal{I} = (Q, \longrightarrow_{\mathcal{I}}, q_{in})$ is an infinite sequence of transitions

$$(q_{in}, \check{\Box}) = (q_0, \delta_0) \xrightarrow{i_0/o_0}_{\mathcal{I}} (q_1, \delta_1) \xrightarrow{i_1/o_1}_{\mathcal{I}} \cdots (q_k, \delta_k) \xrightarrow{i_k/o_k}_{\mathcal{I}} \cdots \quad . \tag{3.1}$$

The *input stream* associated with the computation in (3.1) is obtained from $i_0, i_1, \ldots$ by omitting all occurrences of λ, and the *output stream* associated with the computation in (3.1) is obtained from $o_0, o_1, \ldots$ by omitting all occurrences of λ. A pair $(\vec{x}, \vec{y}) \in \Sigma^{\omega} \times \Sigma^{\omega}$ is an *interaction pair* associated with $\mathcal{I}$ if there exists a computation of $\mathcal{I}$ with $\vec{x}$ as input stream and $\vec{y}$ as output stream. The set of all interaction pairs associated with an ITM $\mathcal{I}$ is called its *interactive behaviour*. In Section 3.2 we present a more refined view on its behaviour when we associate with every ITM a transition system. A computation of the form in (3.1) is *interactive* iff, for all $k \in \mathbb{N}$, if $i_k \neq \lambda$, then there exists $\ell \geq k$ such that $o_{\ell} \neq \lambda$. The computation in (3.1) is *input-active* iff $i_k \neq \lambda$ for all $k \in \mathbb{N}$.

An ITM satisfies the *interactiveness* condition if all its computations are interactive. Clearly, if a deterministic ITM $\mathcal{I}$ satisfies the interactiveness condition, then its interactive behaviour is total, in the sense that for every $\vec{x} \in \Sigma^{\omega}$ there is at least one $\vec{y} \in \Sigma^{\omega}$ such that $(\vec{x}, \vec{y})$ is an interaction pair of $\mathcal{I}$. Note that we only consider the computations with non-empty inputs, therefore, the input-active condition is necessary. By confining our attention to the input-active computations—which, in the terminology of [60], corresponds to adopting the full environmental activity postulate—, we may then associate with every such ITM an ω-translation: we say that ITM $\mathcal{I}$ produces $\vec{y}$ on input $\vec{x}$ if $(\vec{x}, \vec{y})$ is the interaction pair associated with an input-active computation of $\mathcal{I}$.

Definition 3.2 (interactively computable ω-translation)**.** An ω-translation $\phi : \Sigma^{\omega} \to \Sigma^{\omega}$ is *interactively computable* iff there exists a deterministic ITM satisfying the interactiveness condition that produces $\phi(\vec{x})$ on input $\vec{x}$ for all $\vec{x} \in \Sigma^{\omega}$.

Van Leeuwen and Wiedermann present in [60] a characterisation of the interactively computable ω-translations by showing that they can be approximated by classically computable partial functions on finite sequences over Σ. For finite and infinite sequences $\vec{x}$ and $\vec{y}$, we write $\vec{x} \prec \vec{y}$ if $\vec{x}$ is a finite and strict prefix of $\vec{y}$, and $\vec{x} \preceq \vec{y}$ if $\vec{x} \prec \vec{y}$ or $\vec{x} = \vec{y}$. We use the following definition of monotonic functions and limit-continuous functions.

Definition 3.3 (monotonic and limit-continuous functions)**.** 1. A partial function

$$f : \Sigma^* \rightharpoonup \Sigma^*$$

is *monotonic* if for all $\vec{x}, \vec{y} \in \Sigma^*$ such that $\vec{x} \prec \vec{y}$ and $f(\vec{y})$ is defined, it holds that $f(\vec{x})$ is defined as well and $f(\vec{x}) \preceq f(\vec{y})$.

2. A partial function

$$\phi : \Sigma^\omega \rightharpoonup \Sigma^\omega$$

is called *limit-continuous* if there exists a classically computable monotonic partial function $f : \Sigma^* \rightharpoonup \Sigma^*$ such that $\phi(\lim_{k\to\infty} \vec{x}_k) = \lim_{k\to\infty} f(\vec{x}_k)$ for all strictly increasing chains $\vec{x}_1 \prec \vec{x}_2 \prec \cdots \prec \vec{x}_k \prec \cdots$ with $\vec{x}_k \in \Sigma^*$.

In [60] a criterion of the interactively computable ω-translations is presented by using limit-continuous functions.

Theorem 3.4 (interactively computable vs limit-continuality)**.** *A total ω-translation is interactively computable iff it is limit-continuous.*

3.2 Executability of Interactive Turing Machines

In this section we associate a transition system with every ITM, and then prove that it is executable modulo divergence-preserving branching bisimilarity. It is convenient to consider input and output as separate actions in the transition system associated with an ITM. We denote by $?i$ the action of inputting the symbol $i \in \Sigma$, and by $!o$ the action of outputting the symbol $o \in \Sigma$; we let $\mathcal{A}_\tau = \{?i, !o \mid i, o \in \Sigma\} \cup \{\tau\}$.

Definition 3.5 (LTSs associated with ITMs)**.** Let $\mathcal{I} = (Q, \longrightarrow_{\mathcal{I}}, q_{in})$ be an ITM. The transition system $\mathcal{T}(\mathcal{I})$ associated with $\mathcal{I}$ is defined as follows:

1. its set of states is the set $\{(s, \delta) \mid s \in Q \cup \{s_o \mid o \in \Sigma_\lambda, s \in Q\}, \delta \text{ is a tape instance}\}$;
2. its transition relation $\longrightarrow$ is the least relation satisfying, for all $s, t \in Q$, $i, o \in \Sigma_\lambda$, $d, e \in \mathcal{D}_\Box$, and $\delta, \delta_L, \delta_R \in \mathcal{D}_\Box^*$:

- $(s, \delta_L \check{d} \delta_R) \xrightarrow{?i} (t_o, \delta_L^{<} e \delta_R)$ iff $(s, d, i) \longrightarrow_{\mathcal{I}} (t, e, L, o)$ and $i \in \Sigma$,
- $(s, \delta_L \check{d} \delta_R) \xrightarrow{?i} (t_o, \delta_L e\, {}^{>}\delta_R)$ iff $(s, d, i) \longrightarrow_{\mathcal{I}} (t, e, R, o)$ and $i \in \Sigma$,
- $(s, \delta_L \check{d} \delta_R) \xrightarrow{\tau} (t_o, \delta_L^{<} e \delta_R)$ iff $(s, d, i) \longrightarrow_{\mathcal{I}} (t, e, L, o)$ and $i = \lambda$,
- $(s, \delta_L \check{d} \delta_R) \xrightarrow{\tau} (t_o, \delta_L e\, {}^{>}\delta_R)$ iff $(s, d, i) \longrightarrow_{\mathcal{I}} (t, e, R, o)$ and $i = \lambda$,
- $(s_o, \delta) \xrightarrow{!o} (s, \delta)$ iff $o \in \Sigma$, and $(s_o, \delta) \xrightarrow{\tau} (s, \delta)$ iff $o = \lambda$.

3. its initial state is the configuration $(q_{in}, \check{\Box})$.

To show that every transition systems associated with an ITM can be simulated by an RTM, it is convenient to allow RTMs to have transitions of the form $s \xmapsto{a[d/e]S} t$, where S is a stay transition with no movement of the tape head. We refer to such machines as *RTMs with stay transitions*. The operational semantics of RTMs can be extended to an operational semantics for RTMs with stay transitions by adding the clause: $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L \check{e} \delta_R)$ iff $s \xmapsto{a[d/e]S} t$. The transition system of an RTM with stay transitions can be simulated by an RTM up to divergence-preserving branching bisimilarity.

Lemma 3.6 (RTM with stay transitions)**.** *The transition system associated with an RTM with stay transitions is executable up to divergence-preserving branching bisimilarity.*

Proof. We suppose that $\mathcal{M} = (Q, \mapsto, Ini)$ is an RTM with stay transitions, and its transition system is $\mathcal{T}(\mathcal{M})$. We define a normal RTM $\mathcal{M}' = (Q_1, \mapsto_1, Ini_1)$ that simulates $\mathcal{T}(\mathcal{M})$ as follows, for all $s, t \in Q$ and $d, e \in \mathcal{D}_{\Box}$:

1. $Q_1 = Q \cup \{s_t \mid s, t \in Q\}$;
2. $s \xmapsto{a[d/e]L}_1 t$ iff $s \xmapsto{a[d/e]L} t$;
3. $s \xmapsto{a[d/e]R}_1 t$ iff $s \xmapsto{a[d/e]R} t$;
4. $s \xmapsto{a[d/e]L}_1 s_t$ and $s_t \xmapsto{\tau[d/d]R}_1 t$ iff $s \xmapsto{a[d/e]S} t$; and
5. $Ini_1 = Ini$.

Then it is straightforward to show that $\mathcal{T}(\mathcal{M}') \leftrightarrow_b^{\Delta} \mathcal{T}(\mathcal{M})$. □

The following lemma shows that every transition system associated with an ITM can be simulated by an RTM.

Lemma 3.7 (Simulating transition systems associated with ITMs)**.** *For every ITM $\mathcal{I}$ there exists an RTM $\mathcal{M}$, such that $\mathcal{T}(\mathcal{I}) \leftrightarrow_b^\Delta \mathcal{T}(\mathcal{M})$.*

We let $\mathcal{I} = (Q, \longrightarrow_{\mathcal{I}}, q_{in})$ be an ITM. By Lemma 3.6, it is enough to show that there exists an RTM with stay transitions $\mathcal{M}$ satisfying $\mathcal{T}(\mathcal{M}) \leftrightarrow_{\mathrm{b}}^\Delta \mathcal{T}(\mathcal{I})$. We construct $\mathcal{M} = (\mathcal{Q}, \mapsto, Ini)$ as follows:

1. $\mathcal{Q} = I \cup O$, where $I = Q$ and $O = \{s_o \mid o \in \Sigma_\lambda, s \in Q\}$;

2. the transition relation $\longrightarrow$ is defined by:

   (a) $s \xmapsto{?i[d/e]M} t_o$ if $(s, d, i) \longrightarrow_{\mathcal{I}} (t, e, M, o)$,

   (b) $s_o \xmapsto{!o[e/e]S} s$ for all $s \in \mathcal{S}$, $o \in \Sigma_\lambda$; and

3. $Ini = q_{in}$.

Then according to Definitions 2.11 and 3.5, the transition systems $\mathcal{T}(\mathcal{M})$ and $\mathcal{T}(\mathcal{I})$ are identical, and therefore $\mathcal{T}(\mathcal{M}) \leftrightarrow_{\mathrm{b}}^\Delta \mathcal{T}(\mathcal{I})$.

As a consequence we have the following theorem.

Theorem 3.8 (executability of ITM)**.** *The transition system associated with an ITM is executable modulo divergence-preserving branching bisimilarity.*

3.3 Executable ω-Translations

Recall that an ω-translation is defined to be interactively computable if, and only if, it can be realised by an ITM. RTMs are designed for investigating the expressive power of transition systems, rather than ω-translations, and not every RTM naturally has an ω-translation associated with it. Imposing some restrictions on the formalism of RTMs, however, we shall define a subclass of RTMs with which an ω-translation is naturally associated. The ω-translation realised by such an RTM is then called *executable*, and we shall establish that an ω-translation is interactively computable if, and only if, it is executable.

By analogy to the systems described in the theory of interactive computation, we let the RTMs for ω-translations execute in steps, in such a way that with every step a pair of input and output actions can be associated. With every infinite computation of the RTM we can then associate an interaction pair, and the RTM will thus give rise to an ω-translation.

Definition 3.9 (RTM for ω-translations)**.** Let $\mathcal{A}_\tau = \{?i, !o \mid i, o \in \{0,1\}\} \cup \{\tau\}$, and let $\mathcal{M} = (Q, \mapsto, Ini)$ be an RTM with $\mathcal{A}_\tau$ as its set of labels. Then $\mathcal{M}$ is an *RTM for ω-translations* if it satisfies the following properties:

1. the set of states Q is partitioned into disjoint sets I of input states and E of execution states, i.e., $Q = I \cup E$ and $I \cap E = \emptyset$;

2. the initial state Ini is an input state, i.e., $Ini \in I$;

3. for a transition $s \xmapsto{a[d/e]M} t$, if $s \in I$, then we have $a \in \{?0, ?1\}$ and $t \in E$; if $s \in E$, then we have $a \in \{!0, !1, \tau\}$ and $t \in I$;

4. for all $(s, d) \in E \times \mathcal{D}_\Box$, there is exactly one transition of the form $s \xmapsto{a[d/e]M} t$; and

5. for all $(s, d) \in I \times \mathcal{D}_\Box$, there are exactly two transitions of the form $s \xmapsto{a[d/e]M} t$, one with $a = ?0$ and the other one with $a = ?1$.

In the following lemma we establish some properties of the transition system associated with an RTM for ω-translation.

Lemma 3.10 (I/O LTS)**.** *Let $\mathcal{M}$ be an RTM for ω-translation. Then the transition system $\mathcal{T}(\mathcal{M}) = (\mathcal{S}_\mathcal{M}, \longrightarrow_\mathcal{M}, \uparrow_\mathcal{M})$ satisfies the following properties:*

1. *(*Alternation*) The set of states $\mathcal{S}_\mathcal{M}$ is partitioned into a set of input states $I_\mathcal{M}$ and a set of output states $E_\mathcal{M}$, i.e., $\mathcal{S}_\mathcal{M} = I_\mathcal{M} \cup E_\mathcal{M}$ and $I_\mathcal{M} \cap E_\mathcal{M} = \emptyset$. For every transition $s \xrightarrow{a} s'$, if $s \in I_\mathcal{M}$, then $a \in \{?0, ?1\}$ and $s' \in E_\mathcal{M}$; if $s \in E_\mathcal{M}$, then $a \in \{!0, !1, \tau\}$ and $s' \in I_\mathcal{M}$.*

2. *(*Unambiguity*) For every $s \in E_\mathcal{M}$, there is exactly one outgoing transition $s \xrightarrow{a} s'$ with $a \in \{!0, !1, \tau\}$.*

3. *(*Totality*) For every $s \in I_\mathcal{M}$, there are exactly two outgoing transitions, labelled with $?0$ and $?1$, respectively.*

Proof. Note that a state in $\mathcal{S}_\mathcal{M}$ is a configuration (s, δ) of $\mathcal{M}$, and we can make a partition of the set of all configurations according to the control states. If $s \in I$, then we have $(s, \delta) \in I_\mathcal{M}$; if $s \in E$, then we have $(s, \delta) \in E_\mathcal{M}$, where I and E are defined in Definition 3.9. We show the three properties in the lemma as follows.

1. (Alternation) By condition 1 in Definition 3.9, we have $Q = I \cup E$ and $I \cap E = \emptyset$, from which it follows that $\mathcal{S}_\mathcal{M} = I_\mathcal{M} \cup E_\mathcal{M}$ and $I_\mathcal{M} \cap E_\mathcal{M} = \emptyset$; moreover, by condition 2, for a transition $s \xmapsto{a[d/e]M} t$, if $s \in I$, then we have $a \in \{?0, ?1\}$ and

$t \in E$; if $s \in E$, then we have $a \in \{!0, !1, \tau\}$ and $t \in I$, from which it follows that for every transition $s \xrightarrow{a} s'$, if $s \in I_{\mathcal{M}}$, then $a \in \{?0, ?1\}$ and $s' \in E_{\mathcal{M}}$; if $s \in E_{\mathcal{M}}$, then $a \in \{!0, !1, \tau\}$ and $s' \in I_{\mathcal{M}}$.

2. (Unambiguity) By condition 3 in Definition 3.9, for all (s, d) where $s \in E$ and $d \in \mathcal{D}_{\Box}$, there is exactly one transition $s \xmapsto{a[d/e]M} t$, from which it follows that for every $s \in E_{\mathcal{M}}$, there is exactly one outgoing transition $s \xrightarrow{a} s'$ with $a \in \{!0, !1, \tau\}$.

3. (Totality) By condition 4 in Definition 3.9, for all (s, d) where $s \in I$ and $d \in \mathcal{D}_{\Box}$, there are exactly two transitions of the form $s \xmapsto{a[d/e]M} t$, with $a = ?0$ or $a = ?1$, respectively, which infers that for every $s \in I_{\mathcal{M}}$, there are two outgoing transitions labelled by ?0 and ?1, respectively.

□

We call a transition system that satisfies the conditions of Lemma 3.10 an *i/o labelled transition system* (I/O LTS). Moreover, by an analogy to the interactiveness condition for ITMs, we impose an interactiveness condition on RTMs for ω-translation.

Definition 3.11 (interactive I/O LTS)**.** An i/o transition system is interactive, if for every $s \in \mathcal{S}$ and $s \xrightarrow{?i} s_0$ with $i \in \{0, 1\}$, and for every sequence $s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \cdots$, there exists a natural number k, such that $s_k \xrightarrow{!o} s_{k+1}$ with $a_k = !o$ and $o \in \{0, 1\}$.

An RTM for ω-translation is *interactive* if the associated i/o transition system is.

We define the ω-translation realized by an RTM by defining the ω-translation realized by the i/o transition system associated with it. Let $T = (\mathcal{S}, \longrightarrow, \uparrow)$ be an i/o transition system, let $s \in \mathcal{S}$, and let $\sigma \in \mathcal{A}^{\omega}$, say $\sigma = a_0, a_1, \ldots$; we write $s \xrightarrow{\sigma}$ if there exist $s_0, s_0', s_1, s_1', \ldots \in \mathcal{S}$ such that $s = s_0$, and $s_i \longrightarrow^* s_i' \xrightarrow{a_i} s_{i+1}$ for all $i \geq 0$. (By $\longrightarrow^*$ we denote the reflexive-transitive closure of the relation $\xrightarrow{\tau}$.) If $\sigma \in \mathcal{A}^{\omega}$ and $s \xrightarrow{\sigma}$, then σ is a *weak infinite trace* from s. We denote by $Tr_w^{\infty}(s)$ the set of weak infinite traces from s, i.e.,

$$Tr_w^{\infty}(s) = \{\sigma \in \mathcal{A}^{\omega} \mid s \xrightarrow{\sigma}\} \ .$$

Definition 3.12 (the ω-translation realized by an interactive I/O LTS)**.** Let T be an i/o transition system, and s_0 be the initial state. For $\sigma \in Tr_w^{\infty}(s_0)$, the *input stream realised by* σ is the stream $\vec{x} \in \Sigma^{\omega}$ such that $\vec{x} = x_1 x_2 \ldots$, where $x_j = i$ if $?i$ is the j-th input action in σ, and similarly for the *output stream realized by* σ.

We define $(\vec{x}, \vec{y}) \in \{0, 1\}^{\omega} \times \{0, 1\}^{\omega}$ as the pair of input and output streams realized by σ as follows.

1. Its input stream is $\vec{x} = x_1 x_2 \ldots$, where $x_j = i$, if $?i$ is the j-th input action in σ, and

2. its output stream is $\vec{y} = y_1 y_2 \ldots$, where $y_j = o$, if $!o$ is the j-th output action in σ.

We say that T realizes ω-translation $\phi : \Sigma^\omega \to \Sigma^\omega$ iff, for every $\vec{x} \in \Sigma^\omega$, there exists a trace $\sigma \in Tr_w^\infty(s_0)$ with $\vec{x}$ as its input stream, and for every such trace, its output stream is $\vec{y} = \phi(\vec{x})$.

We can now define when an ω-translation is executable.

Definition 3.13 (executable ω-translation)**.** An ω-translation is executable if it can be realized by an executable i/o transition system.

Note that not every i/o transition system gives an ω-translation. The following example shows that the interactiveness condition is necessary.

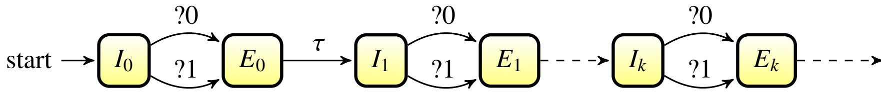


Figure 3.2: Interactivess is necessary for ω-translation

Example 3.14. Let $T = (\mathcal{S}, \longrightarrow, \uparrow)$ be the i/o transition system in Figure 3.2 defined as follows:

1. $\mathcal{S} = \{I_k \mid k \in \mathbb{N}\} \cup \{E_k \mid k \in \mathbb{N}\}$;

2. $\mathcal{S} = \{(I_k, ?0, E_k) \mid k \in \mathbb{N}\} \cup \{(I_k, ?1, E_k) \mid k \in \mathbb{N}\} \cup (E_k, \tau, I_k) \mid k \in \mathbb{N})$;

3. $\uparrow = I_0$.

T satisfies all the conditions in Lemma 3.10, however, it is not interactive. It does not realize any ω-translation, since no output stream is given from T.

The following lemma establishes that we can associate with every interactive i/o transition system an ω-translation.

Lemma 3.15 (interactive I/O LTS)**.** *If an i/o transition system is interactive, then it realises an ω-translation.*

Proof. Let T be an i/o interactive transition system, and let s_0 be the initial state of T. By Definition 3.12, we need to show that there exists an ω-translation ϕ such that for every $\vec{x} \in \Sigma^\omega$, there exists a trace $\sigma \in Tr_w^\infty(s_0)$ with input stream $\vec{x}$, and for every trace with input stream $\vec{x}$, its output stream is $\vec{y} = \phi(\vec{x})$.

By the alternation condition in Lemma 3.10, every $\sigma \in Tr_w^\infty(s_0)$ is of the form $i_0 o_0 i_1 o_1 \dots$ where $i_j \in \{?0, ?1\}$ and $o_j \in \{!0, !1, \tau\}$. Let $\vec{x}$ be an arbitrary input stream, by the totality condition in Lemma 3.10, we can find a trace $\sigma \in Tr_w^\infty(s_0)$ with input stream $\vec{x}$.

Moreover, given a trace σ with an infinite input stream $\vec{x}$, by interactiveness, it always produces an infinite output stream $\vec{y}$.

Finally, by unambiguity, there do not exist two traces sharing the same input stream. It follows that for every trace with input stream $\vec{x}$, its output stream is $\vec{y}$. Hence, we relate with every input stream a unique output stream, in a way, we get a ω-translation from T. □

We mapped every interactive i/o transition system to an ω-translation. Then we imposed an executability restriction on i/o transition systems, which maps every executable and interactive i/o transition system to an executable ω-translation. Now we proceed to establish a correspondence between the executable ω-translations and the interactively computable ω-translations.

It is not hard to show the following lemmas.

Lemma 3.16 (I/O LTSs and ω-translations)**.** *Let T_1 and T_2 be two interactive i/o transition systems, and $T_1 \leftrightarrows_b T_2$. Then they realize the same ω-translation.*

Proof. We let s_1 and s_2 be the initial states of T_1 and T_2, respectively. As $T_1 \leftrightarrows_b T_2$, we have that for every $\sigma \in Tr_w^\infty(s_1)$, there exists a trace $\sigma' \in Tr_w^\infty(s_2)$, and they share the same input and output stream, and vice versa. It follows that T_1 and T_2 realize the same ω-translation. □

Lemma 3.17 (limit of traces of I/O LTS)**.** *Let T be an executable interactive i/o transition system, let s_0 be its initial state, and let g be a function defined as follows:*

$$g : \Sigma^* \to \Sigma^* \ ,$$

satisfying that if $g(x) = y$, then for every $\sigma \in Tr_w^\infty(s_0)$ with input $\vec{x}$ and output stream $\vec{y}$, if $x \prec \vec{x}$, then $y \prec \vec{y}$. Then g is computable.

Proof. We consider a finite trace from s_0, and associate with such a trace its input and output sequences in a similar way as defined in Definition 3.12. By Lemma 3.15, there is only one finite trace with x as its input sequence; let y be the associated output

sequence. By totality, for every $x \in \Sigma^*$, there exists such a finite trace with $y = g(x)$ as the associated output sequence. Therefore, the function g is computable if the finite traces associated with every input sequence x are computable.

As T is executable, the transition relation of i/o transition system is computable. We can design a computable procedure to simulate the execution of T on every input sequence $x \in \Sigma^*$. So g is computable. □

We have the following theorem.

Theorem 3.18 (executable ω-translation)**.** *An ω-translation is executable iff it is a limit-continuous total function.*

Proof. We let ϕ be an ω-translation.

1. For the "only if" part, we assume that ϕ is executable, and show that it is a limit-continuous total function. It suffices to show that there exists a computable total function $g : \Sigma^* \to \Sigma^*$, such that g is monotonic and for all strictly increasing chains $u_1 \prec u_2 \prec \ldots \prec u_t \prec \ldots$ with $u_t \in \Sigma^*$ $(t \geq 1)$, one has $\phi(\lim_{t\to\infty} u_t) = \lim_{t\to\infty} g(u_t)$.

   We assume that ϕ is realized by an executable interactive i/o transition system T, and we let s_0 be the initial state of T. By Lemma 3.17 the following function is computable: $g : \Sigma^* \to \Sigma^*$, satisfying that if $g(x) = y$, then for every $\sigma \in Tr_w^\infty(s_0)$ with input stream $\vec{x}$ and output stream $\vec{y}$, if $x \prec \vec{x}$, then $y \prec \vec{y}$. By unambiguity and totality, g is a monotonic and total computable function.

   Moreover, for a strictly increasing chain $u_1 \prec u_2 \prec \ldots \prec u_t \prec \ldots$ with $u_t \in \Sigma^*$ for $t \geq 1$, the computation of $\lim_{t\to\infty} g(u_t)$ is the execution of a trace σ with the input stream $\lim_{t\to\infty} u_t$. Hence we have $\phi(\lim_{t\to\infty} u_t) = \lim_{t\to\infty} g(u_t)$.

   Thus, g is the computable total function we need, and it follows that ϕ is a computable limit-continuous total function.

2. For the "if" part, we assume that ϕ is a total limit-continuous function, and design an RTM $\mathcal{M}$ to realize this translation. By Theorem 3.4, ϕ is interactively computable by some ITM $\mathcal{M}'$. According to Definition 3.5 and Lemma 3.10, the transition system associated with $\mathcal{M}'$ is an i/o transition system. Moreover, according to Theorem 3.8, it is an executable i/o transition system. Therefore, we have shown that ϕ is an executable ω-translation by Lemma 3.16.

□

By Theorem 3.4 and Theorem 3.18, we have the following corollary.

Corollary 3.19 (executable ω-translation)**.** *An ω-translation is executable iff it is interactively computable.*

Therefore, the classes of computable limit-continuous functions, interactively computable ω-translations and executable ω-translations coincide.

3.4 Advice

In [58], the computational power of evolving interactive systems is studied using ITMs. Particularly, a mechanism called *advice function* is introduced to enhance the computational power of an ITM. In this way, the insertion of external information into the course of a computation is allowed, which leads to a non-uniform operation, which means the behaviour of a machine varies with different advice functions. In this section, we introduce the notion of advice as a process in parallel composition with an RTM, and show that advice processes indeed give the systems more expressive power.

In this section, we consider advices as functions over natural numbers. In order to record a number on the tape, a natural number n is encoded by a sequence of n "1"s ending with a "0". In [58], the notion of ITM with advice is defined as follows.

Definition 3.20 (advice functions)**.** An *advice function* is a function $f : \mathbb{N} \to \mathbb{N}$. An ITM with advice (ITM/A) is equipped with a separate *advice tape* and a distinguished *advice state*. By writing the value of the argument x on the advice tape and by entering the advice state, the value of $f(x)$ will appear on the advice tape in a single step. By this action, the original contents of the advice tape is completely overwritten.

Here we do not put a restriction on the length of the advice function as in [60], since it does not make a difference in the issue of computability, and we are not yet interested in the issue of complexity. Moreover, we do not restrict the advice functions to computable functions. From [60] a computable advice only yields a speedup to the computing procedure, it does not alter the computability of the machine. It is obvious that ITMs with uncomputable advice functions cannot be simulated by any RTM, as uncomputable advice functions cannot be computed by the mechanism of RTMs. As an extension, we equip RTMs with advice processes which enable the simulation of ITM/As.

An advice process A_f is designed to compute the function f, and can interact with an RTM $\mathcal{M}$. As an advice function is not necessarily computable, we cannot associate with every advice process an executable transition system. An RTM $\mathcal{M}$ communicates with A_f as follows. We presuppose an input channel *in* and an output channel *out*. When $\mathcal{M}$ needs to get the result of $f(i)$, it enters a special control state a_f, and starts to

send a sequence of i "1" s and a "0" , which is already written on the tape, to the channel *in*, and then, it receives the result sequence $f(i)$ "1"s and a "0" from *out* channel, and writes them on the tape. This procedure ends in another control state. We can model an advice process as follows.

Definition 3.21 (the advice process for f)**.** Let $f : \mathbb{N} \to \mathbb{N}$ be a function. A_f is the *advice process for* f with transition system $\mathcal{T}(A_f) = (\mathcal{S}, \to, \uparrow)$, where

1. $\mathcal{S} = \{s_i \mid i = 0, 1, 2, \ldots\} \cup \{t_i \mid i = 0, 1, 2, \ldots\}$, and
2. $s_i \xrightarrow{in?1} s_{i+1},\ i = 0, 1, 2, \ldots \quad s_i \xrightarrow{in?0} t_{f(i)},\ i = 1, 2, \ldots$
   $t_i \xrightarrow{out!1} t_{i-1},\ i = 1, 2, \ldots \quad t_0 \xrightarrow{out!0} s_0$
3. $\uparrow = s_0$.

The behaviour of A_f is deterministic. It receives a sequence of i "1"s from the channel *in*, followed by a "0" symbol, indicating the end of the sequence, and then, it produces $f(i)$ "1"s to the channel *out*, also followed by a "0" symbol. This procedure is repeated indefinitely.

The parallel composition of an RTM $\mathcal{M}$ and an advice process A_f for f is written as $[\mathcal{M} \parallel A_f]_C$. The parallel composition is defined in the same way as the parallel composition of two RTMs in Definition 2.14, where $C = \{in, out\}$ is the set of restricted names for communication.

Definition 3.22 (RTMs with advice)**.** Let $\mathcal{M}$ be an RTM and A_f be an advice process for f. We call $[\mathcal{M} \parallel A_f]_C$ a *Reactive Turing Machine with advice (RTM/A)*, where $C = \{in, out\}$.

Note that, since advice functions and advice processes both introduce the power of computing functions on natural numbers, by analogy to Corollary 3.19, we have the following Corollary.

Corollary 3.23. *An ω-translation is realisable by an ITM/A iff it is realisable by an RTM/A.*

By analogy to normal RTMs, we also define a notion of executability with respect to RTM/A.

Definition 3.24 (executability with respect to RTM/A)**.** A transition system T is *executable with advice process A_f* modulo a behavioural equivalence $\equiv$, if there exists an RTM/A $[\mathcal{M} \parallel A_f]_C$ such that $T \equiv \mathcal{T}(\mathcal{M})$.

In the theory of executability, it is a natural question to figure out the expressive power of the labelled transition systems associated with RTM/As. We now proceed to show that every boundedly branching labelled transition system can be simulated by some RTM/A up to divergence-preserving branching bisimilarity, provided that the advice is not restricted to evaluate computable functions, that is, we expect advice processes with the power of computing arbitrary functions.

Let T be any bounded branching transition system (not necessarily effective). Based on a presupposed encoding of its sets of states and actions and its transition relation, let the advice function f_T be such that for the encoding of a state it yields the encoding of the set of all outgoing transitions of that state. It is straightforward to define an RTM that simulates T with the help of f_T. Then we obtain the following result.

Theorem 3.25 (boundedly branching LTSs and RTM/A)**.** *If T is a countable boundedly branching labelled transition system, then there exists an advice process A_f and an RTM/A $[\mathcal{M} \parallel A_f]_C$ such that*

$$\mathcal{T}([\mathcal{M} \parallel A_f]_C) \leftrightarroweq_b^{\Delta} T \ .$$

Proof. We assume that $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ is an $\mathcal{A}_\tau$-labelled transition system, and we assume that it has n distinct action labels and its branching degree is bounded by k. We also assume an encoding $\lceil_\rceil$ that encodes $\mathcal{A}_\tau$ and $\mathcal{S}_T$ as natural numbers. We denote by $\lceil a\rceil$ the encoding of an action a and denote by $\lceil s\rceil$ be the encoding of a state s , and denote by $\lceil x_1, x_2, \ldots, x_n\rceil$ the encoding of an n-tuple $(x_1, x_2, \ldots, x_n)$.

We declare an advice process A_f that realizes the following function:

$$f(\lceil s\rceil) = \lceil a_1, \ldots, a_m, s_1, \ldots, s_m\rceil \ ,$$

where $(a, s) \in \{(a_1, s_1), \ldots, (a_m, s_m)\}$ iff $s \xrightarrow{a}_T s$.

We characterise an outline of the execution of $\mathcal{M}$ as follows. (We omit the position of the tape head since it is not crucial.)

1. We need the following control states: *initial*, *advice*, *decode*, $next_{\mathcal{A}_\tau^{\leq k}}$ ($\mathcal{A}_\tau^{\leq k}$ ranges over all $\mathcal{A}_\tau$ words with length of at most k, i.e., there are 2^k such states), $choose_i$ ($i \leq k$).

2. The execution of $\mathcal{M}$ is as follows, its initial configuration is $(initial, \Box)$.

   (a) In the state *initial*, the machine writes the encoding of the initial state of the transition system $\lceil\uparrow_T\rceil$ on the tape, and reaches *advice* state.

   $$(initial, \Box) \longrightarrow^* (advice, \lceil\uparrow_T\rceil) \ .$$

(b) In the state *advice*, the machine sends the encoding of the current state $\ulcorner s_0 \urcorner$ to the advice process, and gets the encoding of the list of all possible transitions $\ulcorner a_1, \ldots, a_m, s_1, \ldots, s_m \urcorner$ from the advice process.

$$(advice, \ulcorner s_0 \urcorner) \longrightarrow^* (decode, \ulcorner a_1, \ldots, a_m, s_1, \ldots, s_m \urcorner) \ .$$

(c) In the state *decode*, the machine decodes all the actions from the tape, and enters one of the *next* state.

$$(decode, \ulcorner a_1, \ldots, a_m, s_1, \ldots, s_m \urcorner) \longrightarrow^* (next_{\{a_1,\ldots,a_m\}}, \ulcorner s_1, \ldots, s_m \urcorner) \ .$$

(d) In the state $next_{\{a_1,\ldots,a_m\}}$, the machine chooses one of the actions. For every $i = 1, \ldots, m$, there is a transition

$$(next_{\{a_1,\ldots,a_m\}}, \ulcorner s_1, \ldots, s_m \urcorner) \xrightarrow{a_i} (choose_i, \ulcorner s_1, \ldots, s_m \urcorner) \ .$$

(e) In the state $choose_i$, the machine projects the encoding $\ulcorner s_1, \ldots, s_m \urcorner$ to the encoding of the i-th state, and enters *advice* state again.

$$(choose_i, \ulcorner s_1, \ldots, s_m \urcorner) \longrightarrow^* (advice, \ulcorner s_i \urcorner) \ .$$

The above procedure describes the simulation of a step of transition $s_0 \xrightarrow{a_i}_T s_i$ in T. Note that the choice of the transition happens in the state $next_{\{a_1,\ldots,a_m\}}$; all the other states have only one outgoing transition, respectively. Moreover, no infinite τ-transition sequence is introduced for simulation.

In a way, one may verify that $\mathcal{T}([\mathcal{M} \parallel A_f]_C) \leftrightarroweq_b^\Delta T$. □

If we, instead, let the advice function f_T be such that on the code of a pair of a state s and a natural number i that yields the code of the ith outgoing transition of s, then we can extend the simulation to transition systems with countably many states and transitions.

Theorem 3.26 (countable LTSs and RTM/A). *If T is a countable labelled transition system, then there exists an RTM/A $[\mathcal{M} \parallel A_f]_C$ such that*

$$\mathcal{T}([\mathcal{M} \parallel A_f]_C) \leftrightarroweq_b T \ .$$

Proof. We assume that $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ is a countable $\mathcal{A}_\tau$-labelled transition system, and we assume that it has n distinct action labels and it possibly has infinite branching. We also assume an encoding $\ulcorner _ \urcorner$ that encodes $\mathcal{A}_\tau$ and $\mathcal{S}_T$ as natural numbers. We denote by $\ulcorner a \urcorner$ the encoding of an action a and denote by $\ulcorner s \urcorner$ the encoding of a state s , and denote by $\ulcorner x_1, x_2, \ldots, x_n \urcorner$ the encoding of an n-tuple $(x_1, x_2, \ldots, x_n)$.

The transition relation $\longrightarrow_T$ maps a state, for instance, s_0, to a possibly infinite set $\{(a_i, s_i) \mid s_0 \xrightarrow{a_i}_T s_i\}$, denoted by $\longrightarrow_T (s_0)$. We define an order $<_T$ over the elements in the set $s_0 \longrightarrow_T$ such that $(a, s) <_T (a', s')$, if $\ulcorner a, s \urcorner <_T \ulcorner a', s' \urcorner$.

We declare an advice function A_f that realizes the following function:

$$f(\ulcorner s_0, i \urcorner) = \ulcorner a_i, s_i \urcorner \ ,$$

where (a_i, s_i) is the i-th element from $\longrightarrow_T (s_0)$ regarding to $<_T$.

We characterise an outline of the execution of $\mathcal{M}$ as follows. (We omit the position of the tape head since it is not crucial.)

1. We need the following control states: *initial*, *advice*, *decode*, $next_{\{\mathcal{A}_\tau\}}$ ($\{\mathcal{A}_\tau\}$ ranges over all subsets of $\mathcal{A}_\tau$, so there are 2^n such states), $choose_i$ ($i = 1, 2$).

2. The execution of $\mathcal{M}$ is as follows, we use a pair (s, δ) to denote the current configuration of the machine.

   (a) In the state *initial*, the machine writes the encoding of the initial state of the transition system $\ulcorner \uparrow_T \urcorner$ on the tape, and reaches the state *advice*.

   $$(initial, \Box) \longrightarrow^* (advice, \ulcorner \uparrow_T, 1 \urcorner) \ .$$

   (b) In the state *advice*, the machine either increases the counter i by 1, or sends $\ulcorner s_0, i \urcorner$ to the advice, and gets $\ulcorner (a_i, s_i) \urcorner$ from the advice.

   $$(advice, \ulcorner s_0, i \urcorner) \longrightarrow^* (advice, \ulcorner s_0, i + 1 \urcorner), \text{or}$$
   $$(advice, \ulcorner s_0, i \urcorner) \longrightarrow^* (decode, \ulcorner s_0, s_i, a_i \urcorner)$$

   (c) In the state *decode*, the machine decodes the action a_i from the tape, and enters the state $next_{a_i}$.

   $$(decode, \ulcorner s_0, s_i, a_i \urcorner) \longrightarrow^* (next_{a_i}, \ulcorner s_0, s_i \urcorner) \ .$$

   (d) In the state $next_{a_i}$, the machine either performs the action, or changes its current choice to another transition.

   $$(next_{a_i}, \ulcorner s_0, s_i \urcorner) \xrightarrow{\tau} (choose_1, \ulcorner s_0, s_i \urcorner), \text{or}$$
   $$(next_{a_i}, \ulcorner s_0, s_i \urcorner) \xrightarrow{a_i} (choose_2, \ulcorner s_0, s_i \urcorner) \ .$$

   (e) In the state $choose_i$ (i=1,2), the machine projects the encoding $\ulcorner s_1, s_2 \urcorner$ to the encoding of the i-th state, and enters the state *advice* again.

   $$(choose_i, \ulcorner s_1, s_2 \urcorner) \longrightarrow^* (advice, \ulcorner s_i, 1 \urcorner) \ .$$

One can verify that

$$\begin{aligned}\mathcal{R} \quad &= \{(s, s') \mid s \in \mathcal{S}_T, s' = (advice, \lceil s, i \rceil) \text{ or } (decode, \lceil s, a_i, s_i \rceil) \\ &\text{or } (next_{a_i}, \lceil s, s_i \rceil) \text{ or } (choose_1, \lceil s, s_i \rceil) \text{ or } (choose_2, \lceil s_i, s \rceil)\}\end{aligned}$$

is a branching bisimulation relation. Hence, we have $\mathcal{T}([\mathcal{M} \parallel A_f]_C) \leftrightarrow_b T$. □

Note that the transition system associated with an RTM/A is boundedly branching. Hence, by Theorem 2.32, if a transition system has no divergence up to $\leftrightarrow_b^\Delta$ and is unboundedly branching up to $\leftrightarrow_b^\Delta$, then it is not executable modulo $\leftrightarrow_b^\Delta$. It follows that there exist countable unboundedly branching transition systems that cannot be simulated by an RTM/A modulo $\leftrightarrow_b^\Delta$.

3.5 Remarks

We have discussed the relationship between two models of computation that both take interaction into account. We have established that the model of RTMs subsumes and is more expressive than the model of ITMs when it comes to specify behaviour, and coincides with the model of ITMs when defining ω-translations.

Furthermore, we have shown that RTMs admit an extension with advice that facilitates modelling non-uniform behaviour. In [13] it was established that every effective transition system can be simulated by an RTM. Our result that every countable transition system can be simulated by an RTM with advice further confirms the universal expressivity of the notion of RTM.

Finally, I list a couple of points as directions for future work:

1. In [81], a complexity theory for interactive computation has been defined on the basis of ITMs and ω-translations. Clearly, such a complexity theory could also be based on the restricted class of RTMs for ω-translations. Such a complexity theory could then be generalised further towards a complexity theory for general executable behaviour.

2. In [65], I/O automata were introduced as a formal model for describing asynchronous concurrent systems. The interaction between systems is described by input or output actions through particular channels in an I/O automaton. We could extract an input and an output sequence to describe the process of interaction, and in a way, the capability of making interactive computation for an I/O automaton could also be expressed in terms of interactive translations. In the future we intend to make use our notion of interactive I/O LTS to investigate the relationship between I/O automata and RTMs.

3. In computability theory, there is an arithmetical hierarchy for uncomputable functions [76]. The arithmetical hierarchy naturally distinguishes different kinds of functions, which could be advice functions in RTM/As. We can impose this hierarchy on RTM/As, which allows RTMs to equip different classes of advice functions. As a consequence, we get a bunch of variations of RTM/As, as well as the classes of associated transition systems. It could be interesting future work to establish a hierarchy on those transition systems. In a way, we expect a hierarchy on unexecutable transition systems with respect to the advice functions allowed.

Chapter 4

Sequential Composition and Intermediate Termination

This chapter focuses on the executability theory on a process calculus called TCP, which includes two problems, namely, the comparison between pushdown processes context-free processes, and the expressivity regarding to RTMs of a process calculus using the iteration and nesting operators instead of a recursive definition. In the first problem, the pushdown processes refers to the transition systems associated with pushdown automata, and the context-free processes refers to the transition systems associated with context-free grammars. In the second problem, iteration and nesting operators are two alternative choices for a process calculus to specify a transition system with infinite states. They appeared in [18, 22] as a suitable replacement for the recursive definitions. Neither problem has an obvious solution in the current setting of the operational semantics of the sequential composition operator in the presence of intermediate termination. To overcome this difficulty, we propose a revised semantics, and solve the two problems mentioned above.

Sequential composition is a standard operator in many process calculi. The functionality of the sequential composition operator is to concatenate the behaviours of two systems. It has been widely used in many process calculi with the notation "$\cdot$". We illustrate its operational semantics by a TCP process $P \cdot Q$ [4]. If the process P has a transition $P \xrightarrow{a} P'$ for some action label a, then the composition $P \cdot Q$ has the transition $P \cdot Q \xrightarrow{a} P' \cdot Q$. Termination is an important behaviour for models of computation [4]. A semantic distinction between successful and unsuccessful termination in concurrency theory (CT) is especially important for a smooth incorporation of the

classical theory of automata and formal languages (AFT): the distinction is used to express whether a state in an automaton is accepting or not. Automata may even have states that are accepting and may still perform transitions; this phenomenon we call *intermediate termination*. From a concurrency-theoretic point of view, such behaviour is perhaps somewhat unnatural. To be able to express it nevertheless, it starts with having a constant **1** expressing successful termination, and we let an alternative composition inherit the option to terminate from just one of its components. The expression $a.(b+\mathbf{1})$ then denotes the process that does an a-transition and subsequently enters a state that is successfully terminated but can also do a b-transition.

To specify the operational semantics of sequential composition in a setting with an explicit successful termination, usually the following three rules are added: the first one states that the sequential composition $P \cdot Q$ terminates if both P and Q terminate; the second one states that if P admits a transition $P \xrightarrow{a} P'$, then $P \cdot Q$ admits a transition $P \cdot Q \xrightarrow{a} P' \cdot Q$; and the third one states that if P terminates, and there is a transition $Q \xrightarrow{a} Q'$, then we have the transition $P \cdot Q \xrightarrow{a} Q'$.

In this chapter, we discuss a complication resulting from these operational semantics of the sequential composition operator. The complication is that a process expression P with the option to terminate is *transparent* in a sequential context $P \cdot Q$: if P may still perform observable behaviour other than termination, then this may be skipped by doing a transition from Q. There are two complications associated with transparency in our attempts to achieve a smooth integration of process theory and the classical theory of automata and formal languages [8]:

The relationship between context-free processes (i.e., processes that can be specified with a guarded recursive specification over a language with action constants, constants for deadlock (**0**) and successful termination (**1**), and binary operations for sequential and alternative composition) and pushdown automata has been discussed a lot in the literature [9]. We observe that context-free processes may have unbounded branching modulo branching bisimilarity by stacking unboundedly many transparent terms with sequential composition. As far as we know, it is still an open problem whether pushdown processes may have unbounded branching. The best known correspondence between context-free processes and pushdown processes is that every context-free process is equivalent to a pushdown process modulo contrasimulation. It is not known whether unboundedly branching behaviour can or cannot be specified in a pushdown process modulo rooted branching bisimulation or any behavioural equivalence stronger than contrasimulation [9]. In order to improve the result to a finer notion of behavioural equivalence, we need to eliminate the problem of unbounded branching. The culprit for not having a better correspondence between context-free processes and pushdown processes turns out to be the semantics of intermediate termination: this

chapter shows that with just a minor adaptation of the semantics a decent correspondence, even modulo strong bisimilarity, is obtained

Transparency also complicates matters if one wants to specify some form of memory (e.g., a counter, a stack, or a tape) that always has the option to terminate, but at the same time does not lose data. If the standard process algebraic specifications of such memory processes are generalised to a setting with intermediate termination, then either they are not always terminating, or they are 'forgetful' and may nondeterministically lose data. This is a concern when one tries to specify the behaviour of a pushdown automaton or a Reactive Turing Machine in a process calculus [13, 62, 63]. The process calculus TCP with iteration and nesting is Turing complete [18, 21]. Moreover, it follows from the result in [21] that it is reactively Turing powerful if intermediate termination is not considered. However, it is not clear to us how to reconstruct the proof of reactive Turing powerfulness if termination is considered. Due to the forgetfulness on the stacking of transparent process expressions, it is not clear to us how to define a stack or even a counter that is always terminating, which is crucial for establishing the reactive Turing powerfulness.

In order to avoid the (in some cases) undesirable features of unbounded branching and forgetfulness, we propose a revised operational semantics for the sequential composition operator. The modification consists of disallowing a transition from the second component of a sequential composition if the first component is still able to perform a transition. Thus, with the revised operator we avoid the problems mentioned above. We shall prove that every context-free process is bisimilar to a pushdown process, and that TCP with iteration and nesting is reactively Turing powerful modulo divergence-preserving branching bisimilarity (without resorting to recursion) in the revised semantics.

The research presented in this chapter is part of an attempt to achieve a smoother integration of the classical theory of automata and formal languages within concurrency theory . The idea is to recognise that a finite automaton is just a special type of labelled transition system, that more complicate automata (pushdown automata, Turing machines) naturally generate transition systems, and that there is a natural correspondence between regular expressions and grammars on the one hand and certain process calculi on the other hand. In [8, 11, 12] various notions of automata from automata and formal languages modulo branching bisimilarity have been studied. In [10] the correspondence between finite automata and regular expressions extended with parallel composition modulo strong bisimilarity was explored.

The chapter is structured as follows. We first introduce TCP and its variants with the standard version of sequential composition in Section 4.1. Next, we discuss the complications caused by transparency in Section 4.2. Then, in Section 4.3, we propose the revised operational semantics of the sequential composition operator, and show that

rooted divergence-preserving branching bisimulation is a congruence. In Section 4.4, we revisit the relationship between context-free processes and pushdown automata, and show that every context-free process is bisimilar to a pushdown process in our revised semantics. In Section 4.5, we prove that TCP with iteration and nesting is reactively Turing powerful in the revised semantics. In Section 4.6, we draw some conclusions and propose some future work.

4.1 TCP and Variations of TCP

4.1.1 TSP

We start this section by introducing the process calculus *Theory of Sequential Processes* (TSP) that allows us to describe transition systems with behaviours that are sequentially compositional [4].

Let $\mathcal{N}$ be a countably infinite set of names. The set of TSP process expressions $\mathcal{P}$ is generated by the following grammar ($a \in \mathcal{A}_\tau$, $N \in \mathcal{N}$, $P \in \mathcal{P}$):

$$P := \mathbf{0} \mid \mathbf{1} \mid a.P \mid P \cdot P \mid P + P \mid N \ .$$

We briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes *deadlock*, the unsuccessfully terminated process. The constant $\mathbf{1}$ denotes *termination*, the successfully terminated process. For each action $a \in \mathcal{A}_\tau$ there is a unary operator $a.$ denoting action prefix; the process denoted by $a.P$ can do an a-labelled transition to the process P. The binary operator $+$ denotes alternative composition or choice. The binary operator $\cdot$ represents the sequential composition of two processes.

Let P be an arbitrary process expression; we use an abbreviation inductively defined by:

1. $P^0 = \mathbf{1}$; and
2. $P^{n+1} = P \cdot P^n$ for all $n \in \mathbb{N}$.

A recursive specification E is a set of equations $E = \{N \stackrel{\text{def}}{=} P \mid N \in \mathcal{N}, P \in \mathcal{P}\}$, satisfying:

1. for every $N \in \mathcal{N}$ it includes at most one equation with N as left-hand side, which is referred to as the *defining equation* for N; and
2. if some name N' occurs in the right-hand side P' of some equation $N' = P'$ in E, then E must include a defining equation for N'.

An occurrence of a name N in a process expression is *guarded* if the occurrence is within the scope of an action prefix $a.$ for some $a \in \mathcal{A}$ (τ cannot be a guard). A recursive specification E is guarded if all occurrences of names in right-hand sides of equations in E are guarded.

We use structural operational semantics to associate a transition relation with TSP process expressions. In many process calculi, process terms are assumed to contain some variables. A term is *closed* iff it does not contain any free variables. Structural operational semantics induces a transition relation on closed terms. We let $\longrightarrow$ be the $\mathcal{A}_\tau$-labelled transition relation induced on the set of process expressions by operational rules in Table 4.1. Note that we presuppose a recursive specification E.

$$\frac{}{\mathbf{1}\downarrow} \qquad \frac{}{a.P \xrightarrow{a} P}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 + P_2 \xrightarrow{a} P_1'} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 + P_2 \xrightarrow{a} P_2'} \qquad \frac{P_1\downarrow}{P_1 + P_2\downarrow} \qquad \frac{P_2\downarrow}{P_1 + P_2\downarrow}$$

$$\frac{P_1\downarrow \quad P_2\downarrow}{P_1 \cdot P_2\downarrow} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1 \cdot P_2 \xrightarrow{a} P_1' \cdot P_2} \qquad \frac{P_1\downarrow \quad P_2 \xrightarrow{a} P_2'}{P_1 \cdot P_2 \xrightarrow{a} P_2'}$$

$$\frac{P \xrightarrow{a} P' \quad (N \stackrel{\text{def}}{=} P) \in E}{N \xrightarrow{a} P'} \qquad \frac{P\downarrow \quad (N \stackrel{\text{def}}{=} P) \in E}{N\downarrow}$$

Table 4.1: The operational semantics of TSP

Here we use $P \xrightarrow{a} P'$ to denote an a-labelled transition $(P, a, P') \in \longrightarrow$. We say a process expression P' is *reachable* from P is there exist process expressions $P_0, \ldots, P_n$ and labels $a_1, \ldots, a_n$ such that $P = P_0 \xrightarrow{a_1} \cdots \xrightarrow{a_n} P_n = P'$.

Given a TSP process expression P, the transition system $\mathcal{T}(P) = (\mathcal{S}_P, \longrightarrow_P, \uparrow_P, \downarrow_P)$ associated with P is defined as follows:

1. the set of states $\mathcal{S}_P$ consists of all process expressions reachable from P;

2. the transition relation $\longrightarrow_P$ is the restriction to $\mathcal{S}_P$ of the transition relation defined on all process expressions by the structural operational semantics, i.e., $\longrightarrow_P = \longrightarrow \cap (\mathcal{S}_P \times \mathcal{A}_\tau \times \mathcal{S}_P)$;

3. $\uparrow_P = P$; and

4. the set of final states $\downarrow_P$ consists of all process expressions $Q \in \mathcal{S}_P$ such that $Q\downarrow$, i.e., $\downarrow_P = \downarrow \cap \mathcal{S}_P$.

4.1.2 TCP

We also use (a restricted variant of) the process calculus TCP in later sections. It is derived from the *Theory of Communicating Processes* (TCP) introduced in [4] which uses silent steps as the abstraction of encapsulated communication. It is obtained by adding a parallel composition operator to TSP. Let C be a set of *channels* and $\mathcal{D}_\Box$ be a set of *data symbols*. For every subset $C' \subseteq C$, we presuppose a special set of actions $\mathcal{I}_{C'} \subseteq \mathcal{A}_\tau$ defined by: $\mathcal{I}_{C'} = \{c?d, c!d \mid d \in \mathcal{D}_\Box, c \in C'\}$.

The actions $c?d$ and $c!d$ denote the events that a datum d is received or sent along channel c, respectively. We include binary parallel composition operators $[_ \parallel _]_{C'}$ $(C \subseteq C)$. Communication along the channels in C' is enforced and communication results in τ.

Let $\mathcal{N}$ be a countably infinite set of names. The set of TCP process expressions $\mathcal{P}$ is generated by the following grammar ($a \in \mathcal{A}_\tau$, $N \in \mathcal{N}$, $P \in \mathcal{P}$):

$$P := \mathbf{0} \mid \mathbf{1} \mid a.P \mid P \cdot P \mid P + P \mid [P \parallel P]_{C'} \mid N \ .$$

The operational semantics of the parallel composition operators is presented in Table 4.2.

$$\frac{P_1\downarrow \quad P_2\downarrow}{[P_1 \parallel P_2]_{C'}\downarrow}$$

$$\frac{P_1 \xrightarrow{a} P_1' \quad a \notin \mathcal{I}_{C'}}{[P_1 \parallel P_2]_{C'} \xrightarrow{a} [P_1' \parallel P_2]_{C'}} \qquad \frac{P_2 \xrightarrow{a} P_2' \quad a \notin \mathcal{I}_{C'}}{[P_1 \parallel P_2]_{C'} \xrightarrow{a} [P_1 \parallel P_2']_{C'}}$$

$$\frac{P_1 \xrightarrow{c?d} P_1' \quad P_2 \xrightarrow{c!d} P_2' \quad c \in C'}{[P_1 \parallel P_2]_{C'} \xrightarrow{\tau} [P_1' \parallel P_2']_{C'}} \qquad \frac{P_1 \xrightarrow{c!d} P_1' \quad P_2 \xrightarrow{c?d} P_2' \quad c \in C'}{[P_1 \parallel P_2]_{C'} \xrightarrow{\tau} [P_1' \parallel P_2']_{C'}}$$

Table 4.2: The operational semantics of parallel composition in TCP

By analogy to TSP, we also associate with every TCP process a labelled transition system according to the operational semantics given in Table 4.1 and Table 4.2.

In [13], the expressivity of TCP is studied, TCP it is proved to be equivalent to RTM by excluding the sequential composition operator.

Theorem 4.1 (expressivity of TCP)**.** *TCP excluding the sequential composition operator is reactively Turing powerful and executable modulo* $\leftrightarrows_b^\Delta$.

Remark 4.2. The reactively Turing powerfulness result still holds for TCP with the sequential composition operator, but the executability result does not.

As a consequence, we also have the reactive Turing powerfulness of TCP. However, it is not executable modulo divergence-preserving branching bisimilarity by Theorem 2.32, since we may define processes with unboundedly branching degree in TCP. (There is an example in Section 4.2.) We only have its executability modulo the divergence-insensitive variant of branching bisimilarity.

Corollary 4.3 (reactive Turing powerfulness and executability of TCP)**.** *TCP is*

1. *reactively Turing powerful modulo* $\leftrightarrows_b^\Delta$*, and*
2. *executable modulo* $\leftrightarrows_b$*, but not modulo* $\leftrightarrows_b^\Delta$.

4.1.3 TCP with Non-regular Iterators

In this chapter, we aim to find some reactively Turing powerful variants of TCP without recursive specification. Inspired by the work by Bergstra, Bethke and Ponse [18, 21], there are some non-regular iterators that might produce behaviours similar to recursive specification of TCP processes. In particular, we consider TCP without recursion but with iteration and nesting [18, 21] in which a binary nesting operator $^\sharp$ and a Kleene star operator * are added. Different from them, we use a unary Kleene star operator. We define TCP with iteration and nesting (TCP$^\sharp$) with process expressions generated by the following grammar:

$$P := \mathbf{0} \mid \mathbf{1} \mid a.P \mid P \cdot P \mid P + P \mid [P \parallel P]_{C'} \mid P^\sharp P \mid P^* \ .$$

We give the operational semantics of these two operators in Table 4.3.

To get some intuition for the operational interpretation of these operators, note that the processes P^* and $P_1{}^\sharp P_2$ respectively satisfy the following equations modulo strong bisimilarity:

$$\begin{aligned} P^* &= P \cdot P^* + \mathbf{1} \\ P_1{}^\sharp P_2 &= P_1 \cdot (P_1{}^\sharp P_2) \cdot P_1 + P_2 \ . \end{aligned}$$

$$\frac{}{P^* \downarrow} \qquad \frac{P \xrightarrow{a} P'}{P^* \xrightarrow{a} P' \cdot P^*}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 {}^\sharp P_2 \xrightarrow{a} P_1' \cdot (P_1 {}^\sharp P_2) \cdot P_1} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1 {}^\sharp P_2 \xrightarrow{a} P_2'} \qquad \frac{P_2 \downarrow}{P_1 {}^\sharp P_2 \downarrow} \; .$$

Table 4.3: The operational semantics of nesting and iteration

Note that for these operators, having $P \downarrow$ or $P_1 \downarrow$ does not give rise to additional behaviour.

4.2 Transparency

Process expressions that have the option to terminate are said to be *transparent* in a sequential context: if P has the option to terminate and $Q \xrightarrow{a} Q'$, then $P \cdot Q \xrightarrow{a} Q'$ even if P can still do transitions. In this section we shall explain how transparency gives rise to two phenomena that are undesirable in certain circumstances. First, it facilitates the specification of unbounded branching behaviour with a guarded recursive specification over TSP. Second, it gives rise to forgetful stacking of variables, and as a consequence it is not clear how to specify an always terminating half-counter in TCP$^\sharp$.

We first discuss process expressions with unbounded branching. It is well-known from formal language theory that the context-free languages are exactly the languages accepted by pushdown automata. The process-theoretic formulation of this result is that every transition system specified by a TSP specification is language equivalent to the transition system associated with a pushdown automaton and, vice versa, every transition system associated with a pushdown automaton is language equivalent to the transition system associated with some TSP specification. The correspondence fails, however, when language equivalence is replaced by (strong) bisimilarity. The tightest result currently known is that for every context-free process there is a pushdown process to simulate it modulo contrasimulation [9] (the notion of context-free processes and pushdown processes are defined in Section 4.4); but we do not know whether there are stronger behavioural equivalences in the spectrum of [40] for which a simulation exists. Hence, we conjecture that not every context-free process is simulated by a pushdown process modulo branching bisimilarity. An observation is that context-free

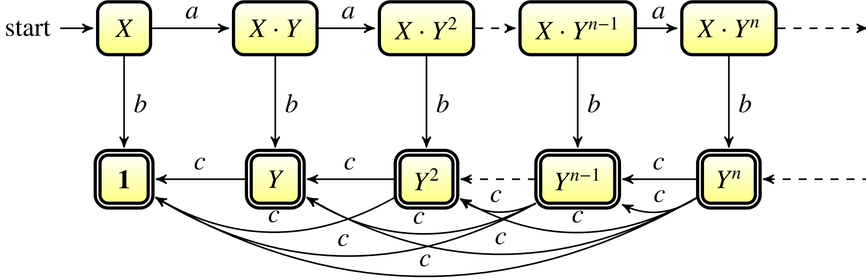


Figure 4.1: A transition system with unbounded branching behaviour

processes may have an unbounded branching degree.

Example 4.4 (unbounded branching)**.** Consider the following process:

$$\begin{aligned} X &= a.X \cdot Y + b.\mathbf{1} \\ Y &= c.\mathbf{1} + \mathbf{1} \ . \end{aligned}$$

The transition system associated with X is illustrated in Figure 4.1. Note that every state in the second row is a terminating state. The state Y^n has n c-labelled transitions to $\mathbf{1}, Y, Y^2, \ldots, Y^{n-1}$, respectively. Therefore, every state in this transition system has finitely many transitions leading to distinct states, but there is no upper bound on the number of transitions from each state. Therefore, we say that this transition system has an unbounded branching degree.

Note that the process defined by the TSP specification above is not strongly bisimilar to a pushdown process since it has an unbounded branching degree, whereas a pushdown process is always boundedly branching. The correspondence does hold modulo contrasimulation [9], and it is an open problem as to whether the correspondence holds modulo branching bisimilarity. In Section 4.4, we show that with a revised operational semantics for sequential composition, we eliminate such unbounded branching and indeed obtain a correspondence between pushdown processes and context-free processes modulo strong bisimilarity.

Next, we discuss the phenomenon of forgetfulness. Bergstra et al. show how one can specify a half counter using iteration and nesting, which then allows them to conclude that the behaviour of a Turing machine can be simulated in the calculus with iteration and nesting (not including recursion) [18, 21].

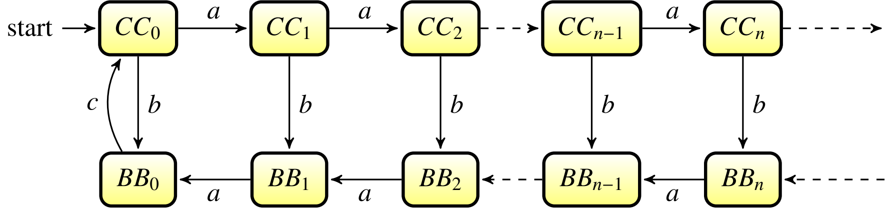


Figure 4.2: The transition system of a half counter

The half counter is specified as follows:

$$\begin{aligned} CC_n &= a.CC_{n+1} + b.BB_n \ (n \in \mathbb{N}) \\ BB_n &= a.BB_{n-1} \ (n \geq 1) \\ BB_0 &= c.CC_0 \ . \end{aligned}$$

The behaviour of a half counter is shown in Figure 4.2. The initial state is CC_0. From CC_0 an arbitrary number of a transitions is possible. After a b-labelled transition, the process performs the same number of a-labelled transitions as before the b-labelled transition, to the state BB_0. In state BB_0, a zero testing transition labelled by c is enabled, leading back to the state CC_0.

An implementation in a calculus with iteration and nesting is provided in [21] as follows.

Example 4.5 (An implementation of the half counter)**.** Consider the following TCP$^\sharp$ process HCC defined by:

$$HCC = ((a^\sharp b) \cdot c)^* \ .$$

We briefly explain the behaviour of the process HCC. We let:

$$\begin{aligned} HCC_n &= (a^\sharp b \cdot a^n \cdot c) \cdot HCC, \\ HBB_n &= a^n \cdot c \cdot HCC \ . \end{aligned}$$

Then its transition system is shown in Figure 4.3. It is straightforward to establish that $((a^\sharp b) \cdot a^n \cdot c) \cdot HCC$ is equivalent to CC_n for all $n \geq 1$ modulo strong bisimilarity, and $(a^n \cdot c) \cdot HCC$ is equivalent to BB_n for all $n \in \mathbb{N}$ modulo strong bisimilarity.

In a context with intermediate termination, one may wonder if it is possible to generalize their result. It is, however, not clear how to specify an always terminating half counter. At least, a naive generalisation of the specification of Bergstra et al. does not do the job. The culprit is forgetfulness.

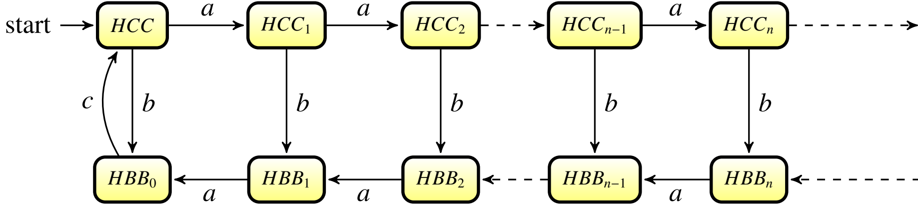


Figure 4.3: An implementation of the half counter in TCP$^\sharp$

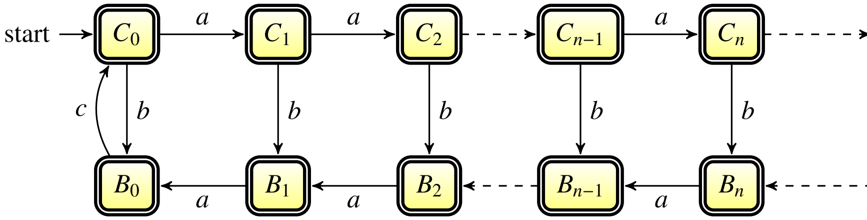


Figure 4.4: The transition system of an always terminating half counter

We define a half counter that terminates in every state as follows:

$$\begin{aligned} C_n &= a.C_{n+1} + b.B_n + \mathbf{1} \quad (n \in \mathbb{N}) \\ B_n &= a.B_{n-1} + \mathbf{1} \quad (n \geq 1) \\ B_0 &= c.C_0 + \mathbf{1} \ . \end{aligned}$$

Its behaviour is illustrated in Figure 4.4. In the following example, we show that due to the phenomenon of forgetfulness, a naive generalization of the implementation from Example 4.5 fails.

Example 4.6 (forgetfulness)**.** Now consider the process HC defined by:

$$HC = ((a + \mathbf{1}).^\sharp(b + \mathbf{1}). \cdot (c + \mathbf{1}))^* \ .$$

We let

$$\begin{aligned} HC_n &= ((a + \mathbf{1})^\sharp(b + \mathbf{1}) \cdot (a + \mathbf{1})^n \cdot (c + \mathbf{1})) \cdot HC, \\ HB_n &= (a + \mathbf{1})^n \cdot (c + \mathbf{1}) \cdot HCC \ . \end{aligned}$$

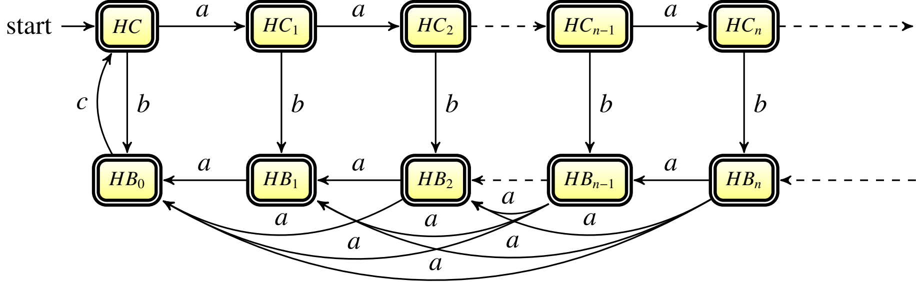


Figure 4.5: A failed implementation of the always terminating half counter in TCP$^\sharp$

Its transition system is illustrated in Figure 4.5. Note that due to transparency, $((a+\mathbf{1})^n\cdot(c+\mathbf{1}))\cdot HC$ is not equivalent to B_n modulo any reasonable behavioural equivalence for $n > 1$ since B_n only has an a-labelled transition to B_{n-1} whereas the other process has at least $n+1$ transitions leading to HC, $(c+\mathbf{1})\cdot HC$, $(a+\mathbf{1})\cdot(c+\mathbf{1})\cdot HC$, ..., $(a+\mathbf{1})^{n-1}\cdot(c+\mathbf{1})\cdot HC$, respectively. This process may choose to "forget" the transparent process expressions that have been stacked using the sequential composition operator. We conjecture that, due to forgetfulness, the always terminating half counter cannot be specified in TCP$^\sharp$.

In Section 4.5, we show that with the revised semantics, it is possible to specify an always terminating half counter and we shall prove that TCP extended with $*$ and $\sharp$ (but without recursion) is reactively Turing powerful.

4.3 A Revised Semantics of the Sequential Composition Operator

In this section, we revise the operational semantics for sequential composition and propose a calculus TCP$^;$. The revised operational semantics resembles similar operators for sequencing discussed in [2] and [23]. We obtain the calculus by replacing the sequential composition operator $\cdot$ by ; in the syntax of TCP. Note that we also use the abbreviation of $P^{n;}$ as we did for the standard version of the sequential composition operator.

The operational rules for ; are given in Table 4.4.

Note that the third rule has a negative premise $P_1 \not\longrightarrow$. Intuitively, this rule is only applicable if there does not exist a closed term P_1' and an action $a \in \mathcal{A}_\tau$ such that

$$\frac{P_1 \downarrow \quad P_2 \downarrow}{P_1;P_2 \downarrow} \qquad \frac{P_1 \xrightarrow{a} P_1'}{P_1;P_2 \xrightarrow{a} P_1';P_2} \qquad \frac{P_1 \downarrow \quad P_2 \xrightarrow{a} P_2' \quad P_1 \not\longrightarrow}{P_1;P_2 \xrightarrow{a} P_2'} .$$

Table 4.4: The revised semantics of sequential composition

the transition $P_1 \xrightarrow{a} P_1'$ is derivable. In the operational semantics, we use negative premises, which has been discussed in [48, 27]. In order to obtain a sound semantics, we need to establish a *stratification* for the transitions [27]. For this purpose, we have to restrict the calculus to guarded recursive specifications. In this section, we provide a stratification for the calculus $\mathrm{TSP}^{;}$. We reformulate the idea in [27] by giving a stratification to the process terms.

Definition 4.7 (Stratification)**.** Let $\mathcal{P}$ be the set of process terms, A function $S : \mathcal{P} \rightarrow \alpha$ where α is an ordinal, is called a *stratification* of $\mathcal{P}$, if for every rule r in the operational semantics, and every substitution $\sigma : V \rightarrow \mathcal{P}$ where V is the set of variables, it holds that:

$$\begin{array}{lcl} \text{for all } P_1 \in \mathit{pprem}(\sigma(r)) & : & S(P_1) \leq S(\mathit{conc}(\sigma(r))) \text{ and} \\ \text{for all } P_1 \in \mathit{nprem}(\sigma(r)) & : & S(P_1) < S(\mathit{conc}(\sigma(r))) \end{array}$$

We use *pprem* to denote the set of terms appears on the left-hand side in a positive premises of a rule, *nprem* to denote the set of terms appears on the left-hand-side in a negative premises of a rule, and *conc* to denote the set of terms appears on the left-hand-side in a conclusion a rule. We apply the above notation to arbitrary substitutions of a rule.

Lemma 4.8 (Stratification for $\mathrm{TSP}^{;}$)**.** *There exists a stratification for guarded recursive specifications in* $\mathrm{TSP}^{;}$.

Proof. We hereby give a stratification to the terms in TSP$^{;}$.

$$\begin{aligned} S(\mathbf{0}) &= 0 \\ S(\mathbf{1}) &= 0 \\ S(a.P) &= 0 \\ S(P_1 + P_2) &= S(P_1) + S(P_2) + 1 \\ S(P_1;P_2) &= S(P_1) + S(P_2) + 1 \\ S(X) &= S(P),\ (X \stackrel{\text{def}}{=} P \in N) \end{aligned}$$

It is straightforward from Definition 4.7 that S is a valid stratification for guarded recursive specifications in TSP$^{;}$. Note that for all terms X with $X \stackrel{\text{def}}{=} P \in N$), we have $S(X) = 0$. We do not have this property for unguarded recursive specifications. □

We can easily convert the above stratification on process terms to adapt the definition of the stratification on transitions in [27] by giving the stratification of a transition by the process term on its left-hand-side. Hence, we may conclude that the revised semantics is well-founded for guarded recursive specifications by Theorem 6.1 in [27]. Moreover, there is also a sound formalisation of this intuition, using the notions of irredundant and well-supported proof, see Section 5 of [41].

As a consequence of the revised semantics, the branching degree of a context-free process is bounded and sequential compositions may have the option to terminate, without being forgetful. We illustrate this idea by the following example.

Example 4.9 (a transition system in the revised semantics)**.** Let us revisit Example **??**. We rewrite it with the revised sequential composition operator:

$$\begin{aligned} X &= a.X;Y + b.\mathbf{1} \\ Y &= c.\mathbf{1} + \mathbf{1}\ . \end{aligned}$$

Its transition system is illustrated in Figure 4.6. Every state in the transition system now has a bounded branching degree. For instance, compared with Figure 4.1, a transition from $Y^{5;}$ to $Y^{2;}$ is abandoned because Y has a transition and only the transition from the first Y in the sequential composition is allowed.

Congruence is an important property to fit a behavioural equivalence into an axiomatic framework. We start from investigating the congruence property of $\leftrightarrows$ with respect to TCP$^{;}$.

Theorem 4.10 ($\leftrightarrows$ is a congruence)**.** *$\leftrightarrows$ is a congruence with respect to TCP$^{;}$.*

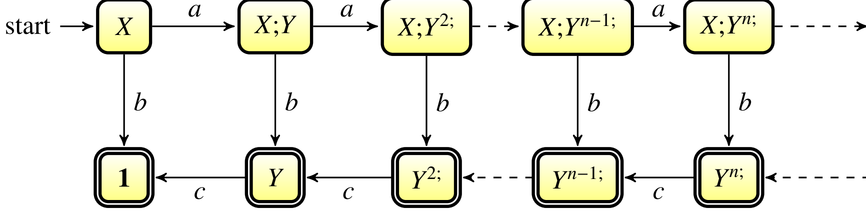


Figure 4.6: A transition system in the revised semantics

Proof. We show that $\leftrightarroweq$ is compatible for each operator $a.,+,;,\|$. For simplicity, we omit the symmetrical cases and the verification of termination condition.

1. Suppose that $P \leftrightarroweq Q$; we show that $a.P \leftrightarroweq a.Q$. To this end, we verify that $\mathcal{R} = \{(a.P, a.Q) \mid P \leftrightarroweq Q\} \cup \leftrightarroweq$ is a strong bisimulation.

   We let $a.P\mathcal{R}a.Q$, then $P \leftrightarroweq Q$; the transition $a.P \xrightarrow{a} P$ is simulated by $a.Q \xrightarrow{a} Q$ with $P\mathcal{R}Q$. Therefore, $\mathcal{R}$ is a strong bisimulation.

2. Suppose that $P_1 \leftrightarroweq Q_1$ and $P_2 \leftrightarroweq Q_2$; we show that $P_1 + P_2 \leftrightarroweq Q_1 + Q_2$. To this end, we verify that $\mathcal{R} = \{(P_1 + P_2, Q_1 + Q_2) \mid P_1 \leftrightarroweq Q_1, P_2 \leftrightarroweq Q_2\} \cup \leftrightarroweq$ is a strong bisimulation.

   We let $P_1+P_2\mathcal{R}Q_1+Q_2$, then we only consider $P_1 \leftrightarroweq Q_1$ and $P_2 \leftrightarroweq Q_2$ (since the other case is trivial). A transition $P_1 + P_2 \xrightarrow{a} P'$ is simulated by $Q_1 + Q_2 \xrightarrow{a} Q'$ in one of the following cases:

   (a) if $P_1 \xrightarrow{a} P'$, then $Q_1 \xrightarrow{a} Q'$ with $P' \leftrightarroweq Q'$; or

   (b) if $P_2 \xrightarrow{a} Q'$, then $Q_2 \xrightarrow{a} Q'$ with $P' \leftrightarroweq Q'$.

   In both cases, we have $P'\mathcal{R}Q'$; so $\mathcal{R}$ is a strong bisimulation.

3. Suppose that $P_1 \leftrightarroweq Q_1$ and $P_2 \leftrightarroweq Q_2$; we show that $[P_1 \parallel P_2]_{C'} \leftrightarroweq [Q_1 \parallel Q_2]_{C'}$. To this end, we verify that $\mathcal{R} = \{([P_1 \parallel P_2]_{C'}, [Q_1 \parallel Q_2]_{C'}) \mid P_1 \leftrightarroweq Q_1, P_2 \leftrightarroweq Q_2\} \cup \leftrightarroweq$ is a strong bisimulation.

   We let $[P_1 \parallel P_2]_{C'}\mathcal{R}[Q_1 \parallel Q_2]_{C'}$, then we only consider $P_1 \leftrightarroweq P_2$ and $Q_1 \leftrightarroweq Q_2$ (since the other case is trivial). A transition $[P_1 \parallel P_2]_{C'} \xrightarrow{a} P'$ is simulated by $[Q_1 \parallel Q_2]_{C'} \xrightarrow{a} Q'$ in one of the following cases:

   (a) if $P_1 \xrightarrow{a} P_1'$, $a \notin \mathcal{I}_{C'}$, then we have $Q_1 \xrightarrow{a} Q_1'$ with $P_1' \leftrightarroweq Q_1'$, and $Q' = [Q_1' \parallel Q_2]_{C'}$; or

(b) if $P_2 \xrightarrow{a} P_2'$, $a \notin \mathcal{I}_{C'}$, then we have $Q_2 \xrightarrow{a} Q_2'$ with $P_2' \leftrightarroweq Q_2'$, and $Q' = [Q_1 \parallel Q_2']_{C'}$; or

(c) if $P_1 \xrightarrow{c?d} P_1'$ and $P_2 \xrightarrow{c!d} P_2'$, then we have $Q_1 \xrightarrow{c?d} Q_1'$ and $Q_2 \xrightarrow{c!d} Q_2'$ with $P_1' \leftrightarroweq Q_1'$ and $P_2' \leftrightarroweq Q_2'$, moreover, $a = \tau$, and $Q' = [Q_1' \parallel Q_2']_{C'}$; or

(d) if $P_1 \xrightarrow{c!d} P_1'$ and $P_2 \xrightarrow{c?d} P_2'$, then we have $Q_1 \xrightarrow{c!d} Q_1'$ and $Q_2 \xrightarrow{c?d} Q_2'$ with $P_1' \leftrightarroweq Q_1'$ and $P_2' \leftrightarroweq Q_2'$, moreover, $a = \tau$, and $Q' = [Q_1' \parallel Q_2']_{C'}$.

In all cases, we have $P'\mathcal{R}Q'$, so $\mathcal{R}$ is a strong bisimulation.

4. Suppose that $P_1 \leftrightarroweq Q_1$ and $P_2 \leftrightarroweq Q_2$; we show that $P_1;P_2 \leftrightarroweq Q_1;Q_2$. To this end, we verify that $\mathcal{R} = \{(P_1;P_2, Q_1;Q_2) \mid P_1 \leftrightarroweq Q_1, P_2 \leftrightarroweq Q_2\} \cup \leftrightarroweq$ is a strong bisimulation.

   We let $P_1;P_2\mathcal{R}Q_1;Q_2$) then we only consider $P_1 \leftrightarroweq Q_1$ and $P_2 \leftrightarroweq Q_2$ (since the other case is trivial). A transition $P_1;P_2 \xrightarrow{a} P'$ is simulated by $Q_1;Q_2 \xrightarrow{a} Q'$ in one of the following cases:

   (a) if $P_1 \xrightarrow{a} P_1'$, then we have $Q_1 \xrightarrow{a} Q_1'$ with $P_1' \leftrightarroweq Q_1'$, and $Q' = Q_1';Q_2$; or

   (b) if $P_1 \not\longrightarrow$, $P_1 \downarrow$ and $P_2 \xrightarrow{a} P_2'$, then we have $Q_1 \not\longrightarrow$, $Q_1 \downarrow$ and $Q_2 \xrightarrow{a} Q_2'$ with $P_2' \leftrightarroweq Q_2'$, and $Q' = Q_2'$.

   In both cases, we have $P'\mathcal{R}Q'$, so $\mathcal{R}$ is a strong bisimulation.

To conclude, $\leftrightarroweq$ is a congruence for TCP$^{;}$. □

Strong bisimulation is considered to be too strong for many cases in concurrency theory. In this thesis, we are more interested in the congruence property for branching bisimulation. However, we observe that the congruence property does not hold in the absence of either the rootedness condition or the divergence-preserving condition. Rootedness condition is required for the nondeterministic choice operator, and divergence-preserving condition is required for the revised sequential composition operator. We have that in the revised semantics, $\leftrightarroweq_{\mathrm{rb}}^{\Delta}$ is a congruence. Note that the congruence property can also be inferred from a recent result of Fokkink, van Glabbeek and Luttik [35].

Theorem 4.11 ($\leftrightarroweq_{\mathrm{rb}}^{\Delta}$ is a congruence)**.** *$\leftrightarroweq_{rb}^{\Delta}$ is a congruence with respect to TCP$^{;}$.*

Proof. We use the following facts:

1. Rooted divergence-preserving branching bisimilarity is also a rooted divergence-preserving branching bisimulation relation since rooted divergence-preserving branching bisimilarity is defined to be the largest rooted divergence-preserving branching bisimulation.

2. Rooted divergence-preserving branching bisimilarity is a subset of divergence-preserving branching bisimilarity since rooted divergence-preserving branching bisimilarity is a rooted divergence-preserving branching bisimulation and therefore it is a divergence-preserving branching bisimulation.

We show that $\underline{\leftrightarrow}^{\Delta}_{rb}$ is compatible for each operator $a.$, $+$, $;$, $\|$.

1. Suppose that $P \underline{\leftrightarrow}^{\Delta}_{rb} Q$; we show that $a.P \underline{\leftrightarrow}^{\Delta}_{rb} a.Q$. To this end, we verify that $\mathcal{R} = \{(a.P, a.Q) \mid P \underline{\leftrightarrow}^{\Delta}_{rb} Q\} \cup \underline{\leftrightarrow}^{\Delta}_{rb}$ is a rooted divergence-preserving branching bisimulation.

   To prove that the pair $(a.P, a.Q)$ with P rooted divergence-preserving branching bisimilar to Q satisfies condition 1 of Definition 2.7, suppose that $a.P \xrightarrow{b} P'$. Then, according to the operational semantics, $b = a$ and $P' = P$. By the operational semantics, we also have that $a.Q \xrightarrow{a} Q$ and, by assumption, P and Q are divergence-preserving branching bisimilar.

   For the termination condition, it is trivially satisfied since both processes do not terminate. The divergence-preserving condition of $\mathcal{R}$ is also trivially satisfied since only an a-labelled transition is allowed from both processes.

2. Suppose that $P_1 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1$ and $P_2 \underline{\leftrightarrow}^{\Delta}_{rb} Q_2$; we show that $P_1 + P_2 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1 + Q_2$. To this end, we verify that $\mathcal{R} = \{(P_1 + P_2, Q_1 + Q_2) \mid P_1 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1,\ P_2 \underline{\leftrightarrow}^{\Delta}_{rb} Q_2\} \cup \underline{\leftrightarrow}^{\Delta}_{rb}$ is a rooted divergence-preserving branching bisimulation relation.

   Suppose that $P_1 + P_2 \xrightarrow{a} P'$; then we have $P_1 \xrightarrow{a} P'$ or $P_2 \xrightarrow{a} P'$. We only consider the first case. Since $P_1 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1$, we have $Q_1 \xrightarrow{a} Q'$ with $P' \underline{\leftrightarrow}^{\Delta}_{b} Q'$. Then we have $Q_1 + Q_2 \xrightarrow{a} Q'$ with $P' \underline{\leftrightarrow}^{\Delta}_{b} Q'$. The same argument holds for the symmetrical case.

   If $P_1 + P_2 \downarrow$, then we have either $P_1 \downarrow$ or $P_2 \downarrow$. Without loss of generality, we suppose that $P_1 \downarrow$. Since $P_1 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1$, we have $Q_1 \downarrow$. Therefore, $Q_1 + Q_2 \downarrow$.

   Moreover, the divergence preservation condition is trivially satisfied.

   Hence, $\mathcal{R}$ is a rooted divergence-preserving branching bisimulation relation.

3. Suppose that $P_1 \underline{\leftrightarrow}^{\Delta}_{rb} Q_1$ and $P_2 \underline{\leftrightarrow}^{\Delta}_{rb} Q_2$; we show that $[P_1 \parallel P_2]_{C'} \underline{\leftrightarrow}^{\Delta}_{rb} [Q_1 \parallel Q_2]_{C'}$. To this end, we verify that $\mathcal{R} = \{([P_1 \parallel P_2]_{C'}, [Q_1 \parallel Q_2]_{C'}) \mid P_1 \underline{\leftrightarrow}^{\Delta}_{rb}$

Q_1, $P_2 \leftrightarroweq^{\Delta}_{rb} Q_2\} \cup \leftrightarroweq^{\Delta}_{rb}$ is a rooted divergence-preserving branching bisimulation relation.

We first show that $\mathcal{R}' = \{([P_1 \parallel P_2]_{C'}, [Q_1 \parallel Q_2]_{C'}) \mid P_1 \leftrightarroweq^{\Delta}_{b} Q_1, P_2 \leftrightarroweq^{\Delta}_{b} Q_2\} \cup \leftrightarroweq^{\Delta}_{b}$ is a divergence-preserving branching bisimulation.

Suppose that $[P_1 \parallel P_2]_{C'} \xrightarrow{a} P'$; then we distinguish several cases.

(a) If $P_1 \xrightarrow{a} P_1'$, $a \notin \mathcal{I}_{C'}$ and $P' = [P_1' \parallel P_2]_{C'}$, then, since $P_1 \leftrightarroweq^{\Delta}_{b} Q_1$, we have $Q_1 \longrightarrow^{*} Q_1'' \xrightarrow{a} Q_1'$ with $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$ and $P_1 \leftrightarroweq^{\Delta}_{b} Q_1''$. Then we have $[Q_1 \parallel Q_2]_{C'} \longrightarrow^{*} [Q_1'' \parallel Q_2]_{C'} \xrightarrow{a} [Q_1' \parallel Q_2]_{C'}$ with $P_1 \leftrightarroweq^{\Delta}_{b} Q_1''$, $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$ and $P_2 \leftrightarroweq^{\Delta}_{b} Q_2$. Thus we have $([P_1' \parallel P_2]_{C'}, [Q_1' \parallel Q_2]_{C'}) \in \mathcal{R}'$ and $([P_1 \parallel P_2]_{C'}, [Q_1'' \parallel Q_2]_{C'}) \in \mathcal{R}'$.

(b) If $P_1 \xrightarrow{c?d} P_1'$, $P_2 \xrightarrow{c!d} P_2'$ and $c \in C'$, then $[P_1 \parallel P_2]_{C'} \xrightarrow{\tau} [P_1' \parallel P_2']_{C'}$. Since $P_1 \leftrightarroweq^{\Delta}_{b} Q_1$ and $P_2 \leftrightarroweq^{\Delta}_{b} Q_2$, we have $Q_1 \longrightarrow^{*} Q_1'' \xrightarrow{c?d} Q_1'$, $Q_2 \longrightarrow^{*} Q_2'' \xrightarrow{c!d} Q_2'$ with $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$, $P \leftrightarroweq^{\Delta}_{b} Q_1''$, $P_2' \leftrightarroweq^{\Delta}_{b} Q_2'$, and $P_2 \leftrightarroweq^{\Delta}_{b} Q_2''$. Then we have $[Q_1 \parallel Q_2]_{C'} \longrightarrow^{*} [Q_1'' \parallel Q_2'']_{C'} \xrightarrow{\tau} [Q_1' \parallel Q_2']_{C'}$ with $P_1 \leftrightarroweq^{\Delta}_{b} Q_1''$, $P_2 \leftrightarroweq^{\Delta}_{b} Q_2''$, $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$ and $P_2' \leftrightarroweq^{\Delta}_{b} Q_2'$. Thus we have $([P_1 \parallel P_2]_{C'}, [Q_1'' \parallel Q_2'']_{C'}) \in \mathcal{R}'$ and $([P_1' \parallel P_2']_{C'}, [Q_1' \parallel Q_2']_{C'}) \in \mathcal{R}'$.

If $[P_1 \parallel P_2]_{C'} \downarrow$, then we have $P_1 \downarrow$ and $P_2 \downarrow$. Since $P_1 \leftrightarroweq^{\Delta}_{b} Q_1$ and $P_2 \leftrightarroweq^{\Delta}_{b} Q_2$, we have $Q_1 \longrightarrow^{*} Q_1' \downarrow$ and $Q_2 \longrightarrow^{*} Q_2' \downarrow$ for some Q_1' and Q_2'. Therefore, $[Q_1 \parallel Q_2]_{C'} \longrightarrow^{*} [Q_1' \parallel Q_2']_{C'} \downarrow$.

Moreover, the divergence-preserving condition is trivially satisfied. Hence, we have $\mathcal{R}'$ is a divergence-preserving branching bisimulation relation.

Now we show that $\mathcal{R}$ is a rooted divergence-preserving branching bisimulation. Suppose that $[P_1 \parallel P_2]_{C'} \xrightarrow{a} P'$; then we distinguish several cases.

(a) If $P_1 \xrightarrow{a} P_1'$ $a \notin \mathcal{I}_{C'}$ and $P' = [P_1' \parallel P_2]_{C'}$, then, since $P_1 \leftrightarroweq^{\Delta}_{rb} Q_1$, we have $Q_1 \xrightarrow{a} Q_1'$ with $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$. Then we have $[Q_1 \parallel Q_2]_{C'} \xrightarrow{a} [Q_1' \parallel Q_2]_{C'}$ with $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$ and $P_2 \leftrightarroweq^{\Delta}_{b} Q_2$. Thus we have $[P_1' \parallel P_2]_{C'} \leftrightarroweq^{\Delta}_{b} [Q_1' \parallel Q_2]_{C'}$.

(b) If the transition is derived from a communication between the parallel components, i.e., $P_1 \xrightarrow{c?d} P_1'$, $P_2 \xrightarrow{c!d} P_2'$ and $c \in C'$, then $[P_1 \parallel P_2]_{C'} \xrightarrow{\tau} [P_1' \parallel P_2']_{C'}$. Since $P_1 \leftrightarroweq^{\Delta}_{rb} Q_1$ and $P_2 \leftrightarroweq^{\Delta}_{rb} Q_2$, we have $Q_1 \xrightarrow{c?d} Q_1'$, $Q_2 \xrightarrow{c!d} Q_2'$ with $P_1' \leftrightarroweq^{\Delta}_{b} Q_1'$ and $P_2' \leftrightarroweq^{\Delta}_{b} Q_2'$. Then we have $[Q_1 \parallel Q_2]_{C'} \xrightarrow{\tau} [Q_1' \parallel$

$Q'_2]_{C'}$ with $P'_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$ and $P'_2 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_2$. Thus we have $[P'_1 \parallel P'_2]_{C'} \,\underline{\leftrightarrow}^{\Delta}_{b}\, [Q'_1 \parallel Q'_2]_{C'}$.

If $[P_1 \parallel P_2]_{C'} \downarrow$, then we have $P_1 \downarrow$ and $P_2 \downarrow$. Since $P_1 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_1$ and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$, we have $Q_1 \downarrow$ and $Q_2 \downarrow$. Therefore, $[Q_1 \parallel Q_2]_{C'} \downarrow$.

Moreover, the divergence-preserving condition is trivially satisfied.

Hence, we have $\mathcal{R}$ is a rooted divergence-preserving branching bisimulation relation.

4. Suppose that $P_1 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_1$ and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$; we show that $P_1;P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_1;Q_2$. To this end, we verify that $\mathcal{R} = \{(P_1;P_2, Q_1;Q_2) \mid P_1 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_1,\ P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2\} \cup \underline{\leftrightarrow}^{\Delta}_{rb}$ is a rooted divergence-preserving branching bisimulation relation.

   We first show that $\mathcal{R}' = \{(P_1;P_2, Q_1;Q_2) \mid P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q_1,\ P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2\} \cup \underline{\leftrightarrow}^{\Delta}_{b}$ is a divergence-preserving branching bisimulation relation.

   Suppose that $P_1;P_2 \xrightarrow{a} P'$; then we distinguish several cases.

   (a) If $P_1 \xrightarrow{a} P'_1$, then $P' = P'_1;P_2$. Since $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q_1$, we have $Q_1 \longrightarrow^* Q''_1 \xrightarrow{a} Q'_1$ with $P'_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$ and $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q''_1$. Then we have $Q_1;Q_2 \longrightarrow^* Q''_1;Q_2 \xrightarrow{a} Q'_1;Q_2$ with $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q''_1$, $P'_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$, and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$. Thus, we have $(P'_1;P_2, Q'_1;Q_2) \in \mathcal{R}'$ and $(P_1;P_2, Q''_1;Q_2) \in \mathcal{R}'$.

   (b) If $P_1 \downarrow$, $P_2 \xrightarrow{a} P'_2$ and $P_1 \not\longrightarrow$. Since $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q_1$ and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$, we have $Q_1 \longrightarrow^* Q'_1 \downarrow$, $Q'_1 \not\longrightarrow$ for some Q'_1 with $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$, and $Q_2 \xrightarrow{a} Q'_2$, with $P'_2 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_2$. Then, we have $Q_1;Q_2 \longrightarrow^* Q'_1;Q_2 \xrightarrow{a} Q'_2$ with $P'_2 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_2$ and $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$. Thus we have $(P'_2, Q'_2) \in \mathcal{R}'$ and $(P_1;P_2, Q'_1;Q_2) \in \mathcal{R}$.

   If $P_1;P_2 \downarrow$, then we have $P_1 \downarrow$ and $P_2 \downarrow$. Since $P_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q_1$ and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$, we have $Q_1 \longrightarrow^* Q'_1 \downarrow$ for some Q'_1 and $Q_2 \downarrow$. Therefore, $Q_1;Q_2 \longrightarrow^* Q'_1;Q_2 \downarrow$.

   Moreover, the divergence preservation condition trivially is satisfied since no infinite τ-transition sequence is introduced in the simulation.

   Hence, $\mathcal{R}$ is a divergence-preserving branching bisimulation relation.

   Now we show that $\mathcal{R}$ is a rooted divergence-preserving branching bisimulation relation.

   We suppose that $P_1;P_2 \xrightarrow{a} P'$, and we distinguish several cases:

   (a) If $P_1 \xrightarrow{a} P'_1$, then $P' = P'_1;P_2$. Since $P_1 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_1$, we have $Q_1 \xrightarrow{a} Q'_1$ with $P'_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$. Then we have $Q_1;Q_2 \xrightarrow{a} Q'_1;Q_2$ with $P'_1 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1$ and $P_2 \,\underline{\leftrightarrow}^{\Delta}_{rb}\, Q_2$. Thus, we have $P'_1;P_2 \,\underline{\leftrightarrow}^{\Delta}_{b}\, Q'_1;Q_2$.

(b) If $P_1 \downarrow$, $P_2 \xrightarrow{a} P_2'$ and $P_1 \not\longrightarrow$. Since $P_1 \leftrightarrow_{rb}^{\Delta} Q_1$ and $P_2 \leftrightarrow_{rb}^{\Delta} Q_2$, we have $Q_1 \downarrow$, $Q_2 \xrightarrow{a} Q_2'$, with $P_2' \leftrightarrow_{b}^{\Delta} Q_2'$, and $Q_1 \not\longrightarrow$. Then, we have $Q_1;Q_2 \xrightarrow{a} Q_2'$ with $P_2' \leftrightarrow_{b}^{\Delta} Q_2'$.

If $P_1;P_2 \downarrow$, then we have $P_1 \downarrow$ and $P_2 \downarrow$. Since $P_1 \leftrightarrow_{rb}^{\Delta} Q_1$ and $P_2 \leftrightarrow_{rb}^{\Delta} Q_2$, we have $Q_1 \downarrow$ and $Q_2 \downarrow$. Therefore, $Q_1;Q_2 \downarrow$.

Moreover, we verify that the divergence preservation condition is satisfied.

Hence, we have $\mathcal{R}$ is a rooted divergence-preserving branching bisimulation relation.

□

Unlike the divergence-preserving variant of rooted branching bisimilarity, the more standard variant that does not require divergence-preservation ($\leftrightarrow_{rb}$) is not a congruence for TCP$^{;}$. We give a counterexample as follows.

Example 4.12 ($\leftrightarrow_{rb}$ is not a congruence for TCP$^{;}$.)**.** Consider the following TCP$^{;}$ process:

$$\begin{aligned} P_1 &= \tau.\mathbf{1} \\ P_2 &= \tau.P_2 \\ Q &= a.\mathbf{1} \ . \end{aligned}$$

We have $P_1 \leftrightarrow_{rb} P_2$ but not $P_1;Q \leftrightarrow_{rb} P_2;Q$, for $P_1;Q$ can do an a-transition after the τ-transitions, whereas $P_2;Q$ can only do τ transitions.

We also define a version of TCP with iteration and nesting (TCP$^{\sharp}$) in the revised semantics. By removing the facility of recursive specification and adding the operations $*$ and $\sharp$, we get TCP$^{\sharp}$. The operational rules for $*$ and $\sharp$ are obtained by replacing, in the rules in Figure 4.3, all occurrences of $\cdot$ by ;, see Table 4.5.

4.4 Context-free Processes and Pushdown Process

The relationship between context-free processes and pushdown processes has been studied in the literature [30, 9]. We consider the process calculus *Theory of Sequential Processes* (TSP$^{;}$). We define context-free processes as follows:

Definition 4.13 (context-free processes)**.** A *context-free process* is the strong bisimulation equivalence class of the transition system generated by a finite guarded recursive specification over TSP$^{;}$.

$$\frac{}{P^* \downarrow} \qquad \frac{P \xrightarrow{a} P'}{P^* \xrightarrow{a} P';P^*}$$

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1{}^\sharp P_2 \xrightarrow{a} P_1';(P_1{}^\sharp P_2);P_1} \qquad \frac{P_2 \xrightarrow{a} P_2'}{P_1{}^\sharp P_2 \xrightarrow{a} P_2'} \qquad \frac{P_2 \downarrow}{P_1{}^\sharp P_2 \downarrow}$$

Table 4.5: The revised semantics of iteration and nesting

Note that there is a method to rewrite every context-free process into Greibach normal form [6], which is also valid in the revised semantics. In this paper, we only consider context-free processes in Greibach normal form , i.e., defined by guarded recursive specifications of the form

$$X = \sum_{i \in I_X} \alpha_i.\xi_i (+\mathbf{1}) \ .$$

In this form, every right-hand side of every equation consists of a number of summands, indexed by a finite set I_X (the empty sum denotes $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $\alpha_i.\xi_i$, where ξ_i is the sequential composition of names (the empty sequence denotes $\mathbf{1}$).

It is well-known that context-free grammars and pushdown automata generate the same class of languages [54]. In concurrency theory, we focus on correspondence modulo other behavioural equivalence relations like bisimulation. We shall show that every context-free process is equivalent to a pushdown process modulo strong bisimilarity. The notion of pushdown automaton is defined as follows:

Definition 4.14 (pushdown automata)**.** A *pushdown automaton (PDA)* is a 7-tuple $(Q, \Sigma, \mathcal{D}, \mapsto, Ini, Z, Fin)$, where

1. Q is a finite set of *states*,
2. Σ is a finite set of *input symbols*,
3. $\mathcal{D}$ is a finite set of *stack symbols*,
4. $\mapsto \subseteq \mathcal{S} \times \mathcal{D} \times \Sigma \times \mathcal{D}^* \times \mathcal{S}$ is a finite *transition relation*, (we write $s \xmapsto{a[d/\delta]} t$ for $(s, d, a, \delta, t) \in \mapsto$),

5. $\mathit{Ini} \in \mathcal{S}$ is the *initial state*,

6. $Z \in \mathcal{D}$ is the *initial stack symbol*, and

7. $\mathit{Fin} \subseteq \mathcal{S}$ is the set of *accepting states*.

We use a sequence of stack symbols $\delta \in \mathcal{D}^*$ to represent the contents of a stack. We associate with every pushdown automaton a labelled transition system. The bisimulation equivalence classes of transition systems associated with pushdown automata are referred to as *pushdown processes*.

Definition 4.15 (pushdown processes)**.** Let $\mathcal{M} = (\mathcal{Q}, \Sigma, \mathcal{D}, \mapsto, \mathit{Ini}, Z, \mathit{Fin})$ be a PDA. The *transition system* $\mathcal{T}(\mathcal{M}) = (\mathcal{S}_{\mathcal{T}}, \longrightarrow_{\mathcal{T}}, \uparrow_{\mathcal{T}}, \downarrow_{\mathcal{T}})$ associated with $\mathcal{M}$ is defined as follows:

1. its set of states is the set $\mathcal{S}_{\mathcal{T}} = \{(s, \delta) \mid s \in \mathcal{Q}, \delta \in \mathcal{D}^*\}$ of all configurations of $\mathcal{M}$,

2. its transition relation $\longrightarrow_{\mathcal{T}} \subseteq \mathcal{S}_{\mathcal{T}} \times \mathcal{A}_\tau \times \mathcal{S}_{\mathcal{T}}$ is the relation satisfying, for all $a \in \Sigma$, $d \in \mathcal{D}$, $\delta, \delta' \in \mathcal{D}^*$: $(s, d\delta) \xrightarrow{a}_{\mathcal{T}} (t, \delta'\delta)$ iff $s \xmapsto{a[d/\delta']} t$,

3. its initial state is the configuration $\uparrow_{\mathcal{T}} = (\mathit{Ini}, Z)$, and

4. its set of terminating states is the set $\downarrow_{\mathcal{T}} = \{(s, \delta) \mid s \in \mathcal{Q},\ s \in \mathit{Fin},\ \delta \in \mathcal{D}^*\}$.

In order to simulate a context-free process using a pushdown process modulo strong bisimilarity, we need to design a pushdown process such that the associated states and the transitions have a correspondence with the ones associated with the context-free process. Moreover, in a context-free process, we should distinguish terminating states and non-terminating ones, but we are not able to obtain this information without checking every name that appears in a sequential composition. To decide the correct states to terminate in a pushdown process, we introduce a mechanism to check the appearance of names by using marked names. Recall that a context-free process is defined by a recursive specification in Greibach normal form; all states of the context-free process are denoted by sequences of names defined in this recursive specification. Note that a sequence of names denotes a terminating state only if all names have the option to terminate. Hence, to be able to determine whether a configuration of the pushdown automaton should have the option to terminate, we need to know whether all names currently on the stack have the option to terminate. We annotate the states of the pushdown automaton with the subset of names currently on the stack. We shall use the stack to record the sequence of names corresponding to the current state. The deepest occurrence of a name on the stack is marked and we shall include special transitions in the automaton for the treatment of marked names. If a marked name is removed from

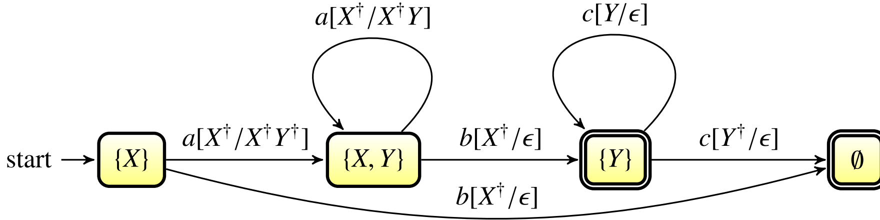


Figure 4.7: A PDA to simulate the process in Figure 4.6

the stack, then, intuitively, it should be removed from the set annotating the state from the set. On the other hand, if a name not in the set is added to the stack, then we shall mark that name and add that name to the set annotating the state. As an example, we introduce a PDA as in Example 4.16 to simulate the process in Figure 4.6 modulo $\leftrightarrows$ as follows.

Example 4.16 (simulating CFP by PDP)**.** Consider a PDA $(Q, \Sigma, \mathcal{D}, \mapsto, \mathit{Ini}, Z, \mathit{Fin})$ with

1. $Q = \{\emptyset, \{X\}, \{Y\}, \{X, Y\}\}$;
2. $\Sigma = \{a, b, c\}$;
3. $\mathcal{D} = \{X, Y, X^\dagger, Y^\dagger\}$
4. $\mapsto = \{(\{X\}, X^\dagger, a, X^\dagger Y^\dagger, \{X, Y\}), (\{X\}, X^\dagger, b, \epsilon, \emptyset), (\{X, Y\}, X^\dagger, a, X^\dagger Y, \{X, Y\}),$ $(\{X, Y\}, X^\dagger, b, \epsilon, \{Y\}), (\{Y\}, Y, c, \epsilon, \{Y\}), (\{Y\}, Y^\dagger, c, \epsilon, \emptyset), \}$;
5. $\mathit{Ini} = \{X\}$;
6. $Z = X^\dagger$; and
7. $\mathit{Fin} = \{\emptyset, \{Y\}\}$.

Its behaviour is shown in Figure 4.7. In the state $\{X\}$, the PDA may choose to stack a Y symbol by an a-labelled transition, or to make a b-labelled transition to enter a terminating state. In the state $\{X, Y\}$, the symbols X and Y are included in its stack, and it may choose to make an a-labelled transition to stack more Y's or choose to make a b-labelled transition to pop the symbol X, and enters the state $\{Y\}$. Finally, in the state $\{Y\}$, the PDA has the option to terminate; moreover, the PDA only has Y as its stack symbol, and it can only make a c-labelled transition to reduce the number of Y's by 1 until it reaches the last one where it could make a c-labelled action and terminate.

To obtain a general result, we consider a context-free process given by a set of names $\mathcal{V} = \{X_0, X_1, \ldots, X_m\}$ with X_0 as the initial state, where

$$X_j = \sum_{i \in I_{X_j}} \alpha_{ij}.\xi_{ij}(+\mathbf{1}) \ .$$

We briefly explain how every process expression ξ is simulated by a configuration of $\mathcal{M}$ such that the sequence of names in ξ is stored in the stack. The first appearance of every name from the bottom of the stack is marked with $\dagger$. The state is marked by a set that contains all the names in ξ. A state is terminating if and only if all the names in the set from the subscript of the state are terminating.

We introduce the following auxiliary functions:

1. $length : \mathcal{V}^* \to \mathbb{N}$, $length(\xi)$ is the length of ξ;
2. $get : \mathcal{V}^* \times \mathbb{N} \to \mathcal{V}$, $get(\xi, i)$ is the i-th name of ξ;
3. $suffset : \mathcal{V}^* \times \mathbb{N} \to 2^{|\mathcal{V}|}$, $suffset(\xi, i) = \{get(\xi, j) \mid j = i+1, \ldots, length(\xi)\}$ computes the set that contains all the names in the suffix which starts from the i-th name of ξ.

We define a PDA $\mathcal{M} = (\mathcal{Q}, \Sigma, \mathcal{D}, \mapsto, Ini, Z, Fin)$ to simulate the transition system associated with X_0 as follows:

1. $\mathcal{Q} = \{D \mid D \subseteq \mathcal{V}\}$;
2. $\Sigma = \mathcal{A}_\tau$;
3. $\mathcal{D} = \mathcal{V} \cup \{X^\dagger \mid X \in \mathcal{V}\}$;
4. the set of transitions $\mapsto$ is defined as follows:

$$\begin{aligned} \mapsto \quad = \quad & \{(D, X_j^\dagger, \alpha_{ij}, \delta(D, X_j^\dagger, \xi_{ij}), merge(D, X_j^\dagger, \xi_{ij})) \mid i \in I_{X_j},\ j = 1, \ldots, n,\ D \subseteq \mathcal{V}\} \\ \cup \quad & \{(D, X_j, \alpha_{ij}, \delta(D, X_j, \xi_{ij}), merge(D, X_j, \xi_{ij})) \mid i \in I_{X_j},\ j = 1, \ldots, n,\ D \subseteq \mathcal{V}\} \ ; \end{aligned}$$

   $\delta(D, X, \xi)$ denotes the string of symbols from ξ to push into the stack in the state D with X as the top symbol; and $\|\ (D, X, \xi)$ denotes the resulting state obtained from a transition that make the above pushing operation; $\delta(D, X_j^\dagger, \xi_{ij})$ is a string defined as follows: for $k = 1, \ldots, length(\xi_{ij})$, we let $X_k = get(\xi_{ij}, k)$,

   (a) if $X_k \notin (D \backslash \{X_j\}) \cup suffset(\xi_{ij}, k)$, then the k-th symbol of $\delta(D, X_j^\dagger, \xi_{ij})$ is $X_k^\dagger$,

   (b) otherwise, the k-th symbol of $\delta(D, X_j^\dagger, \xi_{ij})$ is X_k,

$\delta(D, X_j, \xi_{ij})$ is a string of length $length(\xi_{ij})$ defined as follows: for $k = 1, \ldots, length(\xi_{ij})$, we let $X_k = get(\xi_{ij}, k)$,

(a) if $X_k \notin D \cup suffset(\xi_{ij}, k)$, then the k-th symbol of $\delta(D, X_j, \xi_{ij})$ is $X_k^\dagger$,

(b) otherwise, the k-th symbol of $\delta(D, X_j, \xi_{ij})$ is X_k, and

we also define $merge(D, X_j^\dagger, \xi_{ij}) = (D \backslash \{X_j\}) \cup suffset(\xi_{ij}, 0)$ and $merge(D, X_j, \xi_{ij}) = D \cup suffset(\xi_{ij}, 0)$;

5. $Ini = \{X_0\}$; $Z = X_0^\dagger$;

6. $Fin = \{D \mid \text{for all } X \in D, X \downarrow\}$.

We have the following result:

Lemma 4.17. $\mathcal{T}(X_0) \leftrightarrows \mathcal{T}(\mathcal{M})$.

Proof. We first define an auxiliary function $stack : \mathcal{V}^* \to \mathcal{D}^*$. We let $\xi \in \mathcal{V}^*$ and for $k = 1, \ldots, length(\xi)$, we let $X_k = get(\xi, k)$. $stack(\xi)$ is given by:

1. if $X_k \notin suffset(\xi, k)$, then the k-th element of $stack(\xi)$ is $X_k^\dagger$;

2. otherwise, the k-th element of $stack(\xi)$ is X_k,

Note that $stack(X\xi)$ and $stack(\xi)$ share the same suffix of length $length(\xi)$, and we use this fact to show that the relation

$$\mathcal{R} = \{(\xi, (suffset(\xi, 0), stack(\xi))) \mid \xi \in \mathcal{V}^*\} ,$$

is a strong bisimulation. We assume that ξ is not an empty sequence, since it is trivial if ξ is an empty sequence.

If $\xi = X_j \xi$ for some $0 \leq j \leq m$, then it has the following transitions:

$$X_j \xi' \xrightarrow{\alpha_{ij}} \xi_{ij} \xi', \ i \in I_{X_j} .$$

We need to show that they are simulated by the transitions:

$$(suffset(\xi, 0), stack(\xi)) \xrightarrow{\alpha_{ij}} (suffset(\xi_{ij}\xi', 0), stack(\xi_{ij}\xi')), \ i \in I_{X_j} ,$$

in a way, we have $(x_{ij}, (suffset(\xi_{ij}\xi', 0), stack(\xi_{ij}\xi'))) \in \mathcal{R}$.

We consider the configuration $(suffset(\xi, 0), stack(\xi))$, we distinguish two cases according to whether the symbol on top of the stack is the deepest occurrence of that symbol on the stack, or not.

1. If $get(stack(\xi), 1) = X_j^\dagger$, then $\mathcal{M}$ has the transition

$$(suffset(\xi, 0), X_j^\dagger, \alpha_{ij}, \delta(suffset(\xi, 0), X_j^\dagger, \xi_{ij}), merge(suffset(\xi, 0), X_j^\dagger, \xi_{ij})) \ .$$

   In the new configuration, the stack is $S = \delta(suffset(\xi, 0), X_j^\dagger, \xi_{ij})stack(\xi')$. We verify that $S = stack(\xi_{ij}\xi')$. Note that they share the same suffix $stack(\xi')$. We only need to verify the first $length(\xi_{ij})$ elements. For the l-th element, we let $X_l = get(\xi_{ij}, l)$, and we distinguish with two cases.

   (a) If $X_l \notin (suffset(\xi, 0)\backslash\{X_j\}) \cup suffset(\xi_{ij}, l)$, then the l-th element of S is $X_l^\dagger$. Since $get(stack(\xi), 1) = X_j^\dagger$, from the definition of *stack*, we have $X_j \notin suffset(\xi, 1) = suffset(\xi', 0)$. Therefore, $suffset(\xi, 0)\backslash\{X_j\} = suffset(\xi', 0)$. In this case, $X_l \notin suffset(\xi', 0) \cup suffset(\xi_{ij}, l)$. Moreover, we have $X_l \notin suffset(\xi_{ij}\xi', l)$, therefore, the l-th element of $stack(\xi_{ij}\xi')$ is also $X_l^\dagger$.

   (b) Otherwise, then the l-th element of S is X_l. By the definition of *stack*, we get that the l-th element of $stack(\xi_{ij}\xi')$ is also X_l.

   Moreover, we verify that the new state $merge(suffset(\xi, 0), X_j^\dagger, \xi_{ij}) = suffset(\xi_{ij}\xi', 0)$. Note that we have

$$merge(suffset(\xi, 0), X_j^\dagger, \xi_{ij}) = (suffset(\xi, 0)\backslash\{X_j\}) \cup suffset(\xi_{ij}, 0)$$
$$= suffset(\xi', 0) \cup suffset(\xi_{ij}, 0) = suffset(\xi_{ij}\xi', 0) \ .$$

   Hence, we have $(suffset(\xi, 0), stack(\xi)) \xrightarrow{\alpha_{ij}} (suffset(\xi_{ij}\xi', 0), stack(\xi_{ij}\xi'))$.

2. if $get(stack(\xi), 1) = X_j$, then $\mathcal{M}$ has the transition

$$(suffset(\xi, 0), X_j, \alpha_{ij}, \delta(suffset(\xi, 0), X_j, \xi_{ij}), merge(suffset(\xi, 0), X_j, \xi_{ij})) \ ,$$

   In the new configuration, the new stack is $S = \delta(suffset(\xi, 0), X_j, \xi_{ij})stack(\xi')$. We verify that $S = stack(\xi_{ij}\xi')$. Note that they share the same suffix $stack(\xi')$. We only need to verify the first $length(\xi_{ij})$ elements. For the l-th element, we let $X_l = get(\xi ij, l)$, and we distinguish with two cases.

   (a) If $X_l \notin (suffset(\xi, 0)) \cup suffset(\xi_{ij}, l)$, then the l-th element of S is $X_l^\dagger$. Since $get(stack(\xi), 1) = X_j$, from the definition of *stack*, we have $X_j \in suffset(\xi, 1) = suffset(\xi', 0)$. Therefore, $suffset(\xi, 0) = suffset(\xi', 0)$. In this case, $X_l \notin suffset(\xi', 0) \cup suffset(\xi_{ij}, l)$. Moreover, we have $X_l \notin suffset(\xi_{ij}\xi', l)$, therefore, the l-th element of $stack(\xi_{ij}\xi')$ is also $X_l^\dagger$.

(b) Otherwise, then the l-th element of S is X_l. By the definition of *stack*, we get that the l-th element of $stack(\xi_{ij}\xi')$ is also X_l.

Moreover, we verify that the new state $merge(suffset(\xi, 0), X_j, \xi_{ij}) = suffset(\xi_{ij}\xi', 0)$. Note that we have

$$merge(suffset(\xi, 0), X_j, \xi_{ij}) = suffset(\xi, 0) \cup suffset(\xi_{ij}, 0)$$
$$= suffset(\xi', 0) \cup suffset(\xi_{ij}, 0) = suffset(\xi_{ij}\xi', 0) \ .$$

Hence, we have $(suffset(\xi, 0), stack(\xi)) \xrightarrow{\alpha_{ij}} (suffset(\xi_{ij}\xi', 0), stack(\xi_{ij}\xi'))$.

We conclude from the above two cases that the transitions are indeed simulated.

Using a similar analysis, we also have all the transitions from $(suffset(\xi, 0), stack(\xi))$ are simulated by $X_j\xi'$.

Now we consider the termination condition. $\xi \downarrow$ iff for all $X \in suffset(\xi, 0)$, $X \downarrow$. Note that $(suffset(\xi, 0), stack(\xi)) \downarrow$ iff for all $X \in suffset(\xi, 0)$, $X \downarrow$. Therefore, termination condition is also verified.

Hence, we have $\mathcal{T}(X_0) \mathrel{\underline{\leftrightarrow}} \mathcal{T}(\mathcal{M})$. □

We have the following theorem.

Theorem 4.18 (simulating CFP by PDP)**.** *For every name X defined in a guarded recursive specification in Greibach normal form there exists a PDA $\mathcal{M}$, such that $\mathcal{T}(X) \mathrel{\underline{\leftrightarrow}} \mathcal{T}(\mathcal{M})$.*

Note that the converse of this theorem does not hold in general, that is, not every pushdown process can be simulated by a context-free process modulo $\mathrel{\underline{\leftrightarrow}}$. A counterexample was established by F. Moller in [70]. We rephrase the example as follows:

Example 4.19. Consider a PDA $(Q, \Sigma, \mathcal{D}, \mapsto, Ini, Z, Fin)$ with

1. $Q = \{p, q\}$;
2. $\Sigma = \{a, b, c\}$;
3. $\mathcal{D} = \{X\}$
4. $\mapsto = \{(p, X, a, XX, p), (p, X, b, \epsilon, X), (p, X, c, \epsilon, q), (q, X, b, \epsilon, q)\}$;
5. $Ini = p$;
6. $Z = X$; and

7. $Fin = q$.

The transition system associated with the above PDA cannot be specified by any context-free process. The language given by the above PDA is $\{a^n b^m c b^{n-m} \mid m, n \in \mathbb{N},\ m \leq n\}$ which is not context-free. Therefore, there is no context-free process which is language equivalent to the above PDA. Hence, there is no context-free process which is strongly bisimilar to the above PDA either. Moreover, this counterexample is also valid modulo $\leftrightarrows_{b}$.

4.5 Executability in the Context of Termination

In this section, we shall discuss the theory of executability. We shall prove that TCP$^\sharp$ is reactively Turing powerful in the context of termination. Our aim in this section is to prove that all executable processes can be specified, up to divergence-preserving branching bisimilarity in TCP$^\sharp$. TCP$^\sharp$ is obtained from TCP by removing recursive definitions and adding the iteration and nesting operators.

To see that TCP$^\sharp$ is executable modulo branching bisimilarity, it suffices to observe that the transition systems associated with TCP$^\sharp$ processes are effective. Thus we can apply the result from [13] and conclude that they are executable modulo $\leftrightarrows_{b}$.

Next we show that TCP$^\sharp$ is reactively Turing powerful by specifying the transition system associated with a reactive Turing machine in TCP$^\sharp$ modulo $\leftrightarrows_{b}^{\Delta}$. Inspired from [18], the proof proceeds in five steps:

1. We first specify an always terminating half counter.

2. Then we show that every regular process can be specified in TCP$^\sharp$.

3. Next we use two half counters and a regular process to encode an always terminating stack.

4. With two stacks and a regular process we can specify a tape.

5. Finally we use a tape and a regular control process to specify an RTM.

We first recall the infinite specification in TSP$^{;}$ of an always terminating half counter from Figure 4.4.

We provide a specification of a half counter in TCP$^\sharp$ as follows:

$$HC = ((a + \mathbf{1})^\sharp (b + \mathbf{1});(c + \mathbf{1}))^*$$

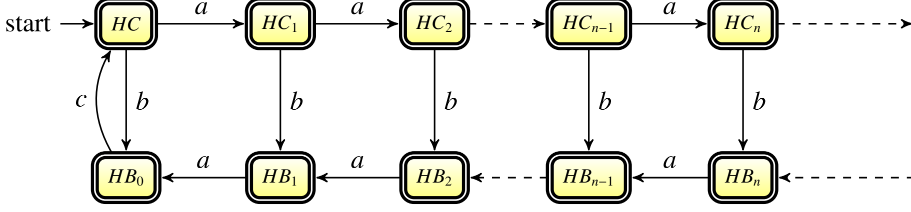


Figure 4.8: An implementation of the always terminating half counter in TCP$^\sharp$

To illustrate its behaviour, we let

$$\begin{aligned} HC_n &= ((a+\mathbf{1})^\sharp(b+\mathbf{1});(a+\mathbf{1})^{n;};(c+\mathbf{1}));HC, \\ HB_n &= (a+\mathbf{1})^{n;};(c+\mathbf{1});HC \ . \end{aligned}$$

We exhibit its transition system in Figure 4.8.

We have the following lemma:

Lemma 4.20 (the always terminating half-counter)**.** $C_0 \leftrightarrows_b^\Delta HC$.

Proof. We verify that $HC \leftrightarrows_b^\Delta C_0$. Consider the following relation:

$$\begin{aligned} \mathcal{R}_1 &= \{(C_0, HC)\} \cup \{(C_n, (a+\mathbf{1})^\sharp(b+\mathbf{1});(a+\mathbf{1})^{n;};(c+\mathbf{1});HC) \mid n \geq 1\} \\ &\cup \ \{(B_n, (a+\mathbf{1})^{n;};(c+\mathbf{1});HC) \mid n \in \mathbb{N}\} \ . \end{aligned}$$

We let $\mathcal{R}_2$ be the inverse of $\mathcal{R}_1$. We show that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ (the symmetrical closure of $\mathcal{R}_1$) is a divergence-preserving branching bisimulation as follows:

Note that $\mathcal{R}$ satisfies the divergence-preserving condition since there is no infinite sequence of τ transitions. In this proof, we only treat the pairs in $\mathcal{R}_1$, since we can use the symmetrical argument for the pairs in $\mathcal{R}_2$. We first consider the pair (C_0, HC). Note that C_0 has the following transitions:

$$C_0 \xrightarrow{a} C_1, \text{ and}$$
$$C_0 \xrightarrow{b} B_0 \ ,$$

which are simulated by:

$$HC \xrightarrow{a} (a+\mathbf{1})^\sharp(b+\mathbf{1});(a+\mathbf{1});(c+\mathbf{1});HC, \text{ and}$$
$$HC \xrightarrow{b} (c+\mathbf{1});HC \ ,$$

with $(C_1, (a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1});(c + \mathbf{1});HC) \in \mathcal{R}$ and $(B_0, (c + \mathbf{1});HC) \in \mathcal{R}$. Moreover, we have $C_0 \downarrow$ and $HC \downarrow$.

Now we consider the pair $(C_n, (a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1})^{n;};(c + \mathbf{1});HC)$, with $n \geq 1$. Note that C_n has the following transitions:

$$C_n \xrightarrow{a} C_{n+1}, \text{ and}$$
$$C_n \xrightarrow{b} B_n \quad ,$$

which are simulated by:

$$(a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1})^{n;};(c + \mathbf{1});HC \xrightarrow{a} (a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1})^{n+1;};(c + \mathbf{1});HC, \text{ and}$$
$$(a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1})^{n;};(c + \mathbf{1});HC \xrightarrow{b} (a + \mathbf{1})^{n;};(c + \mathbf{1});HC \quad ,$$

with $(C_{n+1}, (a + \mathbf{1})^\sharp(b + \mathbf{1});(a+\mathbf{1})^{n+1;};(c+\mathbf{1});HC) \in \mathcal{R}$ and $(B_n, (a+\mathbf{1})^{n;};(c+\mathbf{1});HC) \in \mathcal{R}$. Moreover, we have $C_n \downarrow$ and $(a + \mathbf{1})^\sharp(b + \mathbf{1});(a + \mathbf{1})^{n;};(c + \mathbf{1});HC \downarrow$.

Now we proceed to consider the pair $(B_0, (c + \mathbf{1});HC)$. Note that B_0 has the following transition:

$$B_0 \xrightarrow{c} C_0 \quad ,$$

which is simulated by:

$$(c + \mathbf{1});HC \xrightarrow{c} HC \quad ,$$

with $(C_0, HC) \in \mathcal{R}$. Moreover, we have $B_0 \downarrow$ and $(c + \mathbf{1});HC \downarrow$.

Next we consider the pair $(B_n, (a + \mathbf{1})^{n;};(c + \mathbf{1});HC)$, with $n \geq 1$. Note that B_n has the following transition:

$$B_n \xrightarrow{a} B_{n-1} \quad ,$$

which is simulated by:

$$(a + \mathbf{1})^{n;};(c + \mathbf{1});HC \xrightarrow{a} (a + \mathbf{1})^{n-1;};(c + \mathbf{1});HC \quad ,$$

with $(B_{n-1}, (a + \mathbf{1})^{n-1;};(c + \mathbf{1});HC) \in \mathcal{R}$. Moreover, we have $B_n \downarrow$ and $(a + \mathbf{1})^{n;};(c + \mathbf{1});HC \downarrow$.

Hence, we have $C_0 \mathrel{\underline{\leftrightarrow}}_{\mathrm{b}}^{\Delta} HC$. □

Next we show that every regular process can be specified in TCP$^\sharp$ modulo $\underline{\leftrightarrow}_{\mathrm{b}}^{\Delta}$. A regular process is given by $P_i = \sum_{j=1}^{n} \alpha_{ij};P_j + \beta_i \, (i = 1, \ldots, n)$ where α_{ij} and β_i are finite sums of actions from $\mathcal{A}_\tau$ and possibly with a $\mathbf{1}$-summand. We have the following lemma.

Lemma 4.21 (regular processes)**.** *Every regular process can be specified in TCP$^\sharp$ modulo $\underline{\leftrightarrow}_b^\Delta$.*

Proof. We consider a regular process with a finite set of action labels $\mathcal{A}_\tau$ which is given by $P_i = \sum_{j=1}^n \alpha_{ij};P_j + \beta_i$ $(i = 1, \ldots, n)$ where α_{ij} and β_i are finite sums of actions from $\mathcal{A}_\tau$. We let $c!0, c!1, \ldots, c!(n+1), c?0, c?1, \ldots, c?(n+1)$ be labels that are not in $\mathcal{A}_\tau$.

Consider the following process:

$$\begin{aligned}
G_i &= \sum_{j=1}^n \alpha_{ij};(c!j + \mathbf{1}) + \beta_i;(c!0 + \mathbf{1}) \\
M &= \left(\sum_{j=1}^n (c?j + \mathbf{1});G_j + (c!(n+1) + \mathbf{1});(c?(n+1) + \mathbf{1})\right)^\sharp (c?0 + \mathbf{1}) \\
N &= \left(\sum_{j=1}^{n+1} (c?j + \mathbf{1});(c!j + \mathbf{1})\right)^\sharp ((c?0 + \mathbf{1});(c!0 + \mathbf{1}))
\end{aligned}$$

Note that ; is associative modulo $\underline{\leftrightarrow}_\mathrm{b}^\Delta$ and we suppose that ; binds stronger than +. We verify that $P_i \underline{\leftrightarrow}_\mathrm{b}^\Delta [G_i;M \parallel N]_{\{c\}}$. We let

$$Q = \left(\sum_{j=1}^n (c?j + \mathbf{1});G_j + (c!(n+1) + \mathbf{1});(c?(n+1) + \mathbf{1})\right)$$

and

$$O = \left(\sum_{j=1}^{n+1} (c?j + \mathbf{1});(c!j + \mathbf{1})\right) .$$

We let

$$\begin{aligned}
\mathcal{R}_1 &= \{(P_i, [G_i;M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \mid k \in \mathbb{N},\ i = 1, \ldots, n\} \\
&\cup \{(P_i, [(c!i + \mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \mid k \in \mathbb{N},\ i = 1, \ldots, n\} \\
&\cup \{(P_i, [M;Q^{k;} \parallel (c!i + \mathbf{1});N;O^{k+1;}]_{\{c\}}) \mid k \in \mathbb{N},\ i = 1, \ldots, n\} \\
&\cup \{(\mathbf{1}, [(c!0 + \mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \mid k \in \mathbb{N}\} \\
&\cup \{(\mathbf{1}, [M;Q^{k;} \parallel (c!0 + \mathbf{1});O^{k;}]_{\{c\}}) \mid k \in \mathbb{N}\} \\
&\cup \{(\mathbf{1}, [Q^{k;} \parallel O^{k;}]_{\{c\}}) \mid k \in \mathbb{N}\} \\
&\cup \{(\mathbf{1}, [(c?(n+1) + \mathbf{1});Q^{k;} \parallel (c!(n+1) + \mathbf{1});O^{k;}]_{\{c\}}) \mid k \in \mathbb{N}\} \ ;
\end{aligned}$$

and we let $\mathcal{R}_2$ be the inverse of $\mathcal{R}_1$. We show that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ is a divergence-preserving branching bisimulation. We shall only verify the pairs in $\mathcal{R}_1$ in this proof since $\mathcal{R}$ is symmetrical.

For the set of pairs $\{(P_i, [G_i;M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \mid k \in \mathbb{N}, i = 1, \ldots, n\}$, note that P_i has the following transitions: $P_i \xrightarrow{a} P_j$ if a is a summand of α_{ij}, or $P_i \xrightarrow{a} \mathbf{1}$ if a is a summand of β_j.

The first transition is simulated by the following transitions:

$$\begin{aligned}&[G_i;M;Q^{k;} \parallel N;O^{k;}]_{\{c\}} \xrightarrow{a} [(c!j + \mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}\\ &\xrightarrow{\tau} [M;Q^{k;} \parallel (c!j + \mathbf{1});N;O^{k+1;}]_{\{c\}}\\ &\xrightarrow{\tau} [G_j;M;Q^{k+1;} \parallel N;O^{k+1;}]_{\{c\}}\ .\end{aligned}$$

If $k \geq 1$, then the second transition is simulated by the following transitions:

$$\begin{aligned}&[G_i;M;Q^{k;} \parallel N;O^{k;}]_{\{c\}} \xrightarrow{a} [(c!0 + \mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}\\ &\xrightarrow{\tau} [M;Q^{k;} \parallel (c!0 + \mathbf{1});O^{k;}]_{\{c\}} \xrightarrow{\tau} [Q^{k;} \parallel O^{k;}]_{\{c\}}\\ &\xrightarrow{\tau} [(c?(n + 1) + \mathbf{1});Q^{k-1;} \parallel (c!(n + 1) + \mathbf{1});O^{k-1;}]_{\{c\}} \xrightarrow{\tau} [Q^{k-1;} \parallel O^{k-1;}]_{\{c\}}\\ &\longrightarrow^{*} \mathbf{1}\ ;\end{aligned}$$

otherwise, if $k = 0$, then the second transition is simulated by:

$$\begin{aligned}&[G_i;M \parallel N]_{\{c\}} \xrightarrow{a} [(c!0 + \mathbf{1});M \parallel N]_{\{c\}}\\ &\xrightarrow{\tau} [M \parallel (c!0 + \mathbf{1})]_{\{c\}} \xrightarrow{\tau} \mathbf{1}\ .\end{aligned}$$

We have that $(P_j, [(c!j{+}\mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \in \mathcal{R}$, $(P_j, [M;Q^{k;} \parallel (c!j{+}\mathbf{1});N;O^{k+1;}]_{\{c\}}) \in \mathcal{R}$, $(P_j, [G_j;M;Q^{k+1;} \parallel N;O^{k+1;}]_{\{c\}}) \in \mathcal{R}$, $(\mathbf{1}, [(c!0{+}\mathbf{1});M;Q^{k;} \parallel N;O^{k;}]_{\{c\}}) \in \mathcal{R}$, $(\mathbf{1}, [M;Q^{k;} \parallel (c!0{+}\mathbf{1});O^{k;}]_{\{c\}}) \in \mathcal{R}$, $(\mathbf{1}, [Q^{k;} \parallel O^{k;}]_{\{c\}})$, $(\mathbf{1}, [(c?(n{+}1){+}\mathbf{1});Q^{k;} \parallel (c!(n{+}1){+}\mathbf{1});O^{k;}]_{\{c\}}) \in \mathcal{R}$ and $(\mathbf{1}, \mathbf{1}) \in \mathcal{R}$ for all $k \in \mathbb{N}$ and $i, j = 1, \ldots, n$.

One can easily verify that all the other pairs satisfy the condition of branching bisimulation. The relation $\mathcal{R}$ also satisfies the divergence-preserving condition since no infinite τ-transition sequence is allowed from any process defined in $\mathcal{R}$.

Therefore, we get a finite specification of every regular process in TCP$^{\sharp}$ modulo $\underline{\leftrightarrow}_b^{\Delta}$. □

Now we show that a stack can be specified by a regular process and two half counters. We first give an infinite specification in TSP$^{;}$ of a stack as follows:

$$\begin{aligned}S_{\epsilon} &= \Sigma_{d\in\mathcal{D}_{\square}} push?d.S_d + pop!\square.S_{\epsilon} + \mathbf{1}\\ S_{d\delta} &= pop!d.S_{\delta} + \Sigma_{e\in\mathcal{D}_{\square}} push?e.S_{ed\delta} + \mathbf{1}\ .\end{aligned}$$

Recall that $\mathcal{D}_\square$ is a finite set of symbols. We suppose that $\mathcal{D}_\square$ contains N symbols (including $\square$). We use ϵ to denote the empty sequence. We inductively define an encoding from a sequence of symbols to a natural number $\lceil _ \rceil : \mathcal{D}_\square{}^* \to \mathbb{N}$ as follows:

$$\lceil \epsilon \rceil = 0 \quad \lceil d_k \rceil = k \quad (k = 1, 2, \ldots, N) \quad \lceil d_k \sigma \rceil = k + N \times \lceil \sigma \rceil \ .$$

Thus we are able to encode the contents of a stack in terms of natural numbers recorded by half counters. We define a stack in TCP$^\sharp$ as follows:

$$\begin{array}{rcl}
S & = & [X_\emptyset \parallel P_1 \parallel P_2]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \\
P_j & = & ((a_j!a + \mathbf{1})^\sharp (b_j!b + \mathbf{1});(c_j!c + \mathbf{1}))^* \ (j = 1, 2) \\
X_\emptyset & = & (\Sigma_{j=1}^N ((push?d_j + \mathbf{1});(a_1?a + \mathbf{1})^{j;};(b_1 + \mathbf{1});X_j) + pop!\square)^* \\
X_k & = & \Sigma_{j=1}^N ((push?d_j + \mathbf{1});Push_j) + (pop!d_k + \mathbf{1});Pop_k \quad (k = 1, 2, \ldots, N) \\
Push_k & = & \mathit{Shift1to2};(a_1?a + \mathbf{1})^{k;};\mathit{NShift2to1};X_k \quad (k = 1, 2, \ldots, N) \\
Pop_k & = & (a_1?a + \mathbf{1})^{k;};\mathit{1/NShift1to2};\mathit{Test}_\emptyset \\
\mathit{Shift1to2} & = & ((a_1?a + \mathbf{1});(a_2?a + \mathbf{1}))^*;(c_1?c + \mathbf{1});(b_2?b + \mathbf{1}) \\
\mathit{NShift2to1} & = & ((a_2?a + \mathbf{1});(a_1?a + \mathbf{1})^{N;})^*;(c_2?c + \mathbf{1});(b_1?b + \mathbf{1}) \\
\mathit{1/NShift1to2} & = & ((a_1?a + \mathbf{1})^{N;};(a_2?a + \mathbf{1}))^*;(c_1?c + \mathbf{1});(b_2?b + \mathbf{1}) \\
\mathit{Test}_\emptyset & = & (a_2?a + \mathbf{1});(a_1?a + \mathbf{1});\mathit{Test}_1 + (c_2?c + \mathbf{1});X_\emptyset \\
\mathit{Test}_1 & = & (a_2?a + \mathbf{1});(a_1?a + \mathbf{1});\mathit{Test}_2 + (c_2?c + \mathbf{1});X_1 \\
\mathit{Test}_2 & = & (a_2?a + \mathbf{1});(a_1?a + \mathbf{1});\mathit{Test}_3 + (c_2?c + \mathbf{1});X_2 \\
& \vdots & \\
\mathit{Test}_N & = & (a_2?a + \mathbf{1});(a_1?a + \mathbf{1});\mathit{Test}_1 + (c_2?c + \mathbf{1});X_N \ .
\end{array}$$

We have the following result.

Lemma 4.22 (the always terminating stack)**.** $S_\epsilon \ \underline{\leftrightarrow}_b^\Delta \ S$.

Proof. We define some auxiliary process:

$$\begin{array}{rcl}
P_j(0) & = & ((a_j!a + \mathbf{1})^\sharp (b_j!b + \mathbf{1});(c_j!c + \mathbf{1}))^* \ (j = 1, 2) \\
P_j(n) & = & (a_j!a + \mathbf{1})^\sharp (b_j!b + \mathbf{1});(a_j!a + \mathbf{1})^{n;};(c_j!c + \mathbf{1}));P_j, \ (j = 1, 2; \ n = 1, 2, \ldots) \\
Q_j(n) & = & (a_j!a + \mathbf{1})^{n;};(c_j!c + \mathbf{1}));P_j, \ (j = 1, 2; \ n \in \mathbb{N}) \ .
\end{array}$$

P_0 and P_1 behave as two half counters.

We let $\mathcal{R}_1 = \{(S_\epsilon, S)\} \cup \{(S_{d_j\delta}, [X_j;X_\epsilon \parallel Q_1(m) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}) \mid j = \lceil d_j \rceil, m = \lceil d_j\delta \rceil, d \in \mathcal{D}_\square, \delta \in \mathcal{D}_\square^*\}$. We let $\mathcal{R}_2$ be inverse of $\mathcal{R}_1$. We verify that $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2 \cup \underline{\leftrightarrow}_b^\Delta$

is a divergence-preserving branching bisimulation relation. For simplicity, we only verify the simulation of the transitions from one direction and the other direction is then trivial.

Note that S_ϵ has the following transitions:

$$S_\epsilon \xrightarrow{push?d_j} S_{d_j} \text{ for all } j = 1, 2, \ldots, N, \text{ and}$$
$$S_\epsilon \xrightarrow{pop!\Box} S_\epsilon \ .$$

They are simulated by the following transitions:

$$\begin{aligned}
&S \xrightarrow{push?d_j} [(a_1?a + \mathbf{1})^{j};(b_1 + \mathbf{1});X_j;X_\epsilon \parallel P_1(0) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \\
&\longrightarrow^* [(b_1 + \mathbf{1});X_j;X_\epsilon \parallel P_1(j) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \\
&\longrightarrow^* [X_j;X_\epsilon \parallel Q_1(j) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \text{ for all } j = 1, 2, \ldots, N, \text{ and} \\
&S \xrightarrow{pop!\Box} S \ .
\end{aligned}$$

We only consider the first case, since the second transition is trivial. We have

$$(S_{d_j}, [X_j;X_\epsilon \parallel Q_1(j) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}) \in \mathcal{R} \ .$$

We denote the sequence of transitions

$$[(a_1?a + \mathbf{1})^{j};(b_1 + \mathbf{1});X_j;X_\epsilon \parallel P_1(0) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \longrightarrow^* \\ [X_j;X_\epsilon \parallel Q_1(j) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \in \mathcal{R}$$

by $s_0 \longrightarrow^* s_m$. It is obvious that $s_0 \leftrightarrow_b^\Delta s_m$. Therefore, $S \xrightarrow{push?d_j} s_0$, and $s_0 \leftrightarrow_b^\Delta s_m$ with $(S_{d_j}, s_m) \in \mathcal{R}$.

Note that $S_{d_j\delta}$ has the following transitions:

$$S_{d_j\delta} \xrightarrow{push?d_k} S_{d_k d_j \delta} \text{ for all } k = 1, 2, \ldots, N, \text{ and}$$
$$S_{d_j\delta} \xrightarrow{pop!d_j} S_{d_k\delta'}, \text{ where } d_k\delta' = \delta \ .$$

They are simulated by the following transitions:

$[X_j;X_\epsilon \parallel Q_1(\lceil d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\xrightarrow{push?d_k} [Push_k;X_\epsilon \parallel Q_1(\lceil d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [(a_1?a+\mathbf{1})^{k;};NShift2to1;X_k;X_\epsilon \parallel P_1(0) \parallel Q_2(\lceil d_j\delta\rceil)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [NShift2to1;X_k;X_\epsilon \parallel P_1(\lceil d_k\rceil) \parallel Q_2(\lceil d_j\delta\rceil)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [X_k;X_\epsilon \parallel Q_1(\lceil d_k d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$ for all $d_j, d_k \in \mathcal{D}_\Box$, $\delta \in \mathcal{D}_\Box^*$ and
$[X_j;X_\epsilon \parallel Q_1(\lceil d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\xrightarrow{pop!d_j} [Pop_j;X_\epsilon \parallel Q_1(\lceil d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [1/NShift1to2;Test_\emptyset;X_\epsilon \parallel Q_1(\lceil d_j\delta\rceil - k) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [Test_\emptyset;X_\epsilon \parallel P_1(0) \parallel Q_2(\lceil \delta\rceil)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$
$\longrightarrow^* [X_k;X_\epsilon \parallel Q_1(\lceil d_k\delta'\rceil \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}$ for all $d_j \in \mathcal{D}_\Box$, $\delta \in \mathcal{D}_\Box^*$ and $\delta = d_k\delta'$.

We have

$$(S_{d_k d_j\delta}, [X_k;X_\epsilon \parallel Q_1(\lceil d_k d_j\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}) \in \mathcal{R}$$
$$(S_{d_k\delta'}, [X_k;X_\epsilon \parallel Q_1(\lceil d_k\delta'\rceil \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}) \in \mathcal{R} \ .$$

By using a similar analysis with the previous case, we have that $\mathcal{R}$ is a bisimulation up to $\leftrightarrows_b$. By Lemma 2.9, we have $\mathcal{R} \subseteq \leftrightarrows_b$. Moreover, there is no infinite τ-transition sequence from any process defined above. Therefore, $\mathcal{R} \subseteq \leftrightarrows_b^\Delta$.

Hence, we have $S_\epsilon \leftrightarrows_b^\Delta S$. □

Next we proceed to define a tape process by means of two stacks. We consider the following infinite specification in TSP$^{;}$ of a tape:

$$T_{\delta_L \check{d}\delta_R} = r!d.T_{\delta_L \check{d}\delta_R} + \Sigma_{e\in\mathcal{D}_\Box} w?e.T_{\delta_L \check{e}\delta_R} + L?m.T_{\delta_L^< d\delta_R} + R?m.T_{\delta_L d^>\delta_R} + \mathbf{1} \ .$$

We define the tape process in TCP$^\sharp$ as follows:

$$\begin{aligned}
T &= [T_\Box \parallel S_1 \parallel S_2]_{\{push_1,pop_1,push_2,pop_2\}} \\
T_d &= r!d.T_d + \Sigma_{e\in\mathcal{D}_\Box} w?e.T_e + L?m.Left_d + R?m.Right_d + \mathbf{1} \quad (d \in \mathcal{D}_\Box) \\
Left_d &= \Sigma_{e\in\mathcal{D}_\Box}((pop_1?e+\mathbf{1});(push_2!d+\mathbf{1});T_e) \\
Right_d &= \Sigma_{e\in\mathcal{D}_\Box}((pop_2?e+\mathbf{1});(push_1!d+\mathbf{1});T_e) \ ,
\end{aligned}$$

where S_1 and S_2 are two stacks obtained by renaming *push* and *pop* in S to $push_1$, pop_1, $push_2$ and pop_2, respectively. We establish the following result.

Lemma 4.23 (the always terminating tape)**.** $T_{\check{\Box}} \underline{\leftrightarrow}^{\Delta}_{b} T$.

Proof. We define the following auxiliary processes:

$$
\begin{array}{rcl}
S_1(\delta) & = & [X_{1,k} \parallel Q_1(\lceil\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}} \\
S_2(\delta) & = & [X_{2,k} \parallel Q_1(\lceil\delta\rceil) \parallel P_2(0)]_{\{a_1,a_2,b_1,b_2,c_1,c_2\}}, \text{ where } \delta = d_k\delta' \ .
\end{array}
$$

$X_{1,k}$ and $X_{2,k}$ are obtained by renaming *push* and *pop* in X_k to *push₁*, *pop₁*, *push₂* and *pop₂* respectively. We use $\overline{\delta}$ to denote the reverse sequence of δ.

We verify that

$$
\mathcal{R} = \{(T_{\delta_L\check{d}\delta_R}, [T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \mid d \in \mathcal{D}_\Box,\ \delta_L, \delta_R \in \mathcal{D}^*_\Box\} \subseteq \underline{\leftrightarrow}^{\Delta}_{\mathrm{b}} \ .
$$

For simplicity, we only verify the simulation of the transitions from $T_{\delta_L\check{d}\delta_R}$ and the other direction is then trivial. $T_{\delta_L\check{d}\delta_R}$ has the following transitions:

$$
\begin{array}{ll}
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{r!d} T_{\delta_L\check{d}\delta_R} \\
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{w?e} T_{\delta_L\check{e}\delta_R} \\
\text{for all } e \in \mathcal{D}_\Box & \\
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{L?m} T_{\delta_L^{<}d\delta_R} \text{ if } \delta_L \neq \epsilon \\
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{R?m} T_{\delta_L d^{>}\delta_R} \text{ if } \delta_R \neq \epsilon \\
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{L?m} T_{\epsilon\check{\Box}d\delta_R} \text{ if } \delta_L = \epsilon \text{ and} \\
 & T_{\delta_L\check{d}\delta_R} \xrightarrow{R?m} T_{\delta_L d\check{\Box}\epsilon} \text{ if } \delta_R = \epsilon \ .
\end{array}
$$

They are simulated by the following transitions:

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{r!d} [T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{e?d} [T_e \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$

for all $e \in \mathcal{D}_\Box$

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{L?m} [Left_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$
$$\longrightarrow^* [T_e \parallel S_1(\overline{\delta'_L}) \parallel S_2(d\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}},\ \delta_L = \delta'_L e,\ \text{if}\ \delta_L \neq \epsilon$$

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{R?m} [Right_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$
$$\longrightarrow^* [T_e \parallel S_1(\overline{\delta_L d}) \parallel S_2(\delta'_R)]_{\{push_1,pop_1,push_2,pop_2\}},\ \delta_R = e\delta_R,\ \text{if}\ \delta_R \neq \epsilon$$

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{L?m} [Left_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$
$$\longrightarrow^* [T_\Box \parallel S_1(\epsilon) \parallel S_2(d\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}},\ \text{if}\ \delta_L = \epsilon$$

$$[T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}} \xrightarrow{R?m} [Right_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}$$
$$\longrightarrow^* [T_\Box \parallel S_1(\overline{\delta_L d}) \parallel S_2(\epsilon)]_{\{push_1,pop_1,push_2,pop_2\}},\ \text{if}\ \delta_R = \epsilon\ .$$

We have

$$(T_{\delta_L \check{d} \delta_R}, [T_d \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R},$$
$$(T_{\delta_L \check{e} \delta_R}, [T_e \parallel S_1(\overline{\delta_L}) \parallel S_2(\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R},$$
$$(T_{\delta_L^< d \delta_R}, [T_e \parallel S_1(\overline{\delta'_L}) \parallel S_2(d\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R},$$
$$(T_{\delta_L d^> \delta_R}, [T_e \parallel S_1(\overline{\delta_L d}) \parallel S_2(\delta'_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R},$$
$$(T_{\epsilon \check{\Box} d \delta_R}, [T_\Box \parallel S_1(\epsilon) \parallel S_2(d\delta_R)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R},\ \text{and}$$
$$(T_{\delta_L d \check{\Box} \epsilon}, [T_\Box \parallel S_1(\overline{\delta_L d}) \parallel S_2(\epsilon)]_{\{push_1,pop_1,push_2,pop_2\}}) \in \mathcal{R}\ .$$

By an analysis similar from Lemma 4.22, we have $\mathcal{R}$ is a bisimulation up to $\leftrightarrows_b$. Therefore, $\mathcal{R} \subset \leftrightarrows_b$. Moreover, there is no infinite τ-transition sequence from the processes defined above. Therefore, $\mathcal{R} \subseteq \leftrightarrows_b^\Delta$.

Hence, we have $T_{\check{\Box}} \leftrightarrows_b^\Delta T$. □

Finally, we construct a finite control process for an RTM $\mathcal{M} = (\mathcal{S}_\mathcal{M}, \longrightarrow_\mathcal{M}, \uparrow_\mathcal{M}, \downarrow_\mathcal{M})$ as follows:

$$C_{s,d} = \Sigma_{(s,d,a,e,M,t) \in \longrightarrow_\mathcal{M}} (a.w!e.M!m.\Sigma_{f \in \mathcal{D}_\Box} r?f.C_{t,f})[+\mathbf{1}]_{s\downarrow_\mathcal{M}}\ (s \in \mathcal{S}_\mathcal{M}, d \in \mathcal{D}_\Box)\ .$$

Note that from Lemma 4.21, the above process can be specified in $TCP^\sharp$.

We prove the following lemma.

Lemma 4.24 (the finite control)**.** $\mathcal{T}(\mathcal{M}) \leftrightarroweq_b^\Delta [C_{\uparrow_\mathcal{M},\Box} \parallel T]_{\{r,w,L,R\}}$.

Proof. By the proof of Theorem 4.11, $\leftrightarroweq_\mathrm{b}^\Delta$ is compatible with parallel composition. Therefore, it is enough to show that $\mathcal{T}(\mathcal{M}) \leftrightarroweq_\mathrm{b}^\Delta [C_{\uparrow,\Box} \parallel T_{\check{\Box}}]_{\{r,w,L,R\}}$.

We define a binary relation $\mathcal{R}$ by:

$$\begin{array}{rcl}\mathcal{R} & = & \{((s, \delta_L\check{d}\delta_R), [C_{s,d} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}}) \mid s \in \mathcal{S}_M,\ \delta_L, \delta_R \in \mathcal{D}_\Box^*,\ d \in \mathcal{D}_\Box\} \\ & \cup & \{((s, \delta_L^{<}d\delta_R), [C_{s,f} \parallel T_{\delta_L^{<}d\delta_R}]_{\{r,w,L,R\}}) \mid s \in \mathcal{S}_M,\ \delta_L, \delta_R \in \mathcal{D}_\Box^*,\ d \in \mathcal{D}_\Box,\ \delta_L \neq \epsilon,\ \delta_L = \delta'_L f\} \\ & \cup & \{((s, \delta_L d^{>}\delta_R), [C_{s,f} \parallel T_{\delta_L d^{>}\delta_R}]_{\{r,w,L,R\}}) \mid s \in \mathcal{S}_M,\ \delta_L, \delta_R \in \mathcal{D}_\Box^*,\ d \in \mathcal{D}_\Box,\ \delta_R \neq \epsilon,\ \delta_R = f\delta'_R\} \\ & \cup & \{((s, \check{\Box}\delta_R), [C_{s,\Box} \parallel T_{\check{\Box}\delta_R}]_{\{r,w,L,R\}}) \mid s \in \mathcal{S}_M,\ \delta_R \in \mathcal{D}_\Box^*\} \\ & \cup & \{((s, \delta_L\check{\Box}), [C_{s,\Box} \parallel T_{\delta_L\check{\Box}}]_{\{r,w,L,R\}}) \mid s \in \mathcal{S}_M,\ \delta_L \in \mathcal{D}_\Box^*\}\ .\end{array}$$

We show that $\mathcal{R} \subseteq \leftrightarroweq_\mathrm{b}^\Delta$.

$(s, \delta_L\check{d}\delta_R)$ has the following transitions:

$$\begin{array}{ll}(s, \delta_L\check{d}\delta_R) \xrightarrow{a} (t, \delta_L^{<}e\delta_R) & \text{if } (s, d, a, e, L, t) \in \longrightarrow_\mathcal{M},\ \delta_L \neq \epsilon \\ (s, \delta_L\check{d}\delta_R) \xrightarrow{a} (t, \delta_L e^{>}\delta_R) & \text{if } (s, d, a, e, R, t) \in \longrightarrow_\mathcal{M},\ \delta_R \neq \epsilon \\ (s, \delta_L\check{d}\delta_R) \xrightarrow{a} (t, \check{\Box}e\delta_R) & \text{if } (s, d, a, e, L, t) \in \longrightarrow_\mathcal{M},\ \delta_L = \epsilon \\ (s, \delta_L\check{d}\delta_R) \xrightarrow{a} (t, \delta_L e\check{\Box}) & \text{if } (s, d, a, e, R, t) \in \longrightarrow_\mathcal{M},\ \delta_R = \epsilon\ .\end{array}$$

They are simulated by:

$$\begin{array}{l}[C_{s,d} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \xrightarrow{a} [w!e.L!m.\Sigma_{f\in\mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \\ \longrightarrow^* [C_{t,f} \parallel T_{\delta_L^{<}d\delta_R}]_{\{r,w,L,R\}},\ \text{if } (s, d, a, e, L, t) \in \longrightarrow_\mathcal{M},\ \delta_L \neq \epsilon,\ \delta_L = \delta'_L f \\ [C_{s,d} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \xrightarrow{a} [w!e.R!m.\Sigma_{f\in\mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \\ \longrightarrow^* [C_{t,f} \parallel T_{\delta_L d^{>}\delta_R}]_{\{r,w,L,R\}},\ \text{if } (s, d, a, e, R, t) \in \longrightarrow_\mathcal{M},\ \delta_R \neq \epsilon,\ \delta_R = f\delta'_R \\ [C_{s,d} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \xrightarrow{a} [w!e.L!m.\Sigma_{f\in\mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \\ \longrightarrow^* [C_{t,\Box} \parallel T_{\check{\Box}d\delta_R}]_{\{r,w,L,R\}},\ \text{if } (s, d, a, e, L, t) \in \longrightarrow_\mathcal{M},\ \delta_L = \epsilon \\ [C_{s,d} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \xrightarrow{a} [w!e.R!m.\Sigma_{f\in\mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L\check{d}\delta_R}]_{\{r,w,L,R\}} \\ \longrightarrow^* [C_{t,\Box} \parallel T_{\delta_L d\check{\Box}}]_{\{r,w,L,R\}},\ \text{if } (s, d, a, e, L, t) \in \longrightarrow_\mathcal{M},\ \delta_R = \epsilon\ .\end{array}$$

We apply a similar analysis to other pairs in $\mathcal{R}$. Using the proof strategy similar to Lemma 4.22, it is straightforward to show that $\mathcal{R}$ is a bisimulation up to $\leftrightarroweq_\mathrm{b}$. Hence, we have $\mathcal{R} \subset \leftrightarroweq_\mathrm{b}$. Moreover, using a similar strategy in the proof to show the reactive

Turing powerfulness of the π-calculus from Section 5.2(See also [61]), we can show that $\mathcal{R}$ satisfies the divergence-preserving condition. For every infinite τ-transition sequence in $\mathcal{T}(\mathcal{M})$, we can find an infinite τ-transition sequence in the transition system induced from $[C_{\uparrow_{\mathcal{M}},\Box} \parallel T]_{\{r,w,L,R\}}$. Therefore, $\mathcal{R} \subseteq \leftrightarrow_{\mathrm{b}}^{\Delta}$.

Hence, we have $\mathcal{T}(\mathcal{M}) \leftrightarrow_{\mathrm{b}}^{\Delta} [C_{\uparrow_{\mathcal{M}},\Box} \parallel T]_{\{r,w,L,R\}}$. □

Concluding, we have proved the following theorem.

Theorem 4.25 (the reactive Turing powerfulness of TCPN)**.** *$TCP^{\sharp}$ is reactively Turing powerful modulo $\leftrightarrow_{b}^{\Delta}$.*

4.6 Remarks

The results established in this chapter show that a revision of the operational semantics of sequential composition leads to a smoother integration of process theory and the classical theory of automata and formal languages. In particular, the correspondence between context-free processes and pushdown processes can be established up to strong bisimilarity, which does not seem to hold with the more standard operational semantics of sequential composition in a setting with intermediate termination [4]. Furthermore, the revised operational semantics of sequential composition also seems to work better in combination with the recursive operations of [21]. We conjecture that it is not possible to specify an always terminating counter or stack in a process calculus with iteration and nesting if the original operational semantics of sequential composition is used.

There are also some disadvantages to the revised operational semantics.

1. First of all, the negative premise in the operational semantics gives well-known formal complications in determining whether some process does, or does not, admit a transition. For instance, consider the following unguarded recursive specification:

$$X = X;Y + \mathbf{1}$$
$$Y = a.\mathbf{1} \ .$$

   It is not a priori clear whether an a-transition is possible from X: if X *only* has the option to terminate, then $X;Y$ can do the a-transition from Y, but then also X can do the a-transition, contradicting the assumption that X *only* has the option to terminate. Therefore, we have to restrict the semantics to guarded recursive specifications.

2. Second, as we have illustrated in Section 4.3, rooted branching bisimilarity is not compatible with respect to the new sequential composition operation. The divergence-preserving condition is required for the congruence property.

3. Finally, note that $(a.\mathbf{1} + \mathbf{1});b.\mathbf{1}$ is not strongly bisimilar to $(a.\mathbf{1};b.\mathbf{1}) + (\mathbf{1};b.\mathbf{1})$, and hence the operator ; does not distribute from the right over +. It is to be expected that there is no finite sound and ground-complete set of equational axioms for the process calculus $\text{TCP}^{;}$ with respect to strong bisimilarity without using an auxiliary operator. We leave for future work to further investigate the equational theory of sequential composition.

Another interesting future work is to establish the reactive Turing powerfulness on other process calculi with non-regular iterators based on the revised semantics of the sequential composition operator. For instance, we could consider the pushdown operator "\$" and the back-and-forth operator "⇆" introduced by Bergstra and Ponse in [21]. They are given by the following equations:

$$
\begin{aligned}
P_1{}^{\$}P_2 &= P_1;(P_1{}^{\$}P_2);(P_1{}^{\$}P_2) + P_2\\
P_1{}^{\leftrightarrows}P_2 &= P_1;(P_1{}^{\leftrightarrows}P_2);P_2 + P_2 \ .
\end{aligned}
$$

By analogy to the nesting operator, we can also give them some proper rules of operational semantics, and then use the calculus obtained by the revised semantics to define other versions of terminating counters. Moreover, we expect to establish their reactive Turing powerfulness.

Chapter 5

RTM and the π-Calculus

The π-calculus [68, 77] is a well-known process calculus for the formal specification of the behaviour of reactive systems in concurrency theory.

Research in concurrency theory has focussed on defining expressive process specification formalisms, modal logics, studying suitable behavioural equivalences, etc. Expressivity questions have also been addressed extensively in concurrency theory, especially in the context of the π-calculus (see, e.g., [47, 38]), but mostly pertaining to the so-called *relative expressivity* of process calculi. The absolute expressivity of process calculi, and in particular the question as to which interactive behaviour can actually be executed by a conventional computing system, has received less attention. In this chapter, we consider the expressivity of the π-calculus with respect to the model of Reactive Turing Machines.

We confirm that the π-calculus is expressive: every executable behaviour can be specified in the π-calculus up to divergence-preserving branching bisimilarity [43, 42], which is the finest behavioural equivalence discussed in van Glabbeek's seminal paper on behavioural equivalences [40]. Although divergence-preserving branching bisimilarity is not the behavioural equivalence that is most used in the π-calculus, we choose it for it is the finest one that fits in the theory of executability. Our proof explains how an arbitrary Reactive Turing Machine can be specified in the π-calculus. The specification consists of a component that specifies the behaviour of the tape memory, and a component that specifies the behaviour of the finite control of the Reactive Turing Machine under consideration. The specification of the behaviour of the tape memory is generic and elegantly uses the link mobility feature of the π-calculus.

We also prove that the converse is not true: it is possible to specify, in the π-calculus, transition systems that are not executable up to divergence-preserving branch-

ing bisimilarity. We shall analyze the discrepancy and identify two causes.

The first cause is that the π-calculus presupposes an infinite supply of names, which is technically essential both for the way input is modelled and for the way fresh name generation is implemented. The infinite supply of names in the π-calculus gives rise to an infinite alphabet of actions. The presupposed alphabet of actions of a Reactive Turing Machine is, however, purposely kept finite, since allowing Reactive Turing Machines to have an infinite alphabet of actions arguably leads to an unrealistic model of executability. As an alternative, we shall therefore investigate the executability of π-calculus behaviour subject to finitely many names, considering only the observable behaviour of a π-calculus term that refers to a finite subset of the set of names. The underlying assumption is that any realistic system will be based on a finite alphabet of input symbols.

The second cause is that, even under a finite name restriction, the transition system associated with a π-calculus term may still have unbounded branching. Transition systems with unbounded branching are not executable up to divergence-preserving branching bisimilarity, but unbounded branching behaviour can be simulated at the expense of sacrificing divergence preservation. We shall establish that, given a finite name restriction, the behaviour associated with a π-term is always executable up to the divergence-insensitive variant of branching bisimilarity.

We shall investigate the possibility of extending the Reactive Turing Machines with an infinite alphabet of actions in Chapter 6. To this end, an infinite alphabet of states or an infinite alphabet of data symbols has to be introduced. We shall see that such a machine is capable of simulating any effective transition system, thus the π-calculus can be simulated. However, such a model is unrealistic. According to the work in [26], a Turing machine with atoms is capable of dealing with the formalism of an infinite alphabet of names. As an analogy to Turing machine with atoms, we define Reactive Turing Machines with atoms, and show that every π-calculus term can be simulated by a Reactive Turing Machine with atoms up to the divergence-insensitive version of branching bisimilarity.

This chapter is organized as follows. In Section 5.1, we recall the operational semantics of the π-calculus with replication. In Section 5.2, we prove the reactive Turing power of the π-calculus modulo divergence-preserving branching bisimilarity: a finite specification of Reactive Turing Machines in the π-calculus is proposed and verified. In Section 5.3, we discuss the executability of transition systems associated with π-calculus processes. We first argue that the π-calculus is not executable in general. Then, we establish that every behaviour specified by the π-calculus restricted to finitely many names is executable modulo the divergence-insensitive variant of branching bisimilarity, but not modulo divergence-preserving branching bisimilarity. The chapter ends with a discussion of related work and some conclusions in Section 5.4.

5.1 The π-Calculus

5.1.1 Syntax

The π-calculus was proposed by Milner, Parrow and Walker in [69] as a language to specify processes with link mobility. The expressivity of many variants of the π-calculus has been extensively studied. In this thesis, we shall consider the basic version presented in the textbook by Sangiorgi and Walker [77], but the match prefix operator from [69] is not considered here. We recapitulate some definitions from [77] below and refer to the book for detailed explanations.

We presuppose a countably infinite set $\mathcal{N}$ of names; we use strings of lower case letters for elements of $\mathcal{N}$. Furthermore, we use upper case letters for π-calculus processes (which are also referred to as π-terms). The *prefixes*, *processes* and *summations* of the π-calculus are, respectively, defined by the following grammar:

$$\begin{array}{lcl} \pi & := & \overline{x}\, y \mid x(z) \mid \tau \qquad (x, y, z \in \mathcal{N}) \\ P & := & M \mid P \,|\, P \mid (z)P \mid\ !P \\ M & := & \mathbf{0} \mid \pi.P \mid M + M \ . \end{array}$$

We briefly explain the meaning of each notation above. $\mathbf{0}$ denotes the empty process; $\overline{x}\, y$ represents the event of sending a name y along the channel x; $x(z)$ represents the event of receiving a name along the channel x. τ represents an internal action; $+$ denotes a choice between two behaviours; $|$ is the parallel composition operator; $(z)P$ denotes a restriction of the name z in P; and $!P$ denotes the replication of P.

In $x(z).P$ and $(z)P$, the displayed occurrence of the name z is *binding* with scope P. An occurrence of a name in a process is *bound* if it is, or lies within the scope of, a binding occurrence in P; otherwise it is *free*. We use $\mathsf{fn}(P)$ to denote the set of names that occur free in P, and $\mathsf{bn}(P)$ to denote the set of names that occur bound in P.

We use $P\{z/y\}$ to denote a π-term obtained by substituting every occurrence of y to z in P; and we use $P\{\vec{z}/\vec{y}\}$ to denote a substitution of a sequence of names, i.e., $P\{\vec{z}/\vec{y}\} = (\dots(P\{z_1/y_1\})\{z_2/y_2\}\dots)\{z_n/y_n\}$ for $\vec{z} = (z_1, z_2, \dots, z_n)$ and $\vec{y} = (y_1, y_2, \dots, y_n)$.

A finite number of changes of bound names in a π-term is often called an α-*conversion*. We rephrase the definition in [77] as below.

Definition 5.1 (α-conversion)**.** Let P be a π-term, then Q is an α-*conversion* of P if it is obtained by a finite number of substitution operations of bound names from P, i.e., there exists $\vec{y} = (y_1, \dots, y_n)$, $\vec{z} = (z_1, \dots, z_n)$,$y_1, \dots, y_n \in \mathsf{bn}(P)$ and $Q = P\{\vec{z}/\vec{y}\}$. We write $P =_\alpha Q$ if P and Q are two π-terms that are α-convertible.

We call the equivalence class of the α-convertible terms of a π-term P the *α equivalence class* of P, which is defined as:

$$[P]_\alpha = \{Q \mid P =_\alpha Q\} \ .$$

Two α-convertible terms has the same behaviour, so we consider them as equivalent. Moreover, in Chapter 6, we shall introduce nominal sets to deal with infinite sets having finite quotients up to α-conversion.

Remark 5.2. For convenience, we sometimes want to abbreviate interactions that involve the transmission of no name at all, or more than one name. Instead of giving a full treatment of the polyadic π-calculus (see [77]), we define the following abbreviations:

$$\begin{aligned}
(z_1, \ldots, z_n)P &\stackrel{\text{def}}{=} (z_1)\ldots(z_n)P, \\
\overline{x}\langle y_1, \ldots, y_n\rangle.P &\stackrel{\text{def}}{=} (w)\overline{x}\,w.\overline{w}\,y_1.\cdots\overline{w}\,y_n.P \quad (w \notin \mathsf{fn}(P)), \text{ and} \\
x(z_1, \ldots, z_n).P &\stackrel{\text{def}}{=} x(w).w(z_1).\cdots w(z_n).P, \text{ for } n \in \mathbb{N}^+ \ .
\end{aligned}$$

5.1.2 Structural Operational Semantics

We define the operational behaviour of π-calculus processes by means of the structural operational semantics in Table 5.1, in which a ranges over the set of actions of the π-calculus

$$\mathcal{A}_\pi = \{x\,y, \overline{x}\,y, \overline{x}\,(z) \mid x, y, z \in \mathcal{N}\} \cup \{\tau\} \ .$$

The rules in Table 5.1 define on π-terms an $\mathcal{A}_\pi$-labelled transition relation $\longrightarrow$. We now briefly explain the effect of each rule. PREFIX consists of three rules for input, output and internal action prefixes, respectively. SUM states the rules for alternative choice; the process is free to choose one of the summands to perform a transition. PAR are the rules for parallel composition without communication, the process may perform a transition from one of its components. COM illustrates the communication of free names, and CLOSE illustrates the communication of bound names; a communication may happen between two parallel components, where one of them sends a name and the other receives it through an arbitrary channel. RES explains the effect of restriction, i.e., a transition may happen if it does not contain a restricted name. OPEN states the effect of sending a bound name; it leads to a scope extrusion regarding that bound name. REP shows the semantics of the replication operator; it always generates new instances of processes. ALPHA is the rule for α-conversion, i.e., α-convertible terms share the same set of transitions.

PREFIX

$$\frac{}{\tau.P \xrightarrow{\tau} P} \qquad \frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \frac{}{x(y).P \xrightarrow{xz} P\{z/y\}}$$

SUM

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

PAR

$$\frac{P \xrightarrow{a} P'}{P \mid Q \xrightarrow{a} P' \mid Q}\ \mathsf{bn}(a) \cap \mathsf{fn}(Q) = \emptyset$$

$$\frac{Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{a} P \mid Q'}\ \mathsf{bn}(a) \cap \mathsf{fn}(P) = \emptyset$$

COM

$$\frac{P \xrightarrow{\bar{x}y} P',\ Q \xrightarrow{xy} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \frac{P \xrightarrow{xy} P',\ Q \xrightarrow{\bar{x}y} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

CLOSE

$$\frac{P \xrightarrow{\bar{x}(z)} P',\ Q \xrightarrow{xz} Q'}{P \mid Q \xrightarrow{\tau} (z)(P' \mid Q')}\ z \notin \mathsf{fn}(Q) \qquad \frac{P \xrightarrow{xz} P',\ Q \xrightarrow{\bar{x}(z)} Q'}{P \mid Q \xrightarrow{\tau} (z)(P' \mid Q')}\ z \notin \mathsf{fn}(P)$$

RES

$$\frac{P \xrightarrow{a} P'}{(z)P \xrightarrow{a} (z)P'}\ z \notin a$$

OPEN

$$\frac{P \xrightarrow{\bar{x}z} P'}{(z)P \xrightarrow{\bar{x}(z)} P'}\ z \neq x$$

REP

$$\frac{P \xrightarrow{a} P'}{!P \xrightarrow{a} P' \mid !P} \qquad \frac{P \xrightarrow{\bar{x}y} P',\ P \xrightarrow{xy} P''}{!P \xrightarrow{\tau} (P' \mid P'') \mid !P} \qquad \frac{P \xrightarrow{\bar{x}(z)} P',\ P \xrightarrow{xz} P''}{!P \xrightarrow{\tau} (z)(P' \mid P'') \mid !P}$$

ALPHA

$$\frac{P \xrightarrow{a} P'}{Q \xrightarrow{a} P'}\ Q =_\alpha P$$

Table 5.1: Structural operational semantics of the π-calculus

We can associate with every π-term P an $\mathcal{A}_\pi$-labelled transition system $\mathcal{T}(P) = (\mathcal{S}_P, \longrightarrow_P, P)$ as follows.

Definition 5.3 (the LTS associated with a π-term)**.** Let P be a π-term. The transition system associated with P is $\mathcal{T}(P) = (\mathcal{S}_P, \longrightarrow_P, \uparrow_P)$, where

1. $\mathcal{S}_P = \{Q \mid Q \in \mathit{Reach}(P)\}$ is the set of all reachable π-terms from P by the operational semantics;
2. $\longrightarrow_P = \{(P, a, Q) \mid P \xrightarrow{a} Q\}$ is the set of transitions between all reachable π-terms; and
3. $\uparrow_P = P$ is the initial state.

We use the following example to illustrate the semantics of the π-calculus and an important feature of *link mobility*.

Example 5.4. We define three π-calculus processes as follows:

$$\begin{aligned} P &= \overline{x}\, z.v(w).\mathbf{0} \\ Q &= x(y).y(u).\mathbf{0} \\ R &= \overline{z}\, w.\mathbf{0}\ . \end{aligned}$$

Consider a process $(z)(P \mid R) \mid Q$, P and Q share a common channel name x, and P and R share a bound name z as illustrated in Figure 5.1. By the structural operational semantics, we have the following transition:

$$(z)(P \mid R) \mid Q \xrightarrow{\tau} v(w).\mathbf{0} \mid (z)(\overline{z}\, w.\mathbf{0} \mid z(u).\mathbf{0}) = P' \mid (z)(R \mid Q'\{z/y\})\ ,$$

where $P' = v(w).\mathbf{0}$ and $Q' = y(u).\mathbf{0}$. After the transition, P' and $Q'\{z/y\}$ do not share the channel name x, and the bound name z is now shared by R and $Q'\{z/y\}$. In other words, the link between the first component and the third component is broken and the link between the first component and the second component is moved to the second component and the third component of the parallel composition during the above τ-transition. We call this phenomenon *link mobility*. We shall take advantage of this feature to simulate the linking structure of the tape of an RTM.

5.1.3 Compatibility

The following lemma establishes that divergence-preserving branching bisimilarity is compatible with restriction and parallel composition. This will be a useful property when establishing the correctness of our simulation of RTMs in the π-calculus, in the next section.

Figure 5.1: An example to illustrate the link mobility of the π-calculus

Lemma 5.5 (compatibility with restriction and parallel composition)**.** *For all π-terms P, P', Q, and Q':*

1. *if $P \leftrightarrow_b^\Delta P'$, then $(x)P \leftrightarrow_b^\Delta (x)P'$;*
2. *if $P \leftrightarrow_b^\Delta P'$ and $Q \leftrightarrow_b^\Delta Q'$, then $P \mid Q \leftrightarrow_b^\Delta P' \mid Q'$.*

Proof. 1. It is straightforward to verify that the relation

$$\mathcal{R} = \{((x)P, (x)P') \mid P \text{ and } P' \text{ are } \pi\text{-terms s.t. } P \leftrightarrow_{\mathrm{b}}^\Delta P'\} \cup \leftrightarrow_{\mathrm{b}}^\Delta$$

is a divergence-preserving branching bisimulation relation.

2. We define the relation $\mathcal{R}$ by

$\{(P|Q, P'|Q') \mid P, P', Q, \text{ and } Q' \text{ are } \pi\text{-terms s.t. } P \leftrightarrow_{\mathrm{b}}^\Delta P' \text{ and } Q \leftrightarrow_{\mathrm{b}}^\Delta Q'\} \cup \leftrightarrow_{\mathrm{b}}^\Delta$;

we verify that $\mathcal{R}$ is a divergence-preserving branching bisimulation. To this end, we first suppose that $P \mid Q \xrightarrow{a} R$, and distinguish three cases according to which operational rule is applied last in the derivation of this transition:

(a) if $P \xrightarrow{a} P_1$ and $R = P_1 \mid Q$, then, since $P \leftrightarrow_{\mathrm{b}}^\Delta P'$, there exist P_1'' and P_1', such that $P' \longrightarrow^* P_1'' \xrightarrow{a} P_1'$ with $P' \leftrightarrow_{\mathrm{b}}^\Delta P_1''$ and $P_1 \leftrightarrow_{\mathrm{b}}^\Delta P_1'$. Thus $P' \mid Q' \longrightarrow^* P_1'' \mid Q \xrightarrow{a} P_1' \mid Q$, and, according to the definition of $\mathcal{R}$, we have $P \mid Q \mathrel{\mathcal{R}} P_1'' \mid Q$ and $P_1 \mid Q \mathrel{\mathcal{R}} P_1' \mid Q$.

(b) If $P \xrightarrow{\bar{x}y} P_1$, $Q \xrightarrow{xy} Q_1$, $R = P_1 \mid Q_1$, and $a = \tau$, then, since $P \leftrightarrow_{\mathrm{b}}^\Delta P'$ and $Q \leftrightarrow_{\mathrm{b}}^\Delta Q'$, there exist P_1'', P_1', Q_1'' and Q_1' such that $P' \longrightarrow^* P_1'' \xrightarrow{\bar{x}y} P_1'$ with $P \leftrightarrow_{\mathrm{b}}^\Delta P_1''$ and $P_1 \leftrightarrow_{\mathrm{b}}^\Delta P_1'$, and $Q' \longrightarrow^* Q_1'' \xrightarrow{xy} Q_1'$ with $Q \leftrightarrow_{\mathrm{b}}^\Delta Q_1''$ and $Q_1 \leftrightarrow_{\mathrm{b}}^\Delta Q_1'$. Hence, it follows that $P' \mid Q' \longrightarrow^* P_1'' \mid Q_1'' \xrightarrow{\tau} P_1' \mid Q_1'$, with $P \mid Q \mathrel{\mathcal{R}} P_1'' \mid Q_1''$ and $P_1 \mid Q_1 \mathrel{\mathcal{R}} P_1' \mid Q_1'$.

(c) If $P \xrightarrow{\bar{x}(z)} P_1$, $Q \xrightarrow{x\,z} Q_1$, $R = (z)(P_1 \mid Q_1)$, and $a = \tau$, then, since $P \mathrel{\underline{\leftrightarrow}}_b^{\Delta} P'$ and $Q \mathrel{\underline{\leftrightarrow}}_b^{\Delta} Q'$, there exist P_1'', P_1', Q_1'' and Q_1' such that $P' \longrightarrow^* P_1'' \xrightarrow{\bar{x}(z)} P_1'$ with $P \mathrel{\underline{\leftrightarrow}}_b^{\Delta} P_1''$ and $P_1 \mathrel{\underline{\leftrightarrow}}_b^{\Delta} P_1'$, and $Q' \longrightarrow^* Q_1'' \xrightarrow{x\,z} Q_1'$ with $Q \mathrel{\underline{\leftrightarrow}}_b^{\Delta} Q_1''$ and $Q_1 \mathrel{\underline{\leftrightarrow}}_b^{\Delta} Q_1'$. Hence, it follows that $P' \mid Q' \longrightarrow^* P_1'' \mid Q_1'' \xrightarrow{\tau} (z)(P_1' \mid Q_1')$, with $P \mid Q \,\mathcal{R}\, P_1'' \mid Q_1''$ and $(z)(P_1 \mid Q_1) \,\mathcal{R}\, (z)(P_1' \mid Q_1')$.

The symmetric cases can be proved analogously.

Moreover, we argue that the divergence-preserving condition holds, since all the transitions above are mutually simulated without introducing a divergence. If $P \mid Q$ has a divergence, then the divergence is simulated by $P' \mid Q'$ according to our analysis; otherwise, if $P \mid Q$ does not have a divergence, then no divergence can be derived from $P' \mid Q'$.

□

5.2 Reactively Turing Powerfulness of the π-Calculus

In the previous section, we have introduced the π-calculus as a language to specify behaviour of systems with link mobility, and we have proposed RTMs to define a notion of executable behaviour. In this section we prove that every executable behaviour can be specified in the π-calculus up to divergence-preserving branching bisimilarity. To this end, we associate with every RTM $\mathcal{M}$ a π-term P that simulates the behaviour of $\mathcal{M}$ up to divergence-preserving branching bisimilarity, that is, $\mathcal{T}(\mathcal{M}) \mathrel{\underline{\leftrightarrow}}_b^{\Delta} \mathcal{T}(P)$.

The structure of our specification is illustrated in Figure 5.2. In this figure, each node represents a parallel component of the specification, each labelled arrow stands for a communication channel, and the dashed lines represent the links maintained by the components. Moreover, the equalities on arrows and dashed lines indicate the correspondence between the names as they occur in the definitions of the linked components. The specification consists of a generic finite specification of the behaviour of a tape (parallel components H_k, $B_{l,k}$, C_k and $B_{r,k}$, for $k \in \mathbb{Z}$ in Figure 5.2), and a finite specification of a control process that is specific for the RTM $\mathcal{M}$ under consideration (parallel component S in Figure 5.2). We first discuss the generic specification of the tape in Section 5.2.1; then we discuss how to add a suitable control process specific for $\mathcal{M}$ in Section 5.2.2; and we finally prove that $\mathcal{M}$ is simulated by the parallel composition of the two parts.

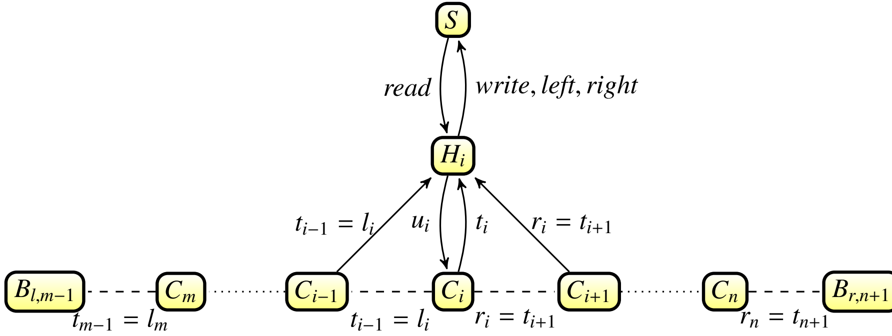


Figure 5.2: Specification of an RTM utilizing the linking structure of the π-calculus

5.2.1 Tape

In [5], the behaviour of the tape of a Turing machine is finitely specified in ACP_τ making use of finite specifications of two stacks. The specification is not easily modified to take intermediate termination into account, and therefore, in [13], an alternative solution is presented, specifying the behaviour of a tape in TCP_τ by using a finite specification of a queue (see also [4]). In the previous chapter (see Section 4.5), we also gave a specification of a tape that is always terminating. That specification essentially used sequential composition (with a revised semantics) and a nesting operator. Neither operation is present in the π-calculus. In this section, we exploit the link passing feature of the π-calculus to give a more direct specification. In particular, we shall model the tape as a collection of cells endowed with a link structure that organises them in a linear fashion. Note that we do not consider termination since the π-calculus does not (explicitly) distinguish between successful and unsuccessful termination.

We first give an informal description of the behaviour of a tape. The state of a tape is characterised by a tape instance $\delta_L \check{d} \delta_R$, with $d \in \mathcal{D}_\Box$ and $\delta_L, \delta_R \in \mathcal{D}_\Box{}^*$, consisting of a finite (but unbounded) sequence of data with the current position of the tape head indicated by ˇ. The tape may then exhibit the following observable actions:

1. $\overline{read}\, d$: the datum under the tape head is output along the channel *read*;
2. *write* e: a datum e is written on the position of the tape head, resulting in a new tape instance $\delta_L \check{e} \delta_R$; and
3. *left*, *right*: the tape head moves one position left or right, resulting in $\delta_L^{\,\check{}} d \delta_R$ or $\delta_L d^{\,\check{}} \delta_R$, respectively.

Henceforth, we assume that tape symbols are included in the set of names, i.e., that $\mathcal{D}_\Box \subseteq \mathcal{N}$.

In our π-calculus specification of a tape, each individual tape cell is specified as a separate component, and there is a separate component modelling the tape head. A tape cell stores a datum d, represented by a free name in the specification, and it has pointers l and r to its left and right neighbour cells. Furthermore, it has two links to the component modelling the tape head: the link u is used by the tape head for updating the datum, and the link t serves as a general communication channel for communicating all relevant information about the cell to the tape head. The following π-term represents the behaviour of a tape cell:

$$\begin{aligned} C &\stackrel{\text{def}}{=} c(t,l,r,u,d).C(t,l,r,u,d) \\ C(t,l,r,u,d) &\stackrel{\text{def}}{=} u(e).\bar{c}\,\langle t,l,r,u,e\rangle.\mathbf{0} + \bar{t}\,\langle l,r,u,d\rangle.\bar{c}\,\langle t,l,r,u,d\rangle.\mathbf{0} \ . \end{aligned}$$

Note that the behaviour of an individual tape cell $C(t,l,r,u,d)$ is as follows: either it receives along channel u an update e for its datum d, after which it recreates itself with datum e in place of d; or it outputs all relevant information about itself (i.e., the links to its left and right neighbours, its update channel u, and the stored datum d) to the tape head along channel t, after which it recreates itself. A cell is created by a synchronisation on name c, by which all relevant information about the cell is passed; we include a component $!C$ for the generation of new incarnations of existing tape cells.

At all times, the number of tape cells will be finite, but there is no a priori bound on the number of tape cells used in an execution of an RTM. To model the unbounded nature of the tape, we define a process B that serves to generate new blank tape cells on either side of the tape whenever needed:

$$\begin{aligned} B &\stackrel{\text{def}}{=} b_l(t,r).(u,l)B_l(t,l,r,u) + b_r(t,l).(u,r)B_r(t,l,r,u) \\ B_l(t,l,r,u) &\stackrel{\text{def}}{=} \bar{t}\,\langle l,r,u,\Box\rangle.(\bar{c}\,\langle t,l,r,u,\Box\rangle.\mathbf{0} \mid \overline{b_l}\,\langle l,t\rangle.\mathbf{0}) \\ B_r(t,l,r,u) &\stackrel{\text{def}}{=} \bar{t}\,\langle l,r,u,\Box\rangle.(\bar{c}\,\langle t,l,r,u,\Box\rangle.\mathbf{0} \mid \overline{b_r}\,\langle t,r\rangle.\mathbf{0}) \ . \end{aligned}$$

Note that B offers the choice to either create a blank tape cell at the left-hand side of the tape through $b_l(t,r)$, or a blank tape cell at the right-hand side of the tape through $b_r(t,l)$. In the first case, suppose the original leftmost cell has the channels t_o and l_o, for itself and its left neighbour, respectively; then for the new cell, we have $t = l_o$ and $r = t_o$, in order to maintain the links to its neighbour. Moreover, at the creation of the new blank cell, two new links are utilized: u is the update channel of the new blank cell, and l will later be used as the link to generate another cell. Thus a new cell is

generated from the process $B_l(t, l, r, u)$ by a synchronisation along channel c, and the cell generator on the left is updated by $\overline{b_l}\langle l, t\rangle.\mathbf{0}$. In the second case, a symmetrical procedure is implemented by $B_r(t, l, r, u)$.

Throughout the simulation of an RTM, the number of parallel components modelling individual tape cells will grow. We shall presuppose a numbering of these parallel components with consecutive integers from some interval $[m, n]$ (m and n are integers such that $m \leq n$), in agreement with the link structure. The numbering is reflected by a naming scheme that adds the subscript i to the links t, l, r, u and d of the ith cell. We abbreviate $C(t_i, l_i, r_i, u_i, d_i)$ by $C_i(d_i)$, and $B_l(t_i, l_i, r_i, u_i)$ and $B_l(t_i, l_i, r_i, u_i)$ by $B_{l,i}$ and $B_{r,i}$, respectively. Let $\vec{d}_{[m,n]} = d_m, d_{m+1}, \ldots, d_{n-1}, d_n$; we define:

$$\begin{aligned} Cells_{[m,n]}(\vec{d}_{[m,n]}) &\stackrel{\text{def}}{=} (b_l, b_r, c)(B_{l,m-1} \mid C_m(d_m) \mid C_{m+1}(d_{m+1}) \mid \\ &\cdots \quad \mid C_{n-1}(d_{n-1}) \mid C_n(d_n) \mid B_{r,n+1} \mid !C \mid !B) \ . \end{aligned}$$

Note that an intuitive illustration of the linking structure in process *Cells* is shown in the bottom row of Figure 5.2.

The component modelling the tape head serves as the interface between the tape cells and the RTM-specific control process. The four channels for tape actions are referring to names $\overline{read}$, *write*, *left*, and *right*. An instance of the tape head process $H(t, l, r, u, d)$ is initiated by the 5 names that parameterise the current cell, which is defined by

$$\begin{aligned} H &\stackrel{\text{def}}{=} h(t, l, r, u, d).H(t, l, r, u, d) \\ H(t, l, r, u, d) &\stackrel{\text{def}}{=} \overline{read}\, d.\overline{h}\langle t, l, r, u, d\rangle.\mathbf{0} + write(e).\overline{u}\, e.\overline{h}\langle t, l, r, u, e\rangle.\mathbf{0} \\ &+ \quad left.l(l', r', u', d').\overline{h}\langle l, l', r', u', d'\rangle.\mathbf{0} \\ &+ \quad right.r(l', r', u', d').\overline{h}\langle r, l', r', u', d'\rangle.\mathbf{0} \ . \end{aligned}$$

The tape head maintains two links to the current cell (a communication channel t and an update channel u), as well as links to its left and right neighbour cells (l and r, respectively). Furthermore, the tape head remembers the datum d in the current cell. The datum d may be output along the *read*-channel. Furthermore, a new datum e may be received through the *write*-channel, which is then forwarded through the update channel u to the current cell. Finally, the tape head may receive instructions to move left or right, which has the effect of receiving information about the left or right neighbours of the current cell through l or r, respectively. In all cases, a new incarnation of the tape head is started, with a call on the h-channel.

Let $\vec{t}_{[m,n]} = t_m, t_{m+1}, \ldots, t_{n-1}, t_n$, let $\vec{u}_{[m,n]} = u_m, u_{m+1}, \ldots, u_{n-1}, u_n$, and let $H_i =$

$H(t_i, l_i, r_i, u_i, d_i)$; we define

$$Tape^{i}_{[m,n]}(\vec{d}_{[m,n]}) \stackrel{\text{def}}{=} (\vec{t}_{[m-1,n+1]}, \vec{u}_{[m,n]})((h)(H_i \mid !H) \mid Cells_{[m,n]}(\vec{d}_{[m,n]})) \ .$$

We prove the following lemmas about the behaviour of our tape specification:

Lemma 5.6. *The following statements are valid:*

1. $(c)(\overline{c}\,\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !C) \leftrightarrows^{\Delta}_{b} (c)(C_i(d_i) \mid !C)$;
2. $(b_l, b_r)(\overline{b_l}\,\langle t_m, r_m\rangle.\mathbf{0} \mid !B) \leftrightarrows^{\Delta}_{b} (b_l, b_r, u_m, l_m)(B_{l,m} \mid !B)$;
3. $(b_l, b_r)(\overline{b_r}\,\langle t_n, l_n\rangle.\mathbf{0} \mid !B) \leftrightarrows^{\Delta}_{b} (b_l, b_r, u_n, r_n)(B_{r,n} \mid !B)$; *and*
4. $(h)(\overline{h}\,\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !H) \leftrightarrows^{\Delta}_{b} (h)(H_i \mid !H)$.

Proof. We just show the first statement. There is only one transition from the process $(c)(\overline{c}\,\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !C)$, which labelled by τ and exactly leads to $(c)(C_i(d_i) \mid !C)$, i.e.,

$$(c)(\overline{c}\,\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !C) \xrightarrow{\tau} (c)(C_i(d_i) \mid !C) \ .$$

Hence, we have $(c)(\overline{c}\,\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !C) \leftrightarrows^{\Delta}_{\mathrm{b}} (c)(C_i(d_i) \mid !C)$.

One may verify in the same way that all the other statements hold. □

We write $P \xrightarrow{a} \leftrightarrows^{\Delta}_{\mathrm{b}} P'$ for "there is a P'' such that $P \xrightarrow{a} P''$ and $P'' \leftrightarrows^{\Delta}_{\mathrm{b}} P'$".

Lemma 5.7. $Tape^{i}_{[m,n]}(\vec{d}_{[m,n]}) \xrightarrow{a} T'$ *if, and only if:*

1. $a = \overline{read}\, d_i$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i}_{[m,n]}(\vec{d}_{[m,n]})$;
2. $a = write\, e$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i}_{[m,n]}(\vec{d}_{[m,i-1]}, e, \vec{d}_{[i+1,n]})$, *where* $e \in \mathcal{D}_{\Box}$;
3. $a = left$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i-1}_{[m,n]}(\vec{d}_{[m,n]})$, *for* $i > m$;
4. $a = left$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i-1}_{[m-1,n]}(\Box, \vec{d}_{[m,n]})$, *for* $i = m$;
5. $a = right$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i+1}_{[m,n]}(\vec{d}_{[m,n]})$, *for* $i < n$; *or*
6. $a = right$ *and* $T' \leftrightarrows^{\Delta}_{b} Tape^{i+1}_{[m,n+1]}(\vec{d}_{[m,n]}, \Box)$, *for* $i = n$.

Proof. $Tape^i_{[m,n]}(\vec{d}_{[m,n]})$ has six possible types of outgoing transitions, namely, reading through the name *read*, writing through the name *write*, moving the head through names *left*, and *right* with or without generating a new tape cell, respectively. We use T' to denote the resulting process. We just argue (only in the first case) that T' is indeed bisimilar to $Tape^i_{[m,n]}(\vec{d}_{[m,n]})$.

$$Tape^i_{[m,n]}(\vec{d}_{[m,n]}) \xrightarrow{\overline{read}\, d_i} (\vec{t}_{[m-1,n+1]}, \vec{u}_{[m,n]})((h)(\overline{h}\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !H) \mid Cells_{[m,n]}) = T' \ .$$

By Lemma 5.6, we have $(h)(\overline{h}\langle t_i, l_i, r_i, u_i, d_i\rangle.\mathbf{0} \mid !H) \leftrightarrow^{\Delta}_{b} (h)(H_i \mid !H)$, so by Lemma 5.5, we have $T' \leftrightarrow^{\Delta}_{b} Tape^i_{[m,n]}(\vec{d}_{[m,n]})$. Moreover, only a finite sequence of τ transitions is introduced here, so we do not introduce any divergence.

One may verify in the same way that all the other statements hold. □

5.2.2 Finite Control

We associate with every RTM $\mathcal{M} = (Q, \mapsto, Ini)$ a finite specification of its control process S. Here m can be either *left* or *right*, and S is defined as follows:

$$S \stackrel{\text{def}}{=} \sum_{s \in Q} s. \sum_{d \in \mathcal{D}_\Box} d.S_{s,d}$$

$$S_{s,d} \stackrel{\text{def}}{=} \sum_{(s,d,a,e,m,t) \in \mapsto} a.\overline{write}\, e.\overline{m}\, .read(f).\bar{t}\, .\overline{f}\, .\mathbf{0} \ .$$

We assume that $Q \subseteq \mathcal{N}$, $\mathcal{D}_\Box \subseteq \mathcal{N}$ and $Q \cap \mathcal{D}_\Box = \emptyset$. We let $\vec{s} = s_1, s_2, \ldots, s_k \in Q$, and $\vec{e} = e_1, e_2, \ldots, e_l \in \mathcal{D}_\Box$ be two vectors that contain all states and data symbols, respectively; we define the control process as:

$$Control_{s,d} \stackrel{\text{def}}{=} (\vec{s}, \vec{e})(S_{s,d} \mid !S), \ s \in Q, d \in \mathcal{D}_\Box \ .$$

On each call of the control process, an instance from the replication of S reads the current state s from the previous stage, and the current datum d. After that, the control process $S_{s,d}$ chooses a transition (s, d, a, e, m, t) from the transition rules of $\mathcal{M}$ that is correlated with the s and d, and it performs a sequence of actions. First, it executes an a-labelled transition, then it writes the updated data to the current cell on the tape, and does the move instructions. Finally, it reads the data f from the new position of the tape, and passes the resulting state t and updated data f to reach another process $S_{t,f}$.

The following lemma illustrates the behaviour of the control process.

Lemma 5.8. *Given an RTM $\mathcal{M} = (\mathcal{Q}, \mapsto, Ini)$ and a control process defined as above, then for every $s \in \mathcal{Q}$ and $d \in \mathcal{D}_\Box$, we have the following transition sequence:*

$$Control_{s,d} \xrightarrow{a} (\vec{s}, \vec{e})(\overline{write}\, e.\overline{m}\, .read(f).\bar{t}\, .\overline{f}\, .\mathbf{0} \mid !S\,) \xrightarrow{\overline{write}\, e} \xrightarrow{\overline{m}} \xrightarrow{read\, f} \underline{\leftrightarrow}^{\Delta}_{b}\ Control_{t,f} \ ,$$

if and only if there is a transition rule $(s, d, a, e, m, t) \in \mapsto$.

Finally, for a given RTM $\mathcal{M}$, we associate with every configuration $(s, \delta_L \check{d} \delta_R)$ a π-term $M_{s,\delta_L \check{d} \delta_R}$, consisting of a parallel composition of the specifications of its tape instance and control process. Let $\vec{r} = read, write, left, right$; we define

$$M_{s,\delta_L \check{d} \delta_R} = (\vec{r})(Control_{s,d} | Tape^{i}_{[m,n]}(\vec{d}_{[m,n]})), \text{ where } \vec{d}_{[m,i-1]} = \delta_L,\ d_i = d, \text{ and } \vec{d}_{[i+1,n]} = \delta_R \ .$$

The following lemma shows that $M_{s,\delta_L \check{d} \delta_R}$ actually simulates every step of a transition of an RTM.

Lemma 5.9. *Given an RTM $\mathcal{M} = (\mathcal{Q}, \mapsto, Ini)$, we associate with every configuration $(s, \delta_L \check{d} \delta_R)$ a specification $M_{s,\delta_L \check{d} \delta_R}$. Moreover, there is a one-to-one correspondence as illustrated in Figure 5.3 between their transitions:*

$$M_{s,\delta_L \check{d} \delta_R} \xrightarrow{a} \underline{\leftrightarrow}^{\Delta}_{b}\ M_{t,\delta'_L \check{f} \delta'_R} \ ,$$

if and only if there is a transition $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta'_L \check{f} \delta'_R)$.

Proof. 1. For the "if" part, we assume that there is a transition of the RTM

$$(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta'_L \check{f} \delta'_R) \ .$$

According to the semantics of the RTM in Definition 2.11, the transition results from the application of the rule $(s, d, a, e, m, t) \in \mapsto$. Then according to Lemma 5.8, we have

$$M_{s,\delta_L \check{d} \delta_R} \xrightarrow{a} (\vec{r}, \vec{s}, \vec{e})(\overline{write}\, e.\overline{m}\, .read(f).\bar{t}\, .\overline{f}\, .\mathbf{0} \mid !S \mid Tape^{i}_{[m,n]}(\vec{d}_{[m,n]})) = M' \ .$$

In the next step we just prove that

$$M' \ \underline{\leftrightarrow}^{\Delta}_{b}\ (\vec{r}, \vec{s}, \vec{e})(\overline{m}\, .read(f).\bar{t}\, .\overline{f}\, .\mathbf{0} | !S | Tape^{i}_{[m,n]}(d_m, \ldots, d_{i-1}, e, d_{i+1}, \ldots, d_n)) = M'' \ .$$

By Lemma 5.7, there exists a T such that

$$M' \xrightarrow{\tau} (\vec{r}, \vec{s}, \vec{e})(\overline{m}\, .read(f).\bar{t}\, .\overline{f}\, .\mathbf{0} \mid !S \mid T') \ ,$$

and $T' \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} Tape^{i}_{[m,n]}(d_m, \ldots, d_{i-1}, e, d_{i+1}, \ldots, d_n)$. Thus $M' \xrightarrow{\tau} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} M''$ according to Lemma 5.5. Moreover, since M' has only one outgoing τ-transition, we have $M' \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} M''$.

Hence, let $T'' = Tape^{i'}_{[m',n']}(\vec{d'}_{[m',n']})$, *where* $\vec{d'}_{[m',n']} = \delta'_L \check{f} \delta'_R$, and we get a transition sequence,

$$\begin{aligned}
M' &\xrightarrow{\tau} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} (\vec{r}, \vec{s}, \vec{e})(\overline{m}\,.read(f).\bar{t}\,.\overline{f}\,.\mathbf{0} \mid !S \mid T') \\
&\xrightarrow{\tau} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} (\vec{r}, \vec{s}, \vec{e})(read(f).\bar{t}\,.\overline{f}\,.\mathbf{0} \mid !S \mid T'') \\
&\xrightarrow{\tau} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} (\vec{r}, \vec{s}, \vec{e})(\bar{t}\,.\overline{f}\,.\mathbf{0} \mid !S \mid T'') \\
&\longrightarrow^{*} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} M_{t, \delta'_L \check{f} \delta'_R} \ .
\end{aligned}$$

by applying Lemma 5.7.

2. For the "only iff" part, we assume that there is a sequence of transitions

$$M_{s, \delta_L \check{d} \delta_R} \xrightarrow{a} \underline{\leftrightarrow}_{\mathrm{b}}^{\Delta} M_{t, \delta'_L \check{f} \delta'_R} \ ,$$

Then, by Lemma 5.8, we have that there is a transition rule $(s, d, a, e, m, t) \in \mapsto$. By Definition 2.11, the RTM has a transition

$$(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta'_L \check{f} \delta'_R) \ .$$

□

Now we proceed to show that the specification simulates the execution of an RTM up to divergence-preserving branching bisimilarity. By Lemma 5.9, every transition of an RTM is simulated by the specification. Moreover, we need to show that every divergence in the transition system of an RTM also leads to a divergence in the simulation. Hence, we shall distinguish the τ-labelled transitions in RTMs from the τ-labelled transitions introduced by the simulation. To this end, we rename the τ to another special label in τ-labelled rules in the RTM.

Lemma 5.10. *Given an RTM $\mathcal{M}$, we have*

$$\mathcal{T}(M_{Ini, \check{\Box}}) \underline{\leftrightarrow}_{b}^{\Delta} \mathcal{T}(\mathcal{M}) \ .$$

Proof. Let $\mathcal{M} = (Q, \mapsto, Ini)$, and let $i \notin \mathcal{A}_\tau$ be a special symbol. We construct an auxiliary RTM $\mathcal{M}' = (Q, \mapsto', Ini)$, where $\mapsto'$ is obtained by replacing every τ-labelled

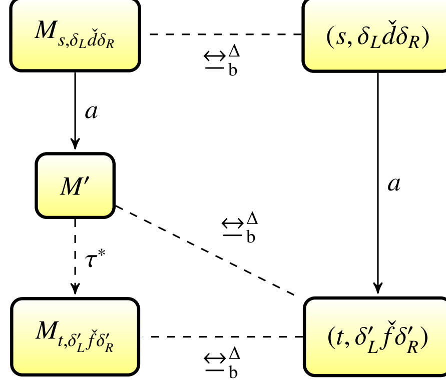


Figure 5.3: Bisimulation relation between $M_{s,\delta_L,\check{d}\delta_R}$ and $(s, \delta_L, \check{d}\delta_R)$

transition in $\mapsto$ by an i-labelled transition. Let $M'_{Ini,\check{\Box}}$ be the π-term associated with the initial configuration of $\mathcal{M}'$ as above.

We consider the relation

$$\mathcal{R}' = \{(M'_{s,\delta_L\check{d}\delta_R}, (s, \delta_L\check{d}\delta_R)) \mid s \in \mathcal{Q},\ \delta_L,\ \delta_R \in \mathcal{D}_\Box^*,\ \check{d} \in \check{\mathcal{D}}_\Box\} \ .$$

$\mathcal{R}'$ is a branching bisimulation up to $\underline{\leftrightarrow}_b$ which could be verified by Lemma 5.9. For then we establish that $\mathcal{T}(M'_{Ini,\check{\Box}}) \underline{\leftrightarrow}_b \mathcal{T}((Ini, \check{\Box}))$ by Lemma 2.9.

Now we proceed to show the branching bisimulation that relates $\mathcal{T}(M'_{Ini,\check{\Box}})$ and $\mathcal{T}((Ini, \check{\Box}))$ is divergence-preserving.

Note that there is no τ-transition in $\mathcal{M}'$, which means $\mathcal{T}((Ini, \check{\Box}))$ has no divergence. Then, by Lemma 5.9, the specification of a certain configuration $M'_{s,\delta_L\check{d}\delta_R}$ can only do transitions labelled with a, where $a \in \mathcal{A} \cup \{i\}$, i.e.

$$M'_{s,\delta_L\check{d}\delta_R} \xrightarrow{a} M' \underline{\leftrightarrow}_b^\Delta M'_{t,\delta'_L\check{f}\delta'_R} \ .$$

Since there is no τ-labelled transition from the term $M'_{t,\delta'_L\check{f}\delta'_R}$, it follows that M' has no divergence either. Hence, none of the terms reachable from $M'_{s,\delta_L\check{d}\delta_R}$ introduce divergence, and we have

$$\mathcal{T}(M'_{Ini,\check{\Box}}) \underline{\leftrightarrow}_b^\Delta \mathcal{T}((Ini, \check{\Box})) \ .$$

Finally, we switch back to $\mathcal{M}$, by changing all the i-labelled transitions to τ, and we let $M_{Ini,\check{\Box}}$ be the specification of the initial state of $\mathcal{M}$. We can also establish that

the relation

$$\mathcal{R}' = \{(M_{s,\delta_L\check{d}\delta_R}, (s, \delta_L\check{d}\delta_R)) \mid s \in \mathcal{Q}, \delta_L, \delta_R \in \mathcal{D}_\Box^*, \check{d} \in \check{\mathcal{D}}_\Box\}$$

is a branching bisimulation up to $\leftrightarrow_b$. Moreover, note that every infinite sequence of the form $\xrightarrow{i}\longrightarrow^*\xrightarrow{i}\longrightarrow^*\cdots$ from $M'_{Ini,\check{\Box}}$ corresponds with an infinite sequence of the form $\xrightarrow{i}\xrightarrow{i}\cdots$ from $\mathcal{M}'$, and vice versa. Additionally, there is no divergence from $M'_{Ini,\check{\Box}}$. Therefore, every infinite τ-labelled sequence from $M_{Ini,\check{\Box}}$ corresponds with an infinite τ-labelled sequence from $\mathcal{M}$. So we can conclude that $\mathcal{R} \subseteq \leftrightarrow_b^\Delta$.

As a remark, the transition systems associated with $M'_{Ini,\check{\Box}}$ and $M_{Ini,\check{\Box}}$ are isomorphic up to a renaming of occurrences of i into τ, and similarly for the transition systems associated with the RTMs, and hence, since the transition systems with the i's do not have divergence at all, the ones in which i's are replaced by τ's only have divergences of a certain shape. These τ-labelled transitions of the RTM are clearly simulated by the π-calculus specification and vice versa. Therefore, the above branching bisimulation is divergence-preserving. □

Thus we have the following expressivity result for the π-calculus.

Theorem 5.11 (reactively Turing powerfulness of the π-calculus)**.** *The π-calculus is reactively Turing powerful modulo* $\leftrightarrow_b^\Delta$.

5.3 Executability of Finite π-Calculus

We have proved that every executable behaviour can be specified in the π-calculus modulo divergence-preserving branching bisimilarity. We now investigate to what extent behaviour specified in the π-calculus is executable. Recall that we have defined executable behaviour as behaviour of an RTM. So, in order to prove that the behaviour specified by a π-term is executable, we need to show that the transition system associated with this π-term is behaviourally equivalent to the transition system associated with some RTM.

5.3.1 A Gap Between RTMs and the π-Calculus

Note that there is an apparent mismatch between the formalisms of RTMs and the π-calculus. On the one hand, the notion of RTM as we have defined it in Section 2.2 presupposes *finite* sets $\mathcal{A}_\tau$ and $\mathcal{D}_\Box$ of actions and data symbols, and also the transition relation of an RTM is *finite*. As a consequence, the transition system associated with

an RTM is finitely branching, and, in fact, its branching degree is bounded by a natural number (see Proposition 2.21). Note that this does not mean that RTMs cannot deal with data of unbounded size; it only means that it has to be encoded using finitely many symbols. The π-calculus, on the other hand, presupposes an infinite set of names by which an infinite set of actions $\mathcal{A}_\pi$ is generated. Furthermore, the transition system associated with a π-term by the structural operational semantics (see Table 5.1) may contain states with an infinite branching degree, due to the rules for input prefix and bound output prefix.

In this section and in Chapter 6, we explore three ways to bridge the gap and establish a simulation of the π-calculus by RTMs.

First, we consider an alternative operational semantics for the π-calculus in which the observable behaviour of every π-term is restricted to some presupposed finite subset of names. We shall see that this restricted variant of the π-calculus is executable up to branching bisimilarity; the simulation in general does require the use of divergence to simulate any unbounded branching that may occur in π-calculus processes.

Then, we consider an extension of the formalism of RTMs allowing infinite sets of actions in Section 6.1. We prove that the π-calculus processes can be simulated by RTMs if both arbitrary infinite sets of actions and infinite sets of states/data symbols are allowed. The result is hardly surprising since every effective transition system can be simulated up to branching bisimilarity by an RTM with only three states if an infinite set of data symbols is allowed (see the proof of Theorem 6.5).

Finally, we consider an extension of the notion of RTMs, referred to as RTMs with atoms in Chapter 6, following the work of Bojańczyk, Klin, Lasota, and Toruńczyk [26]. RTMs with atoms also facilitate infinite sets of actions, states and data symbols, but with the explicit proviso that these infinite sets have no other structure than equality and that they are, in fact, orbit-finite (the notion of sets with atoms and orbit-finite set are defined in Section 6.2). We see that the type of infinity provided by RTMs with atoms is all that is needed to simulate the π-calculus processes.

5.3.2 Restricting the π-Calculus

Now we proceed to consider the first option, which is to propose a restriction on the transition systems associated with π-terms such that they refer only to finitely many actions.

The infinity of the set of actions in the π-calculus arises from the fact that there are infinitely many names available to construct actions. Namely, from an input prefix $a(x)$, we can derive infinitely many input actions with infinitely many distinct names, which are referred to as free input names; and from an output prefix $\overline{a}\,x$ a bounded name x, we can also derive infinitely many output actions with infinitely many distinct names,

which are referred to as bound output names. The free input names allow a process to receive any potential input from the environment and the bound output names give a process the ability to generate unboundedly many distinct private channels to communicate with other processes. For both purposes, infinite branching of the transition system is essential.

First observe that the infinite branching caused by input prefix can be thought of as a technical device in the operational semantics to model the communication of an arbitrary name from one parallel component to another. The name that will be received, can either be a free name of the sending process (a value), or a bound name (a private channel). Since the sending parallel component will only have a finite number of free names, only finitely many values can be communicated.

Second, technically speaking, according to the operational semantics, infinitely many distinct private channels may be communicated when an input prefix synchronises with a bound output prefix, the communicated private channel is not observable, and the resulting π-terms are all equal up to α-conversion, so the only observable effect of the interaction is that after the communication the sending and receiving parties share a private channel of which the name is irrelevant.

Our goal is to investigate to what extent the behaviour specified by an individual π-term is executable. Motivated by the above intuitive interpretation of interaction of a π-term with its environment, we assume that the behaviour specified by that π-term is executed in an environment that may offer data values from some presupposed finite set on its input channels. This assumption seems reasonable as a machine should know in advance which symbols to expect as input. Furthermore, we assume that there is a facility for establishing a private channel between the π-term and its environment. Such a facility could, e.g., be implemented using encryption; we will abstract from its actual implementation. We define a restriction on the transition systems associated with π-terms that is based on these assumptions. We call the π-calculus restricted to finitely many names *the finite π-calculus*. The semantics of the finite π-calculus is defined as follows.

Definition 5.12 (the finite π-calculus). Let $\mathcal{N}' \subseteq \mathcal{N}$ be a finite set of names, let $\mathcal{A}'_\pi = \mathcal{A}_\pi - (\{xy \mid x, y \in \mathcal{N},\ y \notin \mathcal{N}'\} \cup \{\overline{x}(z) \mid x, z \in \mathcal{N}\})$, and let P be a π-term. The transition system associated with P restricted to $\mathcal{N}'$, denoted by $\mathcal{T}(P) \upharpoonright \mathcal{N}'$, is a triple $(\mathcal{S}_P \upharpoonright \mathcal{N}', \longrightarrow_P\upharpoonright \mathcal{N}', P)$, obtained from $\mathcal{T}(P) = (\mathcal{S}_P, \longrightarrow_P, P)$ as follows:

1. $\mathcal{S}_P \upharpoonright \mathcal{N}'$ is the set of states reachable from P by means of transitions that are not labelled by xy with $y \notin \mathcal{N}'$; and
2. $\longrightarrow_P\upharpoonright \mathcal{N}'$ is the restriction of $\longrightarrow_P$ obtained by excluding all transitions labelled by xy with $y \notin \mathcal{N}'$, and relabelling all transitions labelled with $\overline{x}(z)$ $(x, z \in \mathcal{N})$ to

$\nu\overline{x}$, i.e.,

$$\begin{aligned}\longrightarrow_P\upharpoonright \mathcal{N}' &= (\longrightarrow_P \cap (\mathcal{S}_P \upharpoonright \mathcal{N}' \times \mathcal{A}'_\pi \times \mathcal{S}_P \upharpoonright \mathcal{N}')) \\ &\quad \cup\{(s, \nu\overline{x}, t) \mid s, t \in \mathcal{S}_P \upharpoonright \mathcal{N}', s \xrightarrow{\overline{x}(z)}_P t\} \ .\end{aligned}$$

Remark 5.13. Our notion of restriction is introduced to restrict labelled transition systems associated with π-terms to refer to finitely many names. Alternatively, we could have defined a finite version π_{fin} of the π-calculus, presupposing a *finite* set of names $\mathcal{N}$ right from the beginning. If $\mathcal{T}(P)$ is the labelled transition system associated with P according to the operational semantics of π_{fin}, then $\mathcal{T}(P) \upharpoonright \mathcal{N}$ is obtained from $\mathcal{T}(P)$ by replacing all transitions with the label $\overline{x}(z)$ by transitions with the label $\nu\overline{x}$. Apart from this modification, restriction keeps all observable behaviour. Additionally, our modification in Definition 5.12 is arguably less restrictive than this alternative one since the set of names is not set to be finite at the beginning.

Using [77, Lemma 1.4.1], it is straightforward to show that for every π-term the set of actions of the π-calculus appearing as labels in $\mathcal{T}(P) \upharpoonright \mathcal{N}'$ is finite. Furthermore, the transition system associated with a π-term by the operational semantics, and also its restriction according to Definition 5.12 are clearly effective. Hence, as an immediate corollary of Theorem 2.23, we may conclude that the transition system associated with a π-term can be simulated by an RTM modulo (the divergence-insensitive variant of) branching bisimilarity.

Corollary 5.14 (executability of the finite π-calculus modulo $\leftrightarrow_{\mathrm{b}}$)**.** *For every closed π-term P, and for every finite set of input names $\mathcal{N}' \subseteq \mathcal{N}$, there exists an RTM $\mathcal{M}$ such that $\mathcal{T}(P) \upharpoonright \mathcal{N}' \leftrightarrow_b \mathcal{T}(\mathcal{M})$.*

The following example shows that there exist π-terms with which the structural operational semantics associates a transition system without divergence that is unboundedly branching up to $\leftrightarrow_{\mathrm{b}}^{\Delta}$. Note that by Theorem 2.32 such π-terms are not executable modulo divergence-preserving branching bisimilarity.

Example 5.15 (unboundedly branching π-calculus process)**.** Consider the π-process $P \stackrel{\mathrm{def}}{=} (c, i, d, s, \mathit{flip})(\bar{i}\, s.\mathbf{0} \mid \mathit{flip}.\mathbf{0} \mid !C \mid !I \mid !D)$, with C, I and D defined as follows:

$$\begin{aligned} C &\stackrel{\mathrm{def}}{=} c(h, t, b).(\overline{h}\,\langle t, b\rangle.\mathbf{0} + \mathit{flip}.\overline{c}\,\langle h, t, 1\rangle.\mathbf{0}) \\ I &\stackrel{\mathrm{def}}{=} i(h).(\mathit{inc}(h').\overline{c}\,\langle h', h, 0\rangle.\overline{i}\,h'.\mathbf{0} + \mathit{flush}.\overline{\mathit{flip}}\,.\overline{d}\,h.\mathbf{0}) \\ D &\stackrel{\mathrm{def}}{=} d(h).(h(t, b).\overline{b}\,.\overline{d}\,t.\mathbf{0}) \ . \end{aligned}$$

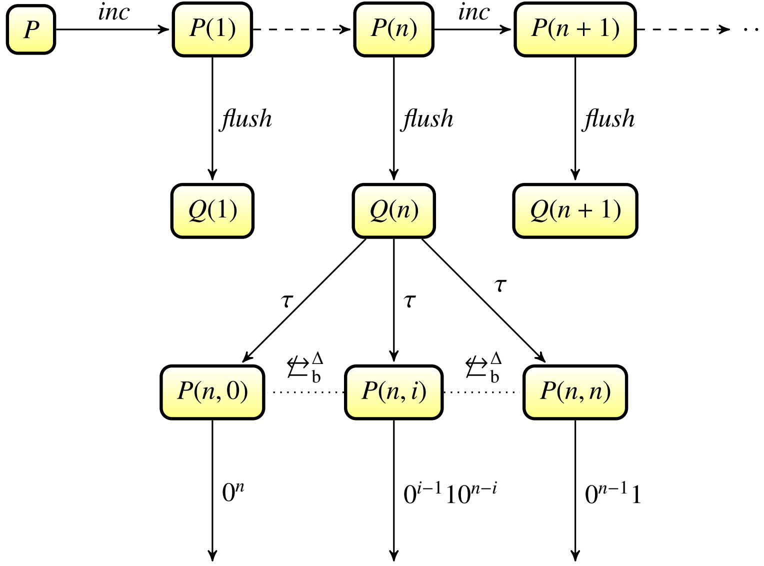


Figure 5.4: A π-calculus process with unbounded branching

The unboundedly branching behaviour of P is illustrated in Figure 5.4. For clarity, we omit deterministic τ-transitions from the figure. Intuitively, the process $!C$ facilitates the generation of a linked list of one-bit cells with a pointer h to the head of the list, a pointer t to the tail of the list, and a bit b. Each cell may either output, along h, the link t to the tail of the list and its bit b, or it may receive the instruction *flip* after which it recreates itself with the value 1. The process I serves as the interface process. It maintains a link to the head of the list. Upon receiving an *inc*-instruction, it adds another one-bit cell to the list, and upon receiving the *flush*-instruction, it flips at most one of the bits, and then calls D. The process D then simply outputs the bits in reverse order.

Consider the state reached after performing n *inc*-actions, followed by a *flush*-action. In this state, the list contains a string of n 0s. We denote the state before performing the *flush* action with a list of n 0s by $P(n)$, and the state after performing a *flush* action by $Q(n)$, i.e., $P(n) \xrightarrow{flush} Q(n)$. The τ-transitions that correspond to the interaction of *flip* between I and one of the *flip*s of one of the one-bit cells or *flip* in the

definition of P have the effect of non-deterministically changing (at most) one of the 0s to a 1. We denote the state with a list of n numbers, where the ith position is 1 and all the others are 0 by $P(n, i)$. Note that there are $n + 1$ such τ-transitions, and since D will subsequently output the sequence in order, the states reached by these τ-transitions are (pairwise) not divergence-preserving branching bisimilar. Hence, it follows that for every n, the transition system associated with P has a reachable state with a branching degree modulo $\leftrightarrow_b^\Delta$ of at least $n + 1$. It follows that the transition system associated with P is unboundedly branching up to $\leftrightarrow_b^\Delta$.

Note that the only names occurring as part of the labels on the transitions in the transition system associated with the π-term P in the preceding example are $\overline{0}$, $\overline{1}$, *inc* and *flush*, so if $\mathcal{N}'$ contains at least these four names, then P satisfies $\mathcal{T}(P) \restriction \mathcal{N}' = \mathcal{T}(P)$. Let us say, in general, that a π-term P has *finitely many observable names* if there exists a finite set $\mathcal{N}' \subseteq \mathcal{N}$ such that $\mathcal{T}(P) \restriction \mathcal{N}' = \mathcal{T}(P)$. Note that, in this case, P cannot have parameterised free inputs, nor bound outputs. For π-terms with finitely many observable names, we have the following corollary as a consequence of a combination of Corollary 5.14 and Example 5.15.

Corollary 5.16 (unexecutability of the finite π-calculus modulo $\leftrightarrow_b^\Delta$)**.** *Every finite π-calculus process P is executable modulo $\leftrightarrow_b$, but there exist finite π-calculus processes that are not executable modulo $\leftrightarrow_b^\Delta$.*

5.4 Remarks

We have investigated the expressivity of the π-calculus in relation to the theory of executability provided by Reactive Turing Machines. The issue of the expressivity of the π-calculus has been extensively studied (see [47] for a comprehensive overview of research in this area). A distinction is usually made between absolute and relative expressivity results. The absolute expressivity results focus on proving the (im)possibility of expressing a computational phenomenon in a calculus; the relative expressivity results are mostly about encoding one calculus in another. Our results pertain to the absolute expressivity of the π-calculus.

We have established that, up to divergence-preserving branching bisimilarity, every executable transition system can be specified in the π-calculus, showing that the π-calculus is reactively Turing powerful. Milner already established in [68] that the π-calculus is Turing powerful, by exhibiting an encoding of the λ-calculus in the π-calculus by which every reduction in the λ-calculus is simulated by a sequence of reductions in the π-calculus. Our result that all executable behaviour can be specified in the π-calculus up to divergence-preserving branching bisimilarity also implies

that the π-calculus is Turing powerful, and thus it subsumes Milner's result. Similarly, in [28] several expressivity results for variants of CCS are obtained via an encoding of Random Access Machines, and also those results only make claims about the computational expressivity of the calculi. Notice that the results in [68] and [28] confirm the computational power of the respective calculi, but do not make a qualitative statement about its interactive expressivity. By showing that Reactive Turing Machines can be faithfully simulated, we at the same time confirm the interactive expressivity of the π-calculus.

In a recent work [37], Fu also proposes to study computation and interaction in an integrated theory. His theory is built on four fundamental principles, rather than on a machine model. One of the contributions of his theory is a calculus including a bare minimum of primitives to be computationally and interactively complete, and he uses it to confirm the completeness of the π-calculus. We leave it for future work to explore the relationship between Fu's theory of interaction and the theory of executability based on Reactive Turing Machines.

We have observed that it is possible to specify behaviour in the π-calculus that is not executable up to any reasonable notion of behavioural equivalence, simply because it uses infinitely many observable names. For the presentation of the π-calculus it is technically important to presuppose an infinite set of names especially to model the feature of dynamic creation of private channels between components. In this chapter, we have shown that a behaviour specified in the π-calculus is executable up to the divergence-insensitive variant of branching bisimilarity if one restricts to finitely many observable names and does not associate a unique identifier with every dynamically created private channel. Allowing RTMs to have an infinite set of actions and either an infinite set of states or an infinite set of data symbols would arguably lead to an unrealistically powerful notion of executability. Moreover, in a real system, private channels between components are likely to be implemented differently, e.g., using some form of encryption.

It has been claimed (e.g., in [34]) that the π-calculus provides a model of computation that is behaviourally more expressive than Turing machines. Our results provide further justification for this claim, and characterise the difference. It should be noted that the difference in expressive power is at the level of interaction (allowing interaction between an unbounded number of components), rather than at the level of computation.

Chapter 6

Nominal Executability

In the previous chapter, we showed that the π-calculus is in general not executable because of the mismatch between the infinite set of names in the π-calculus and the finite set of actions in the definition of RTMs. Moreover, the π-calculus is not unique in relying on an infinite alphabet of actions. A notable number of process calculi leading to transition systems with infinite sets of actions were proposed for various purposes, for instance, the psi-calculus [16], the value-passing calculus [36] and mCRL2 [49]. In this chapter, we therefore investigate extensions of the formalism of Reactive Turing Machines that allows the set of actions to be infinite.

First, we explore a generalised notion of executability based on Reactive Turing Machines that allow an infinite alphabet of actions. We observe that allowing an infinite alphabet only makes sense if we also allow the set of data symbols (or, equivalently, the set of states) to be infinite. Putting no restrictions at all yields a notion of executability that is not discriminating at all: every countable transition system is executable by such an infinitary RTM. The result has two immediate corollaries: Every effective transition system is executable modulo divergence-preserving branching bisimilarity by an infinitary RTM with an effective transition relation, and every computable transition system is executable modulo divergence-preserving branching bisimilarity by an infinitary RTM with a computable transition relation.

Second, we consider a more restricted notion of infinitary executability. Following the research about nominal sets for variable binding with infinite alphabets [39], the notion of Turing machine with atoms was introduced [26]. We define RTMs with atoms as an extension of Turing machines with atoms. RTMs with atoms allow the sets involved in the definition to be infinite, but in a limited way; intuitively, the infinity can only be exploited to generate fresh names in an execution. By using the notion

of legal and orbit-finite set, Turing machines with atoms are allowed to have infinite alphabets, while keeping the transition relation finitely definable and, in fact, finite up to atom automorphism. We say a transition system is nominally executable if it is behaviourally equivalent to a transition system associated with an RTM with atoms. To investigate the notion of nominal executability, we propose a notion of transition system with atoms as a restricted version of transition systems. We show that the transition systems associated with RTMs with atoms satisfy the definition of transition system with atoms.

Finally, we apply the results to draw conclusions about the executability of process calculi. We prove that all π-calculus processes are nominally executable. On the other hand, in mCRL2 it is possible to define behaviours that are not nominally executable. Therefore, nominal executability provides a new expressivity criterion for process calculi involving infinite alphabets.

The chapter is organized as follows. In Section 6.1, we introduce the infinitary RTMs and investigate the associated notion of executability. In Section 6.2, we introduce the notion of sets with atoms. In Section 6.3, we propose the notion of RTM with atoms, and define nominal executability. In Section 6.4, we prove that the transition systems associated with the π-calculus processes are nominally executable. Section 6.5 proposes a notion of labelled transition system with atoms, and shows that the transition systems associated with mCRL2 are not nominally executable . The chapter concludes in Section 6.6, in which a hierarchy of executability and some future work are proposed.

6.1 Infinitary Reactive Turing Machines

In this section, we shall investigate the effect of lifting the finiteness condition imposed on RTMs on the ensued notion of executability. We start with lifting the finiteness condition on the alphabet of actions and the transition relation only. We shall argue by means of an example that this extension is hardly useful, because it is not possible to sufficiently distinguish the computational effect of each action. The next step is, therefore, to also allow an infinite set of data symbols. This, in turn, yields a notion of executability that is too expressive. Finally, we provide two intermediate notions of executability by restricting the transition relations associated with infinitary Reactive Turing Machines to be effective or computable.

6.1.1 Infinitely Many States and Data Symbols

In this section, we allow $\mathcal{A}$ to be a countable infinite set of action labels. Recall from Definition 2.10 that an RTM has a finite set of states Q and a finite transition relation. If we allow RTMs to have infinitely many actions, then, inevitably, we should at least also allow them to have an infinite transition relation. The following example illustrates that we then also either need infinitely many states or infinitely many data symbols.

Example 6.1 (infinite states and data symbols)**.** Consider the $\mathcal{A}_\tau$-labelled transition system $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ in the left of Figure 6.1, where

1. $\mathcal{S}_T = \{\uparrow_T, \downarrow_T\} \cup \{s_a \mid a \in \mathcal{A}\}$, and
2. $\longrightarrow_T = \{(\uparrow_T, a, s_a) \mid a \in \mathcal{A}\} \cup \{(s_a, a, \downarrow_T) \mid a \in \mathcal{A}\}$.

There does not exist an RTM with finitely many states and data symbols that simulates T modulo branching bisimilarity.

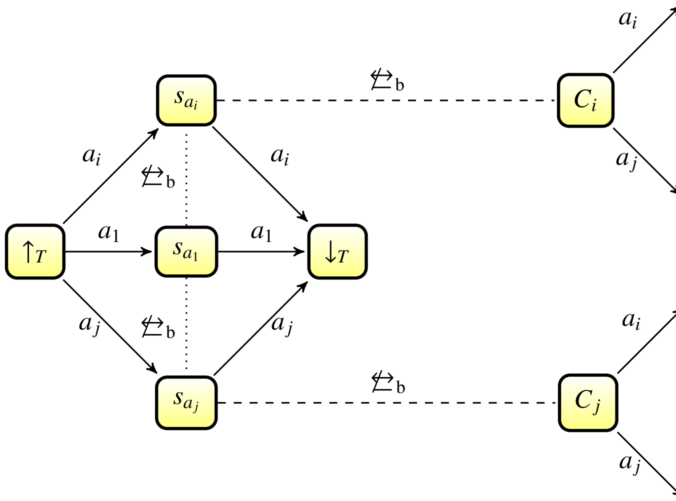


Figure 6.1: A transition system with infinitely many distinct labels

We suppose that $\mathcal{M} = (Q, \mapsto, Ini)$ is an RTM such that $\mathcal{T}(\mathcal{M}) \leftrightarrow_b T$, and we let $\mathcal{A} = \{a_1, a_2, \ldots\}$. The transitions $\uparrow_T \xrightarrow{a_1} s_{a_1}$, $\uparrow_T \xrightarrow{a_2} s_{a_2}, \ldots$ lead to infinitely many states

$s_{a_1}, s_{a_2}, \ldots$, which are all mutually distinct modulo branching bisimilarity since each of them has an outgoing transition with a distinct label.

Let $C = (\uparrow, \check{\square})$ be the initial configuration of $\mathcal{M}$. Assume that we have $C \leftrightarroweq_{\mathrm{b}} \uparrow_T$, so C admits the following transition sequences: $C \longrightarrow^* \xrightarrow{a_1} \longrightarrow^* C_1 \xrightarrow{a_1}$, $C \longrightarrow^* \xrightarrow{a_2} \longrightarrow^* C_2 \xrightarrow{a_2}, \ldots$, where $C_1 \leftrightarroweq_{\mathrm{b}} s_{a_1}$, $C_2 \leftrightarroweq_{\mathrm{b}} s_{a_2}, \ldots$.

The transitions of an RTM are of the form (s, d, a, e, M, t), where $s,\ t \in \mathcal{S}$, and $d,\ e \in \mathcal{D}_\square$; we call the pair (s, d) the trigger of the transition. A configuration $(s', \delta_L \check{d}' \delta_R)$ satisfies the trigger (s, d) if $s = s'$ and $d = d'$. Now we observe that a transition (s, d, a, e, M, t) gives rise to an a-transition from every configuration satisfying its trigger (s, d). Since $\mathcal{S}$ and $\mathcal{D}_\square$ are finite sets, there are finitely many triggers.

So, in the infinite list of configurations $C_1, C_2, \ldots$, there are at least two configurations C_i and C_j, satisfying the same trigger (s, d); these configurations must have the same outgoing transitions.

Note that we cannot have both $C_i \leftrightarroweq_{\mathrm{b}} s_{a_i}$ and $C_j \leftrightarroweq_{\mathrm{b}} s_{a_j}$. Since $C_j \xrightarrow{a_j}$, a transition labelled by a_j is triggered by (s, d). As C_i also satisfies the trigger (s, d), we have the transition $C_i \xrightarrow{a_j}$. Hence $C_i \not\leftrightarroweq_{\mathrm{b}} s_{a_i}$, and we get a contradiction to $\mathcal{T}(\mathcal{M}) \leftrightarroweq_{\mathrm{b}} T$.

6.1.2 Infinitary Reactive Turing Machines

If we allow the set of control states or the set of data symbols to be infinite too, the expressivity of RTMs is greatly enhanced, as we shall see below. We define a notion of infinitary Reactive Turing Machines as follows.

Definition 6.2 (infinitary Reactive Turing Machines)**.** An *infinitary Reactive Turing Machine* (RTM$^\infty$) is a triple $(Q, \mapsto, \mathit{Ini})$, where

1. Q is a countable set of *states*;
2. $\mapsto\ \subseteq Q \times \mathcal{D}_\square \times \mathcal{A}_\tau \times \mathcal{D}_\square \times \{L, R\} \times Q$ is a countable $(\mathcal{D}_\square \times \mathcal{A}_\tau \times \mathcal{D}_\square \times \{L, R\})$-labelled *transition relation* (we write $s \xmapsto{a[d/e]M} t$ for $(s, d, a, e, M, t) \in \longrightarrow$); and
3. $\mathit{Ini} \in \mathcal{S}$ is a distinguished *initial state*.

Remark 6.3. Note that we do not have to require both the set of states and the set of data symbols to be infinite simultaneously. Instead, we only need one of them to be infinite, as in the above definition. Actually, the requirement of having a countable set of states is equally powerful as the requirement of having a countable set of data symbols or having both a countable set of states and a countable set of data symbols.

By analogy to Definition 2.11, we associate with every RTM$^\infty$ a labelled transition system. Similar to the notion of executability ensued from RTMs, we also introduce executability with respect to RTM$^\infty$s. In this chapter, we shall only be interested in the executability with respect to RTM$^\infty$s modulo (divergence-preserving) branching bisimilarity.

Definition 6.4 (executability with respect to RTM$^\infty$)**.** A transition system is *executable by an RTM$^\infty$ modulo (divergence-preserving) branching bisimilarity* if it is (divergence-preserving) branching bisimilar to a transition system associated with some RTM$^\infty$.

As we did for RTMs, we shall discuss to what extent, labelled transition systems can be simulated by RTM$^\infty$s. A transition system is *countable*, if its sets of labels, states and transitions are all countable sets. The following theorem illustrates a characterisation of the expressivity of RTM$^\infty$s, showing that every countable transition system is executable by an RTM$^\infty$ modulo $\leftrightarroweq_{\mathrm{b}}^{\Delta}$.

Theorem 6.5 (countable LTS)**.** *For every countable set $\mathcal{A}_\tau$ and every countable $\mathcal{A}_\tau$-labelled transition system T, there exists an RTM$^\infty$ $\mathcal{M}$ such that $T \leftrightarroweq_{b}^{\Delta} \mathcal{T}(\mathcal{M})$.*

Proof. Let $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ be an $\mathcal{A}_\tau$-labelled countable transition system, and let $\lceil _ \rceil : \mathcal{S}_T \to \mathbb{N}$ be an injective function encoding its states as natural numbers. Then an RTM$^\infty$ $\mathcal{M} = (Q, \mapsto, Ini)$ is defined as follows.

1. $Q = \{s, t, \uparrow\}$ is the set of control states.
2. $\longrightarrow$ is a $(\mathcal{D}_\Box \times \mathcal{A} \times \mathcal{D}_\Box \times \{L, R\})$-labelled transition relation consisting of the following transitions:
   (a) $(\uparrow, \Box, \tau, \lceil \uparrow_T \rceil, R, s)$,
   (b) $(s, \Box, \tau, \Box, L, t)$, and
   (c) $(t, \lceil s_1 \rceil, a, \lceil s_2 \rceil, R, s)$ for every transition $s_1 \xrightarrow{a}_T s_2$.
3. $\uparrow \in Q$ is the *initial state*.

Note that a transition $s_1 \xrightarrow{a} s_2$ is simulated by a sequence

$$(t, \lceil \check{s_1} \rceil \Box) \xrightarrow{a} (s, \lceil s_2 \rceil \check{\Box}) \xrightarrow{\tau} (t, \lceil \check{s_2} \rceil \Box) \enspace .$$

Then one can verify that $\mathcal{T}(\mathcal{M}) \leftrightarroweq_{\mathrm{b}}^{\Delta} T$. Note that $\{(s_T, (t, \lceil \check{s_T} \rceil \Box)) \mid s_T \in \mathcal{S}_T\} \cup \{(s_T, (s, \lceil s_T \rceil \check{\Box})) \mid s_T \in \mathcal{S}_T\}$ is a divergence-preserving branching bisimulation. The divergence-preserving branching bisimilarity is illustrated in Figure 6.2. $\Box$

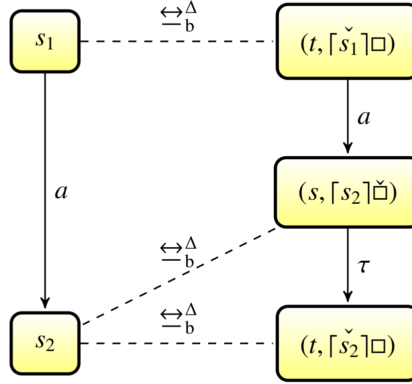


Figure 6.2: Bisimulation relation in the proof of Theorem 6.5

So RTM$^\infty$s are very expressive, and they certainly do not yield a useful model to distinguish between processes that can and cannot be executed. The reason is that we have not yet put a restriction that the transition relation used to define an RTM$^\infty$needs to be computable or effective. As a compromise, we provide two intermediate models by formulating extra requirements on the transition relation of RTM$^\infty$s.

We say that a transition relation of an RTM$^\infty$ is effective, if for every pair of a control state and a data symbol (s, d), the set of subsequent transitions is recursively enumerable, i.e., the function $out(s, d) = \{(a, e, M, t) \mid s \xmapsto{a[d/e]M} t\}$ is recursively enumerable with respect to some encoding. In the proof of Theorem 6.5, if the transition system is effective, then the set of transitions $\{(t, \ulcorner s_1 \urcorner, a, \ulcorner s_2 \urcorner, R, s) \mid s_1 \xrightarrow{a} s_2\}$ is recursively enumerable. It is trivial that all the other transitions are also recursively enumerable. Hence, we get an effective transition relation. We derive the following corollary for the executability of effective transition systems from Theorem 6.5.

Corollary 6.6 (effective LTS)**.** *For every countable set $\mathcal{A}_\tau$ and every effective $\mathcal{A}_\tau$-labelled transition system T, there exists an RTM$^\infty$ $\mathcal{M}$ with an effective transition relation such that $T \underline{\leftrightarrow}^{\Delta}_{b} \mathcal{T}(\mathcal{M})$.*

We say that a transition relation of an RTM$^\infty$ is computable if for every pair of a control state and a data symbol (s, d) the set of subsequent transitions is computable, i.e., the function $out(s, d) = \{(a, e, M, t) \mid s \xmapsto{a[d/e]M} t\}$ is recursive with respect to some encoding. By analogy to Corollary 6.6, we derive the following result from Theorem 6.5.

Corollary 6.7 (computable LTS)**.** *For every countable set $\mathcal{A}_\tau$ and every computable $\mathcal{A}_\tau$-labelled transition system T, there exists an RTM$^\infty$ $\mathcal{M}$ with a computable transition relation such that $T \leftrightarrow_b^\Delta \mathcal{T}(\mathcal{M})$.*

6.2 Sets with Atoms

As we discussed in the previous section, an unrestricted lifting of the finiteness conditions of RTMs leads to an extremely expressive notion of executability. Such a notion is not very useful since every countable transition system is trivially included which makes it not distinguishing at all.

In this section, we introduce a notion of Reactive Turing Machine with atoms (RTM$^{\mathbb{A}}$) as a natural intermediate between RTMs and RTM$^\infty$s. On the one hand, RTM$^{\mathbb{A}}$s will be more expressive than RTMs, since they will admit infinite alphabets, whereas RTMs do not. On the other hand, RTM$^{\mathbb{A}}$s will be less expressive than RTM$^\infty$s, because there will be restrictions imposed that, intuitively, make the alphabets finitely presentable. We introduce a notion of effective transition system with atoms to characterise the transition systems associated with RTM$^{\mathbb{A}}$s modulo branching bisimilarity. We then have a proper model to investigate the executability of process calculi with infinite alphabets (such as the π-calculus).

6.2.1 Equality Atoms

In this section, we introduce the notion of set with atoms as a basis of RTM$^{\mathbb{A}}$. We use the definition of sets with atoms from Bojańczyk et al. [25, 26]. Moreover, Bojańczyk recently introduced many useful concepts related to sets with atoms in a book [24]; we shall also refer to many concepts introduced in this book.

Sets with atoms rely on an infinite set of atoms and sets that contain those atoms, together with some relations and functions on the set of atoms. We present some examples of sets of atoms paired with relations over atoms as follows:

- $(\mathbb{N}, =)$ the natural numbers (or any countably infinite set) with equality,
- $(\mathbb{Q}, \leq)$ the rational numbers with an order, and
- $(\mathbb{Z}, +1)$ the integers with a unary successor function.

The first structure is referred to as *equality atoms*. In this chapter, we only consider equality atoms. For convenience, we just call them sets with atoms. We fix for the remainder of this chapter a countably infinite set $\mathbb{A}$; we call its elements *atoms*. We use the underlined names like $\underline{1}$ or $\underline{a}$ for examples of atoms.

In set theory, a set x is called a well-founded set if the set membership relation is well-founded, i.e., the set membership relation has a minimal element on its transitive closure. We use this notion to define a set with atoms.

Definition 6.8. A *set with atoms* is any set that contains atoms or other sets with atoms, in a well-founded way.

Example 6.9. To give some the intuition for this definition, we present some examples of sets with (equality) atoms.

1. Every set in the traditional sense is a set with atoms, for instance, the empty set $\emptyset$;
2. the set of atoms $\mathbb{A}$ is a set with atoms;
3. ordered pairs or tuples of atoms are sets with atoms; i.e., we use $\{\underline{a}, \{\underline{a}, \underline{b}\}\}$ to denote an ordered pair $(\underline{a}, \underline{b})$; and
4. the set of n-tuples of atoms $\mathbb{A}^n$ and the set of finite sequences of atoms $\mathbb{A}^*$ are sets with atoms.

6.2.2 Legality and Orbit-finiteness

The atoms will allow us to formulate certain finiteness restrictions that are slightly more liberal than simply requiring that sets are finite. To this end we proceed to introduce *legal* and *orbit-finite* sets with atoms.

An *atom automorphism* is a bijection (permutation) on $\mathbb{A}$. For a set with atoms X and an atom automorphism π, by $\pi(X)$ we denote the set obtained by application of π to every atom in X, in elements of X, in elements of elements of X, etc., recursively. For a set of atoms $S \subseteq \mathbb{A}$, if an atom automorphism π is the identity on S, i.e. for all $x \in S$, $\pi(x) = x$, then we call it an *S-automorphism*. We say that S *supports* a set with atoms X iff $X = \pi(X)$ for every S-automorphism π.

Definition 6.10. A set of atoms S is a *support* of a set with atoms X iff S supports X. A set with atoms is called *legal* if it has a finite support, each of its elements has a finite support, and so on recursively.

A set with atoms may contain infinitely many atoms, but legality restricts the extent. We give some examples to illustrate this notion.

Example 6.11.

A finite set of atoms is legal. The support is a set that contains every atom from the finite set.

A co-finite set of atoms is legal. The finite complement of a set that contains every atom from the co-finite set is a support.

The set of all odd natural numbers is not legal with respect to equality atoms. Its support necessarily includes all odd numbers, or all even numbers.

For legal sets with atoms, we further introduce a notion of *orbit-finiteness*, which restricts the number of equivalence classes partitioned by atom automorphisms of a set with atoms. Now we proceed to introduce the notion of *orbit-finite* set. Let x be an element in a set with atoms X, the *orbit of x* is the set

$$\{y \in X \mid y = \pi(x) \text{ for some atom automorphism } \pi\} \enspace .$$

A set with atoms X is partitioned into disjoint *orbits*: elements x and y are in the same orbit iff $\pi(x) = y$ for some atom automorphism π. We give some examples to orbit-finiteness

Example 6.12.

$\mathbb{A}^2$ decomposes into two orbits, the diagonal $\{(a, a) \mid a \in \mathbb{A}\}$ and its complement $\{(a, b) \mid a, b \in \mathbb{A},\ a \neq b\}$. Note that there is no atom automorphism that maps (a, b) to (a, a) if a and b are different atoms.

$\mathbb{A}^*$ has infinitely many orbits as the elements from $\mathbb{A}$, $\mathbb{A}^2, \ldots$ all fall into disjoint orbits.

Orbit-finiteness restricts the number of partitions of a set with atoms with respect to atom automorphism.

Definition 6.13. A set with atoms that is partitioned into finitely many orbits is called an *orbit-finite* set.

Remark 6.14. In Bojańczyk's book [24], the notion of orbit is parameterised by a tuple of atoms $\bar{a}$, namely, an $\bar{a}$-orbit of x is the set

$$\{y \in X \mid y = \pi(x),\ \pi \text{ is an } \bar{a}\text{-automorphism}\} \enspace .$$

For the definition of orbit-finiteness, we do not need to have a tuple of atoms as a parameter. Actually, in [24], it was proved that, for equality atoms, every legal set with atoms X is a finite union of $\bar{a}$-orbits for some tuple $\bar{a}$ that supports X if and only if it is a finite union of $\bar{a}$-orbits for every tuple $\bar{a}$ that supports X. In other words, orbit-finiteness does not depend on the choice of $\bar{a}$.

6.2.3 Definability

In order to utilize the notion of set with atoms in the theory of executability, in particular, to define RTMs using sets with atoms, we require the sets with atoms to be *definable*. In [24], the definability of sets with atoms has been extensively discussed. Hence, we shall recall them in the remaining part of this section.

To syntactically define a set, we need a notion of set builder expressions.

Definition 6.15. Let V be a countably infinite set of variables. Let $\bar{x}$ be a tuple of variables. An *$\bar{x}$-valuation* is a function that maps each variable in $\bar{x}$ to an element in $\mathbb{A}$. The notion of *set builder expression* is inductively defined as follows:

1. *Atom constant.* Every atom $a \in \mathbb{A}$ is a set builder expression. The expression has no free variables, and it represents the atom a.

2. *Variable expression.* A variable $x \in V$ is a set builder expression. The variable x is free in this expression.

3. *Set expression.* Let $\bar{x}$ and $\bar{y}$ be disjoint tuples of variables and let α be a set builder expression with free variables contained in $\bar{x}\bar{y}$, and let ϕ be a first-order formula over $\mathbb{A}$ with free variables contained in $\bar{x}\bar{y}$ which is allowed to use constants from $\mathbb{A}$ (such constants are called *parameters*). Then

$$\{\alpha(\bar{x}\bar{y}) \mid \text{ for } \bar{y} \text{ such that } \phi(\bar{x}\bar{y})\}$$

   is a set builder expression with free variables $\bar{x}$ and bound variables $\bar{y}$.

4. *Union expression.* If $\alpha_1, \ldots, \alpha_n$ are set builder expressions, then so is $\alpha_1 \cup \cdots \cup \alpha_n$.

We use $\mathbb{B}$ to denote the set of all set builder expressions. For a set builder expression α with free variables $\bar{x}$, we define $[\![\alpha]\!]$ to be the function which inputs a tuple of atoms as the valuation of $\bar{x}$ and outputs the corresponding set (or set of sets, etc.). When α has no free variables, then $[\![\alpha]\!]$ is simply a set (or atom).

Definition 6.16. A set with atoms x is *definable* if $x = [\![\alpha]\!]$, where α is a set builder expression without free variables.

A definable set could be an atom, if α is an atom constant, or a set. Moreover, a definable set can be represented using different set builder expressions.

In [24], it was proved that for equality atoms, definable sets are equivalent to hereditarily orbit-finite sets.

Definition 6.17. A *hereditarily orbit-finite set* is a set with atoms which is orbit finite, whose elements are orbit-finite, and so on until the empty set is or an atom is reached.

For example, a set $\{\mathbb{A}^*\}$ is orbit-finite, since its only orbit is $\mathbb{A}^*$. However, it is not hereditarily orbit-finite, since its element $\mathbb{A}^*$ is not an orbit-finite set.

Lemma 6.18. *Every legal and hereditarily orbit-finite set with atoms is definable.*

Taking the π-calculus as an example assuming that the set of names to be the infinite supply of atoms, the set of all π-terms is not definable, since it is not orbit-finite. π-terms with different lengths are in different orbits, in other words, infinitely many distinct structures are involved in the π-calculus. However, the α-equivalence class of a single π-term is definable since it has only one orbit (every α-conversion can be achieved by some automorphisms on names).

6.3 Reactive Turing Machines with Atoms

Bojańczyk et al. [26] defined a notion of Turing machine with atoms based on sets with atoms. Now we generalize this notion by defining a notion of Reactive Turing Machine with atoms. We assume that the sets of action symbols $\mathcal{A}_\tau$ and data symbols $\mathcal{D}_\Box$ are definable (legal and hereditarily orbit-finite) sets with atoms.

Definition 6.19 (Reactive Turing Machines with atoms)**.** A *Reactive Turing Machine with atoms* ($\mathrm{RTM}^{\mathbb{A}}$) is a triple $(Q, \mapsto, \mathit{Ini})$, where

1. Q is a definable set of *states*,
2. $\mapsto \subseteq Q \times \mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\} \times Q$ is a definable $(\mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\})$-labelled transition relation (we write $s \xmapsto{a[d/e]M} t$ for $(s, d, a, e, M, t) \in \mapsto$),
3. $\mathit{Ini} \in Q$ is a distinguished *initial state*.

By analogy to Definition 2.16, we associate with every $\mathrm{RTM}^{\mathbb{A}}$ a labelled transition system, and define a notion of executability with respect to $\mathrm{RTM}^{\mathbb{A}}$. In this chapter, we only consider nominal executability modulo the divergence-insensitive variant of branching bisimilarity.

Definition 6.20 (nominal executability)**.** A transition system is *nominally executable* if it is branching bisimilar to a transition system associated with some $\mathrm{RTM}^{\mathbb{A}}$.

$\mathrm{RTM}^{\mathbb{A}}$s give rise to a less liberal notion of executability compared to the one induced by RTM^{∞}s. The following example gives us an insight in the effect of the legality restriction, which makes $\mathrm{RTM}^{\mathbb{A}}$ a more restrictive notion than RTM^{∞}.

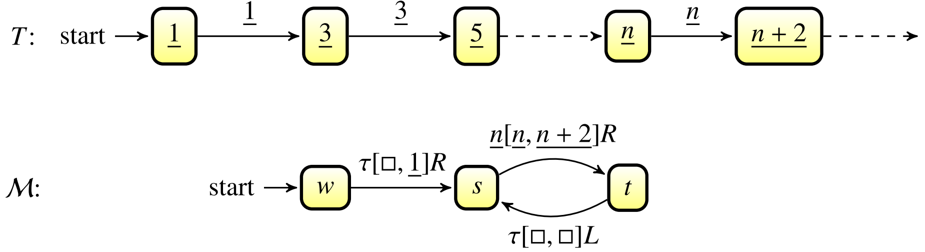


Figure 6.3: Enumerating odd numbers with an RTM$^\infty$

Example 6.21. Assume that $\mathbb{A} = \mathbb{N}$. Consider a transition system $T = (\mathcal{S}, \longrightarrow, \uparrow)$ (see Figure 6.3) defined as follows:

1. $\mathcal{S} = \{\underline{n} \mid n \in \mathbb{N}\}$,
2. $\longrightarrow = \{(\underline{n}, \underline{n}, \underline{n+2}) \mid n \in \mathbb{N}\}$, and
3. $\uparrow = \underline{1}$.

We recognize that T is a transition system that enumerates all the odd numbers. It can be simulated by an RTM$^\infty$ $\mathcal{M} = (\mathcal{Q}, \mapsto, \mathit{Ini})$ (see Figure 6.3) defined as follows:

1. $\mathcal{Q} = \{s, t, w\}$,
2. $\mapsto\ = \{(w, \square, \tau, \underline{1}, R, s)\} \cup \{(s, \underline{n}, \underline{n}, \underline{n+2}, R, t) \mid n \in \mathbb{N}\} \cup \{(t, \square, \tau, \square, L, s)\}$, and
3. $\mathit{Ini} = w$.

Both the set of odd numbers and the set of even numbers supports the above set, however, neither of them is finite. For any finite set S, there exists an S-automorphism that maps an odd number to an even number. Therefore, the above transition relation does not have a finite support. Therefore, the above RTM$^\infty$ is not an RTM$^\mathbb{A}$.

Besides legality, orbit-finiteness also restricts the notion of executability. The transition relations of RTM$^\mathbb{A}$s are restricted to finitely many different orbits up to atom automorphism. As a result, RTM$^\mathbb{A}$s cannot make transitions labelled with tuples of atoms of arbitrary lengths, nevertheless, such transitions can be realized by an RTM$^\infty$.

Example 6.22. Consider an RTM$^\infty$ $\mathcal{M} = (\mathcal{Q}, \mapsto, \mathit{Ini})$ defined as follows:

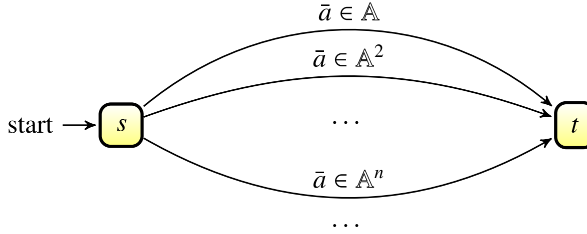


Figure 6.4: A labelled transition system with infinitely many orbits

1. $Q = \{s, t\}$,
2. $\mapsto = \{(s, \Box, \bar{a}, \bar{a}, R, t) \mid \bar{a} \in \mathbb{A}^*\}$, and
3. $Ini = s$.

The transition system associated with $\mathcal{M}$ is illustrated in Figure 6.4. Note that we allow labels and data symbols of this RTM$^\infty$ to be tuples of atoms of arbitrary lengths, which are not orbit-finite. Therefore this RTM$^\infty$ is not an RTM$^\mathbb{A}$.

6.4 Nominal Executability of the π-Calculus

In this section, we prove that the π-calculus is nominally executable modulo (the divergence-insensitive variant of) branching bisimilarity. We show that for every π-calculus process P, there exists an RTM$^\mathbb{A}$ $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarroweq_b \mathcal{T}(P)$.

Recall that we consider the set of names as the set of atoms. Note that in the π-calculus the set of labels is a definable set. Henceforth, we assume the set of data symbols includes the set of labels. We give an encoding from π-terms to $\mathcal{D}_\Box^*$. The encoding allows us to store an arbitrary π-term on the tape.

First of all, we shall present some examples to illustrate the capability of an RTM$^\mathbb{A}$ to manipulate sets with atoms. We propose some fragments of RTM$^\mathbb{A}$s. A fragment of an RTM$^\mathbb{A}$ defines a certain behaviour from a certain configuration, and the idea is that several fragments of RTM$^\mathbb{A}$s can be combined to form a single RTM$^\mathbb{A}$. In the following three examples, we show that an RTM$^\mathbb{A}$ is capable of duplicating a data symbol, generating an fresh atom and rename an atom. These three operations are crucial for an RTM$^\mathbb{A}$ to simulate the transition system associated with a π-calculus process.

Example 6.23. We define some fragments that duplicate non-blank data symbols. Let $\bar{d} \in \mathcal{D}^*$ be a string of data symbols, and let n be the length of $\bar{d}$. We define n fragments of RTM$^{\mathbb{A}}$s, denoted by $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ respectively. We suppose that $\mathcal{M}_i$ is initially set to a configuration $(start, \bar{d}, i)$, where the $\bar{d}$ is its tape instance, and the tape head is on the i-th element of $\bar{d}$. Note that the notation we use here for a configuration is different from that in Definition 2.11. The transition relation associated with $\mathcal{M}_i$ admits a sequence of transitions $(start, \bar{d}, i) \longrightarrow^* (finish, \bar{d}x, n+1)$ satisfying that x is the i-th element of $\bar{d}$ $(i \leq n)$, and the head is at the position of x. The functionality of the RTM$^{\mathbb{A}}$ $\mathcal{M}_i$ is to duplicate the i-th element of a string of symbols.

We first explain our intuition to design a fragment of an RTM$^{\mathbb{A}}$ to fulfill the above requirements. A simple solution is to use the tape head to read the symbol to be duplicated, and then move the tape head towards the destination of the duplication and write the symbol onto the tape. Note that the position of the symbol to be duplicated is different from the destination of the duplication. We need a way to "remember" the symbol while the tape head is moving towards the destination. We introduce a set of control states $\{copy_x \mid x \in \mathcal{D}_\Box\}$ to "remember" the symbols.

We take $\{start, finish\} \cup \{copy_x \mid x \in \mathcal{D}_\Box\}$ as the set of control states of $\mathcal{M}_i$ and the initial configuration is $(start, \bar{a}, i)$. The machine remembers the symbol $x \in \mathcal{D}_\Box$ using a state $copy_x$. Note that the set of states is definable since $\mathcal{D}_\Box$ is a definable set, and it contains all $copy_x$ where x ranges over $\mathcal{D}_\Box$. Then the duplication is realized by the following set of transitions:

$$\begin{aligned} &\{(start, x, \tau, x, R, copy_x) \mid x \in \mathcal{D}_\Box\} \\ \cup&\{(copy_x, y, \tau, y, R, copy_x) \mid x, y \in \mathcal{D}_\Box\} \\ \cup&\{(copy_x, \Box, \tau, x, R, finish) \mid x \in \mathcal{D}_\Box\} \ . \end{aligned}$$

This is a definable set of transitions. The least support equals the least support of $\mathcal{D}_\Box$, and it has finitely many orbits since $\mathcal{D}_\Box$ has finitely many orbits. For instance, the first component $\{(start, x, \tau, x, R, copy_x) \mid x \in \mathcal{D}_\Box\}$ in the union has the same number of orbits as $\mathcal{D}_\Box$, i.e., every $\{(start, x, \tau, x, R, copy_x) \mid x \in O\}$ forms an orbit of the above set, where O is an orbit of $\mathcal{D}_\Box$. We remark that if $\mathcal{D}_\Box = \mathbb{A}$, then the set of transitions has four orbits, namely $\{(start, x, \tau, x, R, copy_x) \mid x \in \mathbb{A}\}$, $\{(copy_x, y, \tau, y, R, copy_x) \mid x, y \in \mathbb{A},\ x \neq y\}$, $\{(copy_x, y, \tau, y, R, copy_x) \mid x, y \in \mathbb{A},\ x = y\}$ and $\{(copy_x, \Box, \tau, x, R, finish) \mid x \in \mathbb{A}\}$.

Example 6.24. We define a fragment that generates a fresh atom. Let $\bar{d} \in \mathcal{D}^*$ be a string of data symbols, and let n be the length of $\bar{d}$. We define a fragment, denoted by $\mathcal{M}_f$. We suppose that $\mathcal{M}_f$ is initially set to a configuration $(start, \bar{d}, n+1)$, where the $\bar{d}$ is its tape instance, and the tape head is directly right of the last symbol of $\bar{d}$. Its transition system admits a sequence of transitions $(start, \bar{d}, n+1) \longrightarrow^* (finish, \bar{d}x, 1)$ satisfying that x is a fresh atom that does not appear in any symbol in $\bar{d}$.

We first explain our intuition to design a fragment of an RTM$^{\mathbb{A}}$ to fulfill the above requirements. Our solution is to first create an arbitrary atom $x \in \mathbb{A}$, and then check whether x is a fresh one, i.e., whether it has already appeared in the tape. In order to check the freshness of x, we introduce a set of states $\{check_x \mid x \in \mathbb{A}\}$ to "remember" the atom while the tape head is moving through the tape to check whether x has appeared in the current data symbol. Whenever we find a symbol that contains x, the machine recreate another atom and repeat the checking procedure. The checking procedure terminates after all the tape symbols have been checked.

We take $\{start, finish\} \cup \{check_x, refresh_x \mid x \in \mathbb{A}\}$ as the set of control states of $\mathcal{M}_f$ and the initial configuration is $(start, \bar{d}, n + 1)$. We use $x \in y$ for $x \in \mathbb{A}$ and $y \in \mathcal{D}_{\Box}$ to denote that x is an atom that appears in y. The machine generates an arbitrary atom and checks whether it is a fresh one by the following set of transitions,

$$
\begin{gathered}
\{(start, \Box, \tau, x, L, check_x) \mid x \in \mathbb{A}\} \\
\cup\{(check_x, y, \tau, y, L, check_x) \mid x \notin y \wedge x \in \mathbb{A} \wedge y \in \mathcal{D}\} \\
\cup\{(check_x, y, \tau, y, L, refresh_x) \mid x \in y \wedge x \in \mathbb{A} \wedge y \in \mathcal{D}\} \\
\cup\{(refresh_x, y, \tau, y, R, refresh_x) \mid x \in \mathbb{A} \wedge y \in \mathcal{D}\} \\
\cup\{(refresh_x, x, \tau, z, L, check_z) \mid x, z \in \mathbb{A}\} \\
\cup\{(check_x, \Box, \tau, \Box, R, finish) \mid x \in \mathbb{A}\} \ .
\end{gathered}
$$

The machine first creates an arbitrary atom x, and then it checks for every symbol on the tape whether it contains x. Note that $\Box$ indicates the end of the sequence of atoms on tape. If the check procedure succeeds, the creation is finished, otherwise, the machine creates another atom and checks again. Note that the above transitions form a definable set, since $\mathcal{D}_{\Box}$ is definable.

Example 6.25. We define a fragment to rename atoms. Let $\bar{d} \in \mathcal{D}^*$ be a string of data symbols, and let n be the length of $\bar{d}$. We define an RTM$^{\mathbb{A}}$, denoted by $\mathcal{M}_r$. We suppose that $\mathcal{M}_r$ is initially set to a configuration $(start, xy\bar{d}, 1)$, where $x, y \in \mathbb{A}$ are two symbols, and the tape head is at x. Its transition system consists of a sequence of transitions $(start, xy\bar{d}, 1) \longrightarrow^* (finish, xy\bar{d}\{y/x\}, n + 1)$ satisfying that every occurrence of x in $\bar{d}$ is renamed to y. The functionality of the RTM$^{\mathbb{A}}$ $\mathcal{M}_r$ is to rename an atom x to y in an arbitrary sequence of data symbols.

We first explain our intuition to design a fragment of an RTM$^{\mathbb{A}}$ to fulfill the above requirements. Our solution is to introduce a set of states $\{rename_x \mid x \in \mathbb{A}\}$ to "remember" the name to be renamed and a set of states $\{rename_{x,y} \mid x, y \in \mathbb{A}\}$ to "remember" the name to be renamed and renamed to. The machine first enters a state $rename_{x,y}$ to represent that a name x should be renamed to y. Then the tape head moves through all the tape symbols and makes the renaming operation.

We take $\{start, finish\} \cup \{rename_x \mid x \in \mathbb{A}\} \cup \{rename_{x,y} \mid x, y \in \mathbb{A}\}$ as the set of control states of $\mathcal{M}_r$ and the initial configuration is $(start, xy\bar{d}, 1)$. The machine renames every symbol in $\bar{d}$ by the following set of transitions,

$$\begin{gathered}
\{(start, x, \tau, x, R, rename_x) \mid x \in \mathbb{A}\} \\
\{(rename_x, y, \tau, y, R, rename_{x,y}) \mid x, y \in \mathbb{A}\} \\
\cup\{(rename_{x,y}, z, \tau, z, R, rename_{x,y}) \mid x \notin z \wedge x, y \in \mathbb{A} \wedge z \in \mathcal{D}\} \\
\cup\{(rename_{x,y}, z, \tau, z\{y/x\}, R, rename_{x,y}) \mid x \in z \wedge x, y \in \mathbb{A} \wedge z \in \mathcal{D}\} \\
\cup\{(rename_{x,y}, \Box, \tau, \Box, L, finish) \mid x, y \in \mathbb{A}\} \ .
\end{gathered}$$

The machine first checks the first symbol (x, y), and enters the state $rename_{x,y}$. Then, the machine searches through every element of $\bar{d}$ and makes a renaming from x to y to a symbol z if x appears in z. Note that the above transitions form a definable set.

In order to simulate the transition system associated with a π-term, we give an encoding $\ulcorner_\urcorner$ from π-terms to $\mathcal{D}_\Box^*$. We assume that the set of action labels $\{x\,y \mid x, y \in \mathcal{N}\} \cup \{\overline{x}\,y \mid x, y \in \mathcal{N}\} \cup \{\overline{x}\,(y) \mid x, y \in \mathcal{N}\} \cup \{\tau\}$, the set of special symbols $\{(,), ., +, |, !, \mathbf{0}\}$ and the set of names $\mathcal{N}$ are subsets of $\mathcal{D}_\Box$. A π-term can be (literally) written on tape with the given set of tape symbols. We use $\ulcorner P \urcorner$ to denote the symbols written on the tape to represent a π-term P.

Moreover, we define a function *Next* to map every π-term to the set of outgoing transitions as follows:

1. $Next(\mathbf{0}) = \emptyset$;

2. $Next(\tau.P) = \{(\tau, Q) \mid \tau.Q =_\alpha \tau.P\}$;

3. $Next(x(y).P) = \{(x\,z, Q\{z/y'\}) \mid z \in \mathcal{N},\ x(y').Q =_\alpha x(y).P\}$;

4. $Next(\overline{x}\,y.P) = \{(\overline{x}\,y, Q) \mid \overline{x}\,y.Q =_\alpha \overline{x}\,y.P\}$;

5. $Next(P_1 + P_2) = Next(P_1) \cup Next(P_2)$;

6. $Next(P_1 \mid P_2) = \{(a, P_1 \mid P_2') \mid (a, P_2') \in Next(P_2),\ \mathsf{bn}(a) \cap \mathsf{fn}(P_1) = \emptyset\}$
$\cup \{(a, P_1' \mid P_2) \mid (a, P_1') \in Next(P_1),\ \mathsf{bn}(a) \cap \mathsf{fn}(P_2) = \emptyset\}$
$\cup \{(\tau, P_1'\{z/y\} \mid P_2') \mid (x\,z, P_1'\{z/y\}) \in Next(P_1),\ (\overline{x}\,z, P_2') \in Next(P_2)\}$
$\cup \{(\tau, P_1' \mid P_2'\{z/y\}) \mid (x\,z, P_2'\{z/y\}) \in Next(P_2),\ (\overline{x}\,z, P_1') \in Next(P_1)\}$
$\cup \{(\tau, (z)P_1'\{z/y\} \mid P_2') \mid (x\,z, P_1'\{z/y\}) \in Next(P_1),\ (\overline{x}\,(z), P_2') \in Next(P_2),\ z \notin \mathsf{fn}(P_2)\}$
$\cup \{(\tau, (z)P_1' \mid P_2'\{z/y\}) \mid (x\,z, P_2'\{z/y\}) \in Next(P_2),\ (\overline{x}\,(z), P_1') \in Next(P_1),\ z \notin \mathsf{fn}(P_1)\}$;

7. $Next((z)P) = \{(a, (z')P') \mid (a, P') \in Next(Q),\ z' \notin a, (z')Q =_\alpha (z)P\}$
$\cup \{(\overline{x}(z'), P') \mid (\overline{x}\,z', P') \in Next(Q),\ (z')Q =_\alpha (z)P\}$;

8. $Next(!P) = \{(a, P' \mid !P) \mid (a, P') \in Next(P)\}$
$\cup \{(\tau, (P' \mid P''\{z/y\}) \mid !P) \mid (\overline{x}\,z, P') \in Next(P),\ (x\,z, P''\{z/y\}) \in Next(P)\}$
$\cup \{(\tau, (z)(P' \mid P''\{z/y\}) \mid !P) \mid (\overline{x}(z), P') \in Next(P),\ (x\,z, P''\{z/y\}) \in Next(P)\}$.

Proposition 6.26. *Let P be a π term, then $Next(P)$ is a definable set.*

Proof. We show that $Next(P)$ is definable by induction on the structure of P.

1. $Next(\mathbf{0}) = \emptyset$. An empty set is trivially definable.

2. $Next(\tau.P) = \{(\tau, Q) \mid \tau.Q =_\alpha \tau.P\}$. Note that $\tau.P$ is definable, therefore, the α-equivalence class of $\tau.P$ is also definable by taking all the (finitely many) bound names in P as parameters to define the set. Hence, $\{(\tau, Q) \mid \tau.Q =_\alpha \tau.P\}$ is definable.

3. $Next(x(y).P) = \{(x\,z, Q\{z/y'\}) \mid z \in \mathcal{N},\ x(y').Q =_\alpha x(y).P\}$. See case 2.

4. $Next(\overline{x}\,y.P) = \{(\overline{x}\,y, Q) \mid \overline{x}\,y.Q =_\alpha \overline{x}\,y.P\}$. See case 2.

5. $Next(P_1 + P_2) = Next(P_1) \cup Next(P_2)$. By the induction hypothesis, $Next(P_1)$ and $Next(P_2)$ are definable, therefore, $Next(P_1) \cup Next(P_2)$ is definable.

6. $Next(P_1 \mid P_2) = \{(a, P_1 \mid P_2') \mid (a, P_2') \in Next(P_2),\ \mathsf{bn}(a) \cap \mathsf{fn}(P_1) = \emptyset\}$
$\cup \{(a, P_1' \mid P_2) \mid (a, P_1') \in Next(P_1),\ \mathsf{bn}(a) \cap \mathsf{fn}(P_2) = \emptyset\}$
$\cup \{(\tau, P_1'\{z/y\} \mid P_2') \mid (x\,z, P_1'\{z/y\}) \in Next(P_1),\ (\overline{x}\,z, P_2') \in Next(P_2)\}$
$\cup \{(\tau, P_1' \mid P_2'\{z/y\}) \mid (x\,z, P_2'\{z/y\}) \in Next(P_2),\ (\overline{x}\,z, P_1') \in Next(P_1)\}$
$\cup \{(\tau, (z)P_1'\{z/y\} \mid P_2') \mid (x\,z, P_1'\{z/y\}) \in Next(P_1),\ (\overline{x}(z), P_2') \in Next(P_2),\ z \notin \mathsf{fn}(P_2)\}$
$\cup \{(\tau, (z)P_1' \mid P_2'\{z/y\}) \mid (x\,z, P_2'\{z/y\}) \in Next(P_2),\ (\overline{x}(z), P_1') \in Next(P_1),\ z \notin \mathsf{fn}(P_1)\}$. Note that each subset connected by the union operator is definable, we conclude that the union is also definable.

7. $Next((z)P) = \{(a, (z')P') \mid (a, P') \in Next(Q),\ z' \notin a, (z')Q =_\alpha (z)P\}$
$\cup \{(\overline{x}(z'), P') \mid (\overline{x}\,z', P') \in Next(Q),\ (z')Q =_\alpha (z)P\}$. Same as the above cases.

8. $Next(!P) = \{(a, P' \mid !P) \mid (a, P') \in Next(P)\}$
$\cup \{(\tau, (P' \mid P''\{z/y\}) \mid !P) \mid (\overline{x}\,z, P') \in Next(P),\ (x\,z, P''\{z/y\}) \in Next(P)\}$
$\cup \{(\tau, (z)(P' \mid P''\{z/y\}) \mid !P) \mid (\overline{x}(z), P') \in Next(P),\ (x\,z, P''\{z/y\}) \in Next(P)\}$. Same as the above cases.

□

Next is used as an auxiliary function in the simulation of a π-term. We shall show later that an $\mathrm{RTM}^{\mathbb{A}}$ is capable of executing every transition from a π-term by enumerating *Next* in an algorithm. Now we give a lemma about *Next* to relate it with the operational semantics of the π-calculus.

Note that the set $Next(P)$ is defined by structural induction on P, in the same way as the structural operational semantics of the π-calculus. Moreover, if a transition $Q \xrightarrow{a} P'$ can be derived from the operational semantics, then by the ALPHA rule, there is a transition $P \xrightarrow{a} P'$ where $P =_\alpha Q$. In the definition of $Next(P)$, we also collect all the possible transitions from an α-equivalence class of a P. We can establish a one-to-one correspondence between the pairs in $Next(P)$ and the transitions obtained from Table 5.1.

Lemma 6.27. *Let P be a π-term. We have $P \xrightarrow{a} Q$ according to the operational semantics in Table 5.1 iff $(a, Q) \in Next(P)$.*

Now we proceed to illustrate the framework of simulating the transition system associated with a π-term P using an $\mathrm{RTM}^{\mathbb{A}}$. We start from a configuration in which $\lceil P \rceil$ is written on the tape. The $\mathrm{RTM}^{\mathbb{A}}$ may choose to produce an α-conversion of P. The machine first nondeterministically finds a bound name in P, and then generates a fresh name, and renames every occurrence of the bound name to the fresh name. An α-conversion is accomplished by repeating the above procedure an arbitrary number of times. Then the machine continues to enumerate the contents of $Next(P)$, and executing a transition according to the contents of *Next*.

As illustrated in Figure 6.5, we shall explain the simulation of a π-term P in 4 steps:

1. apply an α-conversion to P,
2. enumerate a pair from $Next(P)$,
3. execute an action from P, and
4. perform a substitution if necessary to reach the resulting state of the transition.

Firstly, we introduce a procedure to create an α-conversion of a π-term P.

1. The machine first traverses through the $\lceil P \rceil$, until it finds a bound name. The $\mathrm{RTM}^{\mathbb{A}}$ may syntactically recognize a bound name in P with some appropriate representation $\lceil P \rceil$ for P.
2. The machine nondeterministically chooses to either copy the name to a specific place (using the method in Example 6.23), or to continue the traversal procedure.

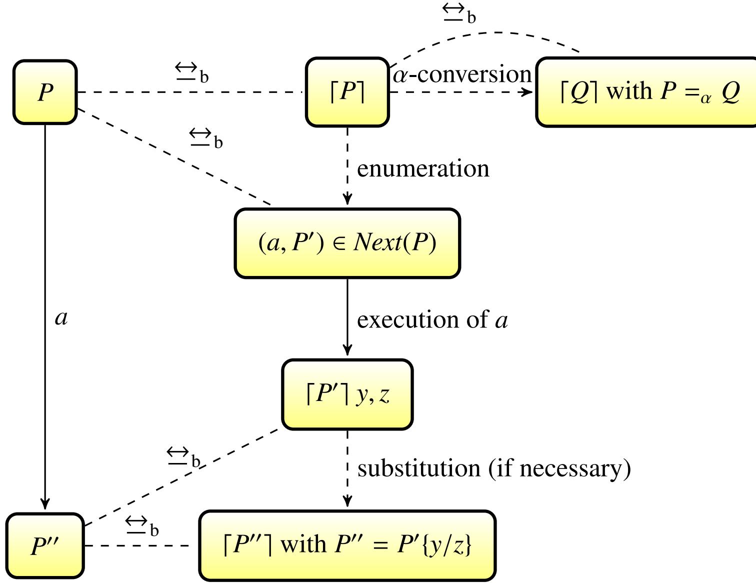


Figure 6.5: Simulation of the transitions from a π-term

3. When it reaches the end of $\lceil P \rceil$ and get a copy of a bound name y, it creates a fresh name z (using the method in Example 6.24).

4. It renames y to z to obtain $\lceil P\{z/y\} \rceil$ (using the method in Example 6.25).

5. The machine repeats the above procedure an arbitrary number of times.

We use the above procedure to generate the transitions of a π-term obtained from its α-conversions. This is the first step of enumerating $Next(P)$.

Secondly, we explain the procedure of enumerating pairs in $Next(P)$. The set $Next(P)$ is defined by induction on the structure of P. The enumeration can be done by a recursive call of the function according to the structure of P. Note that every pair in $Next(P)$ can be encoded as a finite string of symbols in $\mathcal{D}_\Box$. So the result generated by the enumeration can be recorded on the tape. Whenever a pair is generated, the machine has three choices: continuing the enumeration, returning to the initial state, or starting to execute a transition.

Next, we introduce the mechanism of executing a transition from a pair (a, P') that is written on the tape. We use a case distinction on a.

1. If a is of the form $x\,y\overline{x}\,y, \overline{x}\,(y)$, then P' is the resulting process. We introduce the following set of transitions in $\mathrm{RTM}^{\mathbb{A}}$ to execute such a transition:

$$\{(trans, a, a, \Box, R, rename) \mid a = x\,y \vee a = \overline{x}\,y \vee a = \overline{x}\,(y),\ x, y \in \mathcal{N}\} \ .$$

2. If a is τ, then $P'\{\vec{z}/\vec{y}\}$ would be the resulting process ($\vec{y}$ and $\vec{z}$ could be empty tuples). We could use the same method as in the previous case.

Note that the transitions in $\mathrm{RTM}^{\mathbb{A}}$ mentioned above are all definable sets.

Finally, the machine starts to complete the renaming procedure. We suppose that a process $\lceil P' \rceil$ and a sequence of renaming pairs $\{\vec{z}/\vec{y}\}$ is already on the tape. The renaming is done by just repeating the method in Example 6.25.

We have the following theorem.

Theorem 6.28. *For every π-term P, there exists an $\mathrm{RTM}^{\mathbb{A}}$ $\mathcal{M}$, such that $\mathcal{T}(P) \leftrightarrows_b \mathcal{T}(\mathcal{M})$.*

We only provide an intuitive analysis here. We consider the method we illustrated in Figure 6.5. Let P be a π-term, and we suppose that it has a transition $P \xrightarrow{a} P''$. There is a configuration $(start, \lceil P \rceil)$, and it has a sequence of transitions

$$(start, \lceil P \rceil) \longrightarrow^* (trans, (a, P')) \xrightarrow{a} (rename, \lceil P' \rceil) \longrightarrow^* (finish, \lceil P'' \rceil) \ .$$

Note that every τ-transition sequence above has a backward transition to the previous stage. Hence, those τ-transitions does not change the state modulo $\leftrightarrows_{\mathrm{b}}$. Therefore, one is able to establish an $\mathrm{RTM}^{\mathbb{A}}$ $\mathcal{M}$ with $\mathcal{T}(P) \leftrightarrows_{\mathrm{b}} \mathcal{T}(\mathcal{M})$.

Moreover, there is the following corollary about the nominal executability of the π-calculus.

Corollary 6.29. *The π-calculus is nominally executable modulo $\leftrightarrows_b$.*

6.5 Negative Result on mCRL2

In this section, we use the notion of nominal executability to study the expressivity of mCRL2. We show that the transition systems associated with mCRL2 processes are not nominally executable. Therefore, nominal executability gives a difference between mCRL2 and the π-calculus with respect to their expressivity.

6.5.1 LTSs with Atoms

We propose *labelled transition systems with atoms* to characterise nominally executable transition systems. In the view of Example 6.21 and Example 6.22, we exclude the transition systems with an illegal or a non-orbit-finite set of labels. We let $\mathcal{A}_\tau$ be a definable set of labels for the remainder of this section. We define the notion of transition system with atoms as follows:

Definition 6.30 (LTSs with atoms)**.** An $\mathcal{A}_\tau$-labelled transition system $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ is a *transition system with atoms* if $\mathcal{S}_T$ and $\longrightarrow_T$ are legal sets with atoms. We say that a transition system with atoms is K-supported if $K \subset \mathbb{A}$ and K is a support of the sets $\mathcal{S}_T$ and $\longrightarrow_T$.

We observe that a transition system with atoms $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ is K-supported iff for every $(s, a, t) \in \longrightarrow_T$ and for every K-automorphism π_K we have $\pi_K(s, a, t) \in \longrightarrow_T$, where $\pi_K(s, a, t) = (\pi_K(s), \pi_K(a), \pi_K(t))$. This fact also applies on sequences of transitions.

For example, the transition systems associated with π-calculus terms are transition systems with atoms. We consider the set of names as the set of atoms. The set of π-terms and the set of transitions from all the π-terms are sets with atoms with empty supports. The support of the transition system associated with an individual π-term is the set of its free names. Note that the set of free names does not grow by transition [77].

The following theorem shows that nominally executable transition systems are labelled transition systems with atoms.

Theorem 6.31. *For every RTM*$^{\mathbb{A}}$ $\mathcal{M}$*, the associated transition system* $\mathcal{T}(\mathcal{M})$ *is a transition system with atoms.*

Proof. Let $\mathcal{M} = (Q_{\mathcal{M}}, \mapsto_{\mathcal{M}}, Ini_{\mathcal{M}})$, then there exists a finite set of atoms $K \subset \mathbb{A}$ such that, for every $(s, a, d, e, M, t) \in \mapsto_{\mathcal{M}}$, and for every K-automorphism π_K, we have $\pi_K(s, a, d, e, M, t) \in \mapsto_{\mathcal{M}}$. It follows that, K is a support of the set of configurations of $\mathcal{M}$, as well as the transition relation of $\mathcal{T}(\mathcal{M})$. Therefore the transition system $\mathcal{T}(\mathcal{M})$ is legal. □

6.5.2 mCRL2

The formal specification language mCRL2 [49, 50] is widely used to specify and analyze the behaviour of distributed systems. The question arises to what extent the transition systems specified by mCRL2 are executable. The actions in an mCRL2 specification may contain tuples of integers of any arbitrary lengths, which leads to

a set of actions with infinitely many orbits. Moreover, we can also specify transition systems that do not have a finite support in mCRL2. Therefore, we conclude that such transition systems are not nominally executable.

Corollary 6.32. *There exists an mCRL2 specification P, such that the transition system $\mathcal{T}(P)$ is not nominally executable.*

Proof. Consider the following mCRL2 specification:

$$\begin{aligned} &act\, num : \; Nat; \\ &init\, sum\, v : \; Nat\, .\, num(2 * v); \end{aligned}$$

It defines a transition system that includes a set of transitions from the initial state labelled by all even natural numbers as follows:

$$\{(\uparrow, 2n, \downarrow) \mid n \in \mathbb{N}\} \;\; .$$

This transition system does not have a finite support, therefore, it is not an LTS with atoms. By Theorem 6.31, every nominally executable transition systems is a transition system with atoms. So the transition systems associated with mCRL2 processes are not necessarily nominally executable. □

Remark 6.33. Actually, in mCRL2, we can also specify the transition system in 6.3, which is also not a transition system with atoms. Moreover, such a transition system could be simulated in a value-passing calculus that communicates with natural numbers and can make an addition operation on numbers, see [36].

6.6 Remarks

In this chapter, we investigated the notion of executable transition systems associated with RTM^{∞} s and $\mathrm{RTM}^{\mathbb{A}}$ s. We gave some properties for executable transition systems regarding RTM^{∞} and $\mathrm{RTM}^{\mathbb{A}}$. We proposed a notion of nominal executability and used it to distinguish the expressivity of the π-calculus and mCRL2. We got some evidence to show that nominal executability is a proper notion for the study of expressivity on process calculus with nominal sets.

Last but not least, we propose some future work on this issue.

1. An independent characterisation of the classes of nominally executable LTSs is still not clear to us. We need to propose a notion of ”effectiveness” on definable sets, as well as labelled transition systems with atoms.

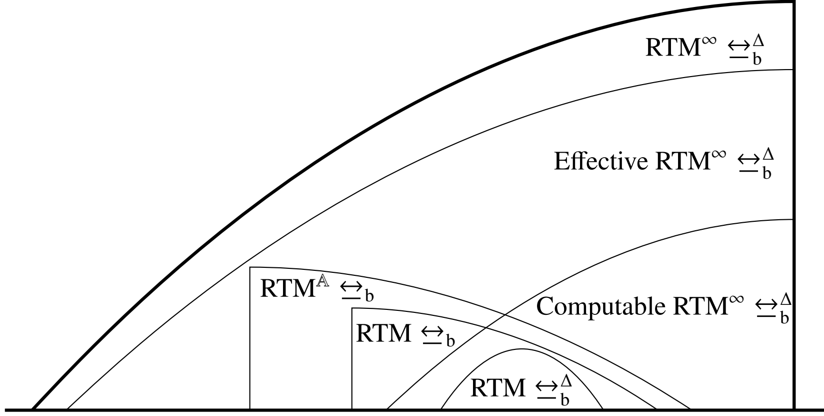


Figure 6.6: A hierarchy of executability

We hereby give an example of "effective" LTS with atoms and show that it is nominally executable which may gives us some ideas of the simulation of an "effective" LTS with atoms.

Example 6.34. Consider the transition system with atoms $T = (\mathcal{S}_T, \longrightarrow_T, \uparrow_T)$ with $\mathcal{A} = \{a \mid a \in \mathbb{A}\} \cup \{(a,b) \mid\in a, b \in \mathbb{A}\}$ defined as follows:

(a) $\mathcal{S}_T = \{\uparrow_T, t\} \cup \{s_a \mid a \in \mathbb{A}\} \cup \{s_{a,b} \mid a, b \in \mathbb{A}\}$, and

(b) $\longrightarrow_T = \{(\uparrow_T, \tau, s_a) \mid a \in \mathbb{A}\} \cup \{(s_a, a, s_{a,b}) \mid a, b \in \mathbb{A}\}$
$\cup \{(s_{a,b}, (a,b), t) \mid a, b \in \mathbb{A}\}$.

We define an RTM$^{\mathbb{A}}$ $\mathcal{M} = (Q_M, \mapsto_M, Ini_M)$ with $\mathcal{D} = \mathcal{A}$ to simulate T as follows:

(a) $Q_M = \{Ini_M, s_1, s_2, s_3\}$, and

(b) $\mapsto_M = \{(Ini_M, \Box, \tau, a, R, s_1) \mid a \in \mathbb{A}\} \cup \{(s_1, \Box, \tau, \Box, L, s_2)\}$
$\cup \{(s_2, a, a, (a,b), R, s_1) \mid a, b \in \mathbb{A}\} \cup \{(s_2, (a,b), (a,b), \Box, R, s_3) \mid a, b \in \mathbb{A}\}$.

$\mapsto_M$ is a legal and orbit-finite set. The least support is the empty set, and it has six orbits, namely,

(a) $\{(Ini_M, \Box, \tau, a, R, s_1) \mid a \in \mathbb{A}\}$,

(b) $\{(s_1, \Box, \tau, \Box, L, s_2)\}$,

(c) $\{(s_2, a, a, (a,b), R, s_1) \mid a, b \in \mathbb{A},\ a = b\}$,

(d) $\{(s_2, a, a, (a, b), R, s_1) \mid a, b \in \mathbb{A},\ a \neq b\}$,

(e) $\{(s_2, (a, b), (a, b), \square, R, s_3) \mid a, b \in \mathbb{A},\ a = b\}$, and

(f) $\{(s_2, (a, b), (a, b), \square, R, s_3) \mid a, b \in \mathbb{A},\ a \neq b\}$.

We verify $\mathcal{T}(\mathcal{M}) \leftrightarrow_b T$ by establishing a branching bisimulation relation as follows:

$$\begin{aligned}\mathcal{R} = \quad & \{((Ini_M, \delta_0, 0), \uparrow_T)\} \cup \{((s_1, \delta_a, 1), s_a) \mid a \in \mathbb{A}\} \\ & \cup\{((s_2, \delta_a, 0), s_a) \mid a \in \mathbb{A}\} \cup \{((s_1, \delta_{a,b}, 1), s_{a,b}) \mid a, b \in \mathbb{A}\} \\ & \cup\{((s_2, \delta_{a,b}, 0), s_{a,b}) \mid a, b \in \mathbb{A}\} \cup \{((s_3, \delta_0, 1), t)\}\ ,\end{aligned}$$

where $\delta_a(0) = a$ and $\delta_a(i) = \square$ for all $i \neq 0$ and for all $a \in \mathbb{A}$; $\delta_{a,b}(0) = (a, b)$ and $\delta_{a,b}(i) = \square$ for all $i \neq 0$ and for all $a, b \in \mathbb{A}$.

According to [24], atoms and set builder expressions which use atom parameters can be written down as bit strings and processed by algorithms (e.g. formalised as Turing machines). Moreover, the paper gives the definition of computable functions and recursively enumerable sets on definable sets. The following questions should be answered in order to arrive at a satisfactory characterisation of nominal executability.

(a) How to enumerate a nominally recursively enumerable definable set under an encoding using an $\mathrm{RTM}^{\mathbb{A}}$ (or Turing machine with atoms)?

(b) How to enumerate the transition relation of an effective LTS with atoms using an $\mathrm{RTM}^{\mathbb{A}}$ (or Turing machine with atoms)?

(c) How to simulate the an effective LTS with atoms using an $\mathrm{RTM}^{\mathbb{A}}$?

As long as we are able to answer the above questions, we aims to a hierarchy of executability as in Figure 6.6:

(a) by Theorem 2.22, the class of executable transition systems by RTMs modulo $\leftrightarrow_b^{\Delta}$ consists of the boundedly branching computable transition system with a finite set of labels;

(b) by Theorem 2.23, the class of executable transition systems by RTMs modulo $\leftrightarrow_b$ consists of the effective transition system with a finite set of labels;

(c) The class of nominally executable transition systems consists of the effective transition system with atoms, provided a proper notion of "effective transition system with atoms";

(d) by Corollary 6.7, the class of executable transition systems by RTM^{∞}s with a computable transition relation modulo $\underline{\leftrightarrow}_{b}^{\Delta}$ is the computable transition system;

(e) by Corollary 6.6,the class of executable transition systems by RTM^{∞}s with an effective transition relation modulo $\underline{\leftrightarrow}_{b}^{\Delta}$ is the effective transition system; and

(f) by Theorem 6.5, the class of executable transition systems by RTM^{∞}s modulo $\underline{\leftrightarrow}_{b}^{\Delta}$ is the countable transition system.

2. The precise characterisation of the transition systems executable by $\text{RTM}^{\mathbb{A}}$s modulo $\underline{\leftrightarrow}_{b}^{\Delta}$ is still open. Further restrictions should be imposed to make it possible to generate all possible transitions of an arbitrary state in the transition system from a single configuration of an $\text{RTM}^{\mathbb{A}}$.

3. It would be interesting to show the existence of a universal $\text{RTM}^{\mathbb{A}}$, such that it is able to simulate the behaviour of every $\text{RTM}^{\mathbb{A}}$ with its encoding.

4. Psi-calculi [16] were introduced to characterise transition systems with nominal data types for data structures and with logical assertions representing facts about data. The adoption of nominal data types provides a natural characterisation of the behaviour executed by $\text{RTM}^{\mathbb{A}}$. An encoding of the π-calculus was proposed in the psi-calculus [17], proving that the psi-calculus is at least as expressive as the π-calculus. It would be interesting to figure out the relationship between the transition systems associated with the psi-calculus and the transition systems associated with $\text{RTM}^{\mathbb{A}}$. We conjecture that as long as the logical assertions used in psi-calculus are semi-decidable, the transition systems associated with the psi-calculus processes are nominally executable.

5. A notion of nominal transition system was proposed by Parrow et al. [73]. Nominal transition systems satisfy the requirements of transition systems with atoms naturally. We did not use the notion of nominal transition system since the predicates for states in Hennessy-Milner logic are ignored in proving the executability. By assuming an appropriate definition of effectiveness, we conjecture that the effective nominal transition systems are nominally executable.

6. The value-passing calculus [36] is a process calculus in which the contents of communications are values chosen from natural numbers. It can be used to specify transition systems that are not nominally executable (such as the one we used in the proof of Corollary 6.32). We could investigate its expressivity with an extended version of nominal executability by imposing some structures on the sets of atoms, e.g., the natural numbers.

Chapter 7

Conclusion

Executability theory provides an integration of computability theory and concurrency theory. This thesis presents some studies of the executability theory which are summarized as follows.

7.1 Robustness

A first basic goal in this thesis is to collect evidence for the robustness of RTMs. To achieve this goal, we compare RTMs with other models of interactive computation, both in the transition system semantics, and in the semantics of other models.

In [13], RTMs is proved to be at least as expressive as PTMs both in transition system semantics modulo branching bisimilarity and in interactive transition system semantics of PTMs.

In Chapter 3 of this thesis, we provide another evidence by showing that RTMs also subsume ITMs both in transition system semantics modulo divergence-preserving branching bisimilarity and in the semantics of stream translations. At a first glance, RTMs and ITMs are incomparable since RTMs give rise to executable transition systems whereas ITMs give rise to interactive computable ω-translations. We use two approaches to compare the two models in a unified semantics. One is to propose a transition system semantics to ITMs. We show that every transition system associated with an ITM is executable modulo divergence-preserving branching bisimilarity, which is illustrated as Theorem 3.8. The other approach is to define a class of RTMs for interactive ω-translations. We show that RTMs and ITMs define the same class of interactive ω-translations, see Corollary 3.19. Van Leeuwen and Wiedermann further

introduced ITMs with advice [60]; we also make an analogy to that notion and define RTMs with advice. We show that RTMs with advice yield more powerful notions of executability, namely, every boundedly branching transition system is executable by some RTM/A modulo divergence-preserving branching bisimilarity and every countable transition system is executable by some RTM/A modulo the divergence-insensitive variant of branching bisimilarity, see Theorem 3.25 and Theorem 3.26.

7.2 Comparison

The second goal of this thesis is to integrate notions in computability theory and concurrency theory, which is the starting point of the theory of executability. In Chapter 4, we investigate the effect of termination to the operational semantics of process calculi, which is one of the bases of concurrency theory. The distinction between successful and unsuccessful termination is important for a smooth integration; in automata theory this distinction is important to express whether a string should be accepted or not. We give an argument that the current semantics of sequential composition does not work ideally in the presence of intermediate termination. The phenomenon of transparency becomes the main obstacle for two problems, namely, showing that pushdown processes could simulate context-free processes modulo strong bisimilarity, and showing that TCP with iteration and nesting is reactively Turing powerful modulo divergence-preserving branching bisimilarity.

In order to solve the above problems, we revise the operational semantics of the sequential composition operator by adding a negative premises. We show that in the revised semantics, every context-free process can be simulated by a pushdown process modulo strong bisimilarity, see Theorem 4.18; and TCP with iteration and nesting is reactively Turing powerful modulo divergence-preserving branching bisimilarity, see Theorem 4.25.

7.3 Expressivity

The final goal of this thesis is to apply the executability theory in evaluating the expressivity of process calculi. We establish a general framework of evaluating the expressivity of process calculi based on executability and reactive Turing powerfulness and parameterised by the choice of behavioural equivalences.

In Chapter 5, we apply this framework on the π-calculus. We show that the π-calculus is reactively Turing powerful modulo divergence-preserving branching bisimilarity by making use of the feature of link mobility in the π-calculus, see Theorem 5.11.

For the executability of the π-calculus, it took us more effort than we expected. The infinity of names in the setting of the π-calculus and the finiteness of set of labels of RTMs makes the π-calculus unexecutable trivially. This result is certainly unsatisfactory. We made a first effort to improve the result by restricting the π-calculus to finitely many names. We show that the restricted π-calculus is executable modulo the divergence-insensitive variant of branching bisimilarity, see 5.16.

In Chapter 6, we made another try of incorporating infinite labels in the formalism of RTMs. We introduce a notion of infinitary RTMs, and show that the associated notion of executability is so strong that every countable transition system is trivially executable modulo divergence-preserving branching bisimilarity, see Theorem 6.5. To obtain a more useable notion of executability with infinite alphabets, we use sets with atoms and propose a notion of RTMs with atoms as well as the associated notion of nominal executability. We establish that the π-calculus is nominally executable modulo the divergence-insensitive variant of branching bisimilarity and we show also that another process calculus mCRL2 is not nominally executable. In this way we obtain some evidence showing that nominal executability is a useful notion in evaluating the expressivity for process calculi with infinite alphabets.

7.4 Future Work

We make a final remark of this thesis by listing some future work in the theory of executability.

1. We need to get more evidence for the robustness of RTMs. There are still quite a number of interactive computation models in the literature which have not been proved to be subsumed by RTMs, for instance:
    (a) Lynch and Tuttle's I/O automaton [65], which is already mentioned in Section 3.5.
    (b) Gurevich's Abstract State Machine [51] also includes a facility of interaction.
    (c) Fu's theory of interaction [37] introduces a minimal requirement of interactive computation models and proposes a notion of subbisimilarity as a standard way of comparing different models of interaction.
    (d) Bergstra and Middelburg's Maurer machines [20] also have a mechanism of interaction and they could simulate Turing machines.

   In order to confirm a concurrent version of the Church-Turing thesis, we need to compare the above theories with the theory of executability, and to show that

these models are equivalent to RTMs in an appropriate semantics (or at least in the labelled transition system semantics).

2. Complexity theory is induced from computability theory by making a constraint on the usage of resources during computation. Executing a certain behaviour in an arbitrary transition system also costs an amount of resources, e.g., time or space. In [81], a complexity model is established based on ITMs. By an analogy, we might be able to evaluate the complexity of RTMs that simulates ITMs. Furthermore, we are interested in a general theory to evaluate the resource consumption of executing a certain behaviour.

3. For the revised semantics of the sequential composition operator, we already got a positive result on its congruence property. In future work, we shall establish an axiomatization for $TSP^{;}$ modulo strong bisimilarity.

4. In Section 4.4, the relationship between pushdown processes and context-free processes is discussed in the revised semantics. However, the correspondence between these models is still not yet clear in the original semantics. The best-known result is that every context-free process can be simulated by a pushdown process modulo contrasimulation [9], but no correspondence has been established modulo any finer notion of behavioural equivalence. We shall leave it as a future work to close this gap.

5. Both TCP and $TCP^{\sharp}$ have been shown to be reactively Turing powerful. As mentioned in Section 4.6, we are interested in finding more alternatives for recursive specification which still keeps the calculus reactively Turing powerful, such as replication, pushdown, and back-and-forth.

6. We could make some extensions to executability theory in two dimensions:

    (a) As discussed in Section 3.5, we could introduce different classes of advice functions for RTM/A. In a way, a hierarchy of RTMs with different advice could be imagined.

    (b) As mentioned in Section 6.6, we are interested in a more complete hierarchy of RTMs with different classes of infinite sets. We already have the executability of infinitary RTMs, and we still need a clear characterisation of nominally executable transition systems independent of $RTM^{\mathbb{A}}$s. Moreover, we could even introduce an RTM that uses natural numbers as its alphabet, and extend our theory to numeral executability in the future. Hence, we are looking forward to establish a correspondence between the value-passing calculus [36] with the theory of executability.

(c) By analogy to the robustness of the theory of executability, we shall also collect evidence for nominal executability, for instance, the existence of a universal $\mathrm{RTM}^{\mathbb{A}}$, and the comparison of $\mathrm{RTM}^{\mathbb{A}}$ with Psi-calculi [16] and nominal transition systems [73].

7. The executability theory could be suitable material for a course about models of computation or a course about concurrency theory. In order to incorporate the executability theory in a course for students, more examples and exercises should be invented. We hereby list a few:

   (a) Show the equivalence between an RTM with a stay transition and a general RTM.

   (b) Show the equivalence between a single-tape RTM and a multi-tape RTM.

   (c) Show the equivalence between an RTM with a two-way infinite tape and an RTM with an one-way infinite tape.

   (d) Give a proper definition of deterministic RTM, and illustrate its relationship with a general RTM,

   (e) In [13], an example of an unexecutable transition system was given. We expect to construct more such transition systems.

A theory of executability has been established by this thesis, nevertheless there is still a long journey towards *the* theory of executability.

Bibliography

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra*, pages 197–292. Elsevier, 2001.

[2] L. Aceto and M. Hennessy. Termination, Deadlock, and Divergence. *J. ACM*, 39(1):147–187, 1992.

[3] L. Aceto, A. Ingólfsdóttir, K. M. Larsen, and J. Srba. *Reactive Systems—Modelling, Specification and Verification*. Cambridge University Press, 2007.

[4] J.C.M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50. Cambridge university press, 2010.

[5] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the Consistency of Koomen's Fair Abstraction Rule. *Theoretical Computer Science*, 51:129–176, 1987.

[6] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of Bisimulation Equivalence for Processes Generating Context-free Languages. *J. ACM*, 40(3):653–682, 1993.

[7] J.C.M Baeten, F. Corradini, and C. Grabmayer. A Characterization of Regular Expressions under Bisimulation. *J. ACM*, 54(2):6, 2007.

[8] J.C.M. Baeten, P.J.L. Cuijpers, B. Luttik, and P.J.A. van Tilburg. A Process-theoretic Look at Automata. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering, Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*, pages 1–33. Springer, 2009.

[9] J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A Context-Free Process as a Pushdown Automaton. In Franck van Breugel and Marsha Chechik, editors,

CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings, volume 5201 of *Lecture Notes in Computer Science*, pages 98–113. Springer, 2008.

[10] J.C.M. Baeten, B. Luttik, T. Muller, and P.J.A. van Tilburg. Expressiveness modulo Bisimilarity of Regular Expressions with Parallel Composition. *Mathematical Structures in Computer Science*, 26:933–968, 2016.

[11] J.C.M. Baeten, B. Luttik, and P.J.A. van Tilburg. Computations and Interaction. In Raja Natarajan and Adegboyega K. Ojo, editors, *ICDCIT*, volume 6536 of *Lecture Notes in Computer Science*, pages 35–54. Springer, 2011.

[12] J.C.M. Baeten, B. Luttik, and P.J.A. van Tilburg. Turing Meets Milner. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2012.

[13] J.C.M. Baeten, B. Luttik, and P.J.A. van Tilburg. Reactive Turing machines. *Information and Computation*, 231:143–166, 2013.

[14] J.C.M. Baeten, B. Luttik, and F. Yang. Sequential Composition in the Presence of Intermediate Termination (Extended Abstract). In Kirstin Peters and Simone Tini, editors, *Proceedings Combined 24th International Workshop on Expressiveness in Concurrency and 14th Workshop on Structural Operational Semantics, EXPRESS/SOS 2017, Berlin, Germany, 4th September 2017*, volume 255 of *EPTCS*, pages 1–17, 2017.

[15] T. Basten. Branching bisimilarity is an equivalence indeed! *Information Processing Letters*, 58(3):141–147, 1996.

[16] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: Mobile Processes, Nominal Data, and Logic. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 39–48. IEEE Computer Society, 2009.

[17] J. Bengtson, M. Johansson, J. Parrow, and B. Victor. Psi-calculi: a Framework for Mobile Processes with Nominal Data and Logic. *Logical Methods in Computer Science*, 7(1), 2011.

[18] J.A. Bergstra, I. Bethke, and A. Ponse. Process Algebra with Iteration and Nesting. *The Computer Journal*, 37(4):243–258, 1994.

[19] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and control*, 60(1-3):109–137, 1984.

[20] J.A. Bergstra and C.A. Middelburg. Simulating Turing Machines on Maurer Machines. *Journal of Applied Logic*, 6(1):1–23, 2008.

[21] J.A. Bergstra and A. Ponse. Non-regular Iterators in Process Algebra. *Theoretical Computer Science*, 269(1):203–229, 2001.

[22] J.A. Bergstra and A. Ponse. Register-machine based processes. *J. ACM*, 48(6):1207–1241, 2001.

[23] B. Bloom. When is Partial Trace Equivalence Adequate? *Formal Aspects of Computing*, 6(3):317–338, 1994.

[24] M. Bojańczyk. *Slightly Infinite Sets*. 2016.

[25] M. Bojańczyk, B. Klin, and S. Lasota. Automata with Group Actions. In *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*, pages 355–364. IEEE Computer Society, 2011.

[26] M. Bojańczyk, B. Klin, S. Lasota, and S. Toruńczyk. Turing Machines with Atoms. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 183–192. IEEE Computer Society, 2013.

[27] R. Bol and J.F. Groote. The Meaning of Negative Premises in Transition System Specifications. *J. ACM*, 43(5):863–914, 1996.

[28] N. Busi, M. Gabbrielli, and G. Zavattaro. On the Expressive Power of Recursion, Replication and Iteration in Process Calculi. *Mathematical Structures in Computer Science*, 19(6):1191–1222, 2009.

[29] J. Cabessa and A.E.P. Villa. The Super-Turing Computational Power of Interactive Evolving Recurrent Neural Networks. In Valeri Mladenov, Petia D. Koprinkova-Hristova, Günther Palm, Alessandro E. P. Villa, Bruno Appollini, and Nikola Kasabov, editors, *Proceedings of ICANN 2013*, volume 8131 of *LNCS*, pages 58–65. Springer, 2013.

[30] D. Caucal. Branching Bisimulation for Context-free Processes. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 12th Conference, New Delhi, India, December 18-20, 1992, Proceedings*, volume 652 of *Lecture Notes in Computer Science*, pages 316–327. Springer, 1992.

[31] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[32] P. Darondeau. Bismulation and Effectiveness. *Information Processing Letters*, 30(1):19–20, 1989.

[33] E.W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569, 1965.

[34] E. Eberbach. The $-calculus Process Algebra for Problem Solving: A Paradigmatic Shift in Handling Hard computational Problems. *Theoretical Computer Science*, 383(2):200–243, 2007.

[35] W. Fokkink, R.J. van Glabbeek, and B. Luttik. Divide and Congruence III: Stability & Divergence. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 15:1–15:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[36] Y. Fu. The Value-Passing Calculus. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 166–195. Springer, 2013.

[37] Y. Fu. Theory of Interaction. *Theoretical Computer Science*, 611:1–49, 2016.

[38] Y. Fu and H. Lu. On the Expressiveness of Interaction. *Theoretical Computer Science*, 411(11-13):1387–1451, 2010.

[39] M.J. Gabbay and A.M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3), 2002.

[40] R.J. van Glabbeek. The Linear Time - Branching Time Spectrum II. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.

[41] R.J. van Glabbeek. The Meaning of Negative Premises in Transition System Specifications II. *J. Log. Algebr. Program.*, 60-61:229–258, 2004.

[42] R.J. van Glabbeek, B. Luttik, and N. Trčka. Branching Bisimilarity with Explicit Divergence. *Fundamenta Informaticae*, 93(4):371–392, 2009.

[43] R.J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.

[44] K.F. Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931.

[45] D.Q. Goldin, S.A. Smolka, P.C. Attie, and E.L. Sonderegger. Turing Machines, Transition Systems, and Interaction. *Information and Computation*, 194(2):101–128, 2004.

[46] D.Q. Goldin, S.A. Smolka, and P. Wegner. *Interactive Computation: The New Paradigm*. Springer Science & Business Media, 2006.

[47] D. Gorla. Towards a Unified Approach to Encodability and Separation Results for Process Calculi. *Information and Computation*, 208(9):1031–1053, 2010.

[48] J.F. Groote. Transition System Specifications with Negative Premises. *Theoretical Computer Science*, 118(2):263–299, 1993.

[49] J.F. Groote, A. Mathijssen, M.A. Reniers, Y. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS), 27.08. - 01.09.2006*, volume 06351 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[50] J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT press, 2014.

[51] Y. Gurevich. Evolving Algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and validation methods*, pages 9–36. Oxford University Press, 1993.

[52] F.C. Hennie and R.E. Stearns. Two-tape Simulation of Multitape Turing Machines. *J. ACM*, 13(4):533–546, 1966.

[53] C.A.R. Hoare. Communicating Sequential Processes. *Commun. ACM*, 21(8):666–677, 1978.

[54] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation - (2. ed.)*. Addison-Wesley series in computer science. Addison-Wesley-Longman, 2001.

[55] H. Huang and F. Yang. An Interpretation of Erlang into Value-Passing Calculus. *Journal of Networks*, 8(7):1504–1513, 2013.

[56] S. C. Kleene. General Recursive Functions of Natural Numbers. *Mathematische annalen*, 112(1):727–742, 1936.

[57] J. van Leeuwen and J. Wiedermann. On Algorithms and Interaction. In Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science 2000, 25th International Symposium, MFCS 2000, Bratislava, Slovakia, August 28 - September 1, 2000, Proceedings*, volume 1893 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2000.

[58] J. van Leeuwen and J. Wiedermann. Beyond the Turing Limit: Evolving Interactive Systems. In Leszek Pacholski and Peter Ruzicka, editors, *SOFSEM 2001: Theory and Practice of Informatics, 28th Conference on Current Trends in Theory and Practice of Informatics Piestany, Slovak Republic, November 24 - December 1, 2001, Proceedings*, volume 2234 of *Lecture Notes in Computer Science*, pages 90–109. Springer, 2001.

[59] J. van Leeuwen and J. Wiedermann. The Turing Machine Paradigm in Contemporary Computing. In *Mathematics unlimited-2001 and beyond*, pages 1139–1155. Springer, 2001.

[60] J. van Leeuwen and J. Wiedermann. A Theory of Interactive Computation. In *Interactive Computation*, pages 119–142. Springer, 2006.

[61] B. Luttik and F. Yang. Executable Behaviour and the π-Calculus. *CoRR*, abs/1410.4512, 2014.

[62] B. Luttik and F. Yang. Executable Behaviour and the π-Calculus (Extended Abstract). In Sophia Knight, Ivan Lanese, Alberto Lluch-Lafuente, and Hugo Torres Vieira, editors, *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015.*, volume 189 of *EPTCS*, pages 37–52, 2015.

[63] B. Luttik and F. Yang. On the Executability of Interactive Computation. In Arnold Beckmann, Laurent Bienvenu, and Natasa Jonoska, editors, *Pursuit of the Universal - 12th Conference on Computability in Europe, CiE 2016, Paris, France, June 27 - July 1, 2016, Proceedings*, volume 9709 of *Lecture Notes in Computer Science*, pages 312–322. Springer, 2016.

[64] B. Luttik and F. Yang. Reactive Turing Machines with Infinite Alphabets. *CoRR*, abs/1610.06552, 2016.

[65] N.A. Lynch and M.R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In Fred B. Schneider, editor, *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, British Columbia, Canada, August 10-12, 1987*, pages 137–151. ACM, 1987.

[66] J.C. Martin. *Introduction to Languages and the Theory of Computation*, volume 4. McGraw-Hill NY, USA, 1991.

[67] R. Milner. *Communication and Concurrency*. PHI Series in computer science. Prentice Hall, 1989.

[68] R. Milner. Functions as Processes. *Mathematical structures in computer science*, 2(2):119–141, 1992.

[69] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I & II. *Information and computation*, 100(1):1–77, 1992.

[70] F. Moller. Infinite Results. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96, Concurrency Theory, 7th International Conference, Pisa, Italy, August 26-29, 1996, Proceedings*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216. Springer, 1996.

[71] C. Palamidessi. Comparing the Expressive Power of the Synchronous and Asynchronous π-calculi. *Mathematical Structures in Computer Science*, 13(5):685–719, 2003.

[72] D.M.R. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *Theoretical Computer Science, 5th GI-Conference, Karlsruhe, Germany, March 23-25, 1981, Proceedings*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[73] J. Parrow, J. Borgström, L. Eriksson, R. Gutkovas, and T. Weber. Modal Logics for Nominal Transition Systems. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015,*

Madrid, Spain, September 1.4, 2015, volume 42 of *LIPIcs*, pages 198–211. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[74] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, University of Bonn, West Germany, 1962.

[75] I.C.C. Phillips. A Note on Expressiveness of Process Algebra. In Geoffrey L. Burn, Simon J. Gay, and Mark Ryan, editors, *Theory and Formal Methods 1993, Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods, Isle of Thorns Conference Centre, Chelwood Gate, Sussex, UK, 29-31 March 1993*, Workshops in Computing, pages 260–264. Springer, 1993.

[76] H. Rogers. *Theory of Recursive Functions and Effective Computability*, volume 5. McGraw-Hill New York, 1967.

[77] D. Sangiorgi and D. Walker. *The Pi-Calculus - a Theory of Mobile Processes*. Cambridge University Press, 2001.

[78] J.C. Shepherdson and H.E. Sturgis. Computability of Recursive Functions. *J. ACM*, 10(2):217–255, 1963.

[79] P.J.A. van Tilburg. *From Computability to Executability: a Process-theoretic View on Automata Theory*. PhD thesis, Eindhovn University of Technology, 2011.

[80] A.M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 2, pages 230–265. Wiley Online Library, 1937.

[81] P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. PhD thesis, Utrecht University, 2006.

[82] J. Wiedermann and J. van Leeuwen. How We Think of Computing Today. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, volume 5028 of *Lecture Notes in Computer Science*, pages 579–593. Springer, 2008.

Index

λ-calculus, 2
ω-translation, 31

$\mathcal{A}$, *see* action symbols
α equivalence class, 92
α-conversion, 91
Abstract State Machines, 27
accepting states, 2
ACP, 3, 27
action symbols, 9
advice function, 41
advice process, 41
arithmetical hierarchy, 2
$\mathcal{A}_\tau$, *see* action symbol,unobservable action
atom automorphism, 120
atoms, 119

behavioural equivalence, 10
bisimulation up to $\leftrightarrows_b$, 13
bisimulation up to $\leftrightarrows_b^\Delta$, 13
blank tape cell, 16
bound name, 91
bounded branching, 21
boundedly branching up to $\leftrightarrows_b^\Delta$, 24
branching bisimulation, 11
branching degree, 20
branching degree up to $\leftrightarrows_b^\Delta$, 24

C, *see* channels
$Conf$, *see* configuration of an RTM
CCS, 3, 27
channels, 17
Church-Turing thesis, 2
communication, 17
complexity theory, 2
component, 30
computability theory, 1
computable functions, 2
computable LTSs, 20
concurrency theory, 1
configuration of an RTM, 16
congruence, 12
context-free process, 68
control state, 2
countable transition system, 117
CSP, 27

$\mathcal{D}$, *see* data symbols
$\mathcal{D}_\Box$, *see* tape symbols
data symbols, 16
definable, 122
divergence, 22
divergence up to $\leftrightarrows_b^\Delta$, 25

divergence-insensitive variant of branching bisimilarity, *see* branching bisimulation
divergence-preserving branching bisimilarity, 11
divergence-preserving condition, 12

effective LTSs, 20
effectively computable function, 20
effectively executable behaviour, 20
enumeration, 22
environment, 30
equality atoms, *see* atoms
executability, 28
executable LTSs, 20

final states, 2, *see* terminating states
finitely branching, 20
forgetfulness, 57
free name, 91

Gödel numbering, 20

half counter, 59

i/o labelled transition system, 37
infinitary Reactive Turing Machines, 116
initial state, 2, 10
interaction, 17
Interactive Turing Machine, 4
interactive Turing machine, 27
interactiveness, 31
intermediate termination, 49
iteration, 51
ITM, *see* Interactive Turing Machine

labelled transition system, *see* transition systems
labelled transition systems with atoms, 133
λ calculus, 26
legal, 120
link mobility, 94
LTS, *see* transition systems

$\mathfrak{M}$, *see* models in concurrency theory
mCRL2, 27
models in concurrency theory, 27

$\mathcal{N}$, *see* names
names, 91
nesting, 51
nominal executability, 114, 123
nominal sets, 113

orbit, 121
orbit-finite, 120

parallel composition on RTMs, 19
parallel composition on transition systems, 18
partial computable function, 22
persistent Turing machine, 27
Persistent Turing Machine, 4
Petri Net, 3
process calculi, 3
PTM, *see* Persistent Turing Machine
pushdown automata, 69
pushdown processes, 70

$\mathfrak{R}$, *see* Reactive Turing Machine
Random Access Machines, 2, 26
reachability, 10
reachable, 10
Reactive Turing Machine, 4, 16
Reactive Turing Machine with advice, 42

Reactive Turing Machine with atoms, 123
reactively Turing powerfulness, 28
recursive functions, 2
reflexive-transitive closure, 10
relative expressivity, 26
rooted divergence-preserving branching bisimulation, 13
rootedness condition, 12
RTM, *see* Reactive Turing Machine, *see* Reactive Turing Machine
RTMs with stay transitions, 34

$\mathcal{S}$, *see* states
set with atoms, 120
states, 9
strong bisimulation, 10
structural operational semantics, 92
support, 120

$\mathcal{T}()$, *see* transition system
tape head, 2, 16
tape instance, 16
tape symbol, 2
tape symbols, 16
τ, *see* unobservable action
TCP, *see* Theory of Communicating Processes
terminating states, 10
the π-calculus, 3, 27, 91
the finite π-calculus, 107
Theory of Communicating Processes, 27, 54
Theory of Sequential Processes, 52, 68
transition, 2, 10
transition relation, 10
transition system, 9
transitive closure, 10
transparency, 56
transparent, 56
Turing machines, 1, 15

unbounded branching behaviour, 56
unboundedly branching up to $\leftrightarrow_{\mathrm{b}}^{\Delta}$, 24
uncomputable functions, 2
unobservable action, 9

value-passing calculus, 27

well-founded set, 120
word, 10

Summary

A Theory of Executability

with a Focus on the Expressivity of Process Calculi

Computability theory and concurrency theory are two of the fundamental research areas in theoretical computer science. Computability theory studies the computational power of computing systems in terms of functions on natural numbers. Many models of computation, e.g., Turing machines, recursive functions, lambda calculus, etc., were shown to be equivalent. Such equivalences provide evidence for the famous Church-Turing thesis, which could be phrased as: a function on the natural numbers is computable by an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

Computability theory does not address the aspect of interaction between systems. In order to study interaction between systems, concurrency theory was proposed. In concurrency theory, the behaviour of a system is mathematically represented as a labelled transition system. To specify and reason about labelled transition systems, many process calculi have been proposed. The community of concurrency theory developed a large variety of methods to measure the expressivity of process calculi. We address a natural question whether we can use the knowledge in computability theory to evaluate the expressivity of process calculi.

In this thesis, we study a theory of executability, which is the integration of computability theory and concurrency theory. It is based on Reactive Turing Machines (RTMs), concurrent variants of Turing machines. Every RTM has an associated labelled transition system. The labelled transition system semantics allows us to define that a transition system is executable when it is behaviourally equivalent to the transition system associated with an RTM. In concurrency theory, there are many behavioural equivalences, differing in the behavioural properties they preserve. From

the perspective of expressivity, it is convenient to consider behavioural equivalence as a parameter of the theory of executability. We aim for results modulo divergence-preserving branching bisimilarity, which is the finest known behavioural equivalence in van Glabbeek's spectrum of equivalences.

In Chapter 2 of the thesis, we first provide some preliminaries for the theory of executability, including the definitions and some basic lemmas about RTMs and executable behaviours. In particular, the role of divergence is discussed, since divergence plays a significant role in the theory of executability. Moreover, a framework of expressivity is proposed. The theory of executability could be applied to measure the expressivity of a process calculus in two aspects. One is the executability, that is, whether every transition system specified in the process calculus is executable modulo some behavioural equivalence; the other one is the reactive Turing powerfulness, that is, whether every executable transition system could be specified in the process calculus modulo some behavioural equivalence.

This thesis proceeds to contribute to four topics in the theory of executability.

Chapter 3 of the thesis gives some evidence for the robustness of executability theory based on RTMs. A comparison is made between RTMs and Interactive Turing Machines (ITMs). An ITM is a model of interactive computation invented by van Leeuwen and Wiedermann. It models interactive computation as a translation over infinite streams. We show that RTMs are at least as expressive as ITMs, both with respect to its semantics in terms of labelled transition systems modulo divergence-preserving branching bisimilarity and also with respect to its semantics in terms of stream translations. For ITMs, van Leeuwen and Wiedermann proposed an extension with a notion of advice, allowing them to model evolving systems such as the Internet. We show that such an extension can also be defined for RTMs.

Chapter 4 of the thesis aims to make some revision of concurrency theory in the context of executability theory. The chapter addresses the semantics of sequential composition in a calculus with intermediate termination. We identify two problems with the standard semantics, namely unbounded branching and forgetfulness. Two issues have been considered in this chapter in the presence of intermediate termination, namely, the relationship between pushdown processes and context-free processes, and the reactive Turing powerfulness of a process calculus with the nesting operator. We cannot resolve unbounded branching and forgetfulness in the classical operational semantics due to transparency. For this reason, we cannot answer the two issues above. We propose a revised semantics of the sequential composition operator in the presence of intermediate termination. In the revised semantics, transparency is eliminated. Thus, we resolve the two problems above in the revised semantics. We show that every context-free process can be simulated by a pushdown process modulo strong bisimilarity, and the process calculus with a nesting operator is reactively Turing powerful

modulo branching bisimilarity.

Chapter 5 of the thesis applies the theory of executability to evaluate the expressivity of the π-calculus. The π-calculus is a widely used process calculus, and its expressivity has been studied extensively in the literature. Most of the results about the expressivity of the π-calculus are about its relative expressivity, that is, the expressivity compared to some other calculus. In the theory of executability, we investigate its absolute expressivity, that is, whether a transition system specified in the π-calculus can be executed by an interactive computing system or not. We show that the π-calculus is reactively Turing powerful modulo divergence-preserving branching bisimilarity. It is not executable since the transition systems associated with π-calculus processes are not limited to a finite set of labels whereas an RTM only accept a finite set of action labels. We investigate the executability of π-calculus by making a compromise that restricts the transition systems associated with π-calculus processes referring to only finitely many names. We show that the π-calculus is executable in the restricted semantics modulo a divergence-insensitive variant of branching bisimilarity.

Chapter 6 of the thesis tries to adapt the theory of executability to a broader domain. In order to apply executability theory to evaluate expressivity of process calculi with infinite alphabets such as the π-calculus, mCRL2 and the value-passing calculus, we extend the theory on the dimension of the infinity of the alphabet. We first propose a notion of an infinitary RTM that allows the sets in the definition of an RTM to be countable. However, it turns out that such an extension hardly makes any sense since every countable transition system is then simply executable. Then, we make some attempts to restrict the transition relation in the RTMs to be effective or computable, which leads to two slightly better notions of executability. Finally, we introduce a theory of nominal executability by using sets with atoms in the definition of RTMs. In the theory of nominal executability, there are two requirements imposed on sets with atoms, namely, legality and hereditary orbit-finiteness. These two requirements do not restrict a set to finitely many elements and still keep the set definable by finitely many elements with finitely many orbits modulo permutation of atoms. We show that the π-calculus is nominally executable modulo a divergence-insensitive variant of branching bisimilarity, but mCRL2 is not. Hence, we have some evidence that the theory of nominal executability is a meaningful notion in the study of expressivity.

Chapter 7 concludes the thesis and proposes some future directions in the research of executability theory. Firstly, there are many models for interactive computation in the literature. Comparing them with RTMs would gain more evidence for the robustness of executability theory. Secondly, an integration of complexity theory and concurrency theory can be an interesting extension of executability theory. Thirdly, the process calculus with the revised sequential composition operator still lacks an axiomatization. Fourthly, the relationship between pushdown processes and context-

free processes is still unclear in the classical semantics. Fifthly, the reactive Turing powerfulness of many process calculi using non-regular iterators other than the nesting operator still needs to be proved. Sixthly, the extensions of RTMs could still be exploited in two dimensions, the choice of advice and the choice of infiniteness.

As a conclusion, the thesis results in some basic building blocks for a theory of executability, namely, the robustness of RTMs, the revision in concurrency theory, the evaluation of expressivity, and the extension of a theory of executability. A step has been made towards a concurrent version of the Church-Turing thesis.

Curriculum Vitae

Fei Yang was born on November 4th, 1988 in Zhenjiang, Jiangsu, China.

After finishing secondary education in 2007 at No. 2 High School of East China Normal University in Shanghai, China, he studied at Shanghai Jiao Tong University in Shanghai, China. He received his bachelor degree in computer science in 2011. Then he studied the master program in the major of computer science at the same university. He worked in the BASICS group and finished his master thesis titled Regularity Problems of Process Rewriting Systems supervised by Prof. Yuxi Fu. He got his master degree in 2014.

In the same year he started a PhD project at Eindhoven University of Technology of which the results are presented in this dissertation.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*. Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*.

Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified

Proofs. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT)*. Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data*. Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences*. Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code*. Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic*. Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems*. Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming*. Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi*. Faculty of Mathematics and Computer Science, TU/e. 2018-11