From Computability to Executability

A process-theoretic view on automata theory

From Computability to Executability

A process-theoretic view on automata theory

Paul van Tilburg

IPA Dissertation Series 2011-11
ISBN: 978-90-386-2630-7
A catalogue record is available from the Eindhoven University of Technology Library

Typeset with LaTeX (TeXLive 2009)
Cover design by Sofie van Schadewijk
Printed by Printservice Eindhoven University of Technology, The Netherlands

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 supported by the Netherlands Organisation for Scientific Research (NWO), project “Models of Computation: Automata and Processes” (nr. 612.000.630).

From Computability to Executability

A process-theoretic view on automata theory

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 27 oktober 2011 om 16.00 uur

door

Paulus Johannes Adrianus van Tilburg

geboren te Breda

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. J.C.M. Baeten

Copromotor:
dr. S.P. Luttik

Contents

Figures

Glossary

This section provides an overview of often used symbols and acronyms. Per item we give a short description and a reference to the (sub)section of its introduction.

Sorts & Variables

$\mathcal{A}$	$a, b, c, \ldots$	actions	(2.1)
$\mathcal{A}_\tau$	$a, b, c, \ldots$	actions (including unobservable)	(2.1)
$\mathcal{A}^*$	w	action sequences, words	(2.1)
$\mathcal{C}$	$c, i, o, \ldots$	channels	(2.2)
$\mathcal{I}$	i, j, k	indices	(2.2.4)
$\mathcal{D}$	$d, e, f, \ldots$	data symbols	(2.2)
$\mathcal{D}^*$	σ, δ, ζ	data symbol sequences, strings	(4.1)
$\mathcal{L}(X)$	L	languages (accepted by X)	(2.1)
	M	automata	(3.1)
$\mathcal{M}(X)$	μ, ν	multisets (over X)	(5.1)
$\mathcal{N}$	$I, N, X, \ldots$	names	(2.2)
$\mathcal{N}^*$	ξ, χ, η, ρ	name sequences	(4.2.1)
$\mathbb{N}$	m, n	natural numbers	(2.1.1)
$\mathcal{P}$	p, q	process expressions	(2.2)
	$E, E_B, E_S, \ldots$	recursive specifications	(2.2)
	R	regular expressions	(3.3)
	$\mathcal{R}$	relations	(2.1.1)
$\mathcal{S}$	$s, t, u, \ldots$	states	(2.1)
$\mathcal{T}(X)$	T	labelled transition systems (associated with X)	(2.1)

Multisets

$\emptyset$	empty multiset	(5.1)
$[\![x]\!]$	singleton multiset	(5.1)
$\mu(x)$	occurrences of x in μ	(5.1)
$x \in \mu$	*same as* $\mu(x) \geq 1$	(5.1)
$\mu \subseteq \nu$	multiset inclusion; $\mu(x) \leq \nu(x)$ for all x	(5.1)
$\mu \uplus \nu$	union of multisets; $(\mu \uplus \nu)(x) = \mu(x) + \nu(x)$	(5.1)
$\mu - \nu$	difference of multisets; $(\mu - \nu)(x) = \mu(x) - \nu(x)$	(5.1)

Actions

ε	empty word	(2.1)
τ	unobservable action	(2.1)
$\surd$	explicit termination action	(4.2.2)
$\#_a(w)$	number of occurrences of action a in word w	(2.1)
$c?d$	receive data element d over channel c	(2.2)
$c!d$	send data element d over channel c	(2.2)
$c⁉d$	communicate data element d over channel c	(2.2)

Data

ε	empty string	(4.1)
$\mathcal{D}_\perp/\mathcal{D}_*/\mathcal{D}_\Box$	stack/bag/tape symbols	(4.1/5.1/6.1)
$\perp$	stack symbol indicating the stack is empty	(4.1)
$*$	bag symbol indicating no element is removed	(5.1)
$\varnothing$	special stack/bag symbol preventing emptiness	(4.1.1)
$\Box$	tape symbol indicating the tape cell is blank	(6.1)
$\#, \vert, [\![,]\!]$	special tape marker symbols	(6.1)
$\delta, \zeta, \theta, \delta_L, \delta_R$	tape strings	(6.1)
$\ulcorner x \urcorner$	coding (of x) into a data string	(6.2.2)

Automata & Transition Systems

$\uparrow$	initial state	(2.1)
$\downarrow$	set of final states	(2.1)
$\rightarrow$	transition relation	(2.1)
$s\downarrow$	termination predicate (for state s)	(2.1)
$\xrightarrow{a}$	transition or step labelled with action a	(2.1)
$\xrightarrow{(a)}$	optional transition	(2.1)
$\xrightarrow{w}\!\!\!\!\!\rightarrow$	multiple transitions, may include unobservable transitions	(2.1)
$\longrightarrow^+$	transitive closure of $\xrightarrow{\tau}$	(2.1)
$\longrightarrow\!\!\!\!\!\rightarrow$	reflexive and transitive closure of $\xrightarrow{\tau}$; *same as* $\xrightarrow{\varepsilon}\!\!\!\!\!\rightarrow$	(2.1)
$\xrightarrow{a[d/\delta]}$	pushdown automaton transition	(4.1)
$\xrightarrow{a[d/\mu]}$	parallel pushdown automaton transition	(5.1)
$\xrightarrow{a[d/e]M}$	reactive Turing machine transition	(6.1)

Equivalences

$\approx$	language equivalence	(2.1)
$\leftrightarrow$	strong bisimilarity	(2.1.1)
$\leftrightarrow_{\not\downarrow}$	strong bisimilarity without termination	(4.2.2)
$\leftrightarrow_{\mathrm{b}}$	branching bisimilarity	(2.1.1)
$\leftrightarrow_{\mathrm{b}}^{\Delta}$	divergence-preserving branching bisimilarity	(2.1.1)
$\leftrightarrow_{\mathrm{rb}}^{\Delta}$	rooted divergence-preserving branching bisimilarity	(2.2.3)

Process Expressions

$\mathbf{0}$	deadlocked or unsuccessfully terminated process	(2.2)
$\mathbf{1}$	empty or successfully terminated process, skip	(2.2)
$a.p$	action prefix	(2.2)
$p.a$	action postfix	(3.2.1)
$p + q$	alternative composition, choice	(2.2)
$p \cdot q$	sequential composition	(2.2)
$p \parallel q$	parallel composition	(2.2)
$p \between q$	left-merge operation	(2.2)
$p \mid q$	communication merge operation	(2.2)
$\partial_c(p)$	encapsulation of communication over channel c	(2.2)
$\tau_c(p)$	abstraction of communication over channel c	(2.2)
$[\![p]\!]_c$	*same as* $\tau_c(\partial_c(p))$	(2.2)
$N \stackrel{\text{def}}{=} p$	defining equation for name N	(2.2)
$(+\,\mathbf{1})$	optional $\mathbf{1}$-summand	(2.2)
$[+\,\mathbf{1}]_C$	conditional $\mathbf{1}$-summand with condition C	(2.2)
$\sum_{i \in \mathcal{I}} p_i$	alternative composition over index set $\mathcal{I}$	(2.2.4)

Acronyms

ACP	Algebra of Communicating Processes	[BK84]
BCP	Basic Communicating Processes	[BBR09]
BPA	Basic Process Algebra	[BK84]
BPP	Basic Parallel Processes	[Chr93]
BSP	Basic Sequential Processes	[BBR09]
CCS	Calculus of Communicating Systems	[Mil80]
CSP	Communicating Sequential Processes	[Hoa85]
DFA	Deterministic finite automaton	(3.1)
EB	Termination on empty bag	(5.1)
ES	Termination on empty stack	(4.1)
FS	Termination on final state	(4.1)
FSEB	Termination on both final state and empty bag	(5.1)
FSES	Termination on both final state and empty stack	(4.1)
FSTB	Termination on both final state and transparent bag	(5.2.1)
GNF	Greibach normal form	(2.2.4)
NFA	Non-deterministic finite automaton	(3.1)
PDA	Pushdown automaton	(4.1)
PPDA	Parallel pushdown automaton	(5.1)
RTM	Reactive Turing machine	(6.1)
TSP	Theory of Sequential Processes	[BBR09]
TCP	Theory of Communicating Processes	[BBR09]

Preface

When I was taught process algebra in my Bachelor curriculum, I was struck by its elegance. Process algebra takes something that is very natural to most of us – we have all been taught mathematics and algebra in high school – and uses it to deal with processes rather than numbers. Regardless of its practical use in software verification, it has always provided me with a clear way to model systems in my mind. This goes beyond models of computer systems and encompasses any kind of system we might encounter in the real and virtual world.

It was exactly this feeling that drew me to the project "Models of Computation: Automata and Processes," which eventually became my Ph.D. research project. Its aim is to integrate automata theory – something taught to every computer science student around the world – with process theory. It provided me with a chance to study the core of process algebra and establish an "improved" theory that included the nowadays very important notion of *interaction* in a clean and systematic manner. It turned out that questions from the process-theoretic point of view were the most interesting, as automata theory mostly ignores the notion of interaction with the environment and focuses on the outcomes rather than the processes or behaviour.

The desire to establish this "improved" theory has led me to the decision to rework all publications written during the course of this project into a monograph. Although there are many unanswered questions, many gaps, and many things left to do, I hope this thesis provides a suitable overview.

Acknowledgements

The stereotype of a Ph.D. student is a student that sits alone in his/her office till late in the evening, digging through papers, trying to find answers to research questions. While there obviously was work that I had to do alone, and there was digging and trying, being a Ph.D. student was nothing like the stereotype I described above. The main difference in my experience is in the word "alone": I was surrounded by colleagues, friends, and family that contributed to a great working atmosphere with lots of social activities. For this I want to thank a lot of people, all this work would not have been possible without them.

First of all, I want to thank my supervisor, my promotor, Jos Baeten for putting me on this path years ago with his process algebra lectures and for stimulating me with big ideas and thoughts ever since. His enthusiasm for the field and also for my project

in particular never failed to motivate me. The same also holds for my daily supervisor, my co-promotor, Bas Luttik. Having been my Master's project supervisor, I knew what to expect from him, but I feel that he exceeded my expectations. He always knew exactly when I was stuck, when I had just realised it myself, and never failed to get me unstuck. Then, there was also his relentless (in a positive way!) feedback, helping me to work towards perfect, correct and clear text. I greatly admire him for being able to give detailed feedback even after the tenth iteration. His feedback has been invaluable and I hope that you, as reader, can see this shine through in this thesis. I really could not have wished for better supervisors!

Special thanks also go to my reading committee: Erik de Vink, Faron Moller, and Jan Willem Klop. They have my sincere gratitude for reading and checking my thesis and their timely response. Special thanks go to Erik de Vink for his detailed feedback which has led to quite an improvement of the thesis. I want to thank Jan Bergstra for taking place in my defence committee, and also Jan Friso Groote for being able to step in at the very last moment to complete the committee.

Because one generally only does a Ph.D. defence once, I wanted to fulfil all the usual traditions by having paranymphs. I want to thank Admar Schoonen for accepting the paranymph duty, but also for all his support throughout the years, as housemate and as friend. You cannot imagine how much talk about all kinds of worries, organisational matters and problems he had to endure. Discussions with him, being an electrical engineer rather than a computer scientist, forced me to think differently about the things I was working on. This has led to greater understanding of the field than I had previously thought possible.

My other paranymph is Alexandra Silva. Because she was one year ahead of me, she could always provide me with valuable advice about the thesis, the defence, and many other organisational issues. But besides that, she has been immensely supportive and has helped me through some rough patches. I still remember the first IPA days during which we met and it has been a series of joyous and "*gezellige*" social events since. I thank her for all the dinners with our joint IPA/CWI friends, and our great, much-needed holiday in the Algarve. Since she seems to think I know some Portuguese, let me put it in other words for her: "*Xana, agradeço-te do fundo do meu coração pela tua ajuda, apoio e amizade!*"

If there is such a notion as a spare or co-paranymph, then the honours must be given to Bram Senders. Bram has always shown great interest in my work and has always been willing to listen to and comment on my struggles, problems and writings. I much admire him for that and also for his thorough review of the thesis draft (twice!) and for providing detailed comments. He did all this outside the working hours of his freelance programming job. Surely, diversity must be one of his strengths!

I am happy that my group of friends does not solely consist of academic and technical people; some variety cannot hurt. One of my non-academic friends is Sofie van Schadewijk, who designed the beautiful cover of this thesis. I want to thank her for being willing to create this nice design. When I started to think about what the cover should look like, I had no idea it would turn out this well! It is quite an achievement given that the topic is quite abstract, and I am pretty sure Sofie underestimates her own talent.

The Spacelabs office in the Potentiaal building has always been a secondary home and working place to me. I want to thank my friends Anne Pijl and Marcel Moreaux for their presence there and their support. I'm guessing they have some idea of how an occasional cup of tea with some (small) talk in an environment that feels like home can help. Other friends related to Spacelabs, one way or another, or to Utopia whom I would like to thank for their support are: Bas Kloet & Henrieke Quarré, Christian Luijten & Marly Luijten-van Geel, Lise Pijl & Erwin Scholtens, Jacco Kwaaitaal, Jarno Ruwen & Marije Schillhorn van Veen, Lotte Oostebrink & Fons Vermeulen, Marcel de Boer & Ello Cuypers, and Sjoerd Simons. Special thanks go to Wouter Lueks, from whom I received much support via Utopia. I think we shared the same goals, issues and attitude while finishing our respective theses and were both ready to help each other out. I wish him good luck with his upcoming Ph.D. student time, I am sure he will succeed.

I also want to thank Eddie van Breukelen, Joke van Oers, and Werend Vrijlandt. We all know it is quite tough to keep in touch relatively frequently with friends from high school and I am happy we made it work.

I must not forget to thank my friend Stefan van der Linden. Without his endless supply of music set recordings and his company at parties I would never have made it through the hours and hours of writing.

Quite a large part of my Ph.D. student period was spent at my student apartment before I moved to my own home. I think that my flatmates and I had a great thing going and sometimes I still miss the good old days. For great memories and support, I want to thank my flatmates: Arwin Goharani, Coen Kuijstermans, Corné Aerts, Frank Boon & Annemiek Consten, Harm van de Ven, Pim Cramer en Ruud van Velzen. Homies4life!

A great thing about the group in which I worked was that it was a member of IPA (Institute for Programming research and Algorithmics), a Dutch national research school. And, because the group was a member, I was a member, albeit unknowingly at first. IPA provides a good way to expand your knowledge of other fields than your own but also to get to know other Ph.D. students across the country. My involvement with IPA was increased by Hugo Jonker, who talked me into becoming a member of the IPA Ph.D. council without me knowing what I was getting myself into. I wish to thank him for that. I also want to thank IPA management, Tijn Borghuis, Michel Reniers, Tim Willemse en Meivan Cheng, for the excellent organisation of the IPA days and (basic? advanced?) courses. They were enjoyable and I look back on them with fond memories, which would have not existed without the presence of my fellow Ph.D. students and friends: Alexandra Silva, Arie Middelkoop, Adam Koprowski, Behnaz Changizi, Carst Tankink, Cyntia Kop, David Costa, Frank Takes, Gijs Kant, Joost Winter, Jörg Endrullis & Rena Bakhshi, José Proença, José Pedro Magalhães, Marijn Schraagen, Mark Timmer-van der Stam, Michiel Helvensteijn, Pim Vullers, Sander Vermolen, Stephanie Kemper, Stijn de Gouw, Yangjing Wang and Young-Joo Moon.

Another highlight in my Ph.D. student time is the summer school in Bertinoro in 2010. Can anyone think of a better place to be? My thanks go to a few foreign friends I met at the summer school for their company: Andrea Cerone, Andrés

Aristizábal, Filippo Bonchi, Jérémy Dubreil, Massimo Callisto, Miguel Andrés, Mário Alvim, Romain Beauxis and Sophia Knight.

Over the years I have had many colleagues; a university is a fluid, ever changing environment. However, special to me are the people whom I have shared offices with. I started out with Michiel van Osch and Nikola Trčka. I want to thank them both for getting me started, their advice and company. I also want to thank Carst Tankink for his company and discussions during his Master's research period. The biggest part of my Ph.D. student time, however, I have spent with Helle Hansen as my officemate. Full of wise advice and always ready to pour me some nice herbal tea, which I will surely miss, I want to thank her and wish her well with the next step in her career in Nijmegen.

Other special colleagues are of course my co-authors. I want to thank Luca Aceto, Anna Ingólfsdóttir, Leonardo Vito and Tim Muller for their participation. I think our papers have turned out really well! Special thanks go to Pieter Cuijpers, who has provided a basis for many of the solutions in this thesis. It has always been inspiring to work with Pieter as he has a slightly different view at things, due to his background. This has often led to ingenious solutions. In addition, I want to thank Clemens Grabmayer for suggesting to us the term *reactive* Turing machine.

Related to my co-authorship was my work visit to Iceland. Luca Aceto arranged my visit to Reykjavík University in 2009. I want to thank him for this opportunity, but also Arnar Birgisson for making this entire trip such a joy. Coincidentally, we met briefly after the IPA days in the Netherlands before he returned to Iceland, just a few months before I planned my trip. As a result of this coincidence, I was able to get in touch with him, stay at his place for a few weeks and tour Iceland together. This is something for which I still feel indebted to him. I want to thank the friends I made in Iceland, Gylfi Þór Guðmundsson, Hanna María Þorgeirsdóttir, Ida Kramarczyk, Matteo Cimini, Páll Rúnar Þráinsson, Sigrún Ammendrup, Stefan Freyr Stefansson, Willard Þór Rafnsson, and Þórhallur Hálfdánarson, for a trip that I will never forget.

I started my work within the group of Formal Methods, in which I felt right at home. The working atmosphere within FM was very good and I enjoyed our joint lunches and FM traditions such as: the informal lunches, Sinterjos and the Christmas Tapas. I want to thank my FM colleagues for heavily contributing to this atmosphere: Bas Luttik, Erik de Vink, Erik Luit, Francien Dechesne, Harsh Beohar, Helle Hansen, Jasen Markovski, Jos Baeten, Kees Huizing, Meivan Cheng, Ronald Middelkoop, Ruurd Kuiper, Simona Orzan, Sonja Georgievska, Suzana Andova, and Wan Fokkink. Special thanks goes to Rob Nederpelt, my internship supervisor from a long time ago, who was always ready to provide me with writing advice and has reviewed some parts of the thesis.

A year ago, the Formal Methods group was merged with the Design and Analysis of Systems (DAS) group into the Formal System Analysis (FSA) group. This joining was almost only organisational in origin, as the FM group already did a lot of things together with the DAS group. I appreciate this joining, our "ver-oedering", and want to thank the fellow Ph.D. students and colleagues that I gained through this merge for their support in many ways: Frank Stappers, Hans Zantema, Jan Friso Groote, Jeroen Keiren, Maciej Gazda, Matthias Raffelsieper, MohammadReza Mousavi, Muhammad

Atif, Neda Noroozi, Rob Hoogerwoord, Sjoerd Cranen, Tim Willemse, Tineke van den Bosch, and Wieger Wesselink. Additional thanks have to go to Aad Mathijssen and Bas Ploeger; they left before the groups merged but I already considered them as my colleagues. I want to especially thank Herman Geuvers for discussion and pointers to related work with respect to Chapter 6.

Besides my colleagues and my friends (through all sorts of connections), there was always family to support me. I am very happy to have received support from so many relatives; they were always ready to help or ask how things were progressing. I want to thank my grandparents, Jan & Nettie, and aunts, uncles, and cousins: Loes & Emiel, Daan, Freek, Margo & Theo, Thijs, Jorieke, Stef, Rob, Bram, Merel, Stijn, and Jan & Jo, for their support.

Finally, the greatest support and love came from my close family. I do not think there is a better set of parents I can wish for. Martin & Ria have always let me find my own way, and I ended up here, finishing my Ph.D. project. The challenge has been overcome with their help and support, and for this I can hope I make them proud in return. I have always felt that, when I got stuck, my mother was ready to jump in and do the work for me if she could. I am also happy to finish my Ph.D. time in the same year as my brother Tom and his girlfriend Jorine who have deserved their Master's degree. They have struggled through a long period of writing, which I can relate to, and I think they can be proud of the result. For all three of us, a new life starts. I wish them much success and luck in the future!

I cannot but hope that I did not forget anyone. (If so, I am very sorry.) I also hope that the long list of people above shows that getting a Ph.D. degree is more than just doing research (alone). I am ever grateful for the opportunities, the knowledge, the experiences, the personal development, and social contacts that I have gained in the process.

Paul van Tilburg
Eindhoven, August 2011

Chapter 1

Introduction

The foundations of computer science were laid in the 1930s, when *computability theory* emerged as the theory that studies which functions are computable. At the core of the computability theory is the theory of automata and formal languages, which provides models of computing agents and means to reason about them. Here we mean by *computing* the application of a deterministic algorithm that transforms input into output. With the advent of the first computer terminal in the 1970s, the uprise of inter-computer networks and multi-processor systems, and the recent introduction of multi-core processors, the notion of *interaction* has become increasingly more important. Concurrency theory, split off from the classical automata theory a few decades ago, provides models of computation similar to the models given by the theory of automata and formal languages, but focuses on concurrent, reactive and interactive systems. Using this theory we can obtain a notion of *executability* on top of computability by additionally considering interaction.

In this thesis we will investigate the integration of the two theories – automata and concurrency theory – by taking prominent results from the field of automata theory and considering them from a process-theoretic perspective. We first discuss the background of both theories in this chapter. Then, we will consider the most prominent similarities and differences between the two theories and indicate what we adopt as leading research questions. Finally, an outline of the contents of the thesis is given per chapter.

1.1 Automata & Formal Language Theory

Automata theory is the study of abstract "mathematical" machines and the computational problems that can be solved using these machines. The theory has its origins in the 1930s, when Turing defined a logical machine to define computable numbers in [Tur37]. This and other models of computation, such as Kleene's *recursive functions* [Kle36] and Church's *λ-calculus* [Chu36], lead to the emergence of computability theory, the branch of mathematical logic that studies the theory of *effectively calculable* (partial) functions. Interestingly, all these models turned out to be equivalent: every effectively calculable function is computable with a Turing

machine, a Kleene recursive function and is λ-definable. This can be considered as evidence for the *Church-Turing thesis* stating that any function that can be computed at all, now and in the future, with any real-world computing device, can be computed with a Turing machine.

Turing's logical machine had a finite number of states, capturing a program, and a tape memory used during execution. Later, several definitions of various kinds of automata were defined by the mathematicians Von Neumann [Neu56] and Kleene [Kle56] to describe neural nets by means of a formal system. These results were based on the neurophysiology research pioneered by McCullough and Pitts [MP43]. The mathematical definitions of automata resulted in the link with *formal language theory*: the study of the purely syntactic aspects of (formal) languages. The first formal language is considered to have been defined by Frege in [Fre79] over one century ago. Chomsky proposed the notion of a formal *grammar* in [Cho56]. While automata provide an operational way to describe computations and languages, grammars accommodate a rather more generative approach. Correspondence results between different kinds of automata and grammars followed and are described in many textbooks on automata and formal language theory, for example see [Sud88, Sip97, Lin01, HMU06].

In [Cho56], Chomsky discerns three classes of languages, which he later extends to four in [Cho59]: regular, context-free, context-sensitive, and recursively enumerable. Taking the corresponding automata as central notion, this thesis will follow the Chomsky hierarchy and develop a process-theoretic view on each class. We will look at process-theoretic analogies of classic results for these classes from automata theory and see if they still hold. If not, we explore what extra conditions are needed to make them hold.

1.2 Concurrency Theory

Concurrency theory is the study of reactive systems, i.e., systems that depend on interaction with their environment during their execution. Petri showed in his thesis [Pet62] that concurrency and interaction may serve to bridge the gap between the theoretically convenient (Turing machine) model of a sequential machine with unbounded memory, and the practically more realistic notion of extensible architecture of components with bounded memory. Towards the end of the 1970s, Milner observed that, for a thorough investigation of concurrency and interaction, it is profitable to study these notions in isolation rather than to try and add them to any of the existing models of computation. One of his desiderata for the design of his algebraic process theories was "that there be *only a single* combinator for combining processes which interact or which coexist" [Mil93]. In particular, the interaction of a computing device with its memory is to be modelled using a symmetric notion of interaction, considering the memory as a separate process.

A large part of the research within the field of concurrency theory is devoted to process theory. In process theory, interaction between systems is treated as a first-class citizen, as it was established by e.g. [Mil80] (see also [Bae05]). It

embodies a powerful composition operator that is used to compose systems in parallel, including their interaction. A system is usually either directly modelled as a *labelled transition system*, or as an expression in a process description language with a well-defined operational semantics that associates a labelled transition system with each expression. Note that the presence of these central notions expose the relationship with automata theory, as finite transition systems and process description languages can be considered as the process-theoretic counterparts of finite automata and grammars. The process description languages, also called *process algebras*, CCS by Milner [Mil80, Mil89], ACP by Bergstra & Klop [BK84] and CSP by Hoare [Hoa85] have been the most prominent for quite some years. Nowadays also the *π-calculus*, a process algebra devised by Milner [Mil99] that can be seen to some extent as the interactive version of the λ-calculus [Chu32], has taken an important place amongst the process theories. In this thesis we use the process theory TCP_τ (Theory of Communicating Processes with τ) [BBR09], which is a generic process algebra encompassing key features of ACP, CCS and CSP.

One of the main contributions of concurrency theory is a richness of *behavioural equivalences* on labelled transition systems that to a more or lesser extent preserve the branching structure. In concurrency theory, behaviours are usually considered modulo a suitable behavioural equivalence. In this thesis we shall mainly use *(divergence-preserving) branching bisimilarity* [GW96], which is the finest behavioural equivalence in Van Glabbeek's spectrum (see [Gla93] for an overview).

1.3 Integration

The theory of automata and formal languages was developed to provide models of computing systems and to reason about them; it even turned out to provide powerful models of computation in general. The theory has been very successful and became widespread. It has many applications and appears in every academic curriculum of computer science. On the other hand, the theory deals with the computation of functions. It can no longer provide a basic model of a computer.

Nowadays, computers are systems that interact continuously not only with us but also with each other; they are non-deterministic, reactive systems. An execution performed by a computer is thus not just a series of steps of an algorithm, but it also involves interaction. It has inherent *non-determinism* and cannot be modelled as a function. Concurrency theory provides exactly this. We can see an execution as a computation plus interaction as modelled in concurrency theory. To illustrate the difference between a computation and an execution, we can say that a Turing machine cannot fly a plane, but a computer can. An automatic pilot cannot know all conditions beforehand, but rather can react to changing conditions real-time.

The goal of this thesis is to investigate the *integration* of automata and process theory, exposing the differences and similarities between them. Because concurrency theory split off from automata theory in the past, some notions are still the same. For example, the notion of a finite automaton is the same as a finite-state transition system; a linear grammar has only minor syntactic differences with a finite recursive

specification over some process algebra. We consider classical definitions and results from automata theory in a process-theoretic setting to make the integration explicit. The attempt at integration hopefully increases the understanding of both theories.

There have been results that consider classical results from a process-theoretic perspective, see for example [HS91, Gro92, BBK93, CHS95, Mol96, Srb01, Sti03]. However, no attempt has been made at *full* integration of the two theories as is done in this thesis. There have also been other attempts to add a notion of interaction to computability theory, see [LW00, GSAS04, GSW06, BGRR07]. But here, the attempts do not take full advantage of the results of concurrency theory. In all formalisations of interaction machines we could find, interaction is added as an asymmetric notion. The focus remains on the computational aspect, and interaction is included as a second-class citizen. In this thesis we want to study a theory of *executability* that treats computation and interaction on an equal footing, because we think that this will lead to a more suitable theory of behaviour of contemporary computing systems. Note that the full integration also has a practical side: the result can be incorporated into a Bachelor course, providing students with an increased understanding of concurrent, reactive systems.

The integration in this thesis includes the reinvestigation of, e.g., the correspondence between finite-state automata, regular languages, regular expressions and regular grammars, and the correspondence between pushdown automata and context-free languages (see [Sud88, Sip97, Lin01, HMU06] for details of these results). We also approach the classes of languages from a different angle and consider the class of so-called parallel pushdown systems. Parallel pushdown systems are obtained by replacing the sequential composition operator used in context-free languages by the typical operator from process theory, the parallel composition.

1.4 Similarities & Differences

As we attempt the full integration, we consider the following important differences in our approach with respect to both automata theory and process theory.

A main difference in approach with respect to automata theory is that we use the semantics of concurrency theory, labelled transition systems, as a central notion. Instead of looking at the classes of languages that are accepted by the various kinds of automata, we look at the classes of *transition systems associated with the automata*. This way, we can choose to divide out a suitable behavioural equivalence to obtain the desired results. For example, languages can still be obtained from the transition systems by dividing out language equivalence. We will see that the way the transition systems are associated with each kind of automaton provides the operational semantics of the automaton. For pushdown automata and parallel pushdown automata we shall consider different *termination conditions* such as termination on final state and termination on empty stack/bag. While the different termination conditions yield the same classes of languages, we will see that they yield different classes of associated transition systems.

A second main difference between automata theory and concurrency theory is that concurrency theory considers language equivalence to be *too coarse* to capture a notion of interaction. Looking at an automaton as a language acceptor, acceptance of a string represents a particular computation of the automaton, and the language is the set of all its computations. But, using language equivalence we abstract from moments of choice within the automaton. As a consequence, the language-theoretic interpretation is only suitable under the assumption that an automaton is a stand-alone computational device; it is unsuitable if some form of interaction of the automaton with its environment (e.g. a user, other automata running in parallel, etc.) may influence its behaviour. Concurrency theory offers other notions of behavioural equivalence. We use the most fine-grained equivalence that preserves the branching structure that the theory currently offers: *divergence-preserving branching bisimulation*. We will see that when we reconsider classical, quite straight-forward results from automata theory, e.g. the correspondence between pushdown automata and context-free grammars, may no longer hold modulo this equivalence. In this case we shall apply restrictions on languages and automata to remedy the situation. Note that in between language equivalence and divergence-preserving branching bisimulation equivalence, there are several other equivalence relations (see [Gla93]). We shall sometimes drop divergence-preservation when this is necessary.

A third difference is that a notion of final state is often missing in concurrency theory. For finite-state automata we have the notion of *intermediate termination*. This means that termination might occur at the same time that the automaton can continue with its computation/execution. Recall that concurrency theory deals with so-called *reactive systems*, which need not terminate but are always on, reacting to stimuli from the environment. As a result, intermediate termination is often neglected in concurrency theory. Using the process theory TCP_τ, which includes notation for a terminating process [BBR09], we obtain a full correspondence with automata theory: a finite-state transition system is exactly a finite automaton. Note that we still fully incorporate the reactive systems approach of concurrency theory: non-terminating behaviour is also relevant behaviour, which is taken into account by allowing for (infinite) recursion. Per kind of automata we will try to find a suitable *specification language*, the process-theoretic counterpart of grammars, and investigate the correspondence between the class of transition systems associated with the automata and the class of transition system associated with the specifications. In [Mol96], Moller presents an overview of the differences in expressive power using labelled transition systems associated with notions that we find in this thesis, such as finite-state automata, pushdown and parallel pushdown automata, several specification languages and Petri nets. We will use and extend results from this paper in the following chapters when we investigate the correspondences between automata and specification languages. We will see that the presence of a terminating process that also allows for continuation of execution makes a process theory too powerful in the sense that a specification language can express more than what can be executed by an automaton; this occurs in particular in combination with sequential composition.

A final difference between automata theory and concurrency theory is that in automata theory for pushdown automata and Turing machines the interaction between the finite-state automaton and its memory is left rather implicit. In the upcoming chapters we will model for each kind of automaton the finite-state automaton and its memory separately by means of a process description, and show that using a parallel operator that allows for communication we obtain a correspondence with the original automaton. This way we make the *interaction explicit*, thus fulfilling Milner's aforementioned desideratum that the interaction of a computing device with its memory should be modelled using a symmetric notion of interaction, modelling the memory as a separate process.

1.5 Thesis Outline

Below we give an outline of the contents of the thesis and summarise the main definitions and contributions of each chapter. Note that Chapters 3, 4, and 6 correspond to classes of the Chomsky hierarchy.

Chapter 2 recapitulates the basic definitions of labelled transitions systems and the behavioural equivalences that are relevant. We also introduce the process theory TCP_τ and several subtheories that are used throughout the thesis.

Chapter 3 discusses finite-state systems. It contains a process-theoretic view on the classical correspondence results between the four ways to describe regular languages: non-deterministic finite automata, deterministic finite automata, regular grammars and regular expressions. A side-goal of this chapter is to recapitulate central notions from automata theory, cast in our process-theoretic framework, as they will reappear in the subsequent chapters. Automata are defined as finite transition systems; regular grammars are defined as finite recursive BSP_τ-specifications called linear specifications. Because regular expressions can be defined as process expressions over TSP^*_τ, a subtheory of TCP_τ extended with the Kleene star, no casting in our process-theoretic framework is needed. However, since the regular expressions are not sufficient to describe all finite automata up to (branching) bisimilarity, we propose regular expressions extended with parallel composition, communication, and encapsulation as the process-theoretic counterparts of regular expressions.

The main definitions and theorems of this chapter are listed in the table below.

Finite-State Systems	
Finite automaton	Definition 3.1
Regular language, finite-state process	Definition 3.3
Linear specification	Definition 3.9
Regular expression	Definition 3.20
Correspondence of finite automata and linear specifications	Theorem 3.13
Correspondence of finite automata and extended regular expressions	Theorem 3.24

Chapter 4 treats pushdown systems. We give semantics to pushdown automata by means of transition systems. As in automata theory, we have to consider two different termination conditions: termination on final state and termination on empty stack. We add to these conditions termination on final state and empty stack and find that up to divergence-preserving branching bisimilarity the transition systems associated with pushdown automata fall apart into different classes. We introduce sequential specifications as the process-theoretic counterpart of context-free languages and investigate the correspondence with the pushdown automata for the different termination conditions. We show that under certain restrictions it is decidable whether two sequential specifications define the same transition system up to bisimilarity. Finally, we make the interaction within a pushdown automaton explicit by giving a finite-state process representing the finite control of the pushdown automaton and putting it in parallel with a stack process.

The main definitions and theorems of this chapter are listed in the table below.

Pushdown Systems	
Pushdown automaton	Definition 4.1
Pushdown transition system	Definition 4.4
Pushdown language, pushdown process	Definition 4.6
Sequential specification	Definition 4.17
Class distinctions for different termination conditions	Theorems 4.9 and 4.14, Examples 4.10 and 4.15
Correspondence of pushdown automata and sequential specifications	Theorems 4.31 and 4.35
Explicit interaction for pushdown automata	Theorems 4.42, 4.43, and 4.46
Decidability of bisimilarity on sequential specifications	Theorem 4.40

Chapter 5 investigates parallel pushdown systems, obtained by analogy from pushdown systems by replacing sequential composition by parallel composition. We define parallel pushdown automata, parallel pushdown transition systems and basic parallel specifications. Following the preceding chapter, we consider the distinct termination conditions for parallel pushdown automata, with termination on empty bag instead of on empty stack, which again lead to different classes of parallel pushdown transition systems. We introduce basic parallel specifications as the process-theoretic counterpart of commutative context-free grammars and investigate the relation between parallel pushdown automata and basic parallel specifications. In contrast with the previous chapter, we show that it is decidable whether two basic parallel specifications define the same transition system up to bisimilarity, without needing to apply restrictions. Finally, we make the interaction within the parallel pushdown automata explicit by giving a finite-state process representing the finite control and putting it in parallel with a bag process.

The main definitions and theorems of this chapter are listed in the following table.

Parallel Pushdown Systems	
Parallel pushdown automaton	Definition 5.1
Parallel pushdown transition system	Definition 5.4
Parallel pushdown language, parallel pushdown process	Definition 5.5
Basic parallel specification	Definition 5.16
Class distinctions for different termination conditions	Theorem 5.9, Examples 5.10, 5.13, and 5.14
Correspondence of parallel pushdown automata and basic parallel specifications	Theorems 5.29 and 5.31
Explicit interaction for parallel pushdown automata	Theorems 5.41, 5.42, 5.43, and 5.45
Decidability of bisimilarity on basic parallel specifications	Theorems 5.36 and 5.38

Chapter 6 studies computable and executable systems and the relation with effective and computable transition systems and Turing machines. For this we present the reactive Turing machine, a classical Turing machine augmented with capabilities for interaction. Classically, Turing machines are associated with recursively enumerable languages and unrestricted grammars. We define transition systems that can be simulated by a reactive Turing machine as executable transition systems, and consider TCP_τ as the process-theoretic version of unrestricted grammars. Instead of reinvestigating this correspondence we investigate the expressiveness of the notion of reactive Turing machines to see if we can still simulate computable transition systems and if it is universal with respect to executable transition systems. Again, we make the interaction within the reactive Turing machine between finite control and tape explicit.

The main definitions and theorems of this chapter are listed in the table below.

Computable & Executable Systems	
Reactive Turing machine	Definition 6.1
Effective & computable transition system	Definition 6.4
Executable process	Definition 6.6
Correspondence of effective & computable transition systems and reactive Turing machines	Theorems 6.22 and Corollary 6.23
Universality of reactive Turing machines	Theorems 6.30 and 6.31
Explicit interaction for reactive Turing machines	Theorem 6.38 and Corollary 6.39

Chapter 7 draws several conclusions and outlines future work.

Chapter 2

Preliminaries

In this chapter, we first briefly recap the basic definitions of labelled transition systems and related notions. Then, we introduce the process theory TCP_τ (Theory of Communicating Processes with τ) and several subtheories used in this thesis.

2.1 Labelled Transition Systems

From here onwards we assume the existence of a countably infinite set of *action symbols* (or just: actions) of which $\mathcal{A}$ is some finite subset. We also fix an *unobservable action* (also called silent or internal action), denoted by the symbol τ, assuming that $\tau \notin \mathcal{A}$; we denote the set $\mathcal{A} \cup \{\tau\}$ as $\mathcal{A}_\tau$. We let a, b, c range over $\mathcal{A}_\tau$.

Definition 2.1. A *labelled transition system* T is defined as a four-tuple $(\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ where:

1. $\mathcal{S}$ is a (possibly infinite) set of states,
2. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ is an $\mathcal{A}_\tau$-labelled *transition relation* on $\mathcal{S}$,
3. $\uparrow \in \mathcal{S}$ is the *initial* state,
4. $\downarrow \subseteq \mathcal{S}$ is the set of *final* states.

If $(s, a, t) \in \rightarrow$, we write $s \xrightarrow{a} t$. If s is a final state, i.e., $s \in \downarrow$, we write $s\downarrow$. △

Furthermore, we abbreviate the statement '$s \xrightarrow{a} t$ or ($a = \tau$ and $s = t$)' with $s \xrightarrow{(a)} t$. We denote the transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow^+$, and we denote the reflexive-transitive closure of $\xrightarrow{\tau}$ by $\longrightarrow\!\!\!\!\!\rightarrow$.

Definition 2.2. Let T be a labelled transition system and let s, t be states in T. We define an (input) *word* w as a sequence of actions, i.e. $w = a_1 \cdots a_n \in \mathcal{A}^*$, and let ε denote the empty word; we write $s \xrightarrow{w}\!\!\!\!\!\rightarrow t$ if there exist states $s_0, \ldots, s_n$ in T such that $s = s_0 \longrightarrow\!\!\!\!\!\rightarrow \xrightarrow{a_1} \longrightarrow\!\!\!\!\!\rightarrow s_1 \cdots \longrightarrow\!\!\!\!\!\rightarrow \xrightarrow{a_n} \longrightarrow\!\!\!\!\!\rightarrow s_n = t$.

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

We will use the notation $\#_a(w)$ to count the occurrences of some action a in word w. Note that always $\#_\tau(w) = 0$.

If we consider transition systems, we can collect all words that lead from the initial state to a final state. In automata theory, this collection is called a *language*.

DEFINITION 2.3. Let $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a transition system. The *language* $\mathcal{L}(T)$ *accepted by* T is defined as

$$\mathcal{L}(T) = \{\, w \in \mathcal{A}^* \mid \exists s \in \downarrow \text{ such that } \uparrow \xrightarrow{w}\!\!\!\!\!\rightarrow\, s \,\} \ .$$

The transition systems T_1 and T_2 are *language equivalent* (notation: $T_1 \approx T_2$) iff $\mathcal{L}(T_1) = \mathcal{L}(T_2)$. △

2.1.1 Behavioural equivalences

We first define bisimilarity, originally proposed by Park in [Par81], extended with conditions for termination. This equivalence relation treats silent transitions as ordinary transitions; it is therefore often referred to as *strong bisimilarity*.

DEFINITION 2.4. Let $T_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be transition systems. A *bisimulation* between T_1 and T_2 is a binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that $\uparrow_1 \mathcal{R} \uparrow_2$ and, for all actions $a \in \mathcal{A}_\tau$ and states s_1 and s_2, $s_1 \mathcal{R} s_2$ implies

1. if $s_1 \xrightarrow{a} s_1'$ then there exists s_2' such that $s_2 \xrightarrow{a} s_2'$ and $s_1' \mathcal{R} s_2'$,
2. if $s_2 \xrightarrow{a} s_2'$ then there exists s_1' such that $s_1 \xrightarrow{a} s_1'$ and $s_1' \mathcal{R} s_2'$,
3. if $s_1\downarrow$ then $s_2\downarrow$ and vice versa.

The transition systems T_1 and T_2 are *bisimilar* (notation: $T_1 \leftrightarroweq T_2$) if there exists a bisimulation between T_1 and T_2. △

A result from concurrency theory is that language equivalence is arguably too coarse for reactive systems, because it abstracts from all moments of choice (see, e.g., [BBR09]). In concurrency theory many alternative behavioural equivalences have been proposed; we refer to [Gla93] for a classification.

The bisimilarity behavioural equivalence might be considered too strong, as it does not abstract from silent, internal transitions. Therefore, most results of this thesis are modulo *branching bisimilarity* [GW96], which is the finest behavioural equivalence in Van Glabbeek's linear time – branching time spectrum [Gla93]. We shall consider both the divergence-insensitive and the divergence-preserving variant. By taking divergence into account, most of our results do not depend on fairness assumptions; these assumptions are needed if systems contain loops of internal transitions. (The divergence-preserving variant is called *branching bisimilarity with explicit divergence* in [Gla93, GW96], but in this thesis we prefer the term *divergence-preserving* branching bisimilarity.)

DEFINITION 2.5. Let $T_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be transition systems. A *branching bisimulation* between T_1 and T_2 is a binary relation $\mathcal{R} \subseteq \mathcal{S}_1 \times \mathcal{S}_2$ such that $\uparrow_1 \mathcal{R} \uparrow_2$ and, for all states s_1 and s_2, $s_1 \mathcal{R} s_2$ implies

1. if $s_1 \xrightarrow{a}_1 s_1'$, then there exist $s_2', s_2'' \in \mathcal{S}_2$ such that $s_2 \longrightarrow\!\!\!\!\!\rightarrow_2 s_2'' \xrightarrow{(a)}_2 s_2'$, $s_1 \mathcal{R} s_2''$ and $s_1' \mathcal{R} s_2'$;

2. if $s_2 \xrightarrow{a}_2 s_2'$, then there exist $s_1', s_1'' \in \mathcal{S}_1$ such that $s_1 \longrightarrow\!\!\!\!\rightarrow_1 s_1'' \xrightarrow{(a)}_1 s_1'$, $s_1'' \mathcal{R} s_2$ and $s_1' \mathcal{R} s_2'$;
3. if $s_1\downarrow_1$, then there exists s_2' such that $s_2 \longrightarrow\!\!\!\!\rightarrow_2 s_2'$, $s_1 \mathcal{R} s_2'$ and $s_2'\downarrow_2$; and
4. if $s_2\downarrow_2$, then there exists s_1' such that $s_1 \longrightarrow\!\!\!\!\rightarrow_1 s_1'$, $s_1' \mathcal{R} s_2$ and $s_1'\downarrow_1$.

The transition systems T_1 and T_2 are *branching bisimilar* (notation: $T_1 \leftrightarrows\!\!\!\!\!=_{\mathrm{b}} T_2$) if there exists a branching bisimulation between T_1 and T_2.

A branching bisimulation $\mathcal{R}$ between T_1 and T_2 is *divergence-preserving* if, for all states s_1 and s_2, $s_1 \mathcal{R} s_2$ implies

5. if there exists an infinite sequence $(s_{1,i})_{i\in\mathbb{N}}$ such that $s_1 = s_{1,0}$, $s_{1,i} \xrightarrow{\tau} s_{1,i+1}$ and $s_{1,i} \mathcal{R} s_2$ for all $i \in \mathbb{N}$, then there exists a state s_2' such that $s_2 \longrightarrow^+ s_2'$ and $s_{1,i} \mathcal{R} s_2'$ for some $i \in \mathbb{N}$; and
6. if there exists an infinite sequence $(s_{2,i})_{i\in\mathbb{N}}$ such that $s_2 = s_{2,0}$, $s_{2,i} \xrightarrow{\tau} s_{2,i+1}$ and $s_1 \mathcal{R} s_{2,i}$ for all $i \in \mathbb{N}$, then there exists a state s_1' such that $s_1 \longrightarrow^+ s_1'$ and $s_1' \mathcal{R} s_{2,i}$ for some $i \in \mathbb{N}$.

The transition systems T_1 and T_2 are *divergence-preserving branching bisimilar* (notation: $T_1 \leftrightarrows\!\!\!\!\!=^{\Delta}_{\mathrm{b}} T_2$) if there exists a divergence-preserving branching bisimulation between T_1 and T_2. △

It has been proved that branching bisimilarity is an equivalence relation on labelled transition systems [Bas96]; for divergence-preserving branching bisimilarity this has been shown in [GLT09].

2.1.2 Branching degree, inertness and norm

We will need as auxiliary notions the notion of *inert* τ-transition and the notion of *branching degree* of a state. For a definition we first define (divergence-preserving) branching bisimulation *on* a labelled transition system, and the *quotient* of a labelled transition system by its maximal (divergence-preserving) branching bisimulation.

Let $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a labelled transition system. A (divergence-preserving) branching bisimulation *on* T is a binary relation $\mathcal{R}$ on $\mathcal{S}$ that satisfies conditions 1–4 (conditions 1–6 in the case of divergence-preservation) of Definition 2.5. Let $\mathcal{R}$ be the maximal (divergence-preserving) branching bisimulation on T. Then $\mathcal{R}$ is an equivalence on $\mathcal{S}$; we denote by $[s]_{\mathcal{R}}$ the *equivalence class* of $s \in \mathcal{S}$ with respect to $\mathcal{R}$ and by $\mathcal{S}/\mathcal{R}$ the set of all equivalence classes of $\mathcal{S}$ with respect to $\mathcal{R}$. On $\mathcal{S}/\mathcal{R}$ we can define an $\mathcal{A}_\tau$-labelled transition relation $\rightarrow_{\mathcal{R}}$ by $[s]_{\mathcal{R}} \xrightarrow{a}_{\mathcal{R}} [t]_{\mathcal{R}}$ if, and only if, there exist $s' \in [s]_{\mathcal{R}}$ and $t' \in [t]_{\mathcal{R}}$ such that $s' \xrightarrow{a} t'$. Furthermore, we define $\uparrow_{\mathcal{R}} = [\uparrow]_{\mathcal{R}}$ and $\downarrow_{\mathcal{R}} = \{s \mid \exists s' \in \downarrow \text{ such that } s \in [s']_{\mathcal{R}}\}$. Now, the *quotient* of T by $\mathcal{R}$ is the labelled transition system $T/\mathcal{R} = (\mathcal{S}/\mathcal{R}, \rightarrow_{\mathcal{R}}, \uparrow_{\mathcal{R}}, \downarrow_{\mathcal{R}})$. It is straightforward to prove that each labelled transition system is (divergence-preserving) branching bisimilar to its quotient by its maximal (divergence-preserving) branching bisimulation.

DEFINITION 2.6. An equivalence class of transition systems with respect to divergence-preserving branching bisimilarity is called a *process*. △

EXAMPLE 2.7. The two transition systems in Figure 2.1 are divergence-preserving branching bisimilar; they are two representatives of the same process of which the left-most is the minimal form. ♢

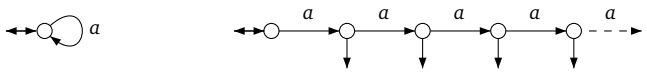

FIGURE 2.1: Two transition systems that belong to the same equivalence class with respect to divergence-preserving branching bisimilarity.

DEFINITION 2.8. Let T be a labelled transition system and let s and t be two states in T. A τ-transition $s \xrightarrow{\tau} t$ is *inert* if s and t are related by the maximal branching bisimulation on T. △

If s and t are distinct states, then an inert τ-transition $s \xrightarrow{\tau} t$ can be *eliminated* from a labelled transition system by: removing all outgoing transitions of s, changing every outgoing transition $t \xrightarrow{a} u$ from t to an outgoing transition $s \xrightarrow{a} u$ from s, changing every incoming transition $u \xrightarrow{a} t$ to t to an incoming transition $u \xrightarrow{a} s$ to s, and removing the state t. This operation yields a labelled transition system that is branching bisimilar to the original labelled transition system.

EXAMPLE 2.9. Consider the labelled transition systems in Figure 2.2. Here, the inert τ-transition from state s to t in the transition system on the left is removed by removing the transition $s \xrightarrow{a} u$ and moving all outgoing transitions of t to s, resulting in the transition system on the right. This is possible because s and t are branching bisimilar. ♢

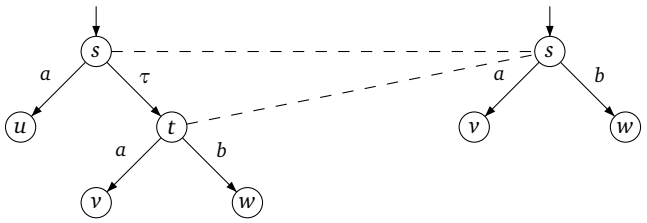

FIGURE 2.2: Removing an inert τ-transition.

To get a notion of branching degree that is preserved up to branching bisimilarity, we define the branching degree of a state as the branching degree of the corresponding equivalence class of states modulo the maximal branching bisimilarity.

DEFINITION 2.10. Let T be a labelled transition system, and let $\mathcal{R}$ be its maximal branching bisimulation. The *branching degree* of a state s in T is the cardinality of the set $\{(a, [t]_{\mathcal{R}}) \mid [s]_{\mathcal{R}} \xrightarrow{a}_{\mathcal{R}} [t]_{\mathcal{R}}\}$ of outgoing edges of the equivalence class of s in the quotient $T/\mathcal{R}$.

We say that T has *finite branching* if all states of T have a finite branching degree. We say that T has *bounded branching* if there exists a natural number $n \geq 0$ such that every state has a branching degree of at most n. △

Branching bisimulations respect branching degrees in the sense that if $\mathcal{R}$ is a branching bisimulation between T_1 and T_2, s_1 is a state in T_1 and s_2 is a state in T_2 such that $s_1 \mathcal{R} s_2$, then s_1 and s_2 have the same branching degree.

DEFINITION 2.11. Let T be a labelled transition system, and let $\mathcal{R}$ be its maximal branching bisimulation. The *norm* of a state s is the minimal number of transitions needed to reach a state that can terminate. We define it formally as follows:

$$\mathrm{norm}(s) = \inf\{\, \mathrm{length}(w) \mid w \in \mathcal{A}^* \text{ such that } s \xrightarrow{w}\!\!\!\!\rightarrow s' \wedge s'\downarrow \,\}\,.$$

Note that this means that if there is no path from state s to a state that can terminate, then $\mathrm{norm}(s) = \infty$. △

2.2 The Process Theory TCP_τ

TCP_τ is a generic process algebra encompassing key features of CSP [Hoa85], CCS [Mil80, Mil89], and ACP [BK84]: it uses prefixing and choice from CCS, parallelism from ACP (including its axiomatisation) with a generalised communication mechanism suitable to model communication over channels, and extends recursion from both CCS and ACP. With respect to the three older algebras, it additionally discerns unsuccessful termination, i.e. deadlock, and successful termination. We introduce an instance of TCP_τ with the specific form of handshaking communication from [BCLT10]. For the full definition, see [BBR09].

We use a finite set $\mathcal{C}$ of *channels* and we assume the existence of a countably infinite set of *data symbols* (or data elements) of which $\mathcal{D}$ is some finite subset; we often let c range over $\mathcal{C}$ and d, e, f range over $\mathcal{D}$. We introduce the set of special actions $\mathcal{A}' = \{\, c?d, c!d, c\mathbin{!?}d \mid d \in \mathcal{D}, c \in \mathcal{C} \,\}$; it is assumed that $\mathcal{A}' \subseteq \mathcal{A}$. Intuitively, the actions $c?d$, $c!d$, $c\mathbin{!?}d$ respectively denote the events that a data element d is received, sent, or communicated along *channel* c. Our instantiated version of TCP_τ can be seen as generic TCP_τ with a fixed, standard handshaking communication function γ, defined as follows:

$$\gamma(c!d, c?d) = c\mathbin{!?}d \qquad \text{for all } c \in \mathcal{D}, d \in \mathcal{D}.$$

This communication function is used throughout the thesis, unless a different communication function is explicitly defined. We assume the existence of a countably infinite set of *names* of which $\mathcal{N}$ is some finite subset; we often let N, but also X and Y, range over $\mathcal{N}$. In literature, names are also often called variables or non-terminals.

The set of *process expressions* $\mathcal{P}(\mathsf{TCP}_\tau)$ is generated by the following grammar ($a \in \mathcal{A}_\tau, N \in \mathcal{N}, c \in \mathcal{C}$):

$$p \;::=\; \mathbf{0} \mid \mathbf{1} \mid a.p \mid p \cdot p \mid p + p \mid p \parallel p \mid p \between p \mid p \mid p \mid \partial_c(p) \mid \tau_c(p) \mid N\,.$$

If a process expression contains no names, we say that the process expression is *closed*.

Let us briefly comment on the operators in this syntax. The constant $\mathbf{0}$ denotes *deadlock*, the unsuccessfully terminated process. The constant $\mathbf{1}$ denotes *skip*, 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 perform an a-transition to the process denoted by p. The binary operator $\cdot$ denotes *sequential composition*. The binary operator $+$ denotes *alternative composition* or *choice*. The binary operator $\|$ denotes *parallel composition*; actions of both arguments are interleaved, and in addition a communication $c?d$ of a data element d on channel c can take place if one argument can do an input action $c?d$ that matches an output action $c!d$ of the other component. The left-merge $\lfloor\!\lfloor$ and communication merge $|$ are auxiliary operators needed for the axiomatisation that we shall present later on. The unary operator $\partial_c(p)$ encapsulates the process p in such a way that all input actions $c?d$ and output actions $c!d$ are blocked (for all data) so that communication on channel c is enforced. Finally, the unary operator $\tau_c(p)$ denotes abstraction from communication over channel c in p by renaming all communications $c?d$ to τ-transitions. We shall abbreviate $\tau_c(\partial_c(p))$ with $\lceil p \rceil_c$.

We will sometimes use the notation $[+p]_C$ to indicate that the *optional summand* with process expression p is only added if condition C holds.

DEFINITION 2.12. A *recursive (TCP_τ-)specification* E is a set of equations of the form: $N \stackrel{\text{def}}{=} p$, with as left-hand side a name N and as right-hand side a (TCP_τ-)process expression p. It is required that a recursive specification E contains, for every $N \in \mathcal{N}$, at most one equation with N as left-hand side; this equation will be referred to as the *defining equation* for N in $\mathcal{N}$. Furthermore, if some name occurs in the right-hand side of some defining equation, then the recursive specification must include a defining equation for it. △

We use Structural Operational Semantics [Plo04] to associate a transition relation with process expressions: let $\rightarrow$ be the $\mathcal{A}_\tau$-labelled transition relation induced on the set of process expressions by the operational rules in Table 2.1. Note that the operational rules presuppose a recursive specification E and a termination predicate $\_\downarrow$.

DEFINITION 2.13. Let E be a recursive specification and let p be a process expression. We define the *labelled transition system* $\mathcal{T}_E(p) = (\mathcal{S}_p, \rightarrow_p, \uparrow_p, \downarrow_p)$ *associated with* p *and* E as follows:

1. the set of states $\mathcal{S}_p$ consists of all process expressions reachable from p;
2. the transition relation $\rightarrow_p$ is the restriction to $\mathcal{S}_p$ of the transition relation $\rightarrow$ defined on all process expressions by the operational rules in Table 2.1, i.e., $\rightarrow_p = \rightarrow \cap (\mathcal{S}_p \times \mathcal{A}_\tau \times \mathcal{S}_p)$.
3. the process expression p is the initial state, i.e. $\uparrow_p = p$; and
4. the set of final states consists of all process expressions $q \in \mathcal{S}_p$ such that $q\downarrow$, i.e., $\downarrow_p = \downarrow \cap \mathcal{S}_p$. △

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

$$\frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p+q \xrightarrow{a} q'} \qquad \frac{p\downarrow}{(p+q)\downarrow} \qquad \frac{q\downarrow}{(p+q)\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p\downarrow \quad q \xrightarrow{a} q'}{p \cdot q \xrightarrow{a} q'} \qquad \frac{p\downarrow \quad q\downarrow}{(p \cdot q)\downarrow}$$

$$\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \mid q \xrightarrow{c\,?\,d} p' \parallel q'} \qquad \frac{p \xrightarrow{c?d} p' \quad q \xrightarrow{c!d} q'}{p \mid q \xrightarrow{c\,?\,d} p' \parallel q'} \qquad \frac{p\downarrow \quad q\downarrow}{(p \mid q)\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \between q \xrightarrow{a} p' \parallel q} \qquad \frac{p \xrightarrow{a} p'}{p \parallel q \xrightarrow{a} p' \parallel q} \qquad \frac{q \xrightarrow{a} q'}{p \parallel q \xrightarrow{a} p \parallel q'} \qquad \frac{p\downarrow \quad q\downarrow}{(p \parallel q)\downarrow}$$

$$\frac{p \xrightarrow{c!d} p' \quad q \xrightarrow{c?d} q'}{p \parallel q \xrightarrow{c\,?\,d} p' \parallel q'} \qquad \frac{p \xrightarrow{c?d} p' \quad q \xrightarrow{c!d} q'}{p \parallel q \xrightarrow{c\,?\,d} p' \parallel q'}$$

$$\frac{p \xrightarrow{a} p' \quad a \neq c?d, c!d}{\partial_c(p) \xrightarrow{a} \partial_c(p')} \qquad \frac{p\downarrow}{\partial_c(p)\downarrow}$$

$$\frac{p \xrightarrow{c\,?\,d} p'}{\tau_c(p) \xrightarrow{\tau} \tau_c(p')} \qquad \frac{p \xrightarrow{a} p' \quad a \neq c\,?\,d}{\tau_c(p) \xrightarrow{a} \tau_c(p')} \qquad \frac{p\downarrow}{\tau_c(p)\downarrow}$$

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

TABLE 2.1: Operational rules for a recursive recursive TCP_τ-specification E and termination predicate $\_\downarrow$ ($a \in \mathcal{A}_\tau, c \in \mathcal{C}, d \in \mathcal{D}$).

Sometimes it is useful to designate an *initial name* for a recursive specification. It is then possible to associate a transition system with a recursive specification without giving the specific process expression. In other words, if I is the initial name of some recursive specification E, then its associated transition system is given by $\mathcal{T}_E(I)$.

In the other direction, if we only have a transition system associated with some recursive specification E and process expression p, it is clear we can always define a recursive specification E' obtained from E by adding initial name I with defining equation $I \stackrel{\text{def}}{=} p$.

We use the guardedness restriction, taken from [BBP94], on recursive specifications throughout the thesis.

DEFINITION 2.14. Let p be a process expression containing the name N. An occurrence of N in p is *τ-guarded* if p has a sub-expression $a.q$, where $a \in \mathcal{A}_\tau$ and q contains this occurrence of N.

We call a recursive specification E *τ-guarded* if for each defining equation $N \stackrel{\text{def}}{=} p_N$ we can obtain, by substituting p_N for N in the specification a finite number of times, the situation that p_N is τ-guarded. △

To guarantee that the specification has a unique solution, we present another restriction depending the operational semantics.

DEFINITION 2.15. A recursive specification E is *τ-founded* (or τ-convergent) if there does not exist a process expression such that $\mathcal{T}_E(p)$ has an infinite τ-path. △

We call a recursive specification *guarded* if it is both τ-guarded and τ-founded.

2.2.1 Subtheories

In the thesis we encounter several subtheories of TCP_τ. A subtheory of TCP_τ has a restricted signature and includes only the operational rules from Table 2.1 relevant for this signature to obtain the associated transition systems.

For the theory BSP_τ (Basic Sequential Processes) the set of process expressions $\mathcal{P}(\mathsf{BSP}_\tau)$ contains all process expressions without occurrences of sequential composition, parallel composition, encapsulation and abstraction; we only have deadlock, skip, prefixing and alternative composition.

For the theory TSP_τ (Theory of Sequential Processes) the set of process expressions $\mathcal{P}(\mathsf{TSP}_\tau)$ contains all the processes expressions without occurrences of parallel composition, encapsulation and abstraction; it can be obtained from $\mathcal{P}(\mathsf{BSP}_\tau)$ by adding sequential composition.

Finally, for the theory BCP_τ (Basic Communicating Processes) the set of processes expressions $\mathcal{P}(\mathsf{BCP}_\tau)$ contains all process expressions without occurrences of sequential composition, encapsulation and abstraction; it can be obtained from $\mathcal{P}(\mathsf{BSP}_\tau)$ by adding parallel composition.

2.2.2 Kleene star

To be able to have regular expressions in our process algebraic framework, we add the unary Kleene star operator ($\_^*$) for iteration to TCP_τ and obtain the theory TCP_τ^*. The Kleene star was originally defined by Kleene in [Kle56] and introduced in a process-theoretic setting by Milner in [Mil84]. (For a discussion of the binary variant of the Kleene star, see [BBP94].) The set of process expressions $\mathcal{P}(\mathsf{TCP}_\tau^*)$ is generated by the original grammar for $\mathcal{P}(\mathsf{TCP}_\tau)$ and the following rule:

$$p ::= \ldots \mid p^* .$$

To associate transition systems with TCP_τ^*-process expressions we extend the operational rules from Table 2.1 with the rules in Table 2.2.

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

TABLE 2.2: Operational rules for the unary Kleene star ($a \in \mathcal{A}_\tau$).

For the subtheory TSP_τ^* the set of process expressions $\mathcal{P}(\mathsf{TSP}_\tau)$ contains all the processes expressions mentioned above, again without occurrences of parallel composition, encapsulation and abstraction.

2.2.3 Axiomatisation

To be able to give concise proofs that certain process expressions are divergence-preserving branching bisimilar, it is convenient to proceed by equational reasoning. We shall use the equations in Table 2.3. See [BBR09] for an explanation of the axioms, and the proof rule RSP, which is based on the assumption that every guarded recursive specification has a unique solution. (Recall that the guardedness of the specifications below follows from the fact that they are τ-guarded and τ-founded, as defined in [BBP94].)

We should, of course, establish that an equational reasoning based on the axioms in Table 2.3 is sound, i.e., that it indeed proves that the equated process expressions are divergence-preserving branching bisimilar. For this it suffices to prove that the axioms in Table 2.3 and RSP are sound with respect to some congruence included in divergence-preserving branching bisimilarity. (Note that, like branching bisimilarity is not a congruence with respect to $+$, divergence-preserving branching bisimilarity is also not a congruence with respect to the operator $+$.) The way we obtain a congruence included in divergence-preserving branching bisimilarity is standard: we define a *rooted* version:

DEFINITION 2.16. A divergence-preserving branching bisimulation $\mathcal{R}$ between T_1 and T_2 is called *rooted* if it meets the following root-conditions for all $a \in \mathcal{A}_\tau$:

1. for all states $s_1' \in \mathcal{S}_1$, whenever $\uparrow_1 \xrightarrow{a} s_1'$, then there exists a state s_2' such that $\uparrow_2 \xrightarrow{a} s_2'$ and $s_1' \mathcal{R} s_2'$;
2. for all states $s_2' \in \mathcal{S}_2$, whenever $\uparrow_2 \xrightarrow{a} s_2'$, then there exists a state s_1' such that $\uparrow_1 \xrightarrow{a} s_1'$ and $s_1' \mathcal{R} s_2'$;
3. if $\uparrow_1\downarrow_1$, then $\uparrow_2\downarrow_2$;
4. if $\uparrow_2\downarrow_2$, then $\uparrow_1\downarrow_1$.

A1	$x+y=y+x$	A6	$x+\mathbf{0}=x$
A2	$(x+y)+z=x+(y+z)$	A7	$\mathbf{0}\cdot x=\mathbf{0}$
A3	$x+x=x$	A8	$x\cdot\mathbf{1}=x$
A4	$(x+y)\cdot z=x\cdot z+y\cdot z$	A9	$\mathbf{1}\cdot x=x$
A5	$(x\cdot y)\cdot z=x\cdot(y\cdot z)$	A10	$a.x\cdot y=a.(x\cdot y)$
M	$x\parallel y=x\between y+y\between x+x\mid y$	B	$a.(\tau.(x+y)+x)=a.(x+y)$
LM1	$\mathbf{0}\between x=\mathbf{0}$	SC1	$x\mid y=y\mid x$
LM2	$\mathbf{1}\between x=\mathbf{0}$	SC2	$x\parallel\mathbf{1}=x$
LM3	$a.x\between y=a.(x\parallel y)$	SC3	$\mathbf{1}\mid x+\mathbf{1}=\mathbf{1}$
LM4	$(x+y)\between z=x\between z+y\between z$	SC4	$(x\parallel y)\parallel z=x\parallel(y\parallel z)$
CM1	$\mathbf{0}\mid x=\mathbf{0}$	SC5	$(x\mid y)\mid z=x\mid(y\mid z)$
CM2	$(x+y)\mid z=x\mid z+y\mid z$	SC6	$(x\between y)\between z=x\between(y\parallel z)$
CM3	$\mathbf{1}\mid\mathbf{1}=\mathbf{1}$	SC7	$(x\mid y)\between z=x\mid(y\between z)$
CM4	$a.x\mid\mathbf{1}=\mathbf{0}$	SC8	$x\between\mathbf{0}=x\cdot\mathbf{0}$
CM5	$c!d.x\mid c?d.y=c\mathbin{?}d.(x\parallel y)$	SC9	$x\between\tau.y=x\between y$
CM6	$a.x\mid b.y=\mathbf{0}$ if $\{a,b\}\neq\{c!d,c?d\}$	SC10	$x\mid\tau.y=\mathbf{0}$
D1	$\partial_c(\mathbf{1})=\mathbf{1}$	T1	$\tau_c(\mathbf{1})=\mathbf{1}$
D2	$\partial_c(\mathbf{0})=\mathbf{0}$	T2	$\tau_c(\mathbf{0})=\mathbf{0}$
D3	$\partial_c(a.x)=\mathbf{0}$ if $a=c?d,c!d$	T3	$\tau_c(a.x)=a.\tau_c(x)$ if $a\neq c?d,c!d$
D4	$\partial_c(a.x)=a.\partial_c(x)$ if $a\neq c?d,c!d$	T4	$\tau_c(a.x)=\tau.\tau_c(x)$ if $a=c?d,c!d$
D5	$\partial_c(x+y)=\partial_c(x)+\partial_c(y)$	T5	$\tau_c(x+y)=\tau_c(x)+\tau_c(y)$

TABLE 2.3: Axioms of the process theory TCP_τ ($a\in\mathcal{A}_\tau, d\in\mathcal{D}$).

The transition systems T_1 and T_2 are *rooted divergence-preserving branching bisimilar* (notation: $T_1\leftrightarrows^{\Delta}_{\mathrm{rb}}T_2$) if there exists a divergence-preserving branching bisimulation between T_1 and T_2 that meets the above mentioned root-conditions. △

In [Trč07], Trčka introduces an equivalence, called *silent bisimulation*, that is an extension of branching bisimulation that preserves deadlock, is divergence sensitive, and incorporates successful termination. As a model he uses doubly labelled transition systems, in which also states are labelled, namely by a list of data propositions that are satisfied. He shows that silent bisimulation is not a congruence with respect to parallel composition in the language κ which is an extension of ACP_τ with data, scoping, guards and the Kleene star. A new equivalence is introduced called *stateless silent bisimulation*, which disregards the labels of the states and coincides with our definition of divergence-preserving branching bisimilarity. For this equivalence it can be proved that it is a congruence.

THEOREM 2.17. *Divergence-preserving branching bisimilarity is a congruence for all operators of* TCP_τ. □

PROOF. We can reuse the proofs for steps 2, 3, 5, 7, and 8 of [Trč07, Theorem 4.3.7] by disregarding the state labels. ■

Because divergence-preserving branching bisimilarity is included in rooted divergence-preserving branching bisimilarity, we have the following proposition that we will use in the proofs that use equational reasoning.

PROPOSITION 2.18. *The equational theory given by Table 2.3 is sound for the model of transition systems modulo divergence-preserving branching bisimilarity.* □

PROOF. First, note that, since divergence-preserving branching bisimilarity is both an equivalence and a congruence, it suffices to check the individual axioms. Second, it is well-known that the axioms are sound for branching bisimilarity. So, we only need to check the divergence-preservation conditions. Because all axioms except for B do not remove or introduce τ-transitions whatsoever, we only need to check axiom B. That axiom B is also sound for the divergence conditions (see conditions 5 and 6 of Definition 2.5) follows easily from inspection of the axiom. ■

Note that the KFAR rule [BBK87] is not a part of the axioms because it implies the removal of τ-loops which would break the divergence-preserving property.

2.2.4 Greibach normal form

In some cases it is useful to have a normal form for process expressions and recursive specifications. We will use a well-known normal form from automata theory: the Greibach normal form, introduced by Greibach in [Gre65].

DEFINITION 2.19. A process expression p is in *Greibach normal form* (GNF) if there exist a finite index set $\mathcal{I}$ such that

$$p = \sum_{i \in \mathcal{I}} a_i.\xi_i \; (+ \mathbf{1}) \,,$$

where $a_i \in \mathcal{A}_\tau$ and ξ_i is a sequence of names ($i \in \mathcal{I}$). The empty sequence denotes $\mathbf{1}$, and the empty summation denotes $\mathbf{0}$.

A recursive specification is in Greibach normal form if all right-hand sides of its defining equations are in Greibach normal form.

We call the Greibach normal form *restricted* if the sequence of names have a length of at most two. △

Classically, the GNF is used for context-free grammars, where the sequences are sequential compositions of non-terminals. In this thesis we use the GNF as a generic normal form. Based on the type of systems and the process theory that we are considering, we use a different interpretation for the sequence of names. Chapter 3 uses a GNF where the sequences of names can either be empty or consist of a single name to obtain the linear normal form. Chapter 4 interprets the sequence as a string, a sequential composition of names, to get the sequential normal form; Chapter 5 interprets the sequence as a multiset, a parallel composition of names, to obtain the basic parallel normal form.

Note that recursive specifications in GNF are automatically τ-guarded. Also note that there is a strong relation between recursive specifications in GNF and their associated transition systems. For example, consider the following recursive TSP_τ-specification in GNF:

$$X \stackrel{\text{def}}{=} a.X \cdot Y + b.\mathbf{1},$$
$$Y \stackrel{\text{def}}{=} c.\mathbf{1}.$$

Intuitively, when we consider the state associated with the name X in the transition system associated with the specification, the defining equation of X lists the possible transitions: an a-transition to a state $X \cdot Y$ and a b-transition to the state $\mathbf{1}$. Note that as a result of the GNF, each state in the associated transition system is denoted by a sequence of names.

Chapter 3

Finite-State Systems

If we consider a computer, or, in general, a computing agent that only has a fixed number of states and no memory except for what can be encoded in the fixed number of states, we call this a *finite-state system*.

In automata theory, the most prominent way used to model these systems is by the notion of the finite automaton. The finite automaton is used to represent the *finite control* of some running program or computation, i.e. the part that manipulates memory, interacts with the environment and can be described in a finite manner. In the upcoming chapters we shall investigate systems that additionally have some kind of external memory to achieve more complicated tasks. However, the finite control will always be present to manipulate the memory.

In this chapter, we present some similarities and differences between automata and process theory. We define well-known notions from automata theory in our process-theoretic setting and investigate the classical results, that are shown up to language equivalence, but now up to (divergence-preserving) branching bisimilarity.

In Section 3.1 we introduce the finite automata. We shall see that, from a process-theoretic point of view, they are actually (non-deterministic) finite labelled transition systems. Automata theory considers the classes of non-deterministic and deterministic finite automata on equal footing, since they can describe the same languages. We shall see, however, that the class of deterministic finite automata is, up to (divergence-preserving) branching bisimilarity, a proper subclass of the class of (non-deterministic) finite automata.

In Section 3.2 we investigate the classical correspondence between finite automata and regular grammars in a process-theoretic setting. Regular grammars are given in our framework as finite recursive BSP_τ-specifications, which we call linear specifications. These linear specifications, having prefixing in the language, only cover the right-linear (regular) grammars. Therefore, we also introduce linear specifications with postfixing to cover left-linear (regular) grammars. We shall see, however, that, up to (divergence-preserving) branching bisimilarity, the class of linear specifications is incomparable with the class of linear specifications with postfixing. We explore the tight correspondence between linear specifications and

finite automata for introductory purposes, as the classical correspondence result between right-linear grammars and finite automata holds even up to isomorphism.

Another prominent correspondence that comes to mind when discussing finite automata is regular expressions. A few decades ago, Milner showed in [Mil84] that up to bisimilarity not all finite automata can be given by a regular expression. In Section 3.3 we extend the language of regular expressions, which we give as closed TSP^*_τ-process expressions, with communication and obtain closed TCP^*_τ-process expressions. We show that we can give a closed TCP^*_τ-process expression that describes each finite automaton up to (divergence-preserving) branching bisimilarity.

This chapter is mainly based on the following publications:

[BCLT10] J. C. M. Baeten, P. J. L. Cuijpers, B. Luttik, and P. J. A. van Tilburg. "A Process-Theoretic Look at Automata". In: *Proceedings of FSEN 2009*. Ed. by F. Arbab and M. Sirjani. LNCS 5961. Springer, 2010, pp. 1–33.

[BLT11a] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. "Computations and Interaction". In: *Proceedings of ICDCIT 2011*. Ed. by R. Natarajan and A. Ojo. LNCS 6536. Springer, 2011, pp. 35–54.

Some material has also been adapted from the following lecture notes and publication:

[Bae11] J. C. M. Baeten. *Models of Computation: Automata and Processes*. Lecture notes 2011.

[BLMT10] J. C. M. Baeten, B. Luttik, T. Muller, and P. J. A. van Tilburg. "Expressiveness modulo Bisimilarity of Regular Expressions with Parallel Composition (extended abstract)". In: *Proceedings of EXPRESS 2010*. Ed. by S. B. Fröschle and F. D. Valencia. EPTCS 41. Open Publishing Association, 2011, pp. 1–15.

3.1 Finite Automata

In Definition 2.1 (on page 9) we have defined the notion of transition systems, the central model of process theory. The central notion of automata theory, the finite automaton, is strongly related to this model. For the finite automaton is just a transition system with a fixed, finite number of states and a finite transition relation, or: finite control.

DEFINITION 3.1. A *finite automaton M* is defined as a five-tuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ where:

1. $\mathcal{S}$ is a finite set of states,
2. $\mathcal{A}$ is a finite set of actions,
3. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{S}$ is a finite $\mathcal{A}_\tau$-labelled *transition relation* on $\mathcal{S}$,
4. $\uparrow \in \mathcal{S}$ is the *initial* state,
5. $\downarrow \subseteq \mathcal{S}$ is the set of *final* states. △

Clearly, from a finite automaton we obtain a transition system by simply omitting $\mathcal{A}$ from the five-tuple and declaring $\rightarrow$ to be an $\mathcal{A}_\tau$-labelled transition relation. In the

remainder of this paper there is no need to make the formal distinction between a finite automaton and the transition system associated with it.

EXAMPLE 3.2. Two examples of finite automata are given in Figure 3.1. The lower automata is a "cleaned up" version (with respect to bisimilarity) of the upper automata where the unreachable state y and inert τ-transitions are removed. ♢

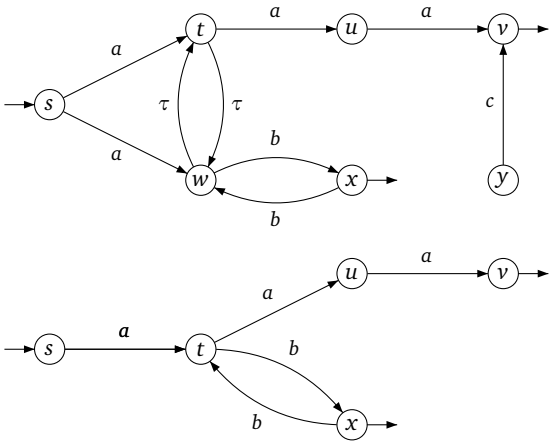

FIGURE 3.1: Two examples of finite automata.

In the theory of automata and formal languages, finite automata are considered as language acceptors. Recall that a finite automaton is a special kind of transition system, so Definition 2.3 (on page 10) applies directly to finite automata. The language of both automata in Figure 3.1 is $\{ab^{2n}aa \mid n \geq 0\} \cup \{ab^{2n-1} \mid n \geq 1\}$.

DEFINITION 3.3. A language $L \subseteq \mathcal{A}^*$ accepted by a finite automaton is called a *regular language*. △

Recall Definition 2.6 (on page 11) that defines processes as divergence-preserving branching bisimilarity equivalence classes of transition systems. If we consider finite-state systems, we are only interested in transition systems that are divergence-preserving branching bisimilar with a finite automaton.

DEFINITION 3.4. A *finite-state process* is a divergence-preserving branching bisimilarity class of transition systems that contains a finite automaton. △

Deterministic finite automata

In the upper automaton in Figure 3.1 it is not determined in which state the automaton is after performing an a-transition from the initial state. So, the notion of finite automaton defined in Definition 3.1 allows for *non-determinism*; it is actually the definition of a non-deterministic finite automaton (NFA).

However, in automata theory the deterministic finite automaton (DFA), a special case of the NFA, also plays a prominent role, for example for parsing.

DEFINITION 3.5. A finite automaton $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ is *deterministic* if, for all states $s, t_1, t_2 \in \mathcal{S}$ and for all actions $a \in \mathcal{A}$, $s \longrightarrow\!\!\!\!\!\rightarrow \xrightarrow{a} t_1$ and $s \longrightarrow\!\!\!\!\!\rightarrow \xrightarrow{a} t_2$ implies $t_1 = t_2$. △

In the theory of automata and formal languages, it is usually also required in the definition of the deterministic finite automaton that the transition relation is *total* in the sense that for all $s \in \mathcal{S}$ and for all $a \in \mathcal{A}$ there exists $t \in \mathcal{S}$ such that $s \xrightarrow{a} t$. The extra requirement is clearly only sensible in the language interpretation of automata; we shall not be concerned with it here.

The upper automaton in Figure 3.1 is non-deterministic and has an unreachable c-transition. The lower automaton is deterministic and does not have unreachable transitions; it is not total.

Up to language equivalence deterministic and non-deterministic automata accept the same languages. See e.g. [HMU06, Theorem 2.12] for a proof of the following theorem.

THEOREM 3.6. *A language L is accepted by some DFA if and only if L is accepted by some NFA.* □

This theorem does not hold if we want to have the result up to branching bisimilarity instead of language equivalence, as is illustrated by the following example.

EXAMPLE 3.7. There exists a finite automaton such that there exists no deterministic finite automaton that is branching bisimilar with it. See Figure 3.2 for such a finite automaton. ♢

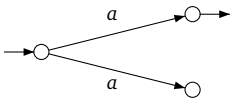

FIGURE 3.2: An example NFA that is not branching bisimilar to any DFA.

Therefore, the class of deterministic finite automata is, up to branching bisimilarity, a proper subclass of the class of finite automata. Because non-determinism is relevant and basic in process theory, we shall not particularly consider deterministic finite automata in our process-theoretic setting from here on.

In automata theory, automata can have silent transitions, usually labelled by ϵ (in [Sip97, HMU06]) or λ (in [Sud88, Lin01]). We prefer the label τ from process theory over ϵ and λ to denote silent, unobservable transitions. While many automata theory textbooks give procedures to remove τ-transitions, up to language equivalence, and this is clearly not possible up to (divergence-preserving) branching bisimilarity. Recall that only inert τ-transitions can be removed (see Definition 2.8 and Example 2.9 on page 12).

3.2 Linear Specifications

In the theory of automata and formal languages, the notion of a *grammar* is used as a syntactic mechanism to describe languages. Grammars were first proposed by Chomsky in [Cho56]. In this chapter, we consider regular grammars, i.e. left- or right-linear grammars, because we are dealing with finite-state systems and finite automata.

Recall that *linear grammar* are grammars where each right-hand side of a production rule has at most one name. A grammar is *right-linear*, when this single name is at the right end, *left-linear* when it is at the left end. We call a grammar *regular* if it is either left- or right-linear.

EXAMPLE 3.8. The following regular grammars generate the language $\{ab^{2n} \mid n \geq 0\}$.

Left-linear:

$$S \to Sbb \mid a$$

Right-linear:

$$S \to aT$$
$$T \to bbT \mid \epsilon$$

♢

The corresponding mechanism in concurrency theory is the notion of recursive specification. For the kind of grammars we are considering, we shall use the process theory BSP_τ (Basic Sequential Processes), which is a subtheory of the theory TCP_τ introduced in Section 2.2. The syntax of the process theory BSP_τ is obtained from that of TCP_τ by omitting sequential composition, parallel composition, encapsulation and abstraction.

DEFINITION 3.9. A *linear specification* over some finite set of names $\mathcal{N}$ is a finite, τ-guarded recursive BSP_τ-specification, i.e. a recursive specification over $\mathcal{N}$ in which only $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}$), $a.\_$ ($a \in \mathcal{A}_\tau$) and $\_ + \_$ are used to build *linear process expressions*. △

It turns out that getting the corresponding linear specification of a right-linear grammar is actually just a matter of changing notation. (We will consider left-linear grammars later on.)

EXAMPLE 3.10. The linear specification that corresponds to the right-linear grammar in Example 3.8 can be given as follows:

$$S \stackrel{\text{def}}{=} a.T\,,$$
$$T \stackrel{\text{def}}{=} b.b.T + \mathbf{1}\,.$$

Production rules have been replaced by defining equations, where the production symbols $\to$ have been replaced by the defined-as symbol $\stackrel{\text{def}}{=}$, non-terminals by names, terminals by prefixing operations $a.\_$, multiple rules for a name by summands of an alternative composition and the empty symbol ϵ by the empty process $\mathbf{1}$. ♢

Additionally, not shown by the example, the absence of a production rule for some non-terminal X is replaced by the equation $X \stackrel{\text{def}}{=} \mathbf{0}$.

Due to the tight relation between linear specifications and right-linear grammars, we can reuse some of the standard procedures, defined for grammars, on recursive specifications. For example, the procedure for associating a linear specification with a finite automaton is discussed next.

3.2.1 Correspondence

In automata theory the following result gives a direct correspondence between finite automata and regular grammars.

THEOREM 3.11. *A language L is regular iff there exists a regular (right-linear) grammar that generates L.* □

The classical proof for this theorem uses in one direction the fact that every regular language is accepted by some (deterministic) finite automaton and gives an algorithm to construct a grammar for this automaton (see, e.g., [Lin01, Theorem 3.4]). In the other direction it can be shown that a finite automaton can be associated with each right-linear grammar (see, e.g., [Lin01, Theorem 3.3]). The proofs hold up to isomorphism for both directions. As the correspondence between specification language and automaton will come up again in subsequent chapters, we repeat the classical proof for illustration purposes in a more process-theoretic setting.

We use the linear specifications defined above as counterparts of right-linear grammars and investigate their associated transition systems. Consider the operational rules in Table 2.1 (on page 15) that are relevant for BSP_τ, for a presupposed recursive specification E. Note that whenever p is a BSP_τ-process expression and $p \xrightarrow{a} q$ then q is again a BSP_τ-process expression. Moreover, q is a subterm of p, or q is a subterm of a right-hand side of the recursive specification E. Thus, it follows that the set of process expressions reachable from a BSP_τ-process expression consists merely of BSP_τ-process expressions, and that it is finite. So the transition system $\mathcal{T}_E(p)$ associated with a BSP_τ-process expression given a recursive BSP_τ-specification E is a finite automaton.

Below we shall also establish the converse, that every finite automaton can be specified, up to isomorphism, by a linear specification. First we illustrate the construction with an example.

EXAMPLE 3.12. Consider the automaton depicted in Figure 3.3 below.

Note that we have labelled each state of the automaton with a unique name; these will be the names of a recursive specification E. We will define each of these names with an equation, in such a way that the transition system $\mathcal{T}_E(S)$ generated by the operational semantics in Table 2.1 (on page 15) is isomorphic and hence (divergence-preserving) branching bisimilar with the automaton in Figure 3.3.

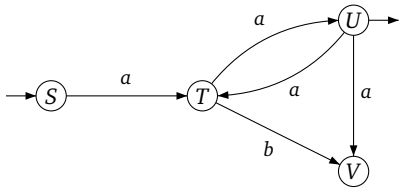

FIGURE 3.3: Another example of a finite automaton.

The recursive specification for the finite automaton in Figure 3.3 is:

$$S \stackrel{\text{def}}{=} a.T \ ,$$
$$T \stackrel{\text{def}}{=} a.U + b.V \ ,$$
$$U \stackrel{\text{def}}{=} a.T + a.V \ + \mathbf{1} \ ,$$
$$V \stackrel{\text{def}}{=} \mathbf{0} \ .$$

The action prefix $a.T$ on the right-hand side of the equation defining S is used to express that S has an a-transition to T. Alternative composition is used on the right-hand side of the defining equation for T to combine the two transitions going out from T. The $\mathbf{1}$-summand on the right-hand side of the defining equation for U indicates that U is a final state. The symbol $\mathbf{0}$ on the right-hand side of the defining equation for V expresses that V is a deadlock state. ♢

We can now give the following correspondence result between finite automata and linear specifications.

THEOREM 3.13. *For every finite automaton M there exists a linear specification E, with initial name I, such that $\mathcal{T}_E(I) \leftrightarrows_b^{\Delta} M$.* □

PROOF. The general procedure is clear from Example 3.12. Let $M = (\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ be some finite automaton. We associate with every state $s \in \mathcal{S}$ a name N_s, and define a recursive specification E on $\{N_s \mid s \in \mathcal{S}\}$ with initial name $N_\uparrow$. The recursive specification E consists of equations of the form

$$N_s \stackrel{\text{def}}{=} \sum_{(s,a,t)\in\rightarrow} a.N_t \ [+ \mathbf{1}]_{s\downarrow} \ ,$$

with the convention that the summation denotes $\mathbf{0}$ if there are no transitions from state s, and the optional $\mathbf{1}$-summand is present if, and only if, $s\downarrow$. It is easily verified that the binary relation $\mathcal{R} = \{(s, N_s) \mid s \in \mathcal{S}\}$ is a (divergence-preserving) branching bisimulation. ■

Incidentally, note that the relation $\mathcal{R}$ in the proof of the above theorem is an isomorphism, so the proof actually establishes that for every finite automaton M there

exists a recursive BSP_τ-specification E and a BSP_τ-process expression p such that the transition system associated with p and E is isomorphic to M.

Linear specifications that are constructed in the way shown in the theorem above are in the linear normal form. We instantiate the definition of the GNF (see Definition 2.19 on page 19) and restrict the sequences to a length of at most one.

DEFINITION 3.14. A linear specification E is in *linear normal form* if each defining equation of name $N \in \mathcal{N}$ is of the following form:

$$N \stackrel{\text{def}}{=} \sum_{i \in \mathcal{I}_N} a_i.N_i \; (+ \; \mathbf{1}) \, .$$

In this form, every right-hand side of every defining equation consists of a number of summands, indexed by a finite set $\mathcal{I}_N$ (the empty sum is $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $a_i.N_i$ with $a_i \in \mathcal{A}_\tau$. △

All linear specifications can be brought into linear normal form.

PROPOSITION 3.15. *For each linear specification E and linear process expression p there exists a linear specification in linear normal form E' such that $\mathcal{T}_{E'}(p) \leftrightarrows_b^\Delta \mathcal{T}_E(p)$.* □

Theorem 3.13 can be viewed as the process-theoretic counterpart of the result from the theory of automata and formal languages that states that every language accepted by a finite automaton is generated by a right-linear grammar. There is no reasonable process-theoretic counterpart of the similar result in the theory of automata and formal languages that every language accepted by a finite automaton is generated by a left-linear grammar, as we shall now explain.

Linear Specifications with Postfixing

To obtain the process-theoretic counterpart of a left-linear grammar, we should replace the action prefixes $a._-$ in BSP_τ by *action postfixes* $-.a$, with the operational rules in Table 3.1. We call this variant: *linear specifications with postfixing*.

$$\frac{p \xrightarrow{b} p'}{p.a \xrightarrow{b} p'.a} \qquad \frac{p\!\downarrow}{p.a \xrightarrow{a} \mathbf{1}}$$

TABLE 3.1: Operational rules for action postfix operators ($a, b \in \mathcal{A}_\tau$).

Analogously with linear specifications, we can define a normal form.

DEFINITION 3.16. A linear specification with postfixing E is in *reversed linear normal form* if each defining equation of name $N \in \mathcal{N}$ is of the following form:

$$N \stackrel{\text{def}}{=} \sum_{i \in \mathcal{I}_N} N_i.a_i \; (+ \mathbf{1}) \,.$$

In this form, every right-hand side of every defining equation consists of a number of summands, indexed by finite sets $\mathcal{I}_N$ (the empty sum denotes $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $N_i.a_i$ with $a_i \in \mathcal{A}_\tau$. △

Note that, if the specification contains names on the right-hand sides, it is unguarded by definition.

Analogously with linear specifications, action postfix distributes over alternative composition and is absorbed by $\mathbf{0}$. It is easy to see that the following holds.

PROPOSITION 3.17. *For each linear specification with postfixing E and process expression p there exists a linear specification with postfixing in reversed linear normal form E' such that $\mathcal{T}_{E'}(p) \leftrightarrows_b^\Delta \mathcal{T}_E(p)$.* □

Given a specification in reversed linear normal form, let p be a process expression which will be of the following form:

$$p = \sum_{i \in \mathcal{I}} N_i.w_i + \sum_{j \in \mathcal{J}} \mathbf{1}.w_j \; (+ \mathbf{1}) \,.$$

By the operation rules we have that if $p \xrightarrow{a} p'$, then $p' \leftrightarrows_{\mathrm{b}} w$ for some sequence of postfixes $w \in \mathcal{A}^*$. (This is because we need to recursively unfold the definition of each name in order to actually perform a transition.) Note that we immediately lose the name in the expression after a transition, and therefore also any form of recursion. Clearly, there exist finite automata that cannot be denoted, up to branching bisimilarity, by a process expression with this property.

EXAMPLE 3.18. Consider for example the finite automaton in Figure 3.4. A process expression denoting it cannot have the above property, for after performing an a-transition there is still a choice between terminating with a b-transition, or performing another a-transition.

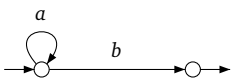

FIGURE 3.4: A finite automaton without a linear specification with postfixing.

We conclude that the automaton in Figure 3.4 cannot be described modulo branching bisimilarity in BSP_τ with action postfix instead of action prefix. ♢

Conversely, with action postfixes instead of action prefixes in the syntax, it is possible to specify transition systems that are not branching bisimilar with a finite automaton.

EXAMPLE 3.19. For instance, consider the recursive specification E over $\{X\}$ consisting of the equation

$$X \stackrel{\text{def}}{=} \mathbf{1} + X.a \ .$$

The transition system associated with X by the operational semantics is depicted in Figure 3.5. Note that in this figure, the initial state is also final.

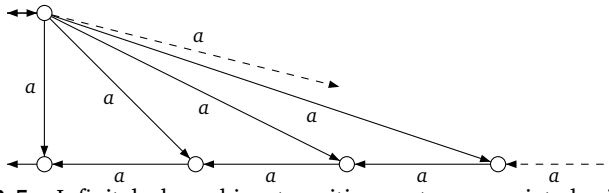

FIGURE 3.5: Infinitely branching transition system associated with an unguarded specification.

It can be proved that the infinitely many states of the depicted transition systems are all distinct modulo branching bisimilarity: each of the states in the bottom in Figure 3.5 has a different norm. It follows that the transition system associated with E is not branching bisimilar to a finite automaton. ♢

We conclude that the classes of processes defined by linear specifications and linear specifications with postfixing do not coincide.

3.3 Regular Expressions

In the previous section we have investigated the classical correspondence results between NFA, DFA and grammars in a process-theoretic setting. In automata theory there is a fourth way to describe a regular language: the regular expressions. Instead of the recursion present in regular specifications, regular expressions include the (unary) *Kleene star* $\_^*$ in their syntax, as introduced by Kleene in [Kle56] to capture repetition in regular behaviours. To obtain regular expressions in our process-theoretic setting, we use an extension of the process theory TSP_τ with the unary Kleene star called TSP^*_τ. (See Table 2.2 on page 17 for the operational rules for the unary Kleene star.)

DEFINITION 3.20. We call a closed TSP^*_τ-process expression a *regular expression*. △

From an automata and formal language point of view the **0** represents the empty language, **1** the empty word or string, and the prefix and alternative composition have their usual meaning.

EXAMPLE 3.21. The regular expression $a.(b.b.\mathbf{1})^* \cdot (a.a.\mathbf{1} + b.\mathbf{1})$ has an associated transition system that is divergence-preserving branching bisimilar with the finite automata in Figure 3.1. ♢

Kleene established in [Kle56] a correspondence between the languages denoted by regular expressions and the languages accepted by finite automata.

THEOREM 3.22. *For every DFA M, there exists a regular expression R such that* $\mathcal{L}(R) = \mathcal{L}(M)$*, and for every regular expression R there exists an NFA M such that* $\mathcal{L}(M) = \mathcal{L}(R)$.□

For the proof in one direction it is assumed that there is some DFA that accepts the language L and then a construction is given that generates a regular expression from the DFA. In the other direction, an NFA is associated with a regular expression. This NFA accepts, by definition, a regular language.

Milner, in [Mil84], showed how regular expressions can be used to describe *behaviour* by directly associating finite automata with them. He then observed that the process-theoretic counterpart of Kleene's theorem fails: there exist finite automata whose behaviours cannot faithfully, i.e., up to bisimilarity, be described by regular expressions. We show a simple example in Figure 3.6 of a finite transition system that is not bisimilar to any transition system that can be associated with a regular expression.

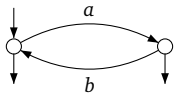

FIGURE 3.6: A finite automaton that has no regular expression up to bisimilarity.

Baeten, Corradini and Grabmayer present in [BCG07] a structural property on finite automata and shown that for the subclass adhering to this property, the so-called *well-behaved* finite automata, it is possible to find a corresponding regular expression up to bisimilarity.

3.3.1 Correspondence

If we want a full correspondence with the class of finite automata, a different approach is required. We present a solution originally published in [BLMT10] where we extend the regular expressions with well-known operators from process theory, parallel composition with communication and encapsulation, and obtain the desired correspondence result between finite automata and closed TCP^*_τ-process expressions. We shall refer to these expressions as *extended regular expressions*

Before we give the actual correspondence result, we show the construction by means of an example. The extended regular expression that we shall associate with the finite automaton will have one parallel component per state of the automaton, representing the behaviour of that state (i.e., which outgoing transitions it has to which other states and whether it is terminating). At any time, one of the parallel components corresponding with the "current state" has control. An a-transition from that current state to a next state corresponds with a communication between the two components yielding the actual a-action. Instead of using the predefined

communication function that we have defined in Section 2.2 we shall use a different, specific communication function for the purposes of this section.

EXAMPLE 3.23. Consider the finite automaton in Figure 3.7 below.

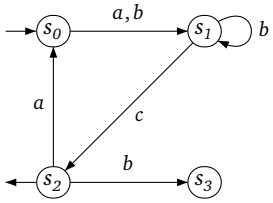

FIGURE 3.7: A finite automaton.

We associate with every state s_i an expression p_i as follows:

$$p_0 = \Big(enter_0.\mathbf{1} \cdot (leave_{a,1}.\mathbf{1} + leave_{b,1}.\mathbf{1})\Big)^* ,$$
$$p_1 = \Big(enter_1.\mathbf{1} \cdot (b.\mathbf{1})^* \cdot leave_{c,2}.\mathbf{1}\Big)^* ,$$
$$p_2 = \Big(enter_2.\mathbf{1} \cdot (leave_{a,0}.\mathbf{1} + leave_{b,3}.\mathbf{1} + \mathbf{1})\Big)^* ,$$
$$p_3 = \big(enter_3.\mathbf{1} \cdot \mathbf{0}\big)^* .$$

By executing an $enter_i$-transition a parallel component p_i can gain control, and by executing a $leave_{\alpha,j}$-transition it may then release control to p_j with action α ($\alpha \in \{a, b, c\}$) as result. Note that loops in the automaton (such as the loop on state s_1) require special treatment as they should not release control to some other state while executing the loop.

We define the communication function in such a way that an $enter_i$ action communicates with a $leave_{\alpha,i}$ action, resulting in the action α. In the case of the example, γ is defined as follows:

$$\gamma(enter_0, leave_{a,0}) = \gamma(leave_{a,0}, enter_0) = a ,$$
$$\gamma(enter_1, leave_{a,1}) = \gamma(leave_{a,1}, enter_1) = a ,$$
$$\gamma(enter_1, leave_{b,1}) = \gamma(leave_{b,1}, enter_1) = b ,$$
$$\gamma(enter_2, leave_{c,2}) = \gamma(leave_{c,2}, enter_2) = c ,$$
$$\gamma(enter_3, leave_{b,3}) = \gamma(leave_{b,3}, enter_3) = b \ ,$$

and it is left undefined otherwise.

Now, let p_0' be the resulting expression after executing the $enter_0$-transition from p_0 (thus gaining control as "current state"), i.e.,

$$p_0' = \mathbf{1} \cdot (leave_{0,1}.\mathbf{1} + leave_{1,1}.\mathbf{1}) \cdot p_0 \ .$$

We define the extended regular expression that simulates the finite automaton in Figure 3.7 as the parallel composition of p_0', p_1, p_2, and p_3, encapsulating the control actions $enter_i$ and $leave_{k,i}$, i.e.,

$$\partial_{\{enter_i, leave_{k,i} \mid 0 \le i \le 3,\ 0 \le k \le 2\}}(p_0' \parallel p_1 \parallel p_2 \parallel p_3) \ . \qquad \diamondsuit$$

Note that the process expressions that are associated with each state are even TSP^*_τ-process expressions, i.e. common regular expressions. We have just added parallelism with communication and encapsulation to obtain the correspondence.

We now present the technique illustrated in the preceding example in full generality. Let $M = (\mathcal{S}, \mathcal{A}, \rightarrow, s_0, \downarrow)$ be a finite automaton, let $\mathcal{S} = \{s_0, \ldots, s_n\}$, and let $\mathcal{A} = \{a_0, \ldots, a_m\}$ be the set of actions occurring on transitions in M. We shall associate with M an extended regular expression p_M that has one parallel component p_i for every state s_i in $\mathcal{S}$. To allow a parallel component to gain and release control, we use a collection of *control actions* $\mathcal{A}_C$ assumed to be disjoint from $\mathcal{A}$, that is defined as

$$\mathcal{A}_C = \{enter_i \mid 0 \leq i \leq n\} \cup \{leave_{k,i} \mid 0 \leq i \leq n,\ 0 \leq k \leq m\} \ .$$

Gaining and releasing control is modelled by the communication function γ on $\mathcal{A} \cup \mathcal{A}_C$ satisfying:

$$\gamma(enter_i, leave_{k,j}) = \gamma(leave_{k,j}, enter_i) = \begin{cases} a_k & \text{if } i = j; \text{ and} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For the specification of the extended regular expressions p_i we need one more definition: for $0 \leq i, j \leq n$ we denote by $\mathcal{K}_{i,j}$ the set of indices of actions occurring as the label on a transition from s_i to s_j, i.e.,

$$\mathcal{K}_{i,j} = \{k \mid s_i \xrightarrow{a_k} s_j\} \ .$$

Now we can specify the extended regular expressions p_i $(0 \leq i \leq n)$ by

$$p_i = \mathbf{1} \cdot \Big(enter_i.\mathbf{1} \cdot \big(\sum_{k \in \mathcal{K}_{i,i}} a_k.\mathbf{1} \big) \cdot \big(\sum_{\substack{1 \leq j \leq n \\ j \neq i}} \sum_{k \in \mathcal{K}_{i,j}} leave_{k,j}.\mathbf{1}\ [+\,\mathbf{1}]_{s_i\downarrow} \big) \Big)^* .$$

By $[+\,\mathbf{1}]_{s_i\downarrow}$ we mean that the summand $+\,\mathbf{1}$ is conditional; it is only included if $s_i\downarrow$. The empty summation denotes $\mathbf{0}$. (We let p_i start with $\mathbf{1}$ to yield a finite automaton associated with p_M which is isomorphic and not just bisimilar with M.)

Note that the parallel component with process expression p_i has a unique transition to gain control, i.e. $p_i \xrightarrow{enter_i} p'_i$, where p'_i denotes:

$$p'_i = \mathbf{1} \cdot \big(\sum_{k \in \mathcal{K}_{i,i}} a_k.\mathbf{1} \big)^* \cdot \big(\sum_{\substack{0 \leq j \leq n \\ j \neq i}} \sum_{k \in \mathcal{K}_{i,j}} leave_{k,j}.\mathbf{1}\ [+\,\mathbf{1}]_{s_i\downarrow} \big) \cdot p_i \ .$$

Assuming that s_0 is the initial state, we now define p_M by

$$p_M = \partial_{\mathcal{A}_C}(p'_0 \parallel p_1 \parallel \cdots \parallel p_n) \ .$$

Clearly, the construction of p_M works for every finite automaton M. The bijection defined by $s_i \mapsto \partial_{\mathcal{A}_C}(p_0 \parallel \cdots \parallel p_{i-1} \parallel p'_i \parallel p_{i+1} \parallel \cdots \parallel p_n)$ is an isomorphism between M and the automaton associated with p_M by the operational semantics (see Table 2.1 on page 15). We shall refer to p_M as the extended regular expression describing M.

THEOREM 3.24. *For every finite automaton M there exists a handshaking communication function γ and extended regular expression p_M such that the transition system associated with p_M is isomorphic with M.* □

PROOF. The bijection defined by $s_i \mapsto \partial_{A_c}(p_0 \parallel \cdots \parallel p_{i-1} \parallel p_i' \parallel p_{i+1} \parallel \cdots \parallel p_n)$ for all $1 \leq i \leq |S|$ is an isomorphism from M to the automaton associated with p_M by the operational semantics in TCP_τ^*. ■

3.4 Conclusions

In this chapter we have investigated the classical correspondence results between the four ways to describe regular languages: NFAs, DFAs, regular grammars and regular expressions. These results can be found in any automata and formal language theory book [Sud88, Sip97, HMU06]; most results are up to isomorphism, but some are up to language equivalence. See Figure 3.8 for a schematic overview.

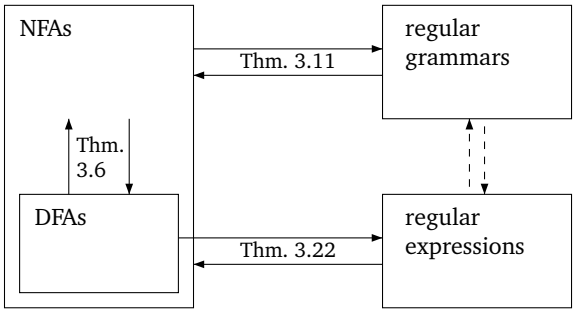

FIGURE 3.8: Classical correspondence results from automata theory.

When we considered these results from a process-theoretic perspective, we have seen that a finite automaton is a finite transition system. Up to bisimilarity the class of deterministic finite automata is smaller than the class of non-deterministic finite automata.

We have seen how regular grammars can be given as linear specifications. This, however, only covers the definition of the right-linear grammars. We can define left-linear grammars as linear specifications with postfixing if we replace the prefixing operations by postfix operations. However, we then get a different class up to branching bisimilarity. As it turns out, there is a full correspondence between finite automata and linear specifications up to branching bisimilarity.

Regular expressions are closed TSP_τ^*-process expressions. However, only the (proper) subclass of well-behaved finite automata can be expressed by regular expressions up to branching bisimilarity. We have extended the syntax of the regular expressions with operators from process theory (parallel composition and encapsulation) to obtain extended regular expressions and we have shown that there

is a full correspondence with finite automata. Interestingly, the construction only needs communication on top of usual regular expressions to work.

Figure 3.9 presents a schematic overview of the correspondence results from a process-theoretic point of view. Note that the correspondence between linear specifications and extended regular expressions is obtained indirectly via the finite automata.

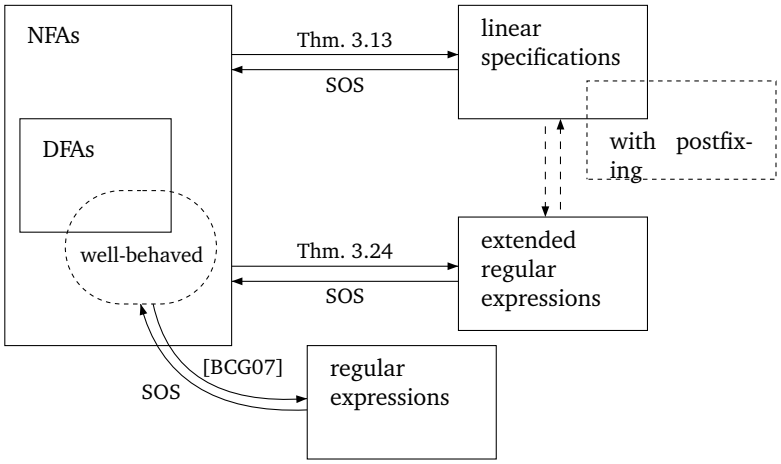

FIGURE 3.9: Correspondence results from a process-theoretic perspective.

In the following chapters we often will see that the classical correspondence results cannot be obtained up to branching bisimilarity or that notions such as context-free or unrestricted grammars do not have a direct process-theoretic version. Instead of loosening restrictions on the syntax or relinquish our strong equivalences to try to reobtain (parts of) the results, we shall extend the syntax with operators that are typically from process theory, such as parallel composition, communication, encapsulation and abstraction.

Chapter 4

Pushdown Systems

In automata and formal language theory it is common practise to characterise languages by means of a finite-state automaton, representing some finite control, which is often augmented by some kind of memory. See for example [Sud88, Sip97, HMU06]. If this memory is absent, the finite-state automata describe the class of regular languages. In case we have a tape as memory, which in fact provides random access to its data, we obtain all recursively enumerable languages. In this chapter we consider finite-state automata augmented with a limited type of memory: a stack. The combination of a finite-state automaton and a stack is called a pushdown automaton.

A classical result in automata and formal language theory is that for every context-free grammar there is a pushdown automaton that describes the same language and vice versa. However, by using this equivalence the language-theoretic approach abstracts from moments of choice and is unsuitable if some form of interaction with the automaton may influence its behaviour. In this chapter we use a process-theoretic approach and give semantics to the pushdown automata by means of associated transition systems. Using the more fine-grained divergence-preserving branching bisimulation equivalence we shall revisit some results from automata theory, amongst which the classical result mentioned previously.

In Section 4.1 we define the pushdown automaton and its associated pushdown transition system. We shall see that up to (divergence-preserving) branching bisimilarity it matters how these notions are defined. The definition of the associated pushdown transition system is given for different termination conditions: termination on final state, on empty stack, and on both final state and empty stack. While these alternative definitions lead to pushdown transition systems that describe the same languages, this is not the case up to (divergence-preserving) branching bisimilarity. We shall compare the different classes of pushdown automata and show that, up to divergence-preserving branching bisimilarity, the class of pushdown transition systems with termination on both final state and empty stack is a proper subclass of the class with termination on final state, and that the class with termination on empty stack is in turn a proper subclass of the class with termination on both final state and empty stack. For the pushdown automata that have an initial state that is also final, the class with termination on empty stack coincides, up to divergence-

branching bisimilarity, with the class with termination on both final state and empty stack.

In Section 4.2 we investigate the classical correspondence result between pushdown automata and context-free grammars in a process-theoretic setting. Context-free grammars are given as finite recursive TSP_τ-specifications, which we call sequential specifications. The choice of TSP_τ as an extension of BSP_τ is a natural one within our framework, as it adds sequential composition to our linear specifications. However, we will see in this chapter that having both sequential composition and the empty process in the specification language causes problems. We will show that only the class of pushdown automata with termination on (final state and) empty stack allows us to obtain a process-theoretic version of the classical correspondence result between pushdown automata and sequential specifications.

It turns out that transition systems associated with sequential specifications can have an unbounded branching degree. We conjecture that in this case there is no correspondence, up to branching bisimilarity, with (associated transition systems of) pushdown automata. We shall therefore propose a restriction on the sequential specifications to get a correspondence with the pushdown automata. As a result, we will discover that these restricted sequential specifications have a correspondence with just a subclass of the pushdown automata, the so-called pop choice-free automata. We will henceforth show that for this subclass there is also a correspondence in the other direction, i.e. with the restricted sequential specifications.

Next, we will investigate the decidability of bisimilarity on processes defined by sequential specifications. We obtain our result by extending earlier results for recursive BPA- and $\mathsf{BPA}_\mathbf{0}$-specifications, which are specifications in subtheories of TSP_τ. It is well-known that it is undecidable whether two context-free grammars generate the same language up to language equivalence. We prove that bisimilarity is decidable on the subclass of transition systems definable by the earlier mentioned restricted sequential specifications, a class that properly includes the $\mathsf{BPA}_\mathbf{0}$-definable transition systems.

In Section 4.3 we define the pushdown automata terminating on (final state and) empty stack by giving a finite recursive TCP_τ-specification consisting of a linear specification representing the finite control and a specification of a stack process. The stack itself is defined by a (restricted) sequential specification and may therefore be considered as the canonical process for this class of specifications. We show that, when these specifications are put in parallel, the associated transition system is divergence-preserving branching bisimilar with the transition system associated with the pushdown automaton. This way we make the communication between the finite control and the stack within a pushdown automaton explicit.

We cannot obtain the same result for pushdown automata terminating on final state using the solution above. We will show that the stack process mentioned previously cannot be reused in this setting if we want to have the result up to (divergence-preserving) branching bisimilarity. So, for this to work, we would need a stack process that can terminate regardless of its contents. We will therefore introduce a new stack process: the always-terminating stack. When we put this new stack process in parallel with the earlier specification of finite control, we can show

that the associated transition system is divergence-preserving branching bisimilar with the transition system associated with the pushdown automaton terminating on final state.

This chapter is mainly based on the following publications:

[BCLT10] J. C. M. Baeten, P. J. L. Cuijpers, B. Luttik, and P. J. A. van Tilburg. "A Process-Theoretic Look at Automata". In: *Proceedings of FSEN 2009*. Ed. by F. Arbab and M. Sirjani. LNCS 5961. Springer, 2010, pp. 1–33.

[BLT11a] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. "Computations and Interaction". In: *Proceedings of ICDCIT 2011*. Ed. by R. Natarajan and A. Ojo. LNCS 6536. Springer, 2011, pp. 35–54.

Some material is also inspired on or adapted from the following lecture notes and publication:

[Bae11] J. C. M. Baeten. *Models of Computation: Automata and Processes*. Lecture notes 2011.

[BCT08] J. C. M. Baeten, P. J. L. Cuijpers, and P. J. A. van Tilburg. "A Context-Free Process as a Pushdown Automaton". In: *Proceedings of CONCUR 2008*. Ed. by F. van Breugel and M. Chechik. LNCS 5201. Springer, 2008, pp. 98–113.

4.1 Pushdown Automata

As an intermediate notion between finite automata and Turing machines, the theory of automata and formal languages treats pushdown automata, which are finite automata extended with a stack as memory. Several definitions of the notion appear in the literature [Sud88, Sip97, HMU06], which are all equivalent in the sense that different kinds of pushdown automata still accept the same (class of) languages.

Recall the definition of a finite set of actions $\mathcal{A}$ and a finite set of data elements $\mathcal{D}$. We add to $\mathcal{D}$ the special symbol $\bot$ to indicate that a stack is empty, assuming that $\bot \notin \mathcal{D}$; we denote the set $\mathcal{D} \cup \{\bot\}$ of *stack symbols* by $\mathcal{D}_\bot$. We denote sequences of data symbols (or strings) by $\mathcal{D}^*$ and sequences of stack symbols by $\mathcal{D}_\bot^*$; we often use δ and ζ to range over $\mathcal{D}^*$ or $\mathcal{D}_\bot^*$ and ε to denote the empty string.

DEFINITION 4.1. A *pushdown automaton* (PDA) M is defined as a six-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \to, \uparrow, \downarrow)$ where:

1. $\mathcal{S}$ is a finite set of states;
2. $\mathcal{A}$ a finite set of actions;
3. $\mathcal{D}$ a finite set of data;
4. $\to \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{D}_\bot \times \mathcal{D}^* \times \mathcal{S}$ is an $\mathcal{A}_\tau \times \mathcal{D}_\bot \times \mathcal{D}^*$-labelled transition relation on $\mathcal{S}$,
5. $\uparrow \in \mathcal{S}$ is the initial state, and
6. $\downarrow \subseteq \mathcal{S}$ is the set of final states. △

If $(s, a, d, \delta, t) \in \to$, we write $s \xrightarrow{a[d/\delta]} t$. The intuitive meaning of such a transition is that if the pushdown automaton M is in state s and data element d is on the top

of the stack, then it can pop d while performing the action a, pushing the string of data elements δ on top of the stack and moving to state t. In the case that $d = \bot$, the meaning of the transition $s \xrightarrow{a[\bot/\delta]} t$ is an *empty-test* such that when the pushdown automaton M is in state s and the stack is empty, the action a can be performed, the string of data elements δ is pushed onto the stack and the automaton moves to state t. Transitions of the form $s \xrightarrow{\tau[d/\delta]} t$ and $s \xrightarrow{\tau[\bot/\delta]} s$ are silent/unobservable transitions of the pushdown automaton that just modify the stack contents.

When considering a pushdown automaton as a language acceptor, it is generally assumed that it starts in its initial state with an empty stack. (Actually, the definition of a PDA in [HMU06, Section 6.1.2] starts in the initial state with a fresh special stack empty symbol Z_0 on the stack which must be removed before terminating. As this removal action will always introduce a choice, by definition of this PDA it is not allowed to put the symbol Z_0 back, it is undesirable from a process-theoretic point of view. Hence, we have deemed it necessary to introduce the empty-test transition.) A computation consists of repeatedly consuming input symbols (or just modifying stack contents without consuming input symbols). When it comes to determining whether or not to accept an input word there are two approaches: "acceptance by final state" (FS) and "acceptance by empty stack" (ES). The first approach accepts a word if the pushdown automaton can move to a configuration with a final state by consuming the word, ignoring the contents of the stack in this configuration. The second approach accepts the word if the pushdown automaton can move to a configuration with an empty stack, ignoring whether the state of this configuration is final or not. Both approaches are equally powerful from a language-theoretic point of view, but not from a process-theoretic point of view, as we shall see below. We shall also consider a third approach in which a configuration is terminating if it consists of a terminating state *and* an empty stack (FSES). We will see in Section 4.1.1 that, from a process-theoretic point of view, the FS, FSES and ES approaches all lead to different notions of transition systems up to (divergence-preserving) branching bisimilarity.

EXAMPLE 4.2. Assume that $\mathcal{A} = \{a, b\}$ and $\mathcal{D} = \{1\}$. The state-transition diagram in Figure 4.1 specifies a pushdown automaton that first can perform a series of a-actions while stacking the data element 1 for each a-action in the state s. Then, it can switch to state t by performing a b-action and removing a data element 1 from the stack followed by performing as many b-actions as there are data elements 1 on the stack.

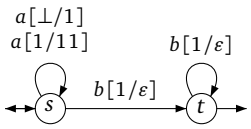

FIGURE 4.1: An example of a pushdown automaton.

Depending on the adopted acceptance condition, the pushdown automaton in Figure 4.1 accepts either the language $\{a^n b^m \mid n \geq m \geq 0\}$ (FS) or the language $\{a^n b^n \mid n \geq 0\}$ (FSES or ES). ♢

To formalise the intuitive behaviour of pushdown automata, we associate with every PDA M a transition system $\mathcal{T}(M)$. For the states of this associated transition system we use configurations defined as follows.

DEFINITION 4.3. A *configuration* of a pushdown automaton M is a pair (s, δ) consisting of a state $s \in \mathcal{S}$, and stack contents $\delta \in \mathcal{D}^*$. The left-most data element of δ represents the top of the stack. △

The associated transition system semantics of PDAs defines an $\mathcal{A}_\tau$-labelled transition relation on configurations such that a PDA-transition $s \xrightarrow{a[d/\delta]} t$ corresponds with an a-labelled transition from a configuration consisting of the PDA-state s and stack contents $d\zeta$. The transition leads to a configuration consisting of the PDA-state t and the stack contents $\delta\zeta$, i.e. the original stack contents with the top element d replaced by δ. A PDA-transition $s \xrightarrow{a[\bot/\delta]} t$ corresponds with an a-labelled transition from a configuration consisting of the PDA-state s and an empty stack, leading to a configuration of the PDA-state t and the stack contents δ.

DEFINITION 4.4. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton. The *transition system* $\mathcal{T}(M)$ *associated with* M is defined as follows:

1. the set of states of $\mathcal{T}(M)$ is the set of configurations $\mathcal{S} \times \mathcal{D}^*$;
2. the transition relation of $\mathcal{T}(M)$ satisfies
   a) $(s, d\zeta) \xrightarrow{a} (t, \delta\zeta)$ iff $s \xrightarrow{a[d/\delta]} t$ for all $s, t \in \mathcal{S}$, $a \in \mathcal{A}_\tau$, $d \in \mathcal{D}$, $\delta, \zeta \in \mathcal{D}^*$, and
   b) $(s, \varepsilon) \xrightarrow{a} (t, \delta)$ iff $s \xrightarrow{a[\bot/\delta]} t$;
3. the initial state of $\mathcal{T}(M)$ is $(\uparrow, \varepsilon)$; and
4. for the set of final states $\downarrow$ we consider three alternative *termination condition*:
   a) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ (the FS interpretation),
   b) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $\zeta = \varepsilon$ (the ES interpretation), and
   c) $(s, \zeta)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ and $\zeta = \varepsilon$ (the FSES interpretation).

A transition system is a *pushdown transition system (according to the FS/ES/FSES interpretation)* if it is associated with a PDA. △

EXAMPLE 4.5. Recall the example PDA in Figure 4.1. The transition system associated with this PDA (according to the ES or FSES interpretation) is shown in Figure 4.2. ♢

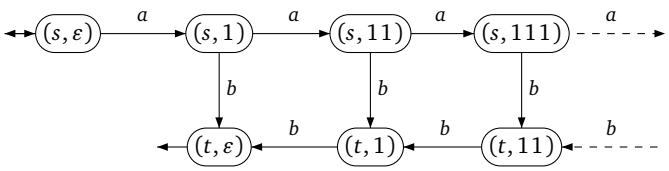

FIGURE 4.2: The transition system associated with the example PDA according to the (FS)ES interpretation.

This definition now gives us the notions of pushdown language and pushdown process (according to the FS/FSES/ES interpretation).

DEFINITION 4.6. A language accepted by a pushdown transition system is called a *pushdown language*.

A *pushdown process (according to the FS/FSES/ES interpretation)* is a divergence-preserving branching bisimilarity class of labelled transition systems containing a pushdown transition system (according to the same interpretation). △

Note that the pushdown languages coincide, up to language equivalence, with the context-free languages.

It is technically convenient to assume that the transitions of a pushdown automaton are composed of two types that perform only a single operation on the stack: either a push or a pop.

DEFINITION 4.7. Let $s,t \in \mathcal{S}$ of some pushdown automaton M. A *push transition* is a transition of the form $s \xrightarrow{a[\bot/d]} t$ or $s \xrightarrow{a[d/ed]} t$ $(d,e \in \mathcal{D})$; a *pop transition* is a transition of the form $s \xrightarrow{a[\bot/\varepsilon]} t$ (the empty-test) or $s \xrightarrow{a[d/\varepsilon]} t$ $(d \in \mathcal{D})$. △

THEOREM 4.8. *For every PDA M there exists a PDA M' that uses only push and pop transitions such that $\mathcal{T}(M) \leftrightarrows_b^{\Delta} \mathcal{T}(M')$.* □

PROOF. It is easy to see that only allowing push and pop transitions in the definition of pushdown automaton yields the same notion of pushdown transition system up to divergence-preserving branching bisimilarity:

1. Eliminate a transition of the form $s \xrightarrow{a[\bot/\delta]} t$, with $\delta = d_n \cdots d_1$ $(n \geq 2)$, by adding fresh states $s_2, \ldots, s_n$ and replacing the transition $s \xrightarrow{a[\bot/\delta]} t$ by transitions

$$s \xrightarrow{a[\bot/d_1]} s_2 \xrightarrow{\tau[d_1/d_2d_1]} \cdots \xrightarrow{\tau[d_{n-2}/d_{n-1}d_{n-2}]} s_n \xrightarrow{\tau[d_{n-1}/d_nd_{n-1}]} t \ .$$

2. Eliminate a transition of the form $s \xrightarrow{a[d/\delta]} t$, with $\delta = d_n \cdots d_1$ $(n \geq 1)$, by adding fresh states $s_1, \ldots, s_n$ and replacing the transition $s \xrightarrow{a[d/\delta]} t$ by transitions $s \xrightarrow{a[d/\varepsilon]} s_1$, $s_1 \xrightarrow{\tau[\bot/d_1]} s_2$ and $s_1 \xrightarrow{\tau[e/d_1e]} s_2$ for all $e \in \mathcal{D}$, and transitions

$$s_2 \xrightarrow{\tau[d_1/d_2d_1]} \cdots \xrightarrow{\tau[d_{n-2}/d_{n-1}d_{n-2}]} s_n \xrightarrow{\tau[d_{n-1}/d_nd_{n-1}]} t \ .$$

Observe that we only get a finite number of additional inert τ-transitions in the associated transition system. ■

Curiously, the stack that is used by the pushdown automaton can be shown to be defined by a pushdown automaton itself. Given a finite set of data $\mathcal{D}$, the stack has an input channel i over which it can receive elements of $\mathcal{D}$ and an output channel o over which it can send elements of $\mathcal{D}$. If the stack is empty, the stack can send the data element $\bot$ over channel o for the purpose of an empty-test.

The stack is defined by a pushdown automaton with one state $\uparrow$ (which is both initial and final) and transitions $\uparrow \xrightarrow{o!\bot[\bot/\varepsilon]} \uparrow$, $\uparrow \xrightarrow{i?d[\bot/d]} \uparrow$, $\uparrow \xrightarrow{i?d[e/de]} \uparrow$, and

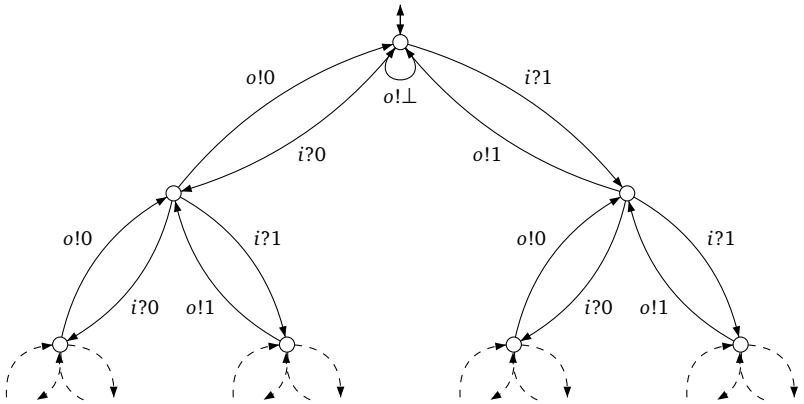

FIGURE 4.3: Stack over $\mathcal{D} = \{0, 1\}$.

$\uparrow \xrightarrow{o!d[d/\varepsilon]} \uparrow$ for all $d, e \in \mathcal{D}$. The associated transition system according to the (FS)ES interpretation of the stack over $\mathcal{D} = \{0, 1\}$ is presented in Figure 4.3.

If we want to model the stack that always terminates, i.e. that terminates regardless of its contents, we can use the PDA specified above but then consider the associated transition system according to the FS interpretation. This transition system is isomorphic with the transition system in Figure 4.3 but each state is final.

4.1.1 Termination Conditions

In the introduction we have already mentioned that from a language-theoretic point of view the different approaches to termination of pushdown automata (FS, ES, FSES) are all equivalent, but not from a process-theoretic point of view.

ES and FSES

First, we argue that the pushdown transition systems according to the ES interpretation form a proper subclass, up to divergence-preserving branching bisimulation, of the pushdown transition systems according to the FSES interpretation.

THEOREM 4.9. *For each pushdown transition system according to the ES interpretation there is, up to divergence-preserving branching bisimilarity, a pushdown transition system according to the FSES interpretation.* □

PROOF. Let T be the transition system associated with a pushdown automaton M according to the ES interpretation. Let M' be the pushdown automaton obtained from M by declaring all states to be final. Then T is also the transition system associated with M' according to the FSES interpretation. ■

When a PDA has an initial state that is also final, we call it *initially terminating*. From a language-theoretic point of view this means that the PDA accepts the empty word (ε); it is said to have the *empty word* property. All pushdown transition

systems according to the ES interpretation can terminate in the initial state, since the pushdown automaton has an empty stack in the initial state by definition. Therefore, they are all initially terminating. This is not the case for pushdown transition systems according to the FSES interpretation, hence, this constitutes a bigger class of transition systems.

EXAMPLE 4.10. Consider the pushdown automaton M in Figure 4.4, which is a modified version of the PDA in Figure 4.1 without an initial state that is also final. The initial state of the associated transition system $\mathcal{T}(M)$ according the FSES interpretation (see Figure 4.5) is not final.

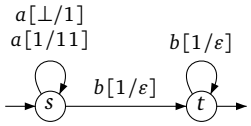

FIGURE 4.4: A pushdown automaton that is not initially terminating.

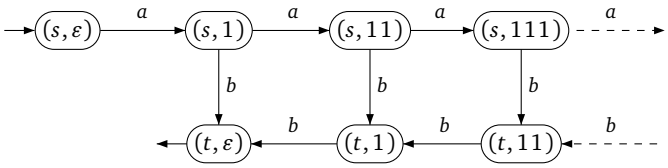

FIGURE 4.5: The transition system associated with the PDA that is not initially terminating according to the FSES interpretation.

The initial state of every pushdown transition system associated with a PDA according to the ES interpretation is always also a final state, because the stack of a PDA is empty in the initial state by definition. Therefore, there can be no pushdown transition system according to the ES interpretation that is branching bisimilar with the pushdown transition system in Figure 4.5. ♢

For pushdown automata that are initially terminating, the class of pushdown transition systems according to the FSES interpretation is the same, up to divergence-preserving branching bisimilarity, as the class according to the ES interpretation. Examples of such pushdown automata are the example PDA in Figure 4.1 and the stack PDA defined before.

We can modify the initially-terminating pushdown automata in such a way that the associated transition system according to the FSES interpretation is branching bisimilar with the transition system associated with the modified PDA according to the ES interpretation. Intuitively, if we go from FSES to ES, the termination condition gets more liberal as we drop the final state requirement. Therefore, we have to ensure that termination on empty stack is still only possible in states that are branching bisimilar to the states originally marked as final. A way to do this is by controlling where the stack becomes empty.

EXAMPLE 4.11. Let us consider the example PDA in Figure 4.6 below and the modified PDA in Figure 4.7.

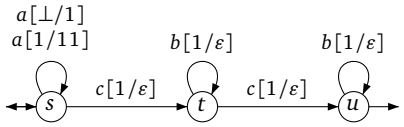

FIGURE 4.6: An example of an initially-terminating pushdown automaton.

By adding the fresh state ↑ and transition $\uparrow \xrightarrow{\tau[\bot/\varnothing]} s$ we put an extra, fresh data element $\varnothing$ on the stack, before the original initial state s, so that the stack can only become empty when we want it to. We replace all transitions performing an empty test by transitions that perform a test on whether the top data element is $\varnothing$, e.g. $s \xrightarrow{a[\varnothing/1\varnothing]} s$. Finally, we add for final states s and u in the original PDA the fresh states $s_\bot, u_\bot$ and four transitions: $s \xrightarrow{\tau[\varnothing/\varepsilon]} s_\bot$ and $u \xrightarrow{\tau[\varnothing/\varepsilon]} u_\bot$ to remove this marker when in the FSES case termination could occur, and $s_\bot \xrightarrow{\tau[\bot/\varnothing]} s$ and $u_\bot \xrightarrow{\tau[\bot/\varnothing]} u$ to put the end-of-stack marker back.

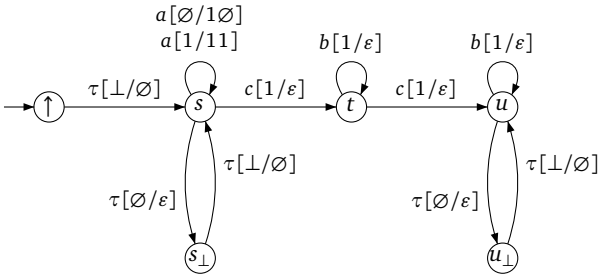

FIGURE 4.7: Modified pushdown automaton for FSES to ES.

The associated transition systems with the original PDA and the modified PDA above are branching bisimilar. However, this modification introduces divergence, as it is possible to infinitely often push and pop the end-of-stack marker. A slightly more complicated modification that preserves divergence is shown in Figure 4.8.

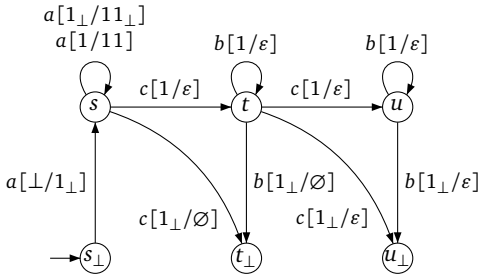

FIGURE 4.8: Modified pushdown automaton for FSES to ES preserving divergence.

For each original state we add a fresh state that encodes that the stack is empty (states $s_\perp$, $t_\perp$ and $u_\perp$). For each data element d we add a fresh data element $d_\perp$; we use these fresh data elements to keep track of when the stack is about to become empty by ensuring that the last data element on the stack is marked. Now, we replace a push transition that performs an empty test by a transition that puts a marked data element on the stack. For example, we replace $s \xrightarrow{a[\perp/1]} s$ by $s_\perp \xrightarrow{a[\perp/1_\perp]} s$. For the other push transitions we add transitions that ensure the last data element on the stack stays marked. For example, for the transition $s \xrightarrow{a[1/11]} s$ we add $s \xrightarrow{a[1_\perp/11_\perp]} s$. For each pop transition we add a transition that moves to "empty stack" counterpart of the destination state if a marked data element is popped. In the example these are the four transitions: $s \xrightarrow{c[1_\perp/\varnothing]} t_\perp$, $t \xrightarrow{b[1_\perp/\varnothing]} t_\perp$, $t \xrightarrow{c[1_\perp/\varepsilon]} u_\perp$, and $u \xrightarrow{b[1_\perp/\varepsilon]} u_\perp$. Note that the transitions that move to $t_\perp$ put the dummy data element $\varnothing$ on the stack, rather than letting it become empty. This is necessary because t is not a final state in the original PDA; only pop transitions to "stack empty" counterpart states for states that are final in the original PDA will let the stack really become empty. ♢

Not shown in the example above is that all newly introduced push transitions from a state $s_\perp$ such that $s \notin \downarrow$ should remove the dummy data element $\varnothing$.

We now show that this modification works universally up to divergence-preserving branching bisimilarity for all PDAs that are initially terminating.

THEOREM 4.12. *For each pushdown transition system according to the FSES interpretation associated with a PDA that is initially terminating there is, up to divergence-preserving branching bisimilarity, a pushdown transition system according to the ES interpretation.* □

PROOF. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be some pushdown automaton that is initially terminating. By Theorem 4.8, we can assume that M only has push and pop transitions. We shall modify M such that the transition system associated with the modified pushdown automaton according to the ES interpretation is divergence-preserving branching bisimilar with the transition system associated with M according to the FSES interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}, \mathcal{D}', \rightarrow', \uparrow, \emptyset)$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a "stack empty" state $s_\perp$ for every state $s \in \mathcal{S}$;
2. $\mathcal{D}'$ is obtained from $\mathcal{D}$ by adding a marked data element $d_\perp$ for each $d \in \mathcal{D}$ and a fresh dummy data element $\varnothing$;
3. $\rightarrow'$ is obtained from $\rightarrow$ by
   a) replacing all push transitions $(s, a, \perp, d, t) \in \rightarrow$ by either $(s_\perp, a, \perp, d_\perp, t) \in \rightarrow'$ if $s \in \downarrow$, or $(s_\perp, a, \varnothing, d_\perp, t) \in \rightarrow'$ if $s \notin \downarrow$,
   b) adding for each push transition $(s, a, d, ed, t) \in \rightarrow$ a push transition $(s, a, d_\perp, ed_\perp, t) \in \rightarrow'$,

c) replacing all pop transitions $(s,a,\bot,\varepsilon,t) \in \rightarrow$ by
$$\begin{cases} (s,a,\varnothing,\varnothing,t) \in \rightarrow' & \text{if } s \notin \downarrow,\ t \notin \downarrow, \\ (s,a,\varnothing,\varepsilon,t) \in \rightarrow' & \text{if } s \notin \downarrow,\ t \in \downarrow, \\ (s,a,\bot,\varnothing,t) \in \rightarrow' & \text{if } s \in \downarrow,\ t \notin \downarrow, \end{cases}$$
leaving the remaining pop transitions if $s \in \downarrow$, $t \in \downarrow$ untouched,

d) adding for each pop transition $(s,a,d,\varepsilon,t) \in \rightarrow$ a pop transition either $(s,a,d_\bot,\varepsilon,t) \in \rightarrow'$ if $t \in \downarrow$ or $(s,a,d_\bot,\varnothing,t) \in \rightarrow'$ if $t \notin \downarrow$.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s,\varepsilon),(s_\bot,\varepsilon)) \mid s \in \downarrow\} \cup \{((s,\varepsilon),(s_\bot,\varnothing)) \mid s \in \mathcal{S} \setminus \downarrow\} \cup \{((s,\delta d),(s,\delta d_\bot)) \mid s \in \mathcal{S}, d \in \mathcal{D}, \delta \in \mathcal{D}^*\}$$

is a divergence-preserving branching bisimulation between the transition associated system with M according to the FSES interpretation and the transition system associated with M' according to the ES interpretation. ■

If we combine the result above with the result of Theorem 4.9 we obtain as a corollary that for pushdown automata that are initially terminating, the class of pushdown transitions systems according to the FSES interpretation is the same, up to divergence-preserving branching bisimilarity, as the class according to the ES interpretation.

FSES and FS

We proceed to argue that the class of pushdown transition systems according to the FSES interpretation is a proper subclass, up to divergence-preserving branching bisimilarity, of the class of pushdown transition systems according to the FS interpretation. The classical proof (see, e.g., [HMU06, Theorems 6.9 and 6.11]) that a pushdown language according to the "acceptance by final state" approach is also a pushdown language according to the "acceptance by empty stack" approach employs τ-transitions in a way that is valid up to language equivalence, but not up to branching bisimilarity. For instance, the construction that modifies a pushdown automaton M into another pushdown automaton M' such that the language accepted by M by final state is accepted by M' by empty stack adds τ-transitions from every final state of M to a fresh state in M' in which the stack is emptied. The τ-transitions introduce, in M', a choice between the original outgoing transitions of the final state in M and termination by going to the fresh state; this choice is not necessarily present in M, and therefore the transition systems associated with M and M' may not be branching bisimilar.

If we want to go from FSES to FS, we drop the empty stack requirement. To still get the same behaviour, intuitively, we would have to add new final states that can only be entered from the original final states by using the empty test. This construction modifies the PDA in a similar way to the construction presented in Example 4.11.

EXAMPLE 4.13. Let us consider the example PDA in Figure 4.6 and the modified PDA in Figure 4.9. We add for each final state in the original PDA the fresh states $s_\perp, u_\perp$ and four empty-test transitions $s \xrightarrow{\tau[\perp/\varepsilon]} s_\perp$ and $u \xrightarrow{\tau[\perp/\varepsilon]} u_\perp$ to detect when in the FSES case termination could occur, and $s_\perp \xrightarrow{\tau[\perp/\varepsilon]} s$ and $u_\perp \xrightarrow{\tau[\perp/\varepsilon]} u$ to be able to return.

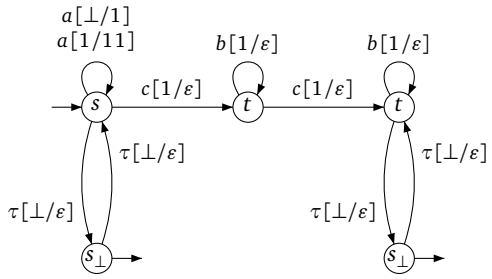

FIGURE 4.9: Modified pushdown automaton for FSES to FS.

This idea leads to a transformation that is correct up to branching bisimilarity, but does not preserve divergence, as it is possible to infinitely often perform empty-test transitions. We present a different approach that preserves divergence in Figure 4.10. This modification is inspired on the modification in Example 4.11 in the sense that it keeps track of when the stack is empty using extra states and marked data elements.

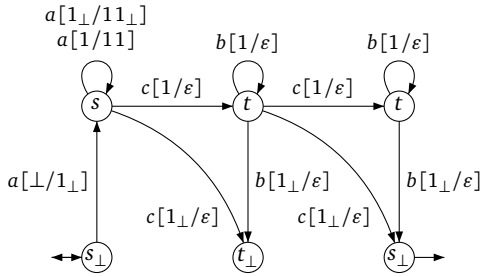

FIGURE 4.10: Modified pushdown automaton for FSES to FS preserving divergence.

The modification is almost the same as from FSES to ES, except that we do not use the dummy data element $\varnothing$ (cf. Figure 4.8). Instead, we only mark the "stack empty" counterpart states final if they correspond to final states in the original PDA. In this example these are $s_\perp$ and $u_\perp$. ♢

We now show that this modification works universally for all PDAs up to divergence-preserving branching bisimilarity.

THEOREM 4.14. *For each pushdown transition system according to the FSES interpretation there is, up to divergence-preserving branching bisimilarity, a pushdown transition system according to the FS interpretation.* □

PROOF. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be some pushdown automaton. By Theorem 4.8 we can assume that M only has push and pop transitions. We shall modify M such that the transition system associated with the modified pushdown automaton according to the FS interpretation is divergence-preserving branching bisimilar to the transition system associated with M according to the FSES interpretation. We define the modified pushdown automaton $M' = (\mathcal{S}', \mathcal{A}, \mathcal{D}', \rightarrow', \uparrow, \downarrow')$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh state $s_\perp$ for every state $s \in \mathcal{S}$;
2. $\mathcal{D}'$ is obtained from $\mathcal{D}$ by adding a marked data element $d_\perp$ for each $d \in \mathcal{D}$;
3. $\rightarrow'$ is obtained from $\rightarrow$ by
   a) replacing all push transitions $(s, a, \perp, d, t) \in \rightarrow$ by $(s_\perp, a, \perp, d_\perp, t) \in \rightarrow'$,
   b) adding for each push transition $(s, a, d, ed, t) \in \rightarrow$ a push transition $(s, a, d_\perp, ed_\perp, t) \in \rightarrow'$,
   c) adding for each pop transition $(s, a, d, \varepsilon, t) \in \rightarrow$ a pop transition $(s, a, d_\perp, \varepsilon, t_\perp) \in \rightarrow'$;
4. $\downarrow'$ is the set $\{ s_\perp \mid s \in \downarrow \}$ of all the newly added states that are counterparts of final states in M.

We leave it to the reader to verify that the relation

$$\mathcal{R} = \{ ((s, \varepsilon), (s_\perp, \varepsilon)) \mid s \in \mathcal{S} \} \cup \{ ((s, \delta d), (s, \delta d_\perp)) \mid s \in \mathcal{S}, d \in \mathcal{D}, \delta \in \mathcal{D}^* \}$$

is a divergence-preserving branching bisimulation between the transition system associated with M according to the FSES interpretation and the transition system associated with M' according to the FS interpretation. ■

Consequently, the class of pushdown transition systems according to the FSES interpretation is at least, up to divergence-preserving branching bisimilarity, a subclass of the class according to the FS interpretation. We can show that it is even a proper subclass.

EXAMPLE 4.15. Consider the pushdown automaton shown in Figure 4.11.

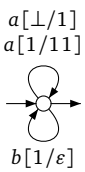

FIGURE 4.11: The counter pushdown automaton.

The transition system associated with it according to the FS interpretation is depicted in Figure 4.12; it has infinitely many terminating configurations. Moreover, no pair of these configurations is branching bisimilar, which we can see by noting that the nth state from the left can perform at most $n-1$ times a b-transition before it has to perform an a-transition again.

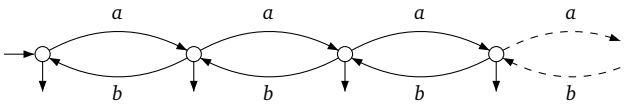

FIGURE 4.12: The transition system associated with PDA of Figure 4.11 according to the FS interpretation.

In contrast with this, note that the transition system associated with the pushdown automaton according to the FSES interpretation, as shown in Figure 4.13, necessarily has finitely many terminating configurations, for the pushdown automaton has only finitely many states and the stack is required to be empty.

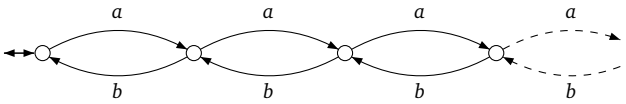

FIGURE 4.13: The transition system associated with automaton of Figure 4.11 according to the FSES interpretation.

This is a property of all pushdown transition systems according to the FSES interpretation. Therefore, there can be no pushdown transition system according to the FSES interpretation that is branching bisimilar to the pushdown transition system in Figure 4.12. ♢

The following mutual relations between the different classes of pushdown transition systems up to divergence-preserving branching bisimilarity have been established. (See also Figure 4.14 for a schematic overview. Note that in the diagram FSES^{it} stands for the class of transition systems according to the FSES interpretation associated with initially-terminating PDAs.)

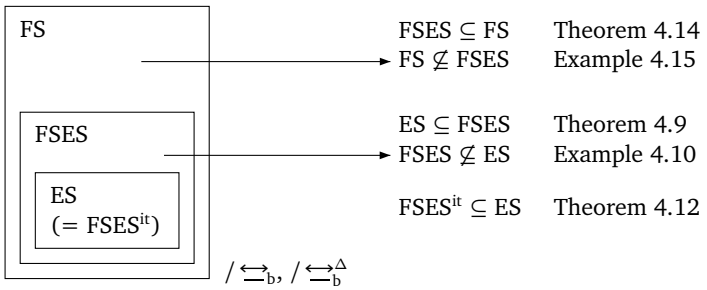

FIGURE 4.14: Overview of the different classes of pushdown transition systems.

COROLLARY 4.16. *The class of pushdown transition systems according to the ES interpretation is a proper subclass, up to divergence-preserving branching bisimilarity, of the class of pushdown transition systems according to the FSES interpretation.*

The class of pushdown transition systems according to the FSES interpretation is a proper subclass, up to divergence-preserving branching bisimilarity, of the class of pushdown transition systems according to the FS interpretation. □

Because the difference between the pushdown transition systems according to the ES and FSES interpretations is only based on whether the associated PDA is initially terminating or not, we will only consider the latter class from here on.

4.2 Sequential Specifications

In the previous chapter we have investigated the link between linear specifications and finite automata. In this section we will introduce the sequential specifications as the process-theoretic counter part of context-free grammars. We then consider a process-theoretic version of the standard result in the theory of automata and formal languages [Sud88, Sip97, HMU06] stating that the class of languages accepted by pushdown automata coincides with the class of languages generated by context-free grammars. This is done by comparing the pushdown transitions systems with the transition systems associated with sequential specifications up to (divergence-preserving) branching bisimilarity. We will first show that it is impossible to obtain this correspondence with the class of pushdown transition systems according to the FS interpretation. Then, we will consider the correspondence for pushdown transition systems according to the FSES interpretation and see that we still have to apply restrictions to both the pushdown automata and sequential specifications if we want to obtain the correspondence. Finally, we look into the decidability of whether two sequential specifications are equal. It is well-known from automata and formal language theory that it is undecidable whether two context-free grammars generate the same language. We will extend earlier work [BBK93, Bos97, Srb01] by showing that it is decidable whether two restricted sequential specifications are bisimilar.

Context-free Grammars

As the process-theoretic counterparts of context-free grammars we shall consider recursive specifications in the subtheory TSP_τ (Theory of Sequential Processes) of TCP_τ, which is obtained from BSP_τ by adding sequential composition $\_ \cdot \_$. Note that TSP_τ can also be seen as the process theory BPA extended with prefixing, $\mathbf{0}$ and $\mathbf{1}$ which also allows for τ-transitions. Processes definable in BPA are often referred to as "context-free processes." The motivation in the literature for this terminology seems to be twofold. On the one hand, it is easy to see that the language associated with a process definable in normed BPA is context-free. On the other hand, context-free grammars in Greibach normal form can be regarded as a BPA-specification by

1. regarding non-terminals as recursion names,
2. regarding a right-hand side $a\xi$ of a production $N \longrightarrow a\xi$ as the sequential composition of the action a and the sequence of non-terminals ξ, and

3. combining the right-hand sides of all productions $N \longrightarrow a_1\xi_1 \mid \cdots \mid a_n\xi_n$ for a non-terminal N with non-deterministic choice to constitute a single right-hand side $a_1\xi_1 + \cdots + a_n\xi_n$ defining the recursion name N.

The resulting recursive specification is guarded and generates a labelled transition system with the same language as the original context-free grammar.

It is well-known from the theory of automata and formal languages (see, e.g., [Sud88, Theorem 5.6.3]) that a context-free grammar can be transformed into Greibach normal form, provided that the grammar does not include so-called useless non-terminals (i.e., non-terminals for which there is no production) and λ-productions (or ϵ-productions). The first restriction is harmless from a language-theoretic point, for there is a language-preserving transformation that eliminates useless non-terminals from a context-free grammar. It is, however, unfortunate from a process-theoretic point of view, for, intuitively, a non-terminal without productions corresponds with a *deadlocked process*. The second restriction is inconvenient even from a language-theoretic point of view, for it excludes all languages with the empty word property.

A thorough investigation of the process theory TSP_τ reinforces the connection between the theory of automata and formal languages on the one hand, and process theory on the other hand. Firstly, it allows a translation of *all* context-free grammars directly into a finite recursive TSP_τ-specification: if there is a λ-production (or ϵ-production) for N, then the right-hand side of the defining equation for N gets a summand $\mathbf{1}$, and if the non-terminal N is useless, then it is defined by the recursion equation $N \stackrel{\text{def}}{=} \mathbf{0}$. Secondly, is possible to define, up to (divergence-preserving) branching bisimilarity, every non-deterministic finite automaton with a finite (guarded) recursive TSP_τ-specification, while it is not possible to define non-deterministic finite automata with intermediate accepting states with a BPA- or $\mathsf{BPA}_\mathbf{0}$-specification.

DEFINITION 4.17. A *sequential specification* over some finite set of names $\mathcal{N}$ is a finite, τ-guarded recursive TSP_τ-specification, i.e. a recursive specification over $\mathcal{N}$ in which only the constructions $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}$), $a._-$ ($a \in \mathcal{A}_\tau$), $\_ \cdot \_$ and $\_ + \_$ are used to build *sequential process expressions* △

EXAMPLE 4.18. The process expression N defined by the sequential specification

$$N \stackrel{\text{def}}{=} \mathbf{1} + a.N \cdot b.\mathbf{1}$$

specifies the pushdown transition system according to the FSES interpretation in Figure 4.13, that is associated with the pushdown automaton in Figure 4.11. ♢

Similarly to context-free grammars, our sequential specifications can be brought into Greibach normal form as well. We can define a normal form for sequential specifications if we instantiate Definition 2.19 (on page 19) with the sequence of names interpreted as a sequential composition of names.

DEFINITION 4.19. A sequential specification E is in *sequential normal form* if each defining equation of a name $N \in \mathcal{N}$ is of the following form:

$$N \stackrel{\text{def}}{=} \sum_{i \in \mathcal{I}_N} a_i.\xi_i \; (+ \mathbf{1}) \,.$$

In this form, every right-hand side of every defining equation consists of a number of summands, indexed by a finite set $\mathcal{I}_N$ (the empty sum is $\mathbf{0}$), each of which is $\mathbf{1}$, or of the form $a_i.\xi_i$ with $a_i \in \mathcal{A}_\tau$ and ξ_i a sequential composition of names; the empty sequential composition is denoted by $\mathbf{1}$. △

It is well-known that all sequential specifications can be brought in sequential normal form.

PROPOSITION 4.20. *For each sequential specification E and sequential process expression p there exists a sequential specification in sequential normal form E' such that $\mathcal{T}_{E'}(p) \leftrightarrows_b^\Delta \mathcal{T}_E(p)$.* □

If the sequences have a length of at most two, we say that the sequential specification is in *restricted* normal form. A proof of the following proposition follows the same lines of the proof of [BBK93, Proposition 4.3].

PROPOSITION 4.21. *For each sequential specification E and sequential process expression p there exists a restricted sequential specification in sequential normal form E' such that $\mathcal{T}_{E'}(p) \leftrightarrows_b^\Delta \mathcal{T}_E(p)$.* □

We can associate transition systems with sequential specifications according to the operational rules in Table 2.1 (on page 15). This also gives us the notion of sequential process.

DEFINITION 4.22. A *sequential process* is a divergence-preserving branching bisimilarity class of labelled transition systems containing a transition system associated with a sequential specification and sequential process expression. △

4.2.1 Correspondence

Now that we have defined sequential specifications as our process-theoretic counterparts of context-free grammars, we can investigate their relation with pushdown automata. That the notion of sequential specification still naturally corresponds with the notion of context-free grammar is confirmed by the following theorem that states the correspondence up to language equivalence. For the proof we refer to [HMU06, Section 6.3].

THEOREM 4.23. *For every pushdown automaton M there exists a sequential specification E, with initial name I, such that $\mathcal{T}(M) \approx \mathcal{T}_E(I)$ according to the FS, ES or FSES interpretation, and, vice versa, for every sequential specification E, with initial name I, there exists a pushdown automaton M such that $\mathcal{T}_E(I) \approx \mathcal{T}(M)$ according to the FS, ES or FSES interpretation.* □

We will now investigate the same result up to divergence-preserving branching bisimilarity. That is, we will compare pushdown transitions systems, according to the FS and FSES interpretations, with transition systems associated with the sequential specifications, given by the SOS rules in Table 2.1 (on page 15). After some definitions we will investigate the correspondence in both directions, first for the FS interpretation and then the FSES interpretation.

Let E be a sequential specification and I be its initial name. We say that E is *simulated by* some PDA M (according to the FS/FSES interpretation), if we have that $\mathcal{T}(M) \leftrightarrows_{\mathrm{b}}^{\Delta} \mathcal{T}_E(I)$. Vice versa, a PDA M (according to the FS/FSES interpretation) is said to be *defined by* a sequential specification E, with initial name I, if $\mathcal{T}_E(I) \leftrightarrows_{\mathrm{b}}^{\Delta} \mathcal{T}(M)$. If we know that there is such a sequential specification for PDA M we say that M is *definable by a sequential specification*.

Let us first consider a prominent PDA or pushdown transition system that can be defined by a sequential specification. Recall the pushdown transition system according to the (FS)ES interpretation of a stack shown in Figure 4.3.

The following infinite recursive specification E_S^∞ specifies the behaviour of the process S_ξ, modelling a stack with as contents the sequence of data elements ξ that receives input over channel i, i.e. when data is pushed, and sends output over channel o, i.e. when data is popped. For the empty stack, we have:

$$S_\varepsilon \stackrel{\text{def}}{=} \mathbf{1} + o!\bot.S_\varepsilon + \sum_{d\in\mathcal{D}} i?d.S_d \ ,$$

and for every non-empty string $d\xi$ ($d \in \mathcal{D}$, $\xi \in \mathcal{D}^*$):

$$S_{d\xi} \stackrel{\text{def}}{=} o!d.S_\xi + \sum_{e\in\mathcal{D}} i?e.S_{ed\xi} \ .$$

However, we would like our stack to be defined by a finite version of this specification to obtain a sequential specification.

DEFINITION 4.24. The following sequential specification defines a stack:

$$S \stackrel{\text{def}}{=} \mathbf{1} + o!\bot.S + \sum_{d\in\mathcal{D}} i?d.S_{\not\varepsilon} \cdot o!d.S \ ,$$
$$S_{\not\varepsilon} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathcal{D}} i?d.S_{\not\varepsilon} \cdot o!d.S_{\not\varepsilon} \ ;$$

we refer to this specification of a stack over $\mathcal{D}$ as E_S. Note that the associated transition system is, up to isomorphism, the same as the pushdown transition system shown in Figure 4.3. △

Note that only the stack PDA according to the FSES interpretation is defined by the sequential specification above. If we take the FS interpretation, we get the stack that can always terminate. We shall see later that in this case the stack PDA is not definable by a sequential specification.

A state of the stack can be characterised by a sequential composition, for example: $S_\not\perp \cdot o!d_n.S_\not\perp \cdot \ldots \cdot o!d_1.S$. An obvious modification to make E_S always terminating would be to ensure that every component of the sequential composition has a $\mathbf{1}$-summand so that termination is always possible.

DEFINITION 4.25. The sequential specification E_{S^f} of the *forgetful stack* over $\mathcal{D}$ is defined as follows:

$$S^f \stackrel{\text{def}}{=} \mathbf{1} + o!\bot.S^f + \sum_{d\in\mathcal{D}} i?d.S^f_\not\perp \cdot (o!d.\mathbf{1}+\mathbf{1}) \cdot S^f \,,$$

$$S^f_\not\perp \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathcal{D}} i?d.S^f_\not\perp \cdot (o!d.\mathbf{1}+\mathbf{1}) \cdot S^f_\not\perp \;;$$

see Figure 4.15 for the, rather contrived, associated transition system. Every node depicted has infinitely many incoming arrows. The dotted arrows only denote some of the outgoing arrows from nodes of level 4. △

Although every state is a final state, we have introduced unwanted behaviour by adding the $\mathbf{1}$-summands. We can "forget" items that are on the stack by popping items that are not the top element. Also the empty-test has lost its meaning as it is always enabled.

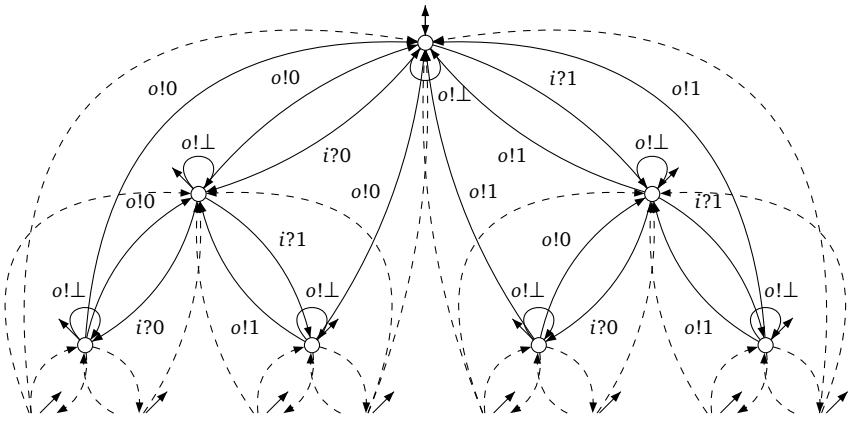

FIGURE 4.15: Forgetful stack over $\mathcal{D} = \{0,1\}$.

Pushdown transition systems according to the FS interpretation

We will now show that, in general, pushdown transition systems according to the FS interpretation cannot be defined by sequential specifications up to (divergence-preserving) branching bisimilarity.

THEOREM 4.26. *There exists a pushdown transition system according to the FS interpretation such that there is no sequential specification with an associated transition system that is (divergence-preserving) branching bisimilar to it.* □

PROOF. We prove by contradiction that the counter pushdown automaton (see Figure 4.11) according to the FS interpretation (see Figure 4.12) is not definable by a sequential specification. Let us first assume that there exists such a sequential specification E. Then, by Proposition 4.20, we can assume that E is in sequential normal form. From the definition of GNF (see Definition 2.19 on page 19) it follows that every state of the transition system associated with E is denoted by a sequential composition of its names. Since the associated transition system should be (divergence-preserving) branching bisimilar with the transition system in Figure 4.12, we now know two things about the names in E:

1. without loss of generality we can assume that all reachable names have a $\mathbf{1}$-summand in their defining equation, and
2. each name has a bounded b-norm, i.e. a maximal number of b-transitions that can be performed from the state associated with the name without performing any a-transitions.

Let n be the maximal b-norm of all names in E. Now, let s be a state that has a b-norm that is larger than n and let ξ be the sequential composition of names that belongs to the (divergence-preserving) branching bisimilar state in the associated transition system of E. Because the b-norm is larger than n, the sequence ξ must contain at least two names that can perform a b-transition, for example X,Y in $\xi_0 X \xi_1 Y \xi_2$. However, because all names have a $\mathbf{1}$-summand, we have that $\xi_0 X \xi_1 Y \xi_2 \xrightarrow{b} \xi_1 Y \xi_2$ and $\xi_0 X \xi_1 Y \xi_2 \xrightarrow{b} \xi_2$, thus leading to two non-bisimilar states. This is not possible in the transition system of the counter PDA. Hence, a sequential specification does not exist. ■

For the remainder of this section, we shall focus on the FSES interpretation. In Section 4.3 we will come back to the FS interpretation.

Pushdown transition systems according to the FSES interpretation

We shall see below that the classical correspondence result with language equivalence replaced by branching bisimilarity still does not hold if we restrict ourselves to the FSES interpretation. In fact, we shall see that there are pushdown transition systems that are not (divergence-preserving) branching bisimilar with the transition system associated with a sequential specification, and that there are also sequential specifications that are not (divergence-preserving) branching bisimilar to a pushdown transition system. We shall first present a restriction on sequential specifications and relate them with a subclass of the pushdown automata and then given this restricted class of pushdown automata achieve the desired equivalence: we shall prove that the *transparency-restricted* sequential specifications correspond with the so-called *pop choice-free* pushdown automata.

On the side of sequential specifications, restricting to the sequential normal form is not sufficient to get the desired correspondence between transition systems associated with sequential specifications and pushdown transition systems.

EXAMPLE 4.27. Consider the following sequential specification, which is in sequential normal form:

$$X \stackrel{\text{def}}{=} a.X \cdot Y + b.\mathbf{1} \ ,$$
$$Y \stackrel{\text{def}}{=} \mathbf{1} + c.\mathbf{1} \ .$$

The transition system associated with X, which is depicted in Figure 4.16, has unbounded branching. ♢

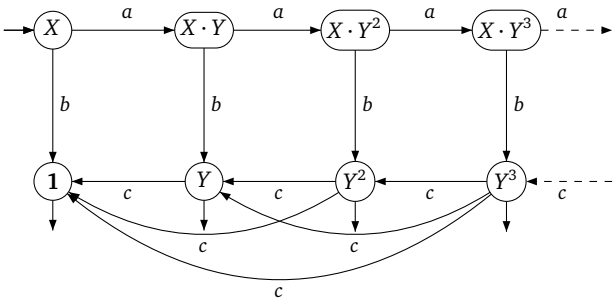

FIGURE 4.16: A transition system with unbounded branching.

Note that if $i \neq j$, then Y^i and Y^j are not bisimilar, since each state Y^i admits up to i consecutive c-transitions. Hence, there does not exist a bound on the branching degree of process expressions reachable from X: each Y^i ($i \in \mathbb{N}$) is reachable and has a branching degree of i. Note how, intuitively, execution of the c-transition from Y^i to Y^j "skips" the behaviour of all intermediate Y^k ($j < k < i$).

A name N in a recursive specification is called *transparent* if its defining equation has a $\mathbf{1}$-summand; otherwise it is called *opaque*. Recall that we had a similar unbounded branching problem with the specification of the forgetful stack (see Definition 4.25) where also all elements of the sequential specification are transparent.

In [BCLT10], we have conjectured that a pushdown transition system cannot have unbounded branching. If we desire a correspondence between sequential specifications and pushdown automata, we shall have to exclude sequential specifications with associated transition systems that have unbounded branching. One way to achieve this is to require that transparent names may only occur as the *last* element of reachable sequential compositions of names.

DEFINITION 4.28. Let E be a sequential specification in sequential normal form. We call such a specification *transparency-restricted* if for all (generalised) sequential compositions of names ξ reachable from a name in E it holds that all but the last name in ξ is opaque. △

While transparency-restrictedness might seem quite a severe restriction on sequential specifications, note that it still allows us to specify useful processes such as

the stack over $\mathcal{D}$ defined in Definition 4.24. While not yet transparency-restricted, it can be defined with a transparency-restricted recursive specification by bringing it in sequential normal form: it suffices to add, for all $d \in \mathcal{D}$, a name T_d to replace $S_{\not\perp} \cdot o!d.\mathbf{1}$.

DEFINITION 4.29. Thus we redefine the the stack over $\mathcal{D}$ by the following transparency-restricted sequential specification:

$$S \stackrel{\text{def}}{=} \mathbf{1} + o!\perp.S + \sum_{d\in\mathcal{D}} i?d.T_d \cdot S \ ,$$
$$T_d \stackrel{\text{def}}{=} o!d.\mathbf{1} + \sum_{e\in\mathcal{D}} i?e.T_e \cdot T_d \ . \qquad \triangle$$

It can easily be seen that the transition system associated with a name in a transparency-restricted specification has bounded branching: the branching degree of a state denoted by a reachable sequential composition of names is equal to the branching degree of its first name, and the branching degree of a name is bounded by the number of summands of the right-hand side of its defining equation.

We are now in a position to establish a process-theoretic counterpart of the correspondence between pushdown automata and context-free grammars. First, we consider the direction from transparency-restricted sequential specification to pushdown automaton. For each specification we can construct a pushdown automaton that simulates it.

EXAMPLE 4.30. Let E be the following sequential specification:

$$X \stackrel{\text{def}}{=} a.X \cdot Y + b.Y + c.\mathbf{1} \ ,$$
$$Y \stackrel{\text{def}}{=} d.\mathbf{1} \ .$$

This specification is in restricted sequential normal form and transparency-restricted as both X and Y are opaque. Figure 4.17 depicts a pushdown automaton with only push and pop transitions that simulates E if we take X as its initial name.

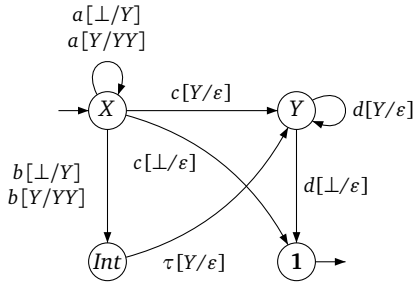

FIGURE 4.17: A pushdown automaton simulating sequential specification E.

We have a state for each name in the specification and two extra states: $\mathbf{1}$ to go to when the stack is empty, and *Int* as an intermediate state to ensure that we only have

push- and pop transitions. (The reason for this will become apparent later.) For each summand of a name in the specification we have a corresponding PDA transition from the state corresponding to the name. Therefore, if we are in a state corresponding with a name, we are simulating the behaviour of that name. For example, for X we have the summands $a.X \cdot Y$, $b.Y$, and $c.\mathbf{1}$. For the summand $a.X \cdot Y$ we add transitions $X \xrightarrow{a[\bot/Y]} X$ and $X \xrightarrow{a[Y/YY]} X$, because if we perform an a-transition from X, we end up in X again with an extra Y on the stack. For the summand $b.Y$ we add the transitions $X \xrightarrow{b[\bot/Y]} \mathit{Int}$, $X \xrightarrow{b[Y/YY]} \mathit{Int}$, and $\mathit{Int} \xrightarrow{\tau[Y/\varepsilon]} Y$. Since the b-transition requires no stack manipulation, we actually just need to go to Y, and this would result in neither a push- or pop transition, we go through an intermediate state. Finally, for the summand $c.\mathbf{1}$ we add the transitions $X \xrightarrow{c[\bot/\varepsilon]} \mathbf{1}$ and $X \xrightarrow{c[Y/\varepsilon]} Y$. If the c-transition is executed, we are done with simulating X. We pop from the stack to see what is next and move to the corresponding state. If the stack is empty, we are done and we move to state $\mathbf{1}$, where we can terminate. ♢

In the example we used the knowledge that only the name Y will ever be stacked. For clarity, all transitions that dealt with the possibility that the name X could be popped from the stack have been omitted. We can generalise the example above to a more formal construction and obtain the following result.

THEOREM 4.31. *For every transparency-restricted sequential specification E, with initial name I, there exists a pushdown automaton M such that $\mathcal{T}(M) \leftrightarrows_b^{\Delta} \mathcal{T}_E(I)$.* □

PROOF. Let E be a transparency-restricted sequential specification over a finite set of names $\mathcal{N}$, and let I be an initial name of E. We define a pushdown automaton $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ as follows:

1. $\mathcal{S}$ consists of all names in $\mathcal{N}$, the symbol $\mathbf{1}$, and an extra intermediate state *Int*;
2. $\mathcal{A}$ consists of all the actions occurring in E;
3. $\mathcal{D}$ consists of the names occurring in E;
4. $\rightarrow$ is defined as follows: for all $a \in \mathcal{A}$
   a) if the right-hand side of the defining equation for a name N has a summand $a.\mathbf{1}$, then $\rightarrow$ has transitions $N \xrightarrow{a[\bot/\varepsilon]} \mathbf{1}$ and $N \xrightarrow{a[N'/\varepsilon]} N'$ $(N' \in \mathcal{N})$,
   b) if the right-hand side of the defining equation for a name N has a summand $a.N'$, then there are transitions $N \xrightarrow{a[d/N'd]} \mathit{Int}$ $(d \in \mathcal{D})$, $N \xrightarrow{a[\bot/N']} \mathit{Int}$ and $\mathit{Int} \xrightarrow{\tau[N'/\varepsilon]} N'$,
   c) if the right-hand side of the defining equation for a name N has a summand $a.N' \cdot N''$, then there are transitions $N \xrightarrow{a[d/N''d]} N'$ $(d \in \mathcal{D})$;
5. $\uparrow$ is the initial name I;
6. $\downarrow$ consists of $\mathbf{1}$ and all names with a $\mathbf{1}$-summand.

Note that the transitions in M are either a pop or a push transition, and that the τ-transitions introduced in the transition system associated with M are inert. We leave it to the reader to verify that the relation

$$\mathcal{R} = \{(N\xi, (N, \xi)), (N\xi, (\mathit{Int}, N\xi)) \mid N \in \mathcal{N}, \xi \in \mathcal{N}^*\} \cup \{(\mathbf{1}, (\mathbf{1}, \varepsilon))\}$$

is a divergence-preserving branching bisimulation between the transition system associated with the sequential specification E for the initial name I and the transition system associated with M according to the FSES interpretation. ■

Note that in the construction in the example and proof above we have that, when some name N is popped, the PDA always ends up in the state labelled N. A more general version of this property turns out to be vital if we want to obtain a correspondence in the other direction.

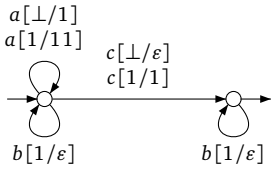

FIGURE 4.18: A pushdown automaton that is not pop choice-free.

Consider the pushdown automaton in Figure 4.18; the associated transition system is shown in Figure 4.19. In [Mol96], Moller proved that this transition system cannot be defined with a recursive BPA-specification. His proof can be modified to show that the transition system is not definable with a sequential specification either.

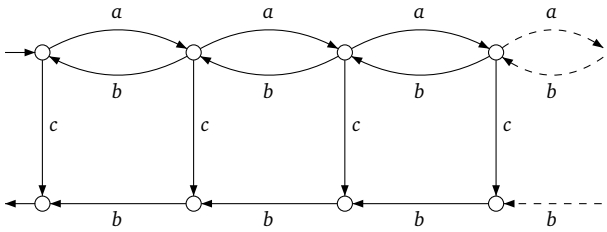

FIGURE 4.19: The transition system associated with the PDA of Figure 4.18.

Note that a push of a data element 1 in the initial state of the pushdown automaton in Figure 4.18 can be popped again in the initial state or in the final state: the choice of where the pop will take place cannot be made at the time of the push. In other words, in the pushdown automaton in Figure 4.18 pop transitions may induce a choice in the associated transition system; we refer to such choice through a pop transition as a *pop choice*. We shall prove below that by disallowing pop choices we define a class of pushdown processes that are definable with sequential specifications.

DEFINITION 4.32. Let M be a pushdown automaton that uses only push and pop transitions. A d-pop transition is a transition $s \xrightarrow{a[d/\varepsilon]} t$, which pops a data element d. We say M is *pop choice-free* iff whenever there are two d-pop transitions $s \xrightarrow{a[d/\varepsilon]} t$ and $s' \xrightarrow{b[d/\varepsilon]} t'$, then $t = t'$. A pushdown transition system is *pop choice-free* if is associated with a pop choice-free pushdown automaton. △

We have not been able to establish that our result is optimal, i.e. that pop choice-freeness is a necessary condition to be able to define it by a sequential specification.

CONJECTURE 4.33. *For each pushdown automaton M there exists a transparency-restricted sequential specification E, with initial name I, such that $\mathcal{T}_E(I) \leftrightarrows_b^{\Delta} \mathcal{T}(M)$ if, and only if, M is* pop choice-free. □

All pushdown automata that can be constructed to simulate a sequential specification according to the proof of Theorem 4.35 are pop choice-free. Now, if we maintain the pop choice-free restriction for the other direction, we get the full correspondence.

EXAMPLE 4.34. Let us consider the example pushdown automaton shown in Figure 4.1 (on page 40). This pushdown automaton is pop choice-free, for both 1-pop transitions lead to the same state t.

Now, consider the following sequential specification that defines the PDA:

$$
\begin{aligned}
N_{s\varepsilon} &\stackrel{\text{def}}{=} \mathbf{1} + a.N_{s1t} \cdot N_{t\varepsilon} \ , \\
N_{t\varepsilon} &\stackrel{\text{def}}{=} \mathbf{1} \ , \\
N_{s1t} &\stackrel{\text{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot N_{t1t} \ , \\
N_{t1t} &\stackrel{\text{def}}{=} b.\mathbf{1} \ ;
\end{aligned}
$$

the initial name of this specification is $N_{s\varepsilon}$. The associated transition system has been depicted in Figure 4.20.

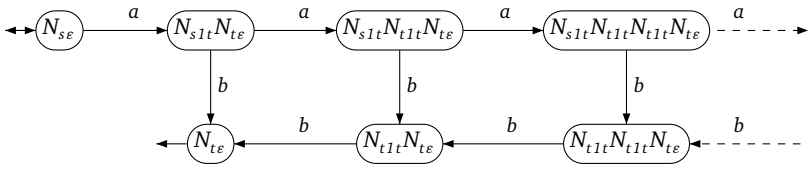

FIGURE 4.20: The transition system associated with sequential specification defining the PDA from Figure 4.1.

The names $N_{s\varepsilon}$ and $N_{t\varepsilon}$ are introduced to encode that we are in state s and t respectively and that the stack is empty. Both names have a $\mathbf{1}$-summand because both states are also final states.

Since we know that the PDA is pop choice-free, we can determine for each data element $d \in \mathcal{D}$ the state we are going to end up in if we pop that data element. In this case there is only the data element 1; after a 1-pop transition we end up in state t. So, we also introduce the names N_{s1t} and N_{t1t} as both states s and t have a 1-pop transition to t. Intuitively, the names encode in which state we are, that a data element 1 is stacked and what state we end up in once it is popped.

We have added summands to the defining equations for each name, given that the PDA only has push and pop transitions. For name $N_{s\varepsilon}$ this is the empty-test (push) transition $s \xrightarrow{a[\bot/1]} t$ for which we have added the summand $a.N_{s1t} \cdot N_{t\varepsilon}$. This summand ensures that after an a-transition we are still in state s, stack the data element 1 and once this is popped we end up in t (and by then the stack is empty). For name N_{s1t} we add a similar summand for the pop transition $s \xrightarrow{a[1/11]} s$. Finally,

we add the summand $b.\mathbf{1}$ to the defining equation of the names N_{s1t} and N_{t1t} because we have the following push transitions: $s \xrightarrow{b[1/\varepsilon]} t$ and $s \xrightarrow{b[1/\varepsilon]} t$. After a b-transition, which happens when data element 1 is popped, we are done with the name and we move to the next name in the sequential composition.

Note that only the names $N_{s\varepsilon}$ and $N_{t\varepsilon}$ have $\mathbf{1}$-summands and that they only occur at the end of the sequential composition. Hence, our sequential specification is transparency-restricted.

We can reduce this specification by removing occurrences of $N_{t\varepsilon}$ (for the right-hand side of the defining equation of this name is just $\mathbf{1}$) and substituting occurrences of N_{t1t} by $b.\mathbf{1}$. We get

$$N_{s\varepsilon} \stackrel{\text{def}}{=} \mathbf{1} + a.N_{s1t} \ ,$$
$$N_{s1t} \stackrel{\text{def}}{=} b.\mathbf{1} + a.N_{s1t} \cdot b.\mathbf{1} \ .$$

Now, we see that $N_{s1t} = (\mathbf{1} + a.N_{s1t}) \cdot b.\mathbf{1} = N_{s\varepsilon} \cdot b.\mathbf{1}$ and therefore we have that $N_{s\varepsilon} = \mathbf{1} + a.N_{s\varepsilon} \cdot b.\mathbf{1}$ which is, up to renaming, equal to the specification we gave before. ♢

We can generalise this example to a more formal construction and obtain the following result.

THEOREM 4.35. *For each pop choice-free pushdown automaton M there exists a transparency-restricted sequential specification E, with initial name I, such that $\mathcal{T}_E(I) \leftrightarrows_b^{\Delta} \mathcal{T}(M)$.* □

PROOF. This proof is an adaptation of the classical proof (see for example [HMU06, Theorem 6.14]) that associates a context-free grammar with a given pushdown automaton. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a pop choice-free pushdown automaton. By Theorem 4.8 we can ensure that M only has push and pop transitions. We define a transparency-restricted specification E with for every state $s \in \mathcal{S}$ a name $N_{s\varepsilon}$ with the following defining equation:

$$N_{s\varepsilon} \stackrel{\text{def}}{=} \sum_{\substack{(s,a,\perp,d,t)\in\rightarrow \\ d\text{-pop to } w}} a.N_{tdw} \cdot N_{w\varepsilon} \ [+\ \mathbf{1}]_{s\downarrow} \ ,$$

and for every state s a name N_{sdt}, if M has transitions that pop data element d leading to the state t, with the following defining equation:

$$N_{sdt} \stackrel{\text{def}}{=} \sum_{\substack{(s,a,d,ed,u)\in\rightarrow \\ e\text{-pop to } w}} a.N_{uew} \cdot N_{wdt} + \sum_{(s,a,d,\varepsilon,t)\in\rightarrow} a.\mathbf{1} \ .$$

Recall that the state w is each time uniquely given because the PDA M is pop choice-free. It is easy to see that the resulting specification is transparency-restricted.

Assuming that that each d_i-pop leads to state s_i $(1 \leq i \leq n)$, we leave it to the reader to verify that the relation

$$\mathcal{R} = \{((s,\varepsilon), N_{s\varepsilon}) \mid s \in \mathcal{S}\} \cup \{((s, d_1 \ldots d_n), N_{sd_1s_1} \cdot \ldots \cdot N_{s_{n-1}d_ns_n} \cdot N_{s_n\varepsilon}) \mid s, \in \mathcal{S}, d_1, \ldots, d_n \in \mathcal{D}\}$$

is a divergence-preserving branching bisimulation and hence $\mathcal{T}_E(N_{\uparrow\varepsilon}) \leftrightarrows^{\Delta}_{\mathrm{b}} \mathcal{T}(M)$. ■

Thus, we have established a correspondence between a pop choice-free pushdown automaton on the one hand, and transparency-restricted sequential specification on the other hand. We thereby cast the classical result of the equivalence of pushdown automata and context-free grammars in terms of transition systems and bisimulation.

COROLLARY 4.36. *For every pop choice-free pushdown automata M there exists a guarded transparency-restricted sequential specification E, with initial name I, such that $\mathcal{T}(M) \leftrightarrows^{\Delta}_{b} \mathcal{T}_E(I)$, and vice versa.* □

PROOF. The result follows from Theorems 4.35 and 4.31. ■

The results presented above only hold for transparency-restricted sequential specifications. In [BCT08] we have established that we can have the correspondence for all sequential specifications, if we step down to a weaker equivalence than branching bisimilarity called contrasimilarity [Gla93, VM01]. In this paper the correspondence was formulated between sequential specifications and a finite-state process put in parallel with a forgetful stack process, thus simulating, up to contrasimilarity, the specifications using a special kind of pushdown automaton. We conjecture that the proof in [BCT08] can be adapted to show that all sequential specifications can be simulated, up to contrasimilarity, using our standard definition of the pushdown automaton (according to the FSES interpretation). For this, we have to move the handling of transparency from the stack to the finite control. This can be done by replacing forgetful popping by non-deterministic popping using τ-transitions.

CONJECTURE 4.37. *For every sequential specification E, with initial name I, there exists a pushdown automaton M such that $\mathcal{T}(M)$ is contrasimilar with $\mathcal{T}_E(I)$.* □

4.2.2 Decidability

It is well-known that it is undecidable whether two context-free grammars generate the same language up to language equivalence. Baeten, Bergstra and Klop have shown in [BBK93] that it *is* decidable for normed processes defined by guarded recursive BPA-specifications, which they consider to be the process-theoretic counterparts of context-free grammars in Greibach normal form, using the finer-grained equivalence of strong bisimilarity. First, several simplified proofs of the result in [BBK93] were presented (see [Cau86, HS91, Gro92]), and then the result was extended by Christensen, Hüttel and Stirling in [CHS95] to the class of *all* processes definable by recursive BPA-specifications. Later it has been proved independently

by Bosscher, in [Bos97], and Srba, in [Srb01], that the problem of deciding whether two $\mathsf{BPA}_\mathbf{0}$-definable processes are strongly bisimilar can be reduced to the problem of deciding whether two BPA-definable processes are strongly bisimilar. Both proofs consist of reducing the problem of deciding whether $\mathsf{BPA}_\mathbf{0}$-definable processes are strongly bisimilar to the problem of deciding whether BPA-definable processes are strongly bisimilar. It follows that strong bisimilarity remains decidable if **0** is added to BPA.

In this section we will consider the decidability of strong bisimilarity on TSP_τ, which is an extension of $\mathsf{BPA}_\mathbf{0}$ with prefixing and, more importantly, the constant **1**. While we would like to have a decidability result for branching bisimilarity (preferably divergence-preserving), we still leave it as an open problem. However, since the decidability of bisimilarity is still an interesting question, we extend earlier work and consider the obtained result as a stepping stone. We reduce the decidability problem to the problem of deciding whether $\mathsf{BPA}_\mathbf{0}$-definable transition systems are bisimilar. This reduction is not trivial because the constant **1** is responsible for a considerable increase of the expressiveness. We refer to [BLMT10] for a study of the increased expressiveness when the constant **1** is added to some well-known process algebras. Recall the sequential specification from Example 4.27 which has an associated transition system (see Figure 4.16) that has unbounded branching due to the presence of the constant **1**.

First, we argue that the proof of [CHS95] for BPA is not, in general, robust for the extension with **1**. Then, we prove that bisimilarity is decidable on the subclass of transition systems definable by the earlier mentioned restricted sequential specification, a class that properly includes the $\mathsf{BPA}_\mathbf{0}$-definable transition systems.

The proof by Christensen, Hüttel and Stirling

We argue that the decidability proof by Christensen, Hüttel and Stirling for BPA in [CHS95] cannot easily be extended to TSP_τ. An important notion in their proof is the notion of *bisimulation base*. Roughly, a bisimulation base is a binary relation $\mathcal{R}$ on processes/transition systems such that its congruence closure with respect to sequential composition (i.e., the least equivalence on processes that contains $\mathcal{R}$ and is compatible with sequential composition) is a bisimilarity. The crucial insight of the proof is that for every finite recursive BPA-specification there exists a *finite* bisimulation base, which consists of two parts:

1. The first part consists of all pairs (X, ξ) with a name X and a sequences ξ of names bisimilar to it. In a BPA-specification, all names have a positive norm, so there can only be finitely many sequences of names ξ_i with the same norm as X.
2. The second part consists of all so-called *indecomposable pairs*, i.e., pairs (ξ, χ) of bisimilar sequences of names that cannot be (non-trivially) split up into smaller pairs $(\xi_1, \chi_1), \ldots, (\xi_n, \chi_n)$ such that $\xi = \xi_1 \cdots \xi_n$ and $\chi = \chi_1 \cdots \chi_n$.
   Clearly, the congruence with respect to sequential composition that is generated by the set of all such indecomposable pairs by definition contains all decompos-

able pairs of bisimilar sequences of names. The argument that the collection of indecomposable pairs is actually finite, is highly nontrivial.

In the original proof every name X has a positive norm, but now it can also have norm 0. Consider for example the defining equation $X \stackrel{\text{def}}{=} a.X + \mathbf{1}$. We have that $X \leftrightarrows X^k$ for any k, so the number of pairs is no longer finite.

Due to the presence of $\mathbf{1}$, the indecomposable pair $(\mathbf{1} \cdot X\xi, (a.\mathbf{1} + \mathbf{1})X\xi)$ where $X \stackrel{\text{def}}{=} a.X + \mathbf{1}$ and ξ can be any sequence, we have an infinite number of indecomposable pairs. Hence, the bisimulation base becomes infinite.

In [Srb01], Srba uses a different approach that reduces the decidability of $\mathsf{BPA}_{\mathbf{0}}$-definable processes to BPA-definable processes. Srba gives a reduction that replaces the deadlocked process $\mathbf{0}$ in some specification by the name D with the defining equation $D \stackrel{\text{def}}{=} d \cdot D$ and provides a relation between the original and translated process. Using this bisimulation preserving translation relation between $\mathsf{BPA}_{\mathbf{0}}$ and BPA and reusing the previously mentioned result by Christian, Hüttel and Stirling for BPA, he shows the decidability of bisimilarity for $\mathsf{BPA}_{\mathbf{0}}$.

In our setting, a straightforward reduction from TSP_{τ}-definable processes to $\mathsf{BPA}_{\mathbf{0}}$-definable processes does not seem possible due to the extra expressive power added by $\mathbf{1}$-summands. Note that replacing prefixing by sequential composition, and replacing a $\mathbf{1}$-summand by a $\surd$-summand for some fresh atomic action $\surd$ (the explicit termination action) does not work in general because it may result in intermediate $\surd$-actions in a $\mathsf{BPA}_{\mathbf{0}}$-defined process that are impossible to relate to intermediate termination in the original TSP_{τ}-defined process.

EXAMPLE 4.38. Consider the following sequential specification:

$$X \stackrel{\text{def}}{=} a.\mathbf{1} + \mathbf{1}\,,$$
$$Y \stackrel{\text{def}}{=} b.\mathbf{1}\,.$$

Now, let $X' \stackrel{\text{def}}{=} a + \surd$ and $Y' \stackrel{\text{def}}{=} b$ be the translated versions for $\mathsf{BPA}_{\mathbf{0}}$. If we have that some $Z \leftrightarrows XY = a.b.\mathbf{1} + b.\mathbf{1}$, then it should hold for our translation that $Z' \leftrightarrows X'Y'$ where $Z' \stackrel{\text{def}}{=} a \cdot b + b$. However, $X'Y' = a \cdot b + \surd \cdot b$ and here it is possible to execute the $\surd$-action while Z' cannot. Obviously, they are not bisimilar. Also, the $\surd$-action is meant to signal termination, but $X'Y'$ can still execute the b-action after it. ♢

So, these intermediate $\surd$-actions that pop up due to this kind of translation form a problem. To ensure that these intermediate $\surd$-actions do not occur, we have to consider a restricted set of sequential specifications. An obvious choice is the transparency-restricted sequential specifications introduced in Definition 4.28 as they will not have intermediate termination behaviour.

We divide the set of names $\mathcal{N}$ for some sequential specification into disjoint subsets called the finitely normed names $\mathcal{N}_{\text{fin}} = \{X \in \mathcal{N} \mid X \text{ is normed}\}$ and the infinitely normed names $\mathcal{N}_{\infty} = \mathcal{N} - \mathcal{N}_{\text{fin}}$. We can further partition the set of finitely normed names $\mathcal{N}_{\text{fin}}$ into the transparent finitely normed names $\mathcal{N}_{\text{fin}}^{+1}$ and the opaque finitely normed names $\mathcal{N}_{\text{fin}}^{-1}$.

A useful property of the class of sequential specifications is that if a name has an infinite norm then by definition we have to end up with another name that has infinite norm after an action has been executed. As a result everything after an infinitely normed name can be removed preserving bisimilarity:

$$\xi X \eta \leftrightarrows \xi X \qquad \text{if } X \in \mathcal{N}_\infty \,.$$

Both this property and the above mentioned transparency-restrictedness leads us to the fact that we can restrict ourselves from here on to states where the labels, which are sequences of names, are elements of the set $(\mathcal{N}_{\text{fin}}^{-1})^*\mathcal{N}_\infty \cup (\mathcal{N}_{\text{fin}}^{-1})^*\mathcal{N}_{\text{fin}}^{+1} \cup (\mathcal{N}_{\text{fin}}^{-1})^*$ or using a more compact notation: $\{\mathbf{1}\} \cup (\mathcal{N}_{\text{fin}}^{-1})^*\mathcal{N}$. Recall that the empty sequence for a sequential composition is denoted by $\mathbf{1}$.

Deciding strong bisimilarity

As mentioned before, the result by Srba in [Srb01] involves a reduction from $\mathsf{BPA}_\mathbf{0}$ to BPA. In this section we give a reduction from TSP_τ to $\mathsf{BPA}_\mathbf{0}$ that preserves and reflects bisimilarity defined by a transparency-restricted sequential specification E.

We recall the syntax of $\mathsf{BPA}_\mathbf{0}$ and give the set of $\mathsf{BPA}_\mathbf{0}$-process expressions $\mathcal{P}(\mathsf{BPA}_\mathbf{0})$ by the following abstract syntax:

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

where a ranges over the set of atomic actions $\mathcal{A}$, and N ranges over the set of names $\mathcal{N}$. So, with respect to TSP_τ we have no constant $\mathbf{1}$ and prefixing. Note that Srba actually uses the symbol δ instead of $\mathbf{0}$ to denote the deadlocked process.

Because $\mathsf{BPA}_\mathbf{0}$ has no explicit termination and prefixing, it has different operational rules. The structural operational semantics of $\mathsf{BPA}_\mathbf{0}$ are given in Table 4.1 below.

$$\frac{}{a \xrightarrow{a} \surd} \qquad \frac{p \xrightarrow{a} p'}{p \cdot q \xrightarrow{a} p' \cdot q} \qquad \frac{p \xrightarrow{a} \surd}{p \cdot q \xrightarrow{a} q}$$

$$\frac{p \xrightarrow{a} p'}{p + q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p + q \xrightarrow{a} q'} \qquad \frac{p \xrightarrow{a} \surd}{p + q \xrightarrow{a} \surd} \qquad \frac{q \xrightarrow{a} \surd}{p + q \xrightarrow{a} \surd}$$

$$\frac{p \xrightarrow{a} p \qquad (N \stackrel{\text{def}}{=} p) \in E}{N \xrightarrow{a} p'} \qquad \frac{p \xrightarrow{a} \surd \qquad (N \stackrel{\text{def}}{=} p) \in E}{N \xrightarrow{a} \surd}$$

TABLE 4.1: Operational rules for a recursive $\mathsf{BPA}_\mathbf{0}$-specification E ($a \in \mathcal{A}_\tau$).

We assume that the specification E is transparency-restricted and in sequential normal form. Now, we reduce the decision problem to the problem of decidability of bisimilarity in $\mathsf{BPA}_{\mathbf{0}}$ as shown in [Srb01].

For the following proofs we fix a fresh action $\surd$ such that $\surd \notin \mathcal{A}$. We define $\mathcal{A}_{\surd} = \mathcal{A} \cup \{\surd\}$ and the translation function $f : \mathcal{P}(\mathsf{TSP}_{\tau}) \to \mathcal{P}(\mathsf{BPA}_{\mathbf{0}})$ as follows:

$$f(\mathbf{0}) = \mathbf{0}\,, \qquad f(\mathbf{1}) = \surd\,,$$
$$f(p+q) = f(p) + f(q)\,, \qquad f(p \cdot q) = f(p) \cdot f(q)\,,$$
$$f(X) = X\,, \qquad f(a.p) = \begin{cases} a & \text{if } p = \mathbf{1}\,, \\ a \cdot f(p) & \text{otherwise}\,. \end{cases}$$

Thus, f simply replaces the $\mathbf{1}$-summands of each transparent name with a $\surd$-summand and changes prefixes into sequential compositions.

If we apply f to the terms of the specification E we get the translated specification $E' = \{X_i \stackrel{\text{def}}{=} f(p_i) \mid X_i \stackrel{\text{def}}{=} p_i \in E\}$. It can be easily seen that the translated guarded recursive specification has the following GNF:

$$X \stackrel{\text{def}}{=} \sum_{i \in \mathcal{I}_X} a_i \cdot \xi_i \;(+\surd) \qquad \text{for all } X \in \mathcal{N}\,.$$

We introduce the variant of (strong) bisimilarity often used in conjunction with $\mathsf{BPA}_{\mathbf{0}}$ that does not take termination into account: let us write $\xi \leftrightarrows\!\!\!\!\!\!\surd\ \chi$ iff (ξ, χ) is in a binary relation $\mathcal{R}$ satisfying, for all $a \in \mathcal{A}_{\surd}$, conditions 1 and 2 of Definition 2.4 (on page 10).

Recall that the structural operational semantics given in Tables 2.1 and 4.1 are actually parametrized by a specification E. For clarity we shall write $\longrightarrow_E$ for the transitive relation and $\downarrow_E$ for the termination predicate associated with E and $\longrightarrow_{E'}$ for the transitive relation associated with E'.

LEMMA 4.39. *Given the specification E and the translated version E' the following holds for every ξ:*

1. $\xi \xrightarrow{a}_E \xi'$ *iff* $\xi \xrightarrow{a}_{E'} \xi'$ *with* $a \neq \surd$,
2. $\xi\!\downarrow_E$ *iff* $\xi \xrightarrow{\surd}_{E'} \surd$. □

PROOF. We prove both statements separately, first from left to right, then from right to left.

1. $\Rightarrow$ If $\xi \xrightarrow{a}_E \xi'$ then there exist ρ, η and some name X that has the defining equation with a summand $a.\xi_i$ for some $i \in \mathcal{I}_X$ such that $\xi = \rho X \eta$, $X \xrightarrow{a} \xi_i$ and $\xi' = \xi_i \eta$. Note that by transparency-restrictedness, $\rho = \mathbf{1}$ and thus $\xi = X\eta$. Then, also in the translated specification X has a summand $a \cdot \xi_i$ and hence $\xi = X\eta \xrightarrow{a}_{E'} \xi_i \eta = \xi'_i$.

   $\Leftarrow$ If $\xi \xrightarrow{a}_{E'} \xi'$ with $a \neq \surd$ then, as in the previous case, $\xi = X\eta$, where η may be empty, but now X has a summand $a \cdot \xi_i$ for some $i \in \mathcal{I}_X$. So $\xi' = \xi_i \eta$, and similarly we have the summand $a.\xi_i$ in the original defining equation and hence $\xi = X\eta \xrightarrow{a}_E \xi\,\eta = \xi'$.

2. $\Rightarrow$ If $\xi\downarrow_E$, then due to transparency-restrictedness ξ consists of one transparent name X. This means that the defining equation of X has a $\mathbf{1}$-summand and consequently the translated version has has a $\surd$-summand. Therefore $\xi \xrightarrow{\surd}_{E'} \surd$.

   $\Leftarrow$ If $\xi \xrightarrow{\surd}_{E'} \surd$, then $\xi = X$ for some name X. This means that the defining equation of X has a $\surd$-summand and consequently the original version has a $\mathbf{1}$-summand in E. Therefore, $\xi\downarrow_E$. ■

Using the properties proved in the lemma above, we can establish the decidability result.

THEOREM 4.40. *Let E be a transparency-restricted sequential specification and ξ, χ be sequences of names reachable from some initial name of E. Then it is decidable whether $\xi \leftrightarrows \chi$.* □

PROOF. Let E' be the translated recursive specification $f(E)$. Because of [Srb01] it is decidable whether $\xi \leftrightarrows_\not\downarrow \chi$ in E'. To be able to decide whether $\xi \leftrightarrows \chi$ it suffices to show that $\xi \leftrightarrows \chi$ in E iff $\xi \leftrightarrows_\not\downarrow \chi$ in E'.

$\Rightarrow$ Suppose $\xi \leftrightarrows \chi$. To establish that $\xi \leftrightarrows_\not\downarrow \chi$ it suffices to prove that the relation $\leftrightarrows_\not\downarrow$ satisfies conditions 1 and 2 of Definition 2.4 (on page 10) for all $a \in \mathcal{A}_\surd$ and all $\xi, \chi \in \mathbb{N}^*$. We will first show that condition 1 holds; the proof of the satisfaction of condition 2 then follows symmetrically. For condition 1 we distinguish two cases:

(a) Suppose $a \in \mathcal{A}$. If $\xi \xrightarrow{a}_{E'} \xi'$ and $a \neq \surd$ then by Lemma 4.39(1) we have $\xi \xrightarrow{a}_E \xi'$. Since $\xi \leftrightarrows \chi$ in E, we also have $\chi \xrightarrow{a}_E \chi'$ and $\xi' \leftrightarrows \chi'$ in E. So, by Lemma 4.39(1) we also have $\chi \xrightarrow{a}_{E'} \chi'$ and $\xi' \leftrightarrows \chi'$ in E.

(b) Suppose $a = \surd$. If $\xi \xrightarrow{\surd}_{E'} \surd$ then by Lemma 4.39(2) we have $\xi\downarrow_E$. Since $\xi \leftrightarrows \chi$ in E also $\chi\downarrow_E$ and by Lemma 4.39(2) we have $\chi \xrightarrow{\surd}_{E'} \surd$.

We have shown for all ξ, χ in E that if the pair (ξ, χ) is in the relation $\leftrightarrows$, then conditions 1 and 2 of Definition 2.4 hold and hence $\xi \leftrightarrows_\not\downarrow \chi$ in E'.

$\Leftarrow$ Suppose $\xi \leftrightarrows_\not\downarrow \chi$. To establish that $\xi \leftrightarrows \chi$ it suffices to prove that the relation $\leftrightarrows_\not\downarrow$ is a bisimulation meeting all conditions of Definition 2.4 for all $a \in \mathcal{A}$ and $\xi, \chi \in \mathbb{N}^*$. We distinguish three cases based on the conditions of Definition 2.4:

1. If $\xi \xrightarrow{a}_E \xi'$ then by Lemma 4.39(1) we have $\xi \xrightarrow{a}_{E'} \xi'$ with $a \neq \surd$. Since $\xi \leftrightarrows_\not\downarrow \chi$ in E', we also have $\chi \xrightarrow{a}_{E'} \chi'$ and $\xi' \leftrightarrows_\not\downarrow \chi'$ in E'. So, by Lemma 4.39(1) we also have $\chi \xrightarrow{a}_E \chi'$ and $\xi' \leftrightarrows_\not\downarrow \chi'$ in E'.
2. By an analogous argument as in the previous case.
3. If $\xi\downarrow_E$ then by Lemma 4.39(2) we have $\xi \xrightarrow{\surd}_{E'} \surd$. Since $\xi \leftrightarrows_\not\downarrow \chi$ in E' also $\chi \xrightarrow{\surd}_{E'} \surd$ and by Lemma 4.39(2) we have $\chi\downarrow_E$.

We have shown for any ξ, χ in E that if the pair (ξ, χ) is in the relation $\leftrightarrows_\not\downarrow$, then all conditions of Definition 2.4 hold and hence $\xi \leftrightarrows \chi$. ■

COROLLARY 4.41. *Bisimilarity is decidable on transparency-restricted sequential specifications.* □

In future work, this decidability result could be extended to the decidability of divergence-preserving branching bisimilarity. Preferably we will also find an extension to decidability for the full class of sequential specifications.

4.3 Explicit Interaction

If we consider the definition of the pushdown automaton, we can discern two components: the finite control and the stack memory. The latter of these two components, the stack memory, seems to have a rather informal definition. In the previous section we have seen that the stack, first given as a pushdown automaton and pushdown transition system, can also be defined by a sequential specification. If we put this specification in parallel with a specification representing the finite control, we can make the interaction with the stack within a pushdown automaton more explicit.

We first consider pushdown automata according to the FSES interpretation. We show that we can translate the finite control of a PDA to a linear specification. Once put in parallel with the sequential specification of the stack, we can define all pushdown transition systems according to the FSES interpretation. Thereafter, we shall consider the other direction.

Recall that transparency-restricted sequential specifications are simulated by pushdown automata. Because we can subsequently give specifications for these pushdown automata, consisting of a linear specification in parallel with the sequential specification of a stack, we can say that every transparency-restricted sequential specification can be defined by a linear specification in parallel with a stack. See also [BCT08] for earlier work that investigated the correspondence between sequential specifications and specifications of finite control in parallel with a stack. The paper shows under what circumstances we can extend the set of pushdown transition systems to incorporate transition systems with unbounded branching. A (partially) forgetful stack is used to deal with transparent names on the stack. Note also that the paper does not require the recursive specifications to be transparency-restricted, but at the cost of using a weaker equivalence (namely contrasimulation) in some cases.

We also cannot obtain the same correspondence result for pushdown automata according to the FS interpretation. Following the reasoning as given in the proof of Theorem 4.26 there exists no sequential specification for the always-terminating stack. This is something that is required if we want to put finite control in parallel with this specification and allow for termination whenever the finite control can do so. (Clearly, the FS interpretation only puts termination conditions on the finite control, in contrast with the FSES and ES interpretation that also put conditions on the stack.) We will use a different approach and use a recursive TCP_τ-specification for the stack that can always terminate. This, of course, comes at the cost of losing the link with sequential specifications that we did have for the FSES interpretation.

4.3.1 According to the FSES Interpretation

We will show that, up to divergence-preserving branching bisimilarity, every pushdown automaton can be specified using the process theory TCP_τ. We do this by showing, for any given PDA, the construction of a finite recursive TCP_τ-specification that defines its behaviour. Our specification will consist of a linear specification of a process that is a translated version of the finite control of the PDA, and a sequential specification of stack memory. We shall prove that the parallel composition of these specifications specifies a transition system that is divergence-preserving branching bisimilar with the transition system associated with the PDA. We remark that we actually only use TCP_τ to arrange the communication between the linear finite control process and the sequential stack process.

Below we will give a translation of the finite control of a PDA into a linear specification E_{fc} and then show that, combined with the sequential specification of the stack process E_S, the correspondence with the original PDA M holds. But first, recall the sequential specification E_S of the stack over $\mathcal{D}$:

$$S \stackrel{\text{def}}{=} \mathbf{1} + o!\bot.S + \sum_{d\in\mathcal{D}} i?d.S_{\not\bot} \cdot o!d.S \;,$$

$$S_{\not\bot} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d\in\mathcal{D}} i?d.S_{\not\bot} \cdot o!d.S_{\not\bot} \;.$$

Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a pushdown automaton. By Theorem 4.8 we can assume that M only has push and pop transitions. We can now define the linear specification E_{fc}, capturing the finite control, i.e. the transition relation, of M. For each $s \in \mathcal{S}$ and $d \in \mathcal{D}_\bot$ we add the name $C_{s,d}$. Each name $C_{s,\bot}$ has the following defining equation:

$$C_{s,\bot} \stackrel{\text{def}}{=} \sum_{(s,a,\bot,d,t)\in\rightarrow} a.C_{t,d}\ [+\mathbf{1}]_{s\downarrow} \;,$$

which corresponds to the empty-test (push) transition and termination when the PDA is in state s and the stack is empty. Each name $C_{s,d}$ ($d \in \mathcal{D}$) has the following defining equation:

$$C_{s,d} \stackrel{\text{def}}{=} \sum_{(s,a,d,e,t)\in\rightarrow} a.i!d.C_{t,e} + \sum_{(s,a,d,\varepsilon,t)\in\rightarrow} a.\sum_{e\in\mathcal{D}_\bot} o?e.C_{t,e} \;,$$

which corresponds, respectively, to the push and pop transitions when the PDA is in state s and data element d is on top of the stack.

Note that the top of the stack is not on the stack but retained by the finite control process.

THEOREM 4.42. *For every pushdown automaton M according to the FSES interpretation there exists a recursive TCP_τ-specification E_M and process expression p defined by a linear specification such that $\mathcal{T}(M) \leftrightarrows_b^\Delta \mathcal{T}_{E_M}(\llbracket p \parallel S \rrbracket_{i,o})$.* □

PROOF. We choose $M = E_{fc} \cup E_S$, where E_{fc} is constructed for M as described above. We present some observations from which it is fairly straightforward to establish that $\mathcal{T}(M) \leftrightarrows_{\mathrm{b}}^{\Delta} \mathcal{T}_{E\cup E_S}(\partial_{i,o}(C_{\uparrow,\perp} \parallel S))$. In our proof we abbreviate the process expression $S \cdot i!d_n.S \cdots i!d_1.S$ by $S_{d_n \cdots d_1}$, with, in particular, $S_\varepsilon = S$. (Recall the infinite specification of the stack given on page 54.)

First, note that whenever $\mathcal{T}(M)$ has a transition $(s,d) \xrightarrow{a} (t,\varepsilon)$, then

$$\partial_{i,o}(C_{s,d} \parallel S_\varepsilon) \xrightarrow{a} \partial_{i,o}((\sum_{e\in\mathcal{D}_\perp} o?e.C_{t,e}) \parallel S_\varepsilon) \xrightarrow{o!\!?\perp} \partial_{i,o}(C_{t,\perp} \parallel S_\varepsilon) \ .$$

The abstraction operator $\tau_{i,o}(\_)$ will rename the transition labelled $o!\!?\perp$ into a τ-transition. So,

$$\partial_{i,o}(C_{s,d} \parallel S_\varepsilon) \xrightarrow{a} \left[(\sum_{e\in\mathcal{D}_\perp} o?e.C_{t,e}) \parallel S_\varepsilon\right]_{i,o} \xrightarrow{\tau} \partial_{i,o}(C_{t,\perp} \parallel S_\varepsilon) \ .$$

This τ-transition is *inert* in the sense that it does not preclude any observable behaviour that was possible before the τ-transition. Such inert τ-transitions can be omitted while preserving branching bisimilarity.

Second, note that whenever $\mathcal{T}(M)$ has a transition $(s,d\zeta) \xrightarrow{a} (t,\zeta)$ with ζ nonempty, say $\zeta = e\zeta'$, then

$$\partial_{i,o}(C_{s,d} \parallel S_\zeta) \xrightarrow{a} \xrightarrow{o!\!?e} \partial_{i,o}(C_{t,e} \parallel S_{\zeta'}) \ ,$$

and, since the second transition is the only step possible after the first a-transition, the τ-transition resulting from applying $\tau_{i,o}(\_)$ is again inert.

Third, note that whenever $\mathcal{T}(M)$ has a transition $(s,d\zeta) \xrightarrow{a} (t,ed\zeta)$, then

$$\partial_{i,o}(C_{s,d} \parallel S_\zeta) \xrightarrow{a} \xrightarrow{i!\!?d} \partial_{i,o}(C_{t,e} \parallel S_{d\zeta}) \ ,$$

and again the τ-transition resulting from applying $\tau_{i,o}(\_)$ is inert.

Finally, note that whenever $\mathcal{T}(M)$ has a transition $(s,\varepsilon) \xrightarrow{a} (t,e)$, then

$$\partial_{i,o}(C_{s,\perp} \parallel S) \xrightarrow{a} \partial_{i,o}(C_{t,e} \parallel S_\varepsilon) \ .$$

Only, single inert τ-steps are removed, no τ-loops are introduce nor removed. Therefore, we have that divergence is preserved. ■

Now, for the other direction. We can show that if we have a process defined by a linear specification that communicates with a stack, we can find a PDA that simulates the behaviour of the two specifications put in parallel.

THEOREM 4.43. *For every linear specification E and linear process expression p there exists a pushdown automaton M according to the FSES interpretation such that $\mathcal{T}_{E\cup E_S}(\partial_{i,o}(p \parallel S)) \leftrightarrows_b^{\Delta} \mathcal{T}(M)$.* □

PROOF. Let E be a linear specification and let p be a linear process expression. We define a pushdown automaton M as follows:

- The set of states, the action alphabet, and the initial and final states are the same as those of the transition system $\mathcal{T}_E(p)$ (which is a finite automaton).
- The data alphabet is the set of data elements $\mathcal{D}$ of the presupposed recursive specification of a stack.
- Whenever $s \xrightarrow{a} t$ in $\mathcal{T}_E(p)$, and $a \neq i!d, o?d$ ($d \in \mathcal{D}$), then $s \xrightarrow{a[\bot/\varepsilon]} t$ and $s \xrightarrow{a[d/d]} t$ for all $d \in \mathcal{D}$;
- whenever $s \xrightarrow{i!d} t$ for some $d \in \mathcal{D}$ in $\mathcal{T}_E(p)$, then $s \xrightarrow{\tau[\bot/d]} t$ and $s \xrightarrow{\tau[e/de]} t$ for all $e \in \mathcal{D}$;
- whenever $s \xrightarrow{o?d} t$ for some $d \in \mathcal{D}$ in $\mathcal{T}_E(p)$, then $s \xrightarrow{\tau[d/\varepsilon]} t$.

We omit the proof that every transition of $\mathcal{T}_{E\cup E_S}(\llbracket p \parallel S \rrbracket_{i,o})$ can be matched by a transition in $\mathcal{T}(M)$ in the sense required by the definition of divergence-preserving branching bisimilarity. ■

We have seen in Section 4.2.1 that (transparency-restricted) sequential specifications can be simulated by a PDA. We have also seen above that each PDA can be defined by a linear specification for the finite control of the PDA and a sequential specification of stack memory, combined in a single specification that allows for communication between both components. Indirectly, we have established that each (transparency-restricted) sequential specification can be written as a linear specification communicating with a stack. Therefore, we can consider the stack, with its sequential specification, as the canonical sequential process.

COROLLARY 4.44. *For every transparency-restricted sequential specification E and sequential expression p there exists a linear specification E_{fc} and linear process expression q such that $\mathcal{T}_E(p) \leftrightarrows_b^\Delta \mathcal{T}_{E_{fc}\cup E_S}(\llbracket q \parallel S \rrbracket_{i,o})$.* □

PROOF. The result follows from Theorems 4.35 and 4.42. ■

The same result was obtained directly for opaque sequential specifications and also for all sequential specifications but for a weaker equivalence, namely contrasimulation, in [BCT08].

4.3.2 According to the FS Interpretation

If we want to make the interaction explicit in a pushdown automaton according to the FS interpretation, we need a stack that can always terminate. As was mentioned before, there is no sequential specification for such a stack. Instead, we present a new stack process that can terminate regardless of its contents. This finite recursive TCP_τ-specification is inspired by the specification of a queue proposed by Baeten and Bergstra in [BB88], which has in turn its origins in the CSP book by Hoare [Hoa85]. It is similar to the tape process that we will see later on in Chapter 6; the stack can be seen as a one-sided tape of which we may only inspect and/or replace the top element.

DEFINITION 4.45. The recursive TCP_τ-specification $E_{S^\downarrow}$ of the *always-terminating stack* over $\mathcal{D}$, with initial name $S^{i,o}_{j,p}$, is defined as follows:

$$S^{i,o}_{j,p} \overset{\text{def}}{=} \mathbf{1} + o!\bot.S^{i,o}_{j,p} + \sum_{d\in\mathcal{D}} i?d.\left[T^{i,o}_{j,p}\, d \parallel S^{j,p}_{i,o}\right]_{j,p},$$

$$T^{i,o}_{j,p}d \overset{\text{def}}{=} \mathbf{1} + o!d.\sum_{f\in\mathcal{D}_\bot} p?f.T^{i,o}_{j,p}f + \sum_{e\in\mathcal{D}} i?e.j!d.T^{i,o}_{j,p}e \qquad (d\in\mathcal{D}),$$

$$T^{i,o}_{j,p}\bot \overset{\text{def}}{=} \mathbf{1} + o!\bot.T^{i,o}_{j,p}\bot + \sum_{d\in\mathcal{D}} i?d.T^{i,o}_{j,p}d,$$

$$T^{j,p}_{i,o}d \overset{\text{def}}{=} \mathbf{1} + o!d.\sum_{f\in\mathcal{D}_\bot} p?f.T^{j,p}_{i,o}f + \sum_{e\in\mathcal{D}} i?e.j!d.T^{j,p}_{i,o}e \qquad (d\in\mathcal{D}),$$

$$T^{j,p}_{i,o}\bot \overset{\text{def}}{=} \mathbf{1} + o!\bot.T^{j,p}_{i,o}\bot + \sum_{d\in\mathcal{D}} i?d.T^{j,p}_{i,o}d. \qquad \triangle$$

Because this stack needs to be a drop-in replacement for our earlier defined stack, it has the same interface: it also receives data elements that are pushed over channel i, sends data elements that are popped over channel o, and can signal over channel o if the stack is empty.

The first time the stack receives a data element, it splits into a top element retaining the data element in parallel with the empty stack. From this moment on, every time a data element is received, a new top element is split off "to the right" to retain the data element that is being replaced by the newly received data element. See Figure 4.21 for a diagram of the always-terminating stack process; depicted is the state when a data element 1 has been pushed.

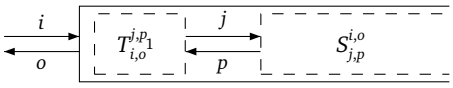

FIGURE 4.21: Diagram of the always-terminating stack specification.

.

If a data element is popped from the left-most top element, all data elements move one position to the left as well. See for example the following trace where data elements 1 and 0 are pushed and then popped:

$$S^{i,o}_{j,p} \xrightarrow{i?0}\!\!\!\!\rightarrow \left[T^{i,o}_{j,p}0 \parallel S^{i,o}_{j,p}\right]_{j,p} \xrightarrow{i?1}\!\!\!\!\rightarrow \left[T^{i,o}_{j,p}1 \parallel \left[T^{j,p}_{i,o}0 \parallel S^{j,p}_{i,o}\right]_{i,o}\right]_{j,p}$$
$$\xrightarrow{o!1}\!\!\!\!\rightarrow \left[T^{i,o}_{j,p}0 \parallel \left[T^{j,p}_{i,o}\bot \parallel S^{j,p}_{i,o}\right]_{i,o}\right]_{j,p} \xrightarrow{o!0}\!\!\!\!\rightarrow \left[T^{i,o}_{j,p}\bot \parallel \left[T^{j,p}_{i,o}\bot \parallel S^{j,p}_{i,o}\right]_{i,o}\right]_{j,p}$$

At the end, we are left with two empty cells. However, it can easily be shown that $\left[T^{i,o}_{j,p}\bot \parallel S^{j,p}_{i,o}\right]_{j,p} \leftrightarrow\!\!\!\!\!\!-^{\Delta}_{\mathrm{b}}\; S^{i,o}_{j,p}$. Thus, the empty cells can be collapsed and removed.

We now reconsider the correspondence results we had for the FSES interpretation, for the FS interpretation. If we go from FSES to ES, we drop the empty stack

requirement; termination needs to happen if the finite control can terminate. We can obtain our results by just replacing the terminating-on-empty stack by the always-terminating stack defined above.

THEOREM 4.46. *For every pushdown automaton M according to the FS interpretation there exists a recursive TCP_τ-specification E_M and process expression p such that $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E_M}(\llbracket p \parallel S^{\downarrow} \rrbracket_{i,o})$.* □

PROOF. We choose $M = E_{fc} \cup E_{S^{\downarrow}}$, where E_{fc} is constructed for M as described in Section 4.3.1. The result follows from Theorem 4.42 and the fact that we use $E_{S^{\downarrow}}$ instead of E_S. ■

Now, for the other direction.

THEOREM 4.47. *For every linear specification E and linear process expression p there exists a pushdown automaton M according to the FS interpretation such that $\mathcal{T}_{E \cup E_{S^{\downarrow}}}(\llbracket p \parallel S \rrbracket_{i,o}) \leftrightarrows_b^{\Delta} \mathcal{T}(M)$.* □

PROOF. The result follows from Theorem 4.43 and the fact that we use $E_{S^{\downarrow}}$ instead of E_S. ■

4.4 Conclusions

In this chapter we have investigated the classical correspondence result between pushdown automata and context-free grammars. To be able to treat this result in a process-theoretic setting, we have associated pushdown transition systems with pushdown automata. In the literature [Sud88, Sip97, HMU06] two distinct termination conditions for pushdown automata are considered: termination on empty stack (ES) and on final state (FS). We have additionally considered termination on both final state and empty stack (FSES). It is well-known that up to language equivalence it does not matter which termination condition is used as they all yield the same class. We can obtain the pushdown languages if we take the pushdown transition systems up to language equivalence. Figure 4.22 gives a schematic overview of the classical correspondence results.

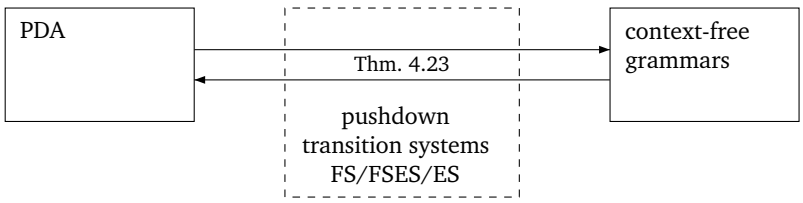

FIGURE 4.22: Classical correspondence results from automata theory.

If we reconsider all results up to (divergence-preserving) branching bisimilarity, we get a much more contrived picture. First, we get different classes of pushdown

transition systems if we take different termination conditions. The class according to the ES interpretation is, up to divergence-preserving branching-bisimilarity, a proper subclass of the class according to the FSES interpretation. Only if consider pushdown automata that are initially terminating, then the class of pushdown transition systems according to the ES interpretation coincides with the class according to the FSES interpretation. The class according to the FSES interpretation is, also up to divergence-preserving branching bisimilarity, a proper subclass of the class according to the FS interpretation. Because of the class differences, we have considered the correspondence results for the FSES and FS classes separately.

We have seen that in our process-theoretic setting context-free grammars can be defined as finite recursive TSP_τ-specifications, which we call sequential specifications. To obtain the correspondence between pushdown automata and sequential specifications we have applied two restrictions. First, we only consider transparency-restricted sequential specifications as a way to prevent unbounded branching. Secondly, we ensure that the pushdown automata are pop choice-free, because it can be shown that there exist non-pop choice-free pushdown automata for which there is no sequential specification. If these two restrictions are applied, we can obtain a correspondence.

Because transparency-restricted sequential specifications play an important role in this chapter, we can wonder if we can decide if two sequential specifications have the same associated transition system up to divergence-preserving branching bisimilarity. We have shown that this is the case for (strong) bisimilarity, extending earlier work for BPA- and $\mathsf{BPA}_\mathbf{0}$-specifications, which are specifications in subtheories of TSP_τ.

We have chosen to translate λ-productions (or ϵ-productions) in context-free grammars by $\mathbf{1}$-summands in sequential specifications. This is mainly done to stay in line with the previous chapter and allow for intermediate termination. However, a different choice could have been to use τ-summands instead. In this case the resulting specification language would always generate opaque sequential specifications and thus have a full correspondence with pushdown automata according to the (FS)ES interpretation.

From a process-theoretic perspective it makes sense to make the interaction in a PDA explicit. We can do this by giving a linear specification representing the finite control of the PDA and put it in parallel with a specification of a stack, allowing communication over an input and output channel for pushing and popping. We have first established this correspondence for pushdown automata according to the FSES interpretation.

Figure 4.23 presents a schematic overview of the correspondence results for the FSES interpretation from a process-theoretic point of view. Note that there is an indirect correspondence between transparency-restricted sequential specifications and the explicit interaction. Because the stack can be defined by a transparency-restricted sequential specification, and all transparency-restricted sequential specifications can be given as a finite-state process communicating with this stack, the stack can be considered as the canonical sequential process.

For the FS interpretation we have seen that there exist pushdown transition systems that have no sequential specification. Hence, we lack a correspondence

result in this case. Note that if we have a PDA that has a pushdown transition system according the FS interpretation that can also be given according to the FSES interpretation, we of course do have a correspondence as described above. The pushdown transition system for the stack according to the FS interpretation also has no sequential specification. Therefore, we resort to a TCP_{τ}-specification of the stack to make the interaction explicit.

See Figure 4.24 for a schematic overview of the correspondence results according to the FS interpretation. Note that, clearly, we also lack the indirect correspondence result between sequential specifications and the explicit interaction.

4.4.1 Future Work

First of all, transparency-restrictedness is too strict. There are finite sequential specifications that are not transparency-restricted but do not have unbounded branching. It should be possible to find a syntactic requirement on sequential specifications such that just a finite sequence of transparent names can be stacked.

On the side of the pushdown automata it is unknown if one can generate pushdown transition systems, up to branching bisimilarity, with or without divergence-preservation, that have unbounded but finite branching. Additionally, we have also not been able to establish that our result is optimal in the sense that a pushdown process is definable by a sequential specification only if it is pop choice-free, although we conjecture that this is the case.

In the previous chapter we have seen that the class of deterministic finite automata accepts the same languages as the class of non-deterministic finite automata, but forms a subclass with respect to branching bisimilarity. It is known that for pushdown automata the languages accepted by deterministic PDAs is a subclass of the languages accepted by non-deterministic PDAs. Intuitively, this is probably also be the case up to branching bisimilarity. However, it would be worthwhile to define deterministic PDAs in our framework and investigate this result using pushdown transition systems.

In [BCT08] we have shown that sequential specifications with unbounded branching can have a correspondence, up to contrasimulation, with a finite-state process communicating with a (partially) forgetful stack. These results could be split up as follows: first a correspondence between sequential specifications and PDAs with a special kind of termination, namely on final state and when the stack contains zero-or-more transparent data elements, and then a correspondence between these PDAs with a special kind of termination and a finite control put in parallel with the (partially) forgetful stack.

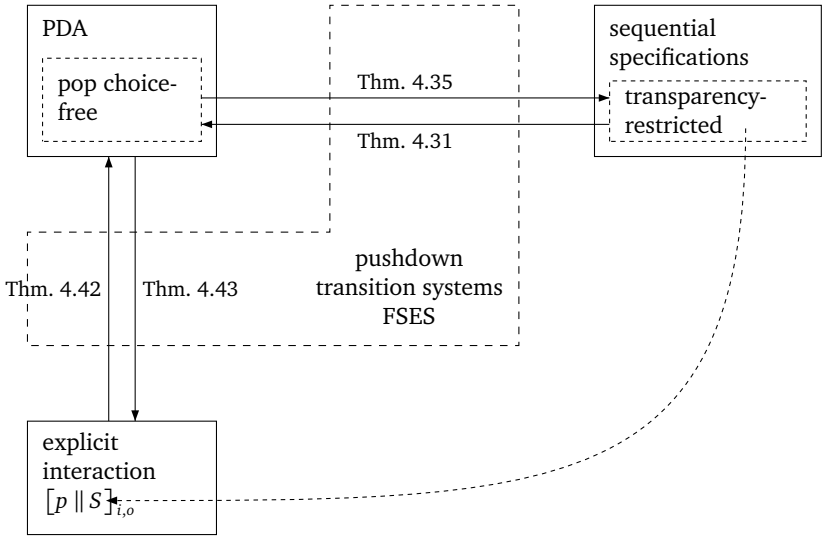

FIGURE 4.23: Correspondence results for the FSES interpretation.

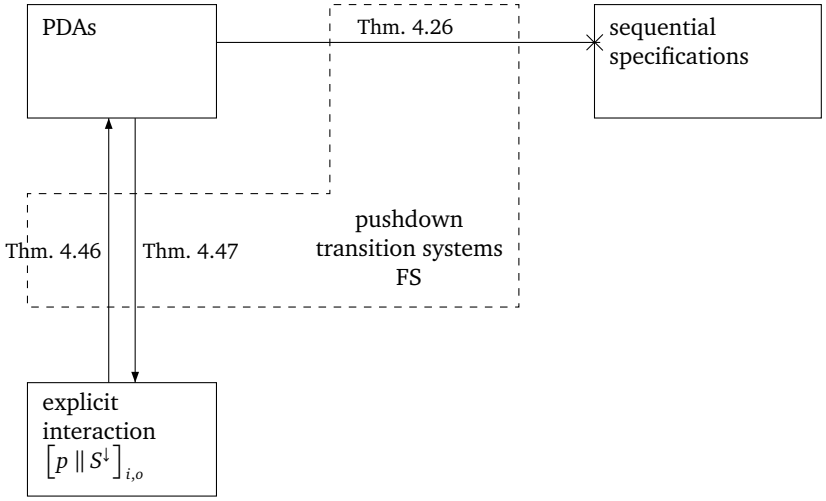

FIGURE 4.24: Correspondence results for the FS interpretation.

Chapter 5

Parallel Pushdown Systems

In this chapter we discuss systems that are a variation on the pushdown systems described in the preceding chapter. Pushdown systems are based on the notion of a pushdown automaton, consisting of finite control and a stack memory. On the stack memory data elements are stored in a sequence and one can only inspect, retrieve, or stack on the top element. In this chapter, we will modify the memory to enable the retrieval of a specific element regardless of its place in the sequence: we let go of the ordered structure and view the sequence of data elements as commutative, i.e. all elements are stored "in parallel"; the memory effectively becomes a bag. We call a pushdown automaton where the stack memory is replaced by a bag memory a *parallel pushdown automaton*. This notion was originally defined by [Mol96] for the class of parallel labelled rewrite systems, i.e. rewrite systems modulo commutativity of concatenation. It has also been called "bag automaton" and "multiset automaton", but we prefer the original name, as it emphasises the relation to pushdown automata, its parallel nature, and not the type of memory that has been used to define or to implement it.

In Section 5.1 we define the parallel pushdown automaton and its associated transition systems. In the definition of the parallel pushdown automaton, the stack memory is replaced by the bag memory. We shall discuss the consequences of this adaptation. Then, similarly as in the previous chapter, we shall investigate different termination conditions: termination on empty bag (EB), on final state (FS), and on both final state and empty bag (FSEB). We will see that the class of pushdown transition systems with termination on empty bag is, up to divergence-preserving branching bisimilarity, a proper subclass of the class with termination on both final state and empty bag. Furthermore, the class with termination on both final state and empty bag is incomparable to the class with termination on final state, again up to divergence-preserving branching bisimilarity. Note that these results are different from what we have seen for pushdown automata in Section 4.1.1 (see also Figure 4.14 on page 50 for the overview).

In Section 5.2 we revisit the correspondence between pushdown automata and context-free grammars, but now in our parallel/bag-oriented setting. We define our commutative context-free grammars as finite recursive BCP_τ-specifications, which

we call basic parallel specifications. Here, the adjective "basic" refers to the fact that we do not allow for communication between parallel components. We will show that opaque and transparent basic parallel specifications can be simulated by parallel pushdown automata, according to the FSEB and FS interpretation respectively. In the case of a specification that is not completely opaque nor transparent we will introduce a new termination condition to the parallel pushdown automaton: termination on both final state and transparent bag (FSTB). The bag is considered to be transparent if it only contains data elements that are marked as transparent. We show that for this termination condition we can simulate any basic parallel specification with a parallel pushdown automaton. For the other direction it was shown by Christensen in [Chr93] that only single-state parallel pushdown automata can be given, up to language equivalence, by a basic parallel specification. We will restrict ourselves to this small subclass of automata and show how they can be defined by basic parallel specifications.

We also investigate the decidability of strong bisimilarity on processes defined by basic parallel specifications. We obtain our results by extending earlier results for recursive specifications over BPP, which is a subtheory of BCP_τ. Christensen, Hirshfeld and Moller proved in [CHM93] that bisimilarity is decidable on processes definable in BPP. The bulk of their proof consists of defining a sound and complete tableau proof system for proving whether two BPP-definable processes are bisimilar. In this section we adapt their tableau proof system with the constant $\mathbf{1}$ to prove decidability of bisimilarity on processes definable by a basic parallel specification. We find that the adaptation requires a careful treatment of the distinction between successful and unsuccessful termination, but it does not result in the kind of difficulties we encountered in the case of sequential specifications. In Section 4.2.2 we only obtained a decidability result for a subclass of the sequential specifications: the transparency-restricted sequential specifications. We shall prove that our extension of the original decidability result for recursive BPP-specifications holds for all basic parallel specifications.

In Section 5.3 we make the communication between the finite control and the bag in a parallel pushdown automaton explicit. We show that every parallel pushdown automaton can be defined by a finite recursive TCP_τ-specification consisting of a linear specification representing the finite control and a specification of a bag process. Depending on the chosen termination condition we use a variant of the bag process defined by a basic parallel specification. The bag may therefore be considered as the canonical process for this class of specifications.

Some material in this chapter is inspired by the following publication:

[BCT09] J. C. M. Baeten, P. J. L. Cuijpers, and P. J. A. van Tilburg. "A Basic Parallel Process as a Parallel Pushdown Automaton". In: *Proceedings of EXPRESS 2008*. Ed. by D. Gorla and T. Hildebrandt. ENTCS 242. Elsevier, 2009, pp. 35–48.

5.1 Parallel Pushdown Automata

Before we start with the definition of the parallel pushdown automaton, we recap the notion of multisets and introduce the notation used in this chapter.

A multiset over some set of elements X, denoted by $\mathcal{M}(X)$, is a function from X to the natural numbers $\mathbb{N}$. For a multiset μ we write $\mu(a) = n$ when the element a occurs n times in μ. For two multisets μ, ν we write $\mu \uplus \nu$ to denote union of multisets such that $(\mu \uplus \nu)(a) = \mu(a) + \nu(a)$. We denote the difference of multisets $\mu - \nu$ such that $(\mu - \nu)(a) = \mu(a) - \nu(a)$ under the assumption that $\mu(a) \geq \nu(a)$. Furthermore, we use $a \in \mu$ to denote the statement that $\mu(a) \geq 1$, and $\mu \subseteq \nu$ to denote that $\mu(a) \leq \nu(a)$ for all a. The multiset $\emptyset$ is the empty multiset, i.e. $\emptyset(a) = 0$ for all a. If the elements of a multiset are enumerated, they are written in between double brackets, e.g. $[\![a, c, a, b]\!]$, analogous to set element enumeration. The singleton multiset is denoted by $[\![a]\!]$.

In the literature, a multiset is also often referred to as a bag. To avoid confusion, we use the term "multiset" to refer to the mathematical object described above and the term "bag" to refer to the type of memory that stores a multiset.

We use a definition of the parallel pushdown automaton that is very similar to the definition of the pushdown automaton (Definition 4.1 on page 39). The main difference is the implicit replacement of the stack memory by the bag memory and subsequently the usage of multisets of symbols instead of strings.

Interestingly, there is more to the replacement of the stack memory by the bag memory. First of all, in the case of the pushdown automaton, transitions can be taken based on the current state and top element of the stack. Since there is no fixed order in the bag memory, it does not have a top element; it is possible to remove any element. (Note that, in the case of the bag, we talk about inserting and removing, rather than pushing and popping.) So, transitions in a parallel pushdown automaton are taken based on the current state and whether some data element $d \in \mathcal{D}$ is available in the bag. Secondly, when the stack is empty a pop of the top element is not possible. Due to its sequential structure, stack memory can been easily equipped with an empty-test: it returns a special symbol ($\bot$) if it is empty when popped. We choose not to equip the bag memory with an empty-test. We will later see that if if we want to be able to define the bag by means of the parallel operator, it has no sequential structure; it cannot tell by itself if it is empty. The only way to check that it is empty would be to try to remove each type of data element and count. Thirdly, recall that pushdown transitions traditionally consist of an action, a removal and an insertion. However, since in case of parallel pushdown transitions removals are impossible when the bag is empty, which we cannot determine, we should allow for pushdown transitions without removal. Therefore, we augment the set of data elements $\mathcal{D}$ with the special symbol $*$ to signify that we do not remove a data element from the bag, assuming that $* \notin \mathcal{D}$; we denote the set $\mathcal{D} \cup \{*\}$ of *bag symbols* by $\mathcal{D}_*$.

Taking these considerations into account, we define the parallel pushdown automaton – inspired by Moller's definition in [Mol96] – as follows.

DEFINITION 5.1. A *parallel pushdown automaton* (PPDA) M is defined as a six-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ where

1. $\mathcal{S}$ is a finite set of states;
2. $\mathcal{A}$ a finite set of actions;
3. $\mathcal{D}$ a finite set of data;
4. $\rightarrow \subseteq \mathcal{S} \times \mathcal{A}_\tau \times \mathcal{D}_* \times \mathcal{M}(\mathcal{D}) \times \mathcal{S}$ is an $\mathcal{A}_\tau \times \mathcal{D}_* \times \mathcal{M}(\mathcal{D})$-labelled transition relation on $\mathcal{S}$,
5. $\uparrow \in \mathcal{S}$ is the initial state, and
6. $\downarrow \subseteq \mathcal{S}$ is the set of final states. △

Similarly to Definition 4.1 (on page 39), if $(s, a, d, \mu, t) \in \rightarrow$, we write $s \xrightarrow{a[d/\mu]} t$. But now the intuitive meaning of this transition is that if the parallel pushdown automaton M is in state s and can remove a data element d from (anywhere in) the bag, then it may do so while performing the action a, replacing datum d by the multiset of data μ and moving to state t. In the case that $d = *$, we have a transition of the form $s \xrightarrow{a[*/\mu]} t$, which means that if M is in state s, it can insert the multiset of data μ into the contents of the bag while performing the action a and moving to state t without inspecting or taking anything from the bag.

In the previous chapter we discussed different termination conditions of the pushdown automata in Section 4.1.1 and compared the mutual relation of the classes of pushdown transition systems, up to (divergence-preserving) branching bisimilarity, according to the ES, FSES and FS interpretation. In this chapter we have the analogous notions for PPDAs with termination on empty bag (EB), final state and empty bag (FSEB), and termination on final state (FS).

EXAMPLE 5.2. Assume that $\mathcal{A} = \{a, b, c\}$ and $\mathcal{D} = \{1\}$. The state-transition diagram in Figure 5.1 specifies a parallel pushdown automaton that can perform a-actions while inserting a data element 1 in the bag for each a-action. When a data element 1 is available in the bag, the parallel pushdown automaton can, in both states, perform a b-action while removing this data element. Only after the c-action is performed, the interleaving of inserting and removing of the data element 1 stops and only the choice to remove and execute the b-action remains. For clarity, the set of data is confined to only one element.

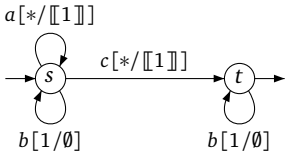

FIGURE 5.1: An example of a parallel pushdown automaton.

Observe that this parallel pushdown automaton is nearly the same as the example of a pushdown automaton in Figure 4.18 (on page 60) that is not pop choice-free. The only minor difference is that the transitions $s \xrightarrow{a[\bot/\llbracket 1 \rrbracket]} s$ and $s \xrightarrow{a[1/\llbracket 11 \rrbracket]} s$, of

which the first uses the empty-test of the stack which is unavailable for the bag, are replaced by the transition $s \xrightarrow{a[*/[\![1]\!]]} s$, an insert transition that does not inspect nor take anything from the bag.

If we disregard these minor difference, we can see that if $\mathcal{D}$ consists of one data element, more specifically if only one type of data element is inserted, then the class of pushdown automata coincides with the class of parallel pushdown automata. This is because a multiset over a set of one element is equal in use to a set or sequence.

Depending on the adopted acceptance condition, the parallel pushdown automaton in Figure 5.1 accepts the language $\{wcw' \mid w \in \{a,b\}^*, w' \in \{b\}^* \wedge \#_a(w)+1 \geq \#_b(w)+\#_b(w')\}$ (FS), or the language $\{wcw' \mid w \in \{a,b\}^*, w' \in \{b\}^* \wedge \#_a(w)+1 = \#_b(w)+\#_b(w')\}$ (FSEB), and for EB we get the same language as for FSEB but it additionally accepts the empty word. ♢

To formalise the intuitive behaviour of pushdown automata, we associate with every PPDA M a transition system $\mathcal{T}(M)$. For the states of this associated transition system we use configurations as defined as follows.

DEFINITION 5.3. A *configuration* of a parallel pushdown automaton M is a pair (s,μ) consisting of a state $s \in \mathcal{S}$, and bag contents (multiset) $\mu \in \mathcal{M}(\mathcal{D})$. △

The associated transition system semantics of PPDAs defines an $\mathcal{A}_\tau$-labelled transition relation on configurations such that a PPDA-transition $s \xrightarrow{a[d/\mu]} t$ corresponds with an a-labelled transition from a configuration consisting of the PPDA-state s and bag contents $[\![d]\!] \uplus \nu$, to a configuration consisting of the PPDA-state t and the bag contents $\mu \uplus \nu$, i.e. the original bag contents with the data element d replaced by the multiset μ.

DEFINITION 5.4. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a parallel pushdown automaton. The *transition system* $\mathcal{T}(M)$ *associated with* M is defined as follows:

1. the set of states of $\mathcal{T}(M)$ is the set of configurations $\mathcal{S} \times \mathcal{M}(\mathcal{D})$;
2. the transition relation of $\mathcal{T}(M)$ satisfies
   a) $(s, [\![d]\!] \uplus \nu) \xrightarrow{a} (t, \mu \uplus \nu)$ iff $s \xrightarrow{a[d/\mu]} t$ for all $s,t \in \mathcal{S}$, $a \in \mathcal{A}_\tau$, $d \in \mathcal{D}$, $\mu, \nu \in \mathcal{M}(\mathcal{D})$, and
   b) $(s,\nu) \xrightarrow{a} (t, \mu \uplus \nu)$ iff $s \xrightarrow{a[*/\mu]} t$;
3. the initial state of $\mathcal{T}(M)$ is $(\uparrow, \emptyset)$; and
4. for the set of final states $\downarrow$ we consider three alternative *termination conditions*:
   a) $(s,\nu)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ (the FS interpretation),
   b) $(s,\nu)\downarrow$ in $\mathcal{T}(M)$ iff $\nu = \emptyset$ (the EB interpretation), and
   c) $(s,\nu)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ and $\nu = \emptyset$ (the FSEB interpretation).

A transition system is a *parallel pushdown transition system (according to the FS/EB/FSEB interpretation)* if it is associated with a PPDA (according to the same interpretation). △

This definition now gives us the notions of parallel pushdown language and parallel pushdown process.

DEFINITION 5.5. A language accepted by a parallel pushdown transition system is called a *parallel pushdown language*.

A *parallel pushdown process (according to the FS/FSEB/EB interpretation)* is a divergence-preserving branching bisimilarity class of transition systems containing a parallel pushdown transition system. △

EXAMPLE 5.6. Recall the example PPDA in Figure 5.1. The transition system associated with this PPDA (according to the FSEB interpretation) is shown in Figure 4.19 (on page 60). ♢

Due to the presence of the special symbol $*$ in PPDA transitions, the notion of insert and remove transitions differs slightly from the notions of push and pop transitions for a PDA.

DEFINITION 5.7. Let $s, t \in \mathcal{S}$ be states of some parallel pushdown automaton M. An *insert transition* is a transition of the form $s \xrightarrow{a[*/[\![d]\!]]} t$ $(d, e \in \mathcal{D})$; a *remove transition* is a transition of the form $s \xrightarrow{a[d/\emptyset]} t$ $(d \in \mathcal{D})$. △

THEOREM 5.8. *For every PPDA M there exists a PPDA M' that uses only insert and remove transitions such that $\mathcal{T}(M) \leftrightarrows_b^{\triangle} \mathcal{T}(M')$.* □

PROOF. It is easy to see that limiting the set of transitions to insert and remove transitions only in the definition of a parallel pushdown automaton yields the same notion of a parallel pushdown transition system up to divergence-preserving branching bisimilarity:

1. Eliminate a transition of the form $s \xrightarrow{a[*/\emptyset]} t$ by adding a fresh state s', replacing the transition by two transitions $s \xrightarrow{a[*/[\![d]\!]]} s' \xrightarrow{\tau[[\![d]\!]/\emptyset]} t$ (with d some arbitrary element in $\mathcal{D}$, assuming that $\mathcal{D} \neq \emptyset$).
2. Eliminate a transition of the form $s \xrightarrow{a[*/\mu]} t$, where $\mu = [\![d_1]\!] \uplus \cdots \uplus [\![d_n]\!]$ $(n > 1)$ for some randomly picked order of data elements, by adding new states $s_2, \ldots, s_n$ and replacing the transition $s \xrightarrow{a[*/\mu]} t$ by the sequence of transitions

$$s \xrightarrow{a[*/[\![d_1]\!]]} s_2 \xrightarrow{\tau[*/[\![d_2]\!]]} \cdots \xrightarrow{\tau[*/[\![d_{n-1}]\!]]} s_n \xrightarrow{\tau[*/[\![d_n]\!]]} t \ .$$

3. Eliminate a transition of the form $s \xrightarrow{a[d/\mu]} t$, where $\mu = [\![d_1]\!] \uplus \cdots \uplus [\![d_n]\!]$ $(n \geq 1)$ for some randomly picked order of data elements, by adding new states $s_1, \ldots, s_n$ and replacing the transition $s \xrightarrow{a[d/\mu]} t$ by transitions $s \xrightarrow{a[d/\emptyset]} s_1$ and the sequence of transitions

$$s_1 \xrightarrow{\tau[*/[\![d_1]\!]]} s_2 \xrightarrow{\tau[*/[\![d_2]\!]]} \cdots \xrightarrow{\tau[*/[\![d_{n-1}]\!]]} s_n \xrightarrow{\tau[*/[\![d_n]\!]]} t \ .$$

Observe that we only get a finite number of additional inert τ-transitions in the associated transition system. ■

Analogously with the stack of a PDA, the bag of a PPDA can also be defined by a parallel pushdown automaton. Given the finite set of data $\mathcal{D}$, the bag has an input

channel i over which it can receive elements of $\mathcal{D}$ and an output channel o over which it can send elements of $\mathcal{D}$.

The bag is defined by a parallel pushdown automaton with one state $\uparrow$ (which is both initial and final) and the transitions $\uparrow \xrightarrow{i?d[*/[\![d]\!]]} \uparrow$ and $\uparrow \xrightarrow{o!d[d/\emptyset]} \uparrow$ for all $d \in \mathcal{D}$. The associated transition system according to the (FS)EB interpretation of the bag over $\mathcal{D} = \{0, 1\}$ is shown in Figure 5.2. Put in contrast with the pushdown transition system with the stack (see Figure 4.3 on page 43), note the absence of the empty test and that we have a grid rather than a tree.

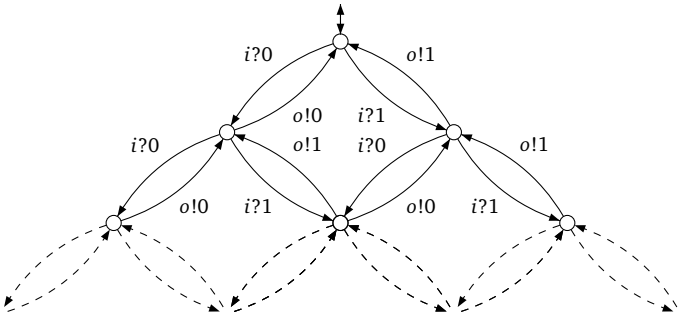

FIGURE 5.2: Bag over $\mathcal{D} = \{0, 1\}$.

If we want to model the bag that always terminates, i.e. that terminates regardless of its contents, we can use the PPDA specified above but then consider the associated transition system according to the FS interpretation. This transition system will be isomorphic with the transition system in Figure 5.2 but each state is final.

5.1.1 Termination Conditions

Recall the results of the differences between classes of pushdown transitions systems according to the FS, FSES and ES interpretations shown in the previous chapter. (See Figure 4.14 on page 50 for the overview.) We shall now investigate the relation between the different classes of parallel pushdown transition systems according to the FS, FSEB and EB interpretations.

FS and FSEB

In the case of the classes of FSEB and EB we can obtain similar results as we have for FSES and ES.

THEOREM 5.9. *For each parallel pushdown transition system according to the EB interpretation there is, up to divergence-preserving bisimilarity, a parallel pushdown transition system according to the FSEB interpretation.* □

PROOF. Let T be the parallel pushdown transition system associated with some PPDA M according to the EB interpretation. Let M' be the PPDA obtained from M by declaring all its states final. Then T is isomorphic with the transition system associated with M' according to the FSEB interpretation. ■

In the other direction we have a result similar as in Example 4.10 (on page 44): transition systems associated with parallel pushdown automata that are not initially terminating cannot be divergence-preserving branching bisimilar with any pushdown transition system according to the EB interpretation.

EXAMPLE 5.10. There exists a pushdown transition system according to the FSEB interpretation such that there is no pushdown transition system according to the EB interpretation that is branching bisimilar with it.

Consider the parallel pushdown automaton M in Figure 5.3. Observe that the initial state of this PPDA is not a final state.

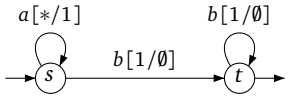

FIGURE 5.3: A parallel pushdown automaton that is not initially terminating.

The associated transition system $\mathcal{T}(M)$ according the FSEB interpretation (see Figure 5.4 below) does not have a initial state which is also final.

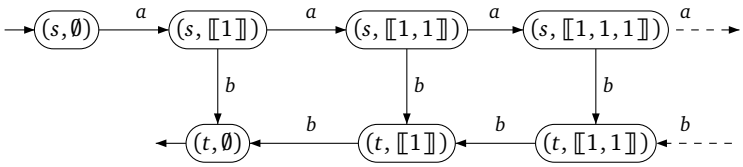

FIGURE 5.4: The transition system associated with the PPDA that is not initially terminating according to the FSEB interpretation.

Because the bag of a PPDA is empty in the initial state by definition, every transition system associated with a PPDA according to the EB interpretation has an initial state which is also a final state. Therefore, there cannot exist a parallel pushdown transition system according to the EB interpretation that is branching bisimilar to the parallel pushdown transition system in Figure 5.4. ♢

For parallel pushdown automata that are initially terminating, we have the same result as for pushdown automata in Example 4.11 (on page 45). The construction described in the proof of that theorem uses a dummy symbol ∅ to control the moment the stack becomes empty. This way it is only allowed to go from a final state where the stack would have been empty to a branching bisimilar, but not divergence-preserving branching bisimilar, state where it really becomes empty. We use a similar technique for parallel pushdown automata, with two differences: we do not have to take the empty-test into account, and we cannot ensure that the bag is really empty because we can reach and remove the dummy symbol at any time. This leads to a slightly simpler construction.

THEOREM 5.11. *For each parallel pushdown transition system according to the FSEB interpretation associated with a PPDA that is initially terminating, there is, up to branching bisimilarity, a parallel pushdown transition system according to the EB interpretation.* □

PROOF. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be some parallel pushdown automaton that is initially terminating. We shall modify M such that the transition system associated with the modified parallel pushdown automaton according to the EB interpretation is branching bisimilar to the transition system associated with M according to the FSEB interpretation. We define the modified parallel pushdown automaton $M' = (\mathcal{S}', \mathcal{A}, \mathcal{D}', \rightarrow', \uparrow', \emptyset)$ as follows:

1. $\mathcal{S}'$ is obtained from $\mathcal{S}$ by adding a fresh initial state $\uparrow'$, and also a fresh state $s^{\downarrow}$ for every final state $s \in \downarrow$;
2. $\mathcal{D}'$ is obtained from $\mathcal{D}$ by adding a fresh dummy symbol $\varnothing$,
3. $\rightarrow'$ is obtained from $\rightarrow$ by
   a) adding a transition $(\uparrow', \tau, *, [\![\varnothing]\!], \uparrow)$,
   b) adding transitions $(s, \tau, \varnothing, \emptyset, s^{\downarrow})$ and $(s^{\downarrow}, \tau, *, [\![\varnothing]\!], s)$ for every $s \in \downarrow$.

Note that the modification of M only introduces inert τ-transitions in the transition system associated with M'. We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((\uparrow, \emptyset), (\uparrow', \emptyset))\} \cup \{((s, \mu), (s, \mu \uplus [\![\varnothing]\!])) \mid s \in \mathcal{S}, \mu \in \mathcal{M}(\mathcal{D})\} \cup \{((s, \mu), (s^{\downarrow}, \mu)) \mid s \in \downarrow, \mu \in \mathcal{M}(\mathcal{D})\}$$

is a branching bisimulation between the transition system associated with M according to the FSEB interpretation and the transition system associated with M' according to the EB interpretation. ■

This modification introduces divergence, as it is possible to infinitely often remove and reinsert the dummy symbol. For PDAs we were able to modify the construction using the empty-test to obtain a result that also preserved divergence, as shown in Theorem 4.12 (on page 46). As we do not have the empty-test in PPDAs, we conjecture that the analogous result for PPDAs does not hold.

CONJECTURE 5.12. *There exists no parallel pushdown transition system according to the EB interpretation that is divergence-preserving branching bisimilar with the parallel pushdown transition system according to the FSEB interpretation associated with the PPDA in Figure 5.3.* □

FSEB and FS

For classes of parallel pushdown transition systems according to the FS and FSEB interpretation we have a slightly different result than for the classes of pushdown transitions systems according to the FS and FSES interpretation: FS and FSEB are incomparable even up to branching bisimilarity.

EXAMPLE 5.13. As an example consider the parallel pushdown automaton shown in Figure 5.5. (This example is the parallel pushdown version of the pushdown automaton in Figure 4.11 on page 49.)

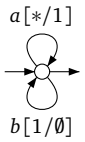

FIGURE 5.5: The counter parallel pushdown automaton.

Let us now assume that there exists a parallel pushdown automaton M that has an associated transition system according the FS interpretation that is branching bisimilar with the associated transition system according to the FSEB interpretation shown in Figure 5.6 below. Let the b-norm of a configuration be the number of b-transitions that can be performed, without performing intermediate a-transitions, until termination can occur.

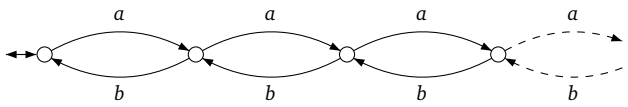

FIGURE 5.6: The transition system associated with the automaton of Figure 5.5 according to the FSEB interpretation.

Because the transition system associated with M is infinite, we can say, without loss of generality, that there exists a state s of M that is infinitely often revisited when performing a-transitions without intermediate b-transitions. Now, let us consider this infinite sequence of configurations with state s. Dickson's Lemma (see [Dic13]) implies that for every infinite sequences of vectors of natural numbers, we have that there exist indices i and j such that $\vec{x}_i \leq \vec{x}_j$ in a point-wise fashion. Because we can consider multisets as vectors of natural numbers, it follows that there are two configurations (s,μ) and (s,ν) in $\mathcal{T}(M)$ such that $\mu \subseteq \nu$. (E.g. let ν be $\mu \uplus \kappa$.) Let the b-norm of the configuration (s,μ) be n and let m be the number of a-transitions necessary get from (s,μ) to (s,ν).

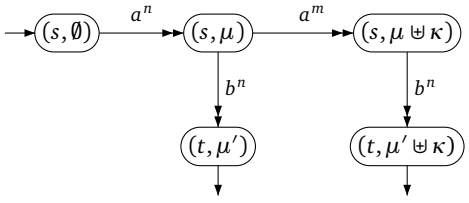

FIGURE 5.7: Schematic overview of an attempted counter PPDA using the FS interpretation.

However, the b-norm of the configuration (s,ν) is also n, because the automaton can go to some terminating state t from state s using only bag contents μ. This

should be $m + n$ if it was branching bisimilar with the associated transition system in Figure 5.6. See Figure 5.7 for a schematic overview of what happens if we try to use the FS interpretation to count.

Hence, there exists no PPDA that has an associated transition system according to the FS interpretation that is branching bisimilar with the associated transition in Figure 5.6. ♢

In the other direction we have a result similar as in Example 4.15 (on page 49).

EXAMPLE 5.14. Reconsider the counter PPDA depicted in Figure 5.5. The associated transition system according to the FS interpretation is the same as for the pushdown version (see Figure 4.12 on page 50). The reason that there is no parallel pushdown transition system according to the FSEB interpretation follows the same argument as in Example 4.15: a parallel pushdown transition system according to the FSEB interpretation has finitely many terminating states, for the PPDA has only finitely many states and the bag needs to be empty, while a parallel pushdown transition system according to the FS interpretation can have infinitely many. ♢

The following mutual relations between the classes up to (divergence-preserving) branching bisimilarity have been established. (See Figure 5.8 for a schematic overview. Note that in the diagram FSEB^{it} stands for the class of transition systems according to the FSEB interpretation associated with initially-terminating PPDAs. Also note that, because the PPDA in Example 5.13 is initially terminating, the example also implicitly shows that EB $\not\subseteq$ FS and therefore the arrow is drawn from EB.)

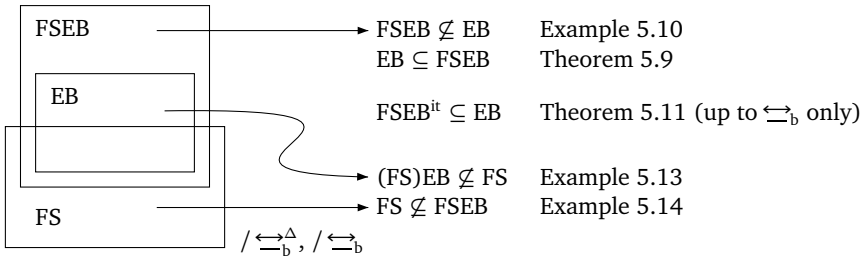

FIGURE 5.8: Overview of the different classes of parallel pushdown transition systems.

COROLLARY 5.15. *The class of parallel pushdown transition systems according to the EB interpretation is a proper subclass, up to divergence-preserving branching bisimilarity, of the class of parallel pushdown transition systems according to the FSEB interpretation.*

The class of parallel pushdown transition systems according to the FSEB interpretation is incomparable with, up to (divergence-preserving) branching bisimilarity, the class of parallel pushdown transition systems according to the FS interpretation. □

Not depicted in Figure 5.8 is the fact that for pushdown transitions that are initially terminating, the class according to the FSEB interpretation is the same, up to branching bisimilarity, as the class according to the EB interpretation.

Note that, had we equipped the bag memory with an empty-test, we would have gotten the same picture as Figure 4.14 on page 50. A similar construction as in the proof of Theorem 4.14 could then show that the class of parallel pushdown transition systems according to the FSEB interpretation is, up to divergence-preserving branching bisimilarity, a proper subclass of the class according to the FS interpretation. Also a similar construction as the proof of Theorem 4.12 could then show that the class of parallel pushdown transition systems according to the FSEB interpretation associated with initially-terminating PPDAs is, up to divergence-preserving branching bisimilarity, equal to the class according to the EB interpretation.

5.2 Basic Parallel Specifications

In Section 4.2 we have seen the sequential process expressions and specifications, which were expressions and finite recursive specifications over TSP_τ. If we replace the sequential composition in TSP_τ by parallel composition we get the subtheory BCP_τ (Basic Communicating Processes) of TCP_τ. We can look upon this specification language as the process-theoretic counterpart of a commutative version of the context-free grammars. We assume that the communication function γ is everywhere undefined. This class of specifications is an extension of BPP (Basic Parallel Processes), introduced by Bergstra and Klop in [BK85] and more thoroughly studied by Christensen in [Chr93]. In [Srb01], Srba extended BPP with deadlock. Here, we will extend it further with the constant $\mathbf{1}$.

DEFINITION 5.16. A *basic parallel specification* over some finite set of names $\mathcal{N}$ is a finite recursive BCP_τ-specification, i.e. a recursive specification over $\mathcal{N}$ in which only the constructions $\mathbf{0}$, $\mathbf{1}$, N ($N \in \mathcal{N}$), $a._-$ ($a \in \mathcal{A}_\tau$), $\_ \parallel \_$ (with an undefined communication function) and $\_ + \_$ are used to build *basic parallel process expressions*. △

EXAMPLE 5.17. The process expression N defined in the basic parallel specification

$$N \stackrel{\text{def}}{=} a.(N \parallel b.\mathbf{1}) + c.\mathbf{1}$$

specifies the parallel pushdown transition system according to the FSEB interpretation in Figure 4.19 (on page 60), which is associated with the parallel pushdown automaton in Figure 5.1. ♢

Our basic parallel specifications can be brought into Greibach normal form. We can define a normal form for basic parallel specifications if we instantiate Definition 2.19 (on page 19) with the sequence of names interpreted as a parallel composition of names.

DEFINITION 5.18. A basic parallel specification E is in *basic parallel normal form* if each defining equation of name $N \in \mathcal{N}$ is of the following form:

$$N \stackrel{\text{def}}{=} \sum_{i \in \mathcal{I}_N} a_i.\xi_i \; (+ \mathbf{1}) \; .$$

In this form, every right-hand side of every defining equation consists of a number of summands, indexed by a finite set $\mathcal{I}_N$ (the empty sum is $\mathbf{0}$), each of which is either $\mathbf{1}$, or of the form $a_i.\xi_i$ with $a_i \in \mathcal{A}_\tau$ and ξ_i a parallel composition of names; the empty parallel composition is denoted by $\mathbf{1}$. △

All basic parallel specifications can be brought in basic parallel normal form. For if we disregard the commutative nature of the parallel composition, we essentially have a sequential specification, i.e. a context-free grammar, for which it is well-known that they can be brought in sequential normal form.

PROPOSITION 5.19. *For each basic parallel specification E and basic parallel process expression p there exists a basic parallel specification in basic parallel normal form E' such that $\mathcal{T}_{E'}(p) \leftrightarrows_b^\Delta \mathcal{T}_E(p)$.* □

We can associate transition systems with basic parallel specifications according to the operational rules in Table 2.1 (on page 15). This gives us also the notion of basic parallel process.

DEFINITION 5.20. A *basic parallel process* is a divergence-preserving branching bisimilarity class of labelled transition systems containing a transition system associated with a basic parallel specification and basic parallel process expression. △

Basic parallel processes were originally defined by Christensen in [Chr93] as the class of processes over a signature including the terminated process, action prefixing, choice and parallel composition. In this thesis we also allow intermediate termination and deadlock.

5.2.1 Correspondence

Example 5.17 already suggests a correspondence between the transition systems associated with basic parallel specifications and parallel pushdown transition systems. We shall investigate the exact nature of this relation in the rest of this section.

Let us first consider a prominent PPDA or parallel pushdown transition system that can be defined by a basic parallel specification. Recall the parallel pushdown transition system according to the (FS)ES interpretation of a bag shown in Figure 5.2.

The following infinite recursive specification E_B^∞ specifies, for the multiset μ, the behaviour of the process B_μ modelling a bag with as contents the multiset of data elements μ that receives input over channel i, i.e. when data is inserted, and sends output over channel o, i.e. when data is removed. For the empty bag, we have:

$$B_\emptyset \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.B_{[\![d]\!]} \; ,$$

and for every non-empty multiset $\mu \in \mathcal{M}(\mathcal{D})$:

$$B_\mu \stackrel{\text{def}}{=} \sum_{d \in \mu} o!d.B_{\mu - [\![d]\!]} + \sum_{e \in \mathcal{D}} i?e.B_{\mu \uplus [\![e]\!]} \ .$$

However, we would like our bag to have a finite version of this specification to obtain a basic parallel specification.

DEFINITION 5.21. The following basic parallel specification defines a bag that can terminate when it is empty:

$$B_{i,o} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.(B_{i,o} \parallel o!d.\mathbf{1}) \ ,$$

which has the same associated transition system, up to isomorphism, as the one shown in Figure 5.2; we refer to this specification of a bag over $\mathcal{D}$ as E_B. △

It can be shown by RSP that the infinite and finite specification yield the same bag process. For the proof we refer to [BW90, Theorem 3.5.3]. Note that the proof is without **1**-summands, but it can easily be extended.

LEMMA 5.22. *We have that* $B_\emptyset \leftrightarrow_b^\Delta B_{i,o}$. □

Note that only the bag PPDA according to the FSEB interpretation is given by the basic parallel specification above. If we consider the bag PPDA according to the FS interpretation, we get the bag that can always terminate, i.e. it can terminate regardless of its contents. The state of the bag when it contains data elements $d_1, \ldots, d_n$ be characterised by a parallel composition, for example: $B_{i,o} \parallel o!d_1.\mathbf{1} \parallel \ldots \parallel o!d_n.\mathbf{1}$ An obvious modification to make E_B always terminating would be to ensure that each parallel component has a **1**-summand so that termination is always possible.

To obtain a specification for the always terminating bag, all we have to do is add **1**-summands to each defining equation of E_B^∞ to obtain a recursive specification $E_{B^t}^\infty$ of a *transparent bag*.

DEFINITION 5.23. The finite version of this specification, E_{B^t} can be defined as follows:

$$B_{i,o}^t \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.\big(B_{i,o}^t \parallel (o!d.\mathbf{1} + \mathbf{1})\big) \ . \qquad \triangle$$

The transition system associated with the specification of the transparent bag above is, up to strong bisimilarity, equal to the transition system associated with the bag PPDA according to the FS specification (see Figure 5.2 and consider it with all states marked final). This is unlike the specification of the forgetful stack (see Definition 4.25 on page 55), that had an associated transition system (see Figure 4.3 on page 43) that was up to branching bisimilarity not equal at all to the transition system associated with the bag PDA according to the FS interpretation.

Note that the specifications of the bag and transparent bag can be easily brought in basic parallel normal form. We just have to replace, for all $d \in \mathcal{D}$, the parallel components, $o!d.\mathbf{1}$ and $o!d.\mathbf{1} + \mathbf{1}$ respectively, by some name E_d with the component itself as the process term of the defining equation.

Now, for the correspondence between parallel pushdown automata and basic parallel specifications, let us first consider the direction from basic parallel specifications to parallel pushdown automata. We will do this in three steps and show up to branching bisimilarity that: opaque specifications can be simulated by parallel pushdown automata according to the FSEB interpretation, transparent specifications by parallel pushdown according to the FS interpretation, and mixed specifications according to the FSTB interpretation introduced below.

Recall that for a recursive specification over a finite set of names $\mathcal{N}$ a name is called transparent if its defining equation has a $\mathbf{1}$-summand; it is called opaque otherwise. Thus we can partition $\mathcal{N}$ into the transparent names $\mathcal{N}^{+1}$ and the opaque names $\mathcal{N}^{-1}$. A recursive specification is transparent if all its names are transparent; it is opaque if all its names are opaque.

Opaque specifications

We can give a construction in a similar way as for the simulation of transparency-restricted specifications by pushdown automata shown in the proof of Theorem 4.35 (on page 62). Let us consider an example first.

EXAMPLE 5.24. Let E be the following basic parallel specification:

$$X \stackrel{\text{def}}{=} a.(X \parallel Y) + b.Y + c.\mathbf{1}\,,$$
$$Y \stackrel{\text{def}}{=} d.\mathbf{1}\,.$$

This specification is in basic parallel normal form and opaque. Figure 5.9 depicts a parallel pushdown automaton that simulates E up to divergence-preserving branching bisimilarity if we take X as its initial name and use the FSEB interpretation.

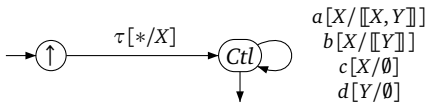

FIGURE 5.9: A parallel pushdown automaton simulating basic parallel specification E.

We have an initial state that puts the initial name in the bag when moving to the state Ctl that handles the control based on the contents of the bag. For each summand of a name in the specification we have a corresponding PPDA transition, labelled with the action of the prefix, that removes the name and inserts all names that are in parallel after the summand of the prefix. For example, for the summand $a.(X \parallel Y)$ of the defining equation of X, we add the transition $Ctl \xrightarrow{a[X/[\![X,Y]\!]]} Ctl$. ♢

The following theorem establishes a complete version of the construction.

THEOREM 5.25. *For each opaque basic parallel specification E, with initial name I, there exists a parallel pushdown automaton M according to the FSEB interpretation such that* $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_E(I)$. □

PROOF. Let E be a basic parallel specification over a finite set of names $\mathcal{N}$, and let I be an initial name of E. By Proposition 5.19 we can assume that E is in basic parallel normal form and that all states in the associated transition system are denoted with multisets of names. We define a parallel pushdown automaton $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ as follows:

1. $\mathcal{S}$ consists of state $\uparrow$ and *Ctl*.
2. $\mathcal{A}$ consists of all the actions occurring in E.
3. $\mathcal{D}$ consists of the names occurring in E.
4. $\rightarrow$ is defined as follows:
   a) for the initial name $I \in \mathcal{N}$, $\rightarrow$ has the transition $\uparrow \xrightarrow{\tau[*/[\![I]\!]]} \mathit{Ctl}$
   b) for each summand $a.\xi$, where $\xi \in \mathcal{M}(\mathcal{N})$ is a parallel composition of names, in the right-hand side of the defining equation for a name N, $\rightarrow$ has the transition $\mathit{Ctl} \xrightarrow{a[N/\xi]} \mathit{Ctl}$.
5. $\uparrow$ is the initial state,
6. $\downarrow$ consists of the state *Ctl*.

Note that the only the only τ-transition introduced in the transition system associated with M is inert. We leave it to the reader to verify that the relation

$$\mathcal{R} = \{(I, (\uparrow, \emptyset))\} \cup \{(\xi, (\mathit{Ctl}, \xi)) \mid \xi \in \mathcal{M}(\mathcal{N})\}$$

is a divergence branching bisimulation between the transition system associated with the basic parallel specification E for the initial name I and the transition system associated with M according to the FSEB interpretation. ■

Transparent specifications

Now, if we have a transparent specification, each defining equation of a name has a **1**-summand. This means that termination is possible in every state of the transition system associated with a transparent specification. If we use a PPDA to simulate this specification, in a similar way as we have shown above, a multiset of names is stored in the bag. However, since all these names are transparent, we should be able to terminate at any moment during the simulation. Hence, by just choosing termination on final state instead of on both final state and empty bag we can obtain the desired result. Note that it should also be possible to always reach a final state. This is the case for our simulator PPDA, as one can always move to the state *Ctl* by means of an inert silent step.

THEOREM 5.26. *For each transparent basic parallel specification E, with initial name I, there exists a parallel pushdown automaton M according to the FS interpretation such that* $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_E(I)$. □

PROOF. The proof follows the lines of the proof of Theorem 5.25. Only now $\mathcal{R}$ is a branching bisimulation between the transition system associated with the basic parallel specification E for the initial name I and the transition system associated with M according to the FS interpretation. ■

Mixed opaque/transparent specifications

We have just seen that for opaque specifications we require for the simulation that the bag is empty before termination can occur. For the transparent specifications we drop the empty bag requirement as we know that during simulation the bag always contains transparent names, i.e. names that may be skipped. However, if we have mixed opaque/transparent specifications, the bag may contain both opaque and transparent names during simulation. So, we would like that the PPDA only terminates if it is in a final state and the bag *only* contains transparent names.

We add the termination condition on final state and transparent bag to the definition of transition systems associated with a PPDA.

DEFINITION 5.27. Let $\mathcal{D}^{-1} \subseteq \mathcal{D}$ be the data elements that are considered to be opaque, and $\mathcal{D}^{+1} = \mathcal{D} \setminus \mathcal{D}^{-1}$ the data elements that are transparent.

If M is a parallel pushdown automaton and $\mathcal{T}(M)$ its associated transition system, then $(s, \nu)\downarrow$ in $\mathcal{T}(M)$ iff $s\downarrow$ and $\nu(d) = 0$ for all $d \in \mathcal{D}^{-1}$ (the *FSTB interpretation*). △

Note that if we define $\mathcal{D}^{-1}$ to be empty (and thus $\mathcal{D}^{+1} = \mathcal{D}$), we obtain termination on final state; the stack can only contain transparent data elements and the requirement $\nu(d) = 0$ ($d \in \mathcal{D}^{-1}$) is always met. If we define $\mathcal{D}^{-1}$ to be equal to $\mathcal{D}$, we obtain termination on both final state and empty stack; the stack can only contain opaque data elements and the requirement $\nu(d) = 0$ ($d \in \mathcal{D}^{-1}$) is only met if $\nu = \emptyset$. However, if $\mathcal{D}^{-1}$ nor $\mathcal{D}^{+1}$ is empty, we conjecture the following.

CONJECTURE 5.28. *There exists a pushdown transition system according to the FSTB interpretation such that there is no pushdown transition system according to the FS nor to the FSEB interpretation that is branching bisimilar with it.* □

The class of pushdown transition systems according to the FSTB interpretation is incomparable to the class according to the FS and FSEB interpretation, as a result.

To simulate a mixed opaque/transparent specification we can again reuse the construction described in the proof of Theorem 5.25.

THEOREM 5.29. *For each basic parallel specification E, with initial name I, there exists a parallel pushdown automaton M according to the FSTB interpretation such that* $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_E(I)$. □

PROOF. The proof follows the lines of the proof of Theorem 5.25. We not only define that $\mathcal{D} = \mathcal{N}$, but also that $\mathcal{D}^{+1} = \mathcal{N}^{+1}$ and thus $\mathcal{D}^{-1} = \mathcal{N}^{-1}$.

Again, $\mathcal{R}$ is a branching bisimulation between the transition system associated with the basic parallel specification E for the initial name I and the transition system associated with M, but this time according to the FSTB interpretation. ■

Note that this result includes the previous two results for opaque and transparent specifications. Indeed, we can take either $\mathcal{D}^{+1}$ or $\mathcal{D}^{-1}$ to be empty and use the preceding correspondence result.

The results for all three classes of specifications hold to up to branching bisimilarity. We think that it should be possible to obtain the result up to divergence-preserving branching bisimilarity by storing additional information in the bag. We leave this to future work.

Now, for the other direction, we have to determine how an arbitrary PPDA can be defined by a basic parallel specification. However, Christensen has shown in [Chr93] that this cannot be done for a simple PPDA such as the one shown in Example 5.10. This is due to the fact that a PPDA with a single state cannot be found for the language accepted by the PPDA in Figure 5.3, i.e. $\{a^n b^n \mid n \geq 1\}$. So, we proceed with the restriction that a PPDA must have a single state and obtain a rather weak correspondence between PPDAs and basic parallel specifications. Note, however, that the constructed PPDA in Example 5.24 is almost single-state, were it not that we have to put the initial variable in the bag.

EXAMPLE 5.30. Consider the counter parallel pushdown automaton in Figure 5.5 that has a single state.

Now, consider the following basic parallel specification that defines this PPDA:

$$N_* \stackrel{\text{def}}{=} \mathbf{1} + a.N_* \parallel N_1 \ ,$$
$$N_1 \stackrel{\text{def}}{=} b.\mathbf{1} \ ;$$

the initial name of this specification is N_*. The associated transition system has been depicted in Figure 5.10.

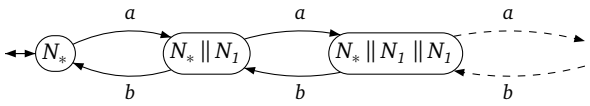

FIGURE 5.10: The transition system associated with the basic parallel specification defining the counter PPDA.

The associated transition system above is isomorphic with the associated transition system of the counter PPDA according to the FSEB interpretation (see also Figure 5.6). If we want the same correspondence for the FS interpretation we have to add an extra $\mathbf{1}$-summand to the defining equation of N_1. ♢

We can generalise this example to a more formal construction and obtain the following result.

THEOREM 5.31. *For every single-state parallel pushdown automaton M there exists a basic parallel specification E, with initial name I, such that $\mathcal{T}_E(I) \leftrightarrows_b^{\Delta} \mathcal{T}(M)$.* □

PROOF. Let $M = (\{\uparrow\}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be a single-state parallel pushdown automaton. We define a basic parallel specification E with a name N_* with the following defining equation:

$$N_* \stackrel{\text{def}}{=} \mathbf{1} + \sum_{(\uparrow, a, *, [\![d_1, \ldots, d_n]\!], \uparrow)} a.N_* \parallel N_{d_1} \parallel \ldots \parallel N_{d_n} ,$$

and for every data element $d \in \mathcal{D}$ a name N_d with the following equation:

$$N_d \stackrel{\text{def}}{=} \sum_{(\uparrow, a, d, [\![d_1, \ldots, d_n]\!], \uparrow)} a.N_{d_1} \parallel \ldots \parallel N_{d_n} ,$$

Note that for transitions that insert nothing, the resulting (empty) parallel composition $N_{d_1} \parallel \ldots \parallel N_{d_n}$ is denoted by $\mathbf{1}$. We choose N_* as the initial name. In case we interpret M according to the FS interpretation, we add a $\mathbf{1}$-summand to each defining equation. We leave it to the reader to verify that the relation

$$\mathcal{R} = \{((\uparrow, [\![d_1, \ldots, d_n]\!]), N_* \parallel N_{d_1} \parallel \ldots \parallel N_{d_n}) \mid d_1, \ldots, d_n \in \mathcal{D}\}$$

is a divergence-preserving branching bisimulation and hence $\mathcal{T}_E(N_*) \leftrightarrows_{\text{b}}^{\Delta} \mathcal{T}(M)$. ■

5.2.2 Decidability

In [CHM93] a tableau decision method is presented to show the decidability of bisimulation equivalence on processes defined by BPP, a subtheory of BCP_τ. In this section, we extend this tableau decision method so that it can also deal with the empty process and the deadlocked process. Similarly as for sequential specifications, we only consider the decidability of strong bisimilarity in this part; we leave the extension to branching bisimilarity (preferably divergence-preserving) to future work. We will briefly discuss the methods, lemmas and theorems involved with using the tableau decision method to decide bisimilarity and mainly focus on the parts where adaptations are needed due to the presence of the constants $\mathbf{0}$ and $\mathbf{1}$.

The main difference is that the constant $\mathbf{0}$ in the paper of Christensen, Hirshfeld and Moller is the identity element for both alternative and parallel composition, while in our setting $\mathbf{0}$ is the identity element for the alternative composition and $\mathbf{1}$ is the identity element for parallel composition. This subtle difference gives rise to some adjustments of the decision method and related proofs:

- In our setting, $\mathbf{0}$ is not the identity element for parallel composition. For example consider the process expression $p = a.\mathbf{1}$. It is clear that $p \parallel \mathbf{0} \leftrightarrows a.\mathbf{0}$, which is not bisimilar to p; the deadlocked process cannot be removed from a parallel composition. We have to ensure that the proof system treats deadlock as a non-removable term.
- Conversely, $\mathbf{1}$ is not an identity element for the alternative composition. To determine if $p + \mathbf{1}$ is bisimilar to q, we have to check that q has a termination option, and thus a $\mathbf{1}$-summand, too.

– Finally, we have a form of synchronised termination in the case of parallel composition. That is, a parallel composition can terminate if all of its components can terminate.

Besides its role as the identity element for parallel composition, the empty process $\mathbf{1}$ allows us to have transparent names in a recursive specification. In the previous chapter we have seen that having transparent names can lead to unbounded branching in the transition system that can be associated with this specification. A requirement for the proof below is that the transition systems associated with the basic parallel specifications have bounded branching. The example below illustrates why this is the case; we refer to [BCT09, Corollary 4.5] for a formal argument.

EXAMPLE 5.32. Let us reconsider Example 4.27 (on page 56), but we replace sequential composition by parallel composition:

$$X \stackrel{\text{def}}{=} a.(X \parallel Y) + b.\mathbf{1}\,,$$
$$Y \stackrel{\text{def}}{=} c.\mathbf{1} + \mathbf{1}\,.$$

(For convenience, we use in this chapter X^n to denote an n-fold parallel composition of X, e.g. $X^3 = X\|X\|X$.) Also in this case the process Y^i can terminate and can perform a c-transition which leads to Y^{i-1}. However, it is not possible to "skip" a name by executing a c-transition from Y^i with $0 \leq i$ and go to Y^j with $0 \leq j < i-1$.$\diamondsuit$

Deciding strong bisimilarity

The tableau decision method is a goal-directed proof system. The method uses inference rules of the form

$$\textit{rule name} \qquad \frac{p = q}{p_1 = q_1 \;\cdots\; p_n = q_n}\; C\,,$$

where p and q are process expressions and C an optional side-condition. The premise $p = q$ is the goal to be achieved whereas the consequents $p_1 = q_1, \ldots, p_n = q_n$ are the subgoals to be established. A *tableau* is a maximal proof tree using a specified set of rules. The rules we use here are shown in Table 5.1. These rules are the ones given in [CHM93] supplemented with the rule SumT to handle the case that there are $\mathbf{1}$-summands along with the summation. When building a tableau and applying the rules, we refer to each premise and/or goal as a *node*. For an example of a tableau, see Example 5.34 later on.

The rule Rec takes care of applying the recursive definition of the name while at the same time *unfolding* a parallel composition. Let $Y_j \stackrel{\text{def}}{=} \sum_{i\in\mathcal{I}_{Y_j}} a_{j,i}.\xi_{j,i}\ (+\,\mathbf{1})$ for $1 \leq j \leq n$. We define the function $\mathrm{unf}_{\mathbf{1}}$, used in the rule Rec to represent the unfolding of $\xi = Y_1 \parallel \ldots \parallel Y_n$, as follows:

$$\mathrm{unf}_{\mathbf{1}}(\xi) = \sum_{j=1}^{n} \sum_{i\in\mathcal{I}_{Y_j}} a_{j,i}.\big(Y_1 \parallel \ldots \parallel Y_{j-1} \parallel \xi_{j,i} \parallel Y_{j+1} \parallel \ldots \parallel Y_n\big)\ [+\,\mathbf{1}]_{\xi\in(\mathcal{N}^{+1})^*}\,.$$

Rec	$\dfrac{\xi = \chi}{\mathrm{unf}_1(\xi) = \mathrm{unf}_1(\chi)}$	
Sum	$\dfrac{\sum_{i=1}^{n} a_i.\xi_i = \sum_{j=1}^{m} b_j.\chi_j}{\{a_i.\xi_i = b_{f(i)}.\chi_{f(i)}\}_{i=1}^{n} \quad \{b_j.\chi_j = a_{g(j)}.\xi_{g(j)}\}_{j=1}^{m}}$ (*) where $f : \{1,\ldots,n\} \mapsto \{1,\ldots,m\}$, $g : \{1,\ldots,m\} \mapsto \{1,\ldots,n\}$	
SumT	$\dfrac{\sum_{i=1}^{n} a_i.\xi_i + \mathbf{1} = \sum_{j=1}^{m} b_j.\chi_j + \mathbf{1}}{\{a_i.\xi_i = b_{f(i)}.\chi_{f(i)}\}_{i=1}^{n} \quad \{b_j.\chi_j = a_{g(j)}.\xi_{g(j)}\}_{j=1}^{m}}$ (**) where $f : \{1,\ldots,n\} \mapsto \{1,\ldots,m\}$, $g : \{1,\ldots,m\} \mapsto \{1,\ldots,n\}$	
Prefix	$\dfrac{a.\xi = a.\chi}{\xi = \chi}$	
SubL	$\dfrac{\xi \parallel \eta = \rho}{\chi \parallel \eta = \rho}$	if $\chi \sqsubset \xi$ and there is a dominated node labelled $\xi = \chi$ or $\chi = \xi$
SubR	$\dfrac{\rho = \xi \parallel \eta}{\rho = \chi \parallel \eta}$	if $\chi \sqsubset \xi$ and there is a dominated node labelled $\xi = \chi$ or $\chi = \xi$

TABLE 5.1: The extended tableau rules.

After applying the Rec rule, one can match summands using the Sum or SumT rule and remove matching prefixes using the Prefix rule. Before applying the Rec rule again, we need to perform a substitution using the SubL and SubR rules on the current node if they can be applied. This is possible if there is a node upward in the tree, called a *dominated node*, with $\xi = \chi$ or $\chi = \xi$ such that $\chi \sqsubset \xi$ for some well-founded ordering $\sqsubset$ that is defined in Definition 5.33 below.

We denote constructed tableaux by $T(\xi = \chi)$ where $\xi = \chi$ is the label of the root; we denote paths by π and nodes by $\mathbf{n}$, possibly with a subscript. If a node is labelled $\xi = \chi$ we write $\mathbf{n} : \xi = \chi$.

Rules may only be applied to nodes that are not *terminal*. A node is terminal if it is either a successful or unsuccessful terminal node. A *successful terminal node* is one labelled either $\xi = \xi$ where ξ may be $\mathbf{1}$ (we assume that the empty multiset

denotes $\mathbf{1}$) or $\mathbf{0} = \mathbf{0}$. We have an *unsuccessful terminal node* if no rule can be applied. The Prefix rule cannot be applied if there is a prefix mismatch, i.e. $a.\xi = b.\chi$ and $a \neq b$. It can also be that the Sum rule cannot be applied, for example when $a.\xi = \mathbf{0}$ or $\mathbf{0} = b.\chi$ or that the SumT cannot be applied because one side has a $\mathbf{1}$-summand but the other side does not. The rules SubL or SubR cannot be applied if the dominated nodes needed for substitution are missing.

To check whether $\xi \leftrightarrows \chi$ holds, we try to find a tableau with $\xi = \chi$ as the root node. If the tableau only has successful terminal nodes, we call it a successful tableau and we have shown that ξ and χ are bisimilar. They are not bisimilar if none of the possible tableaux is successful.

We have to show that the application of rules in a tableau always eventually stops. To show that each tableau is finite, and that there are finitely many tableaux we require a well-founded ordering on the multisets of names. The ordering is used in the side-conditions of the SubL and SubR rules.

In the definition of this ordering we assume that there is some fixed total order on the names: $\mathcal{N} = \{N_1, \ldots, N_n\}$.

DEFINITION 5.33. We define a well-founded (lexicographical) ordering on all multisets of parallel compositions of names $\mathcal{N}$ as follows:

$$N_1^{k_1} \parallel \ldots \parallel N_n^{k_n} \sqsubset N_1^{l_1} \parallel \ldots \parallel N_n^{l_n}$$

iff there exists j such that $k_j < l_j$ and for all $i < j$ we have $k_i = l_i$. △

EXAMPLE 5.34. Let us consider the following recursive specification:

$$N_1 \stackrel{\text{def}}{=} a.(N_2 \parallel N_3) + b.\mathbf{1} + \mathbf{1}\,,$$
$$N_2 \stackrel{\text{def}}{=} a.(N_2 \parallel N_4) + c.\mathbf{1}\,,$$
$$N_3 \stackrel{\text{def}}{=} \mathbf{0}\,,$$
$$N_4 \stackrel{\text{def}}{=} a.N_5 + b.\mathbf{1} + \mathbf{1}\,,$$
$$N_5 \stackrel{\text{def}}{=} a.(N_4 \parallel N_5) + c.N_6\,,$$
$$N_6 \stackrel{\text{def}}{=} \mathbf{0}\,.$$

We fix the total ordering on the names as $N_6 < N_5 < \cdots < N_1$. If we now check whether $N_1 \leftrightarrows N_4$ holds, we can construct the following successful tableau:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{N_1 = N_4}{a.(N_2 \parallel N_3) + b.\mathbf{1} + \mathbf{1} = a.N_5 + b.\mathbf{1} + \mathbf{1}}\,\textsf{Rec}}{a.(N_2 \parallel N_3) = a.N_5}\,\textsf{SumT} \qquad \dfrac{b.\mathbf{1} = b.\mathbf{1}}{\mathbf{1} = \mathbf{1}}\,\textsf{Prefix}}{N_2 \parallel N_3 = N_5}\,\textsf{Prefix}}{a.(N_2 \parallel N_4 \parallel N_3) + c.N_3 = a.(N_4 \parallel N_5) + c.N_6}\,\textsf{Rec}}{a.(N_2 \parallel N_4 \parallel N_3) = a.(N_4 \parallel N_5)}\,\textsf{Sum} \qquad \dfrac{c.N_3 = c.N_6}{N_3 = N_6}\,\textsf{Prefix}}{N_2 \parallel N_4 \parallel N_3 = N_4 \parallel N_5}\,\textsf{Rec}}{N_4 \parallel N_5 = N_4 \parallel N_5}\,\textsf{SubL} \qquad \mathbf{0} = \mathbf{0}$$

In this tableau the SubL rule can be applied because using the well-founded ordering on the parallel compositions we have that $N_5 \sqsubset N_2 \parallel N_3$. ♢

For this tableau decision method to work, we need to show that it is both sound and complete. First, we need to know that tableaux are finite and that there are hence only finitely many tableaux for each pair of process expressions. A proof for the following lemma is already provided by Christensen, Hirshfeld and Moller (as Lemma 3.2 in [CHM93]). We will not repeat the proof here as we did not adapt anything that might affect its validity.

LEMMA 5.35. *Every tableau for $\xi = \chi$ is finite. Furthermore, there is only a finite number of tableaux for $\xi = \chi$.* □

The proof of the following completeness and soundness theorems are also mainly due to Christensen, Hirshfeld and Moller [CHM93].

THEOREM 5.36 (Completeness). *If $\xi \leftrightarrows \chi$ then there exists a successful tableau with root labelled $\xi = \chi$.* □

PROOF. It is easy to see that the added rule SumT is forward sound, i.e. if the premise as well as all nodes above relate bisimilar processes then it is possible to find a set of goals relating bisimilar processes. Because the property holds for the added rule and the unfolding function unf_1 preserves bisimilarity, the proof for this theorem is the same as the proof in [CHM93, Theorem 3.3]. ■

The soundness proof relies on an alternative characterisation of bisimulation taken from [CHM93] and extended with termination conditions.

DEFINITION 5.37. The sequence of bisimulation approximations $\{\leftrightarrows_n\}_{n=0}^{\infty}$ is defined as follows:

- $p \leftrightarrows_0 q$ for all process expression p and q;
- $p \leftrightarrows_{n+1} q$ iff for all $a \in \mathcal{A}_\tau$,
  - if $p \xrightarrow{a} p'$ then there exists q' such that $q \xrightarrow{a} q'$ and $p' \leftrightarrows_n q'$,
  - if $q \xrightarrow{a} q'$ then there exists p' such that $p \xrightarrow{a} p'$ and $p' \leftrightarrows_n q'$,
  - if $p\downarrow$ then $q\downarrow$ and vice versa. △

Using bisimulation approximation sequences we can prove soundness of the tableau method.

THEOREM 5.38 (Soundness). *If there is a successful tableau labelled with root labelled $\xi = \chi$ then $\xi \leftrightarrows \chi$.* □

PROOF. Suppose $T(\xi = \chi)$ is a tableau for $\xi = \chi$, and that $\xi \not\leftrightarrows \chi$. We shall construct a maximal path $\pi = \{\, \mathbf{n_i} : p = q \,\}$ through this tableau starting at the root $\xi = \chi$ in which $p_i \neq q_i$ for each i. Hence the terminal node of this path cannot be successful, so $T(\xi = \chi)$ is not successful.

While constructing π, we shall at the same time construct the sequence of natural numbers $\{ m_i : p_i \not\leftrightarrows_{m_i} q_i \mid p_i \leftrightarrows_j q_i \text{ for all } j < m_i \}$. We shall also prove along the way that this sequence is non-increasing, and strictly decreasing through applications of the rule Prefix. Given $\mathbf{n_i} : p_i = q_i$ and m_i, we get $\mathbf{n_{i+1}} : p_{i+1} = q_{i+1}$ and m_{i+1} according to the following cases:

- If Rec is applied to $\mathbf{n_i}$, then the consequent is $\mathbf{n_{i+1}}$ and $m_{i+1} = m_i$.
- If Sum is applied to $\mathbf{n_i}$, then there must be some consequent $\mathbf{n_{i+1}} : p_{i+1} = q_{i+1}$ with $p_{i+1} \not\leftrightarrows_{m_i} q_{i+1}$ and $p_{i+1} \leftrightarrows_j q_{i+1}$ for all $j < m_i$, so $m_{i+1} = m_i$.
- If SumT is applied to $\mathbf{n_i}$, then there must be some consequent $\mathbf{n_{i+1}} : p_{i+1} = q_{i+1}$ with $p_{i+1} \not\leftrightarrows_{m_i} q_{i+1}$ and $p_{i+1} \leftrightarrows_j q_{i+1}$ for all $j < m_i$, so $m_{i+1} = m_i$.
- If Prefix is applied to $\mathbf{n_i}$, then the consequent is $\mathbf{n_{i+1}}$ and $m_{i+1} = m_i - 1$.
- If SubL is applied to $\mathbf{n_i}$, then $p_i = q_i$ must be of the form $\xi \parallel \eta = \rho$ with dominated node $\xi = \chi$ ($\chi \sqsubset \xi$). Since between $\mathbf{n_j}$ and $\mathbf{n_i}$ there must have been an intervening application of the rule Prefix, we must have that $m_i < m_j$. We take the node $\mathbf{n_{i+1}} : \chi \parallel \eta = \rho$, and show that we have some valid $m_{i+1} \leq m_i$, that is, that $\chi \parallel \eta \not\leftrightarrows \rho$. But this follows from $\xi \leftrightarrows_{m_i} \chi$ and $\xi \parallel \eta \not\leftrightarrows_{m_i} \rho$.
- The arguments for the application of the SubR are identical.

That the above conditions hold of the resulting path is now clear. ■

With the modified tableau decision method for which we have shown that it still generates finitely many finite tableaux and the rules are still sound and complete, we have the desired result.

COROLLARY 5.39. *Bisimilarity is decidable on basic parallel specifications.* □

5.3 Explicit Interaction

In the previous chapter we have made the interaction within the pushdown automaton explicit: a linear specification of the finite control of the pushdown automaton was put in parallel with a sequential specification of the stack. We have seen that this yielded an associated transition system that is divergence-preserving branching bisimilar with the original pushdown transition system. The result was obtained in two steps: first for pushdown automata with termination on both final state and empty stack, and then with termination on final state only. For the latter case we introduced an alternative definition of the stack, as there is no sequential specification of an always terminating stack.

In this section we will do the same for the parallel pushdown automaton and see that the result becomes more clear-cut. However, we obtain results only up to branching bisimilarity due to the fact that removing data elements from a bag is not deterministic. For the parallel pushdown automata with termination on final state and termination on both final state and empty bag, we are able to use the same finite control, and just use a different specification of the bag. These specifications are the basic parallel specifications of the bag and the transparent bag that we have

seen before in Section 5.2.1. Since we are also interested in the relation between basic parallel specifications and the explicit interaction, we also investigate pushdown automata with termination on both final state and transparent bag. For this we shall present the partially transparent bag process, which is a concept in between the bag and the transparent bag.

First, we consider parallel pushdown automata according to the FSEB interpretation. Let $M = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be some parallel pushdown automaton. By Theorem 5.8 we can assume that M only has insert and remove transitions. We can now define the linear specification E_{fc}, capturing the finite control of M. For every state $s \in \mathcal{S}$ we add to E_{fc} a name C_s with the following defining equation ($s, t \in \mathcal{S}$, $a \in \mathcal{A}_\tau$, $d \in \mathcal{D}$):

$$C_s \stackrel{\text{def}}{=} \sum_{(s,a,*,d,t)\in\rightarrow} a.i!d.C_t + \sum_{d\in\mathcal{D}} o?d.C_{s,d} \ [+\ \mathbf{1}]_{s\downarrow} \ ,$$

and for every state $s \in \mathcal{S}$ and data element $d \in \mathcal{D}$ we add to E_{fc} a name $C_{s,d}$ with the following defining equation ($s, t \in \mathcal{S}$, $a \in \mathcal{A}_\tau$, $d \in \mathcal{D}$):

$$C_{s,d} \stackrel{\text{def}}{=} i!d.C_s + \sum_{(s,a,d,\emptyset,t)\in\rightarrow} a.C_t \ .$$

The names C_s ($s \in \mathcal{S}$) handle the insert transitions for state s and the detection whether some data element d can be found in the bag. The names $C_{s,d}$ ($s \in \mathcal{S}$, $d \in \mathcal{D}$) handle the remove transitions for state s given that we know that data element d has been found in the bag. Note that these names also have a summand $i!d.C_s$ to put the data element back in the bag to prevent that the removal of d becomes an irreversible choice. We will see later that this is necessary to prevent the creation of non-inert silent transition once abstraction has been applied over the communication between finite control and bag.

EXAMPLE 5.40. Let us reconsider the parallel pushdown automaton in Figure 5.1 (on page 82). When applying the construction described above we get the following linear specification for the finite control:

$$\begin{aligned} C_s &\stackrel{\text{def}}{=} a.i!1.C_s + c.i!1.C_t + o?1.C_{s,1} \ , \\ C_{s,1} &\stackrel{\text{def}}{=} i!1.C_s + b.C_s \ , \\ C_t &\stackrel{\text{def}}{=} o?1.C_{t,1} + \mathbf{1} \ , \\ C_{t,1} &\stackrel{\text{def}}{=} i!1.C_t + b.C_t \ . \end{aligned}$$

♢

Now, if we put the finite control in parallel with the bag, we can obtain the following result for pushdown automaton with termination on both final state and empty bag.

THEOREM 5.41. *For every parallel pushdown automaton M according to the FSEB interpretation there exists a linear specification E_{fc} and linear process expression p, such that $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E_{fc}\cup E_B}(\partial_{i,o}(p \parallel B))$.* □

PROOF. The specification E_{fc} is constructed for M as described above. We present some observations from which it is fairly straightforward to establish that $\mathcal{T}(M) \leftrightarrows_{\mathrm{b}} \mathcal{T}_{E_{fc} \cup E_B}(\left[p \parallel B\right]_{i,o})$. In our proof we abbreviate the process expression $B \parallel i!d_1.\mathbf{1} \parallel \cdots \parallel i!d_n.\mathbf{1}$ by $B_{[\![d_1,\ldots,d_n]\!]}$, with, in particular, $B_\emptyset = B$. (Recall the infinite specification of the bag given on page 92.)

First, note that the control process for some state s is not allowed to choose which data element to pick until the corresponding action is performed. Therefore, given that the multiset μ is not empty, we have for each $d \in \mu$ that

$$\partial_{i,o}(C_s \parallel B_\mu) \xrightarrow{o?d} \partial_{i,o}(C_{s,d} \parallel B_{\mu - [\![d]\!]}) \xrightarrow{i?d} \partial_{i,o}(C_s \parallel B_\mu)$$

When the abstraction $\tau_{i,o}(\_)$ is applied, we get two inert τ-transitions and obtain the following (intermediate) result:

$$\left[C_s \parallel B_\mu\right]_{i,o} \leftrightarrows_{\mathrm{b}} \sum_{d \in \mu} \left[C_{s,d} \parallel B_{\mu - [\![d]\!]}\right]_{i,o} .$$

Hence, when a data element is removed from the bag, the control process has not made a choice yet as it can always reinsert it. This is different from the interaction between the control process and the stack in the proof of Theorem 4.42 (on page 70), since a pop from the stack is deterministic, i.e. one always receives the top element.

Second, whenever $\mathcal{T}(M)$ has an insert transition $(s, \mu) \xrightarrow{a} (t, \mu \uplus [\![d]\!])$, then

$$\partial_{i,o}(C_s \parallel B_\mu) \xrightarrow{a} \xrightarrow{i?d} \partial_{i,o}(C_t \parallel B_{\mu \uplus [\![d]\!]}) ,$$

and the τ-transition resulting from applying $\tau_{i,o}(\_)$ is inert.

Finally, whenever $\mathcal{T}(M)$ has remove transition $(s, \mu \uplus [\![d]\!]) \xrightarrow{a} (t, \mu)$, we first use the fact that $\left[C_s \parallel B_{\mu \uplus [\![d]\!]}\right]_{i,o} \leftrightarrows_{\mathrm{b}} \left[C_{s,d} \parallel B_\mu\right]_{i,o}$, non-deterministically removing data element d, and then finish with

$$\partial_{i,o}(C_{s,d} \parallel B_\mu) \xrightarrow{a} \partial_{i,o}(C_t \parallel B_\mu) .$$ ■

Now, for the other direction. We can show that if we have a process defined by a linear specification that communicates with the bag, we can find a PPDA that simulates the behaviour of the two specifications put in parallel.

THEOREM 5.42. *For every linear specification E and linear process expression p there exists a parallel pushdown automaton M according to the FSEB interpretation such that $\mathcal{T}_{E \cup E_B}(\left[p \parallel B\right]_{i,o}) \leftrightarrows_b \mathcal{T}(M)$.* □

PROOF. Let E be a linear specification and let p be a linear process expression. We define a parallel pushdown automaton M as follows:

- The set of states, the action alphabet, and the initial and final states are the same as those of the transition system $\mathcal{T}_E(p)$ (which is a finite automaton).
- The set of data symbols is the set of data $\mathcal{D}$ of the presupposed recursive specification of the bag.

– Whenever $s \xrightarrow{a} t$ in $\mathcal{T}_E(p)$, and $a \neq i!d, o?d$ $(d \in \mathcal{D})$, then $s \xrightarrow{a[*/\emptyset]} t$;
– whenever $s \xrightarrow{i!d} t$ for some $d \in \mathcal{D}$ in $\mathcal{T}_E(p)$, then $s \xrightarrow{\tau[*/[\![d]\!]]} t$;
– whenever $s \xrightarrow{o?d} t$ for some $d \in \mathcal{D}$ in $\mathcal{T}_E(p)$, then $s \xrightarrow{\tau[d/\emptyset]} t$.

We omit the proof that every transition of $\mathcal{T}_{E \cup E_B}(\lceil p \parallel B \rceil_{i,o})$ can be matched by a transition in $\mathcal{T}(M)$ in the sense required by the definition of divergence-preserving branching bisimilarity. ■

To obtain the same results for parallel pushdown automata according to the FS interpretation, we only have to replace the bag by the transparent bag (see Definition 5.23). If we apply the same constructions explained above, termination will occur when the final control is in a final state, because the transparent bag can always terminate. We get the following results, but shall omit the proofs.

THEOREM 5.43. *For every parallel pushdown automaton M according to the FS interpretation there exists a linear specification E_{fc} and linear process expression p, such that $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E_{fc} \cup E_{B^t}}(\lceil p \parallel B^t \rceil_{i,o})$, and vice versa.* □

To also have the same results for parallel pushdown automata according to the FSTB interpretation, we have to replace the bag again.

DEFINITION 5.44. Let $\mathcal{D}^{-1} \subseteq \mathcal{D}$ be the data elements that are considered to be opaque, and $\mathcal{D}^{+1} = \mathcal{D} \setminus \mathcal{D}^{-1}$ the data elements that are transparent.

We define the *partially transparent bag*, a mix of the specification of the bag and the transparent bag, by the following basic parallel specification:

$$B^{pt}_{i,o} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}^{-1}} i?d.(B^{pt}_{i,o} \parallel o!d.\mathbf{1}) + \sum_{d \in \mathcal{D}^{+1}} i?d.\big(B^{pt}_{i,o} \parallel (o!d.\mathbf{1} + \mathbf{1})\big) .$$

We refer to this specification as $E_{B^{pt}}$. △

Now, if we apply the same constructions explained above, termination will occur when the final control is in a final state and the partially transparent bag contains no opaque data elements.

THEOREM 5.45. *For every parallel pushdown automaton M according to the FSTB interpretation there exists a linear specification E_{fc} and linear process expression p, such that $\mathcal{T}(M) \leftrightarrows_b \mathcal{T}_{E_{fc} \cup E_{B^{pt}}}(\lceil p \parallel B^{pt} \rceil_{i,o})$, and vice versa.* □

We have seen in Section 5.2.1 that basic parallel specifications can be simulated by a PPDA (according to the FSTB interpretation). We have also seen in the theorem above that each PPDA according to the FSTB interpretation can be defined by a linear specification for the finite control of the PPDA and a basic parallel specification of the partially transparent bag memory, combined in a single specification that allows for communication between both components. Indirectly, we have established that each basic parallel specification can be written as a linear specification communicating with a partially transparent bag. Therefore, we can consider the partially transparent bag, with its basic parallel specification, as the canonical basic parallel process.

COROLLARY 5.46. *For every basic parallel specification E and basic parallel expression p there exists a linear specification E_{fc} and linear process expression q such that*

$$\mathcal{T}_E(p) \leftrightarrows_b \mathcal{T}_{E_{fc} \cup E_{B^{pt}}}(\left[q \parallel B^{pt}\right]_{i,o}) \ . \qquad \square$$

PROOF. The result follows from Theorems 5.29 and 5.45. ■

Note that the same result was obtained directly for basic parallel specifications in [BCT09].

5.4 Conclusions

In this chapter we have followed the lead of the previous chapter and investigated a parallel, commutative version of pushdown automata and context-free languages. We have seen the definition of the parallel pushdown automaton, which is basically a pushdown automaton equipped with a bag memory instead of a stack memory. The replacement of the type of memory leads to subtle differences in semantics with respect to the regular pushdown automata, such as the removal of a data element from the memory not being deterministic and not being able to test whether the memory is empty.

We have investigated the differences in classes of parallel pushdown transition systems if we use different termination conditions: termination on empty bag (EB), on final state (FS), and on both final state and empty bag (FSEB). We have shown that the class according to the EB interpretation is, up to divergence-preserving branching bisimilarity, a proper subclass of the class according to the FSEB interpretation. If we drop divergence-preservation and consider parallel pushdown automata that are initially terminating, then the class of parallel pushdown transition systems according to the EB interpretation coincides with the class according to the FSEB interpretation. The class according to the FSEB interpretation turns out to be incomparable, up to branching bisimilarity, with the class according to the FS interpretation. Unlike for pushdown transition systems, this is also the case without divergence-preservation. Therefore, we have considered the correspondence results both the FSEB and FS class.

We proposed basic parallel specifications as the specification language for the class of parallel pushdown systems. A basic parallel specification is a finite recursive BCP_τ-specification (assuming an empty communication function). This specification language extends a traditional language for similar kinds of systems, called BPP, with $\mathbf{0}$, $\mathbf{1}$ and prefixing. It also is the parallel counterpart of the sequential specifications of the previous chapter where sequential composition is replaced by parallel composition. We have seen that we can find parallel pushdown automata that simulate, up to branching bisimilarity, opaque and transparent basic parallel specifications, respectively by using the FSEB and FS interpretations. To be able to simulate mixed opaque/transparent basic parallel specifications we added the termination condition on both final state and transparent bag (FSTB). This means

that the bag can terminate if it only contains data elements from a designated "transparent" subset of data elements. In the other direction we have seen that only single-state PPDAs can be defined by a basic parallel specification. Hence, the correspondence between PPDAs and basic parallel specifications is rather weak.

As basic parallel specifications play an important role in this chapter, we have shown that it is possible to decide if two basic parallel specifications have the same associated transition system up to (strong) bisimilarity, extending earlier work for BPP-specifications.

From a process-theoretic perspective it makes sense to make the interaction with the bag in a PPDA explicit. We can do this by giving a linear specification representing the finite control of the PPDA and put it in parallel with a specification of a bag, allowing communication over an input and output channel for inserting and removing data elements. We have established this correspondence for parallel pushdown automata according to the FSEB, FS and FSTB interpretations by using the same linear specification of the finite control and respectively the bag, the transparent bag and the partially transparent bag.

Figure 5.11 presents a schematic overview of the correspondence results for all three interpretations from a process-theoretic point of view. Note that there is an indirect correspondence between basic parallel specifications and the explicit interaction. Because the (partially transparent) bag can be defined by a basic parallel specification, and all basic parallel specifications can be given as a finite-state process communicating with this bag, the (partially transparent) bag can be considered as the canonical basic parallel process.

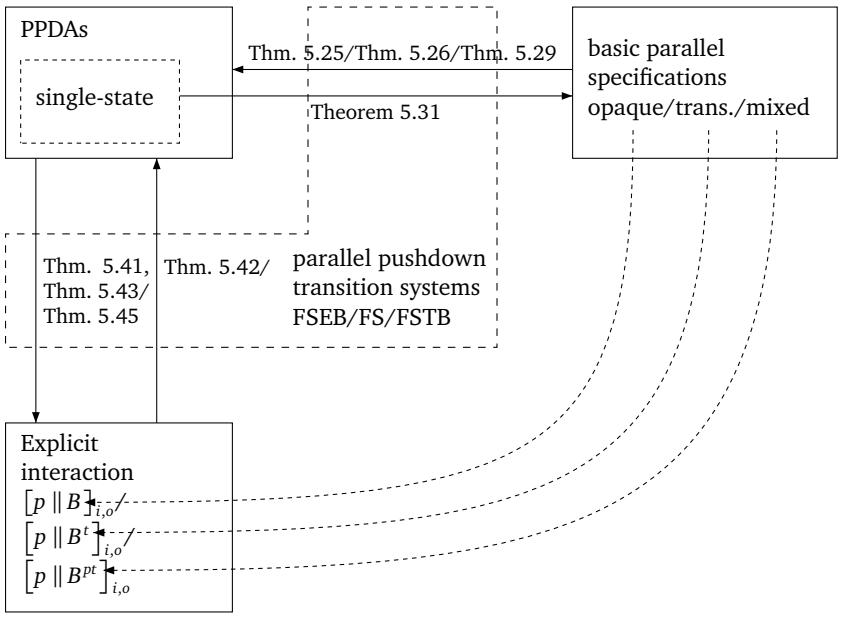

FIGURE 5.11: Correspondence results for the FSEB/FS/FSTB interpretations.

5.4.1 Future Work

The decidability result discussed in Section 5.2.2 should be extended to branching bisimilarity (preferably divergence-preserving). However, this has been an open problem for a long time.

We have seen that only single-state PPDAs can be defined by a basic parallel specification. A more suitable candidate could be Petri nets instead of basic parallel specifications, as was shown by Moller in [Mol96]. However, Hirshfeld and Moller have shown later in [HM01] that there are Petri nets that cannot be simulated by a PPDA. Thus, it is necessary to find a appropriate restriction on Petri nets for the correspondence with PPDAs.

Recall that we use a technique to avoid making a choice when removing something from the bag in the definition of the linear specifications of the finite control of a PPDA. Namely, when some data element is picked from the bag, it can always be put back in the bag. While this makes the initial τ-transition to remove the data element inert up to branching bisimilarity, it does introduce divergence. The question remains whether this can be done without introducing divergence and thus lifting the correspondence results to divergence-preserving branching bisimilarity.

Another question is whether the introduction of the FSTB interpretation is really necessary. In the previous chapter we remarked that it might be possible to simulate mixed opaque/transparent sequential specifications by a PDA with termination on both final state and empty stack. This would be, however, at the cost of switching to the weaker contrasimulation equivalence. It would be worth investigating if such an approach would work here as well as an alternative solution to using the FSTB interpretation.

Finally, in the conclusions of the previous chapter we have suggested to define and investigate deterministic pushdown automata (see page 76). Similarly, it would be interesting to define deterministic parallel pushdown automata and investigate the expressivity of this class using parallel pushdown transition systems.

Chapter 6

Computable & Executable Systems

The Turing machine [Tur37] is widely accepted as a computational model suitable for exploring the theoretical boundaries of computing. It is used in computability theory to formally characterise the notion of *effectively calculable function*. An effectively calculable function is a function for which there exists an algorithm that can calculate its values. It was later shown that the Turing machine characterises the same notion of effectively calculable function as the separately proposed notions of recursive functions by Kleene in [Kle36] and λ-calculus by Church in [Chu36].

Motivated by the existence of universal Turing machines, many textbooks on the theory of computation (e.g., [Sud88, Sip97, HMU06]) present the Turing machine not just as a theoretical model to explain which functions are computable, but, in fact, as an accurate conceptual model of the computer. For instance, Sipser writes in [Sip97] that "[a] Turing machine can do everything a real computer can do." This statement is sometimes referred to as the *strong* Church-Turing thesis, as opposed to the normal Church-Turing thesis according to which every *effectively calculable function* is computable by a Turing machine.

There is, however, a limitation to viewing the Turing machine as a conceptual model of a computer. A Turing machine operates from the assumptions that: (1) all the input it needs for the computation is available on the tape from the very beginning; (2) it performs a terminating computation; and (3) it leaves the output on the tape at the very end. That is, a Turing machine computes a function, and thus it abstracts from two key ingredients of computing: *interaction* and *non-termination*. Nowadays, most computing systems are so-called *reactive systems* [HP89], systems that are generally not meant to terminate and that consist of a number of computing devices that interact with each other and with their environment. A reactive system often unremittingly depends on input, and unremittingly produces output.

Towards the end of the 1970s, Milner observed that, for a thorough investigation of interaction and concurrency, it is profitable to study these notions in isolation rather than to try and add them to any of the existing models of computation. One of his desiderata for the design of CCS was "that there be *only a single* combinator for combining processes which interact or which coexist" [Mil93]. In particular, also the interaction of a computing device with its memory should be modelled using

a symmetric notion of interaction, considering the memory as a separate process. Concurrency theory has provided us with a fundamental understanding of interaction and non-termination.

In Section 6.1 we propose a notion of *reactive* Turing machine (RTM), extending the classical notion of Turing machines with interaction in the style of concurrency theory. The extension consists of a facility to declare every transition to be either *observable*, by labelling it with an action symbol, or unobservable, by labelling it with τ. Typically, a transition labelled with an action symbol models an interaction of the RTM with its environment (or some other RTM), while a transition labelled with τ refers to an internal computation step. Thus, a conventional Turing machine can be regarded as a special kind of RTM in which all transitions are declared unobservable by labelling them with τ.

The semantic object associated with a conventional Turing machine is either the function that it computes, or the formal language that it accepts. The semantic object associated with an RTM is a behaviour, formally represented by a transition system, as we have also done in the previous chapters. A function is said to be *effectively computable* if it can be computed by a Turing machine. By analogy, we say that a behaviour is *effectively executable* if it can be exhibited by a reactive Turing machine. In concurrency theory, behaviours are usually considered modulo a suitable behavioural equivalence. Also in this chapter we shall mainly use (divergence-preserving) branching bisimilarity.

In Section 6.2 we set out to investigate the expressiveness of RTMs up to divergence-preserving branching bisimilarity. We shall present an example of a behaviour that is not effectively executable up to branching bisimilarity. Then, we establish that every computable transition system with a bounded branching degree can be simulated, up to divergence-preserving branching bisimilarity, by an RTM. If the divergence-preservation requirement is dropped, even every effective transition system can be simulated. These results will then allow us to conclude that the behaviour of a parallel composition of RTMs can be simulated on a single RTM.

In Section 6.2.4 we define a suitable notion of universality for RTMs and investigate the existence of universal RTMs. We shall find that, since bisimilarity is sensitive to branching, there are some subtleties pertaining to the branching degree bound associated with each RTM. Up to divergence-preserving branching bisimilarity, an RTM can at best simulate other RTMs with the same or a lower bound on their branching degree. If divergence-preservation is not required, however, then universal RTMs do exist.

In Section 6.3, we consider the correspondence between RTMs and the process theory TCP_τ. We establish that every executable transition system is, again up to divergence-preserving branching bisimilarity, definable by a finite recursive TCP_τ-specification. As we have seen in previous chapters, recursive specifications are the process-theoretic counterparts of grammars in the theory of formal languages. Thus, the result in Section 6.3 may be considered as the process-theoretic version of the correspondence between Turing machines and unrestricted grammars. Furthermore, the finite recursive TCP_τ-specification actually consists of a specification of the finite control of the RTM that interacts with a specification modelling a tape. Thus, as an

interesting corollary, we obtain a specification that makes the conceptual interaction within a reactive Turing machine between its finite control and its tape memory explicit; similar results have also been obtained for pushdown automata and parallel pushdown automata in the previous chapters.

Several extensions of Turing machines with some form of interaction have been proposed in the literature, already by Turing in [Tur39], and more recently, when there was renewed interest in the matter, in [LW00, GSAS04, GSW06, BGRR07, WL08]. The goal in these works is mainly to investigate to what extent interaction may have a beneficial effect on the power of sequential computation. Interaction is, e.g., added by allowing an algorithm to query its environment, or by assuming that the environment periodically writes a write-only input tape and reads a read-only output tape of a Turing machine. Thereby, the focus remains on the computational aspect, and interaction is not treated as a first-class citizen. Our goal, instead, is to achieve integration of automata and concurrency theory that treats computation and interactivity on equal footing.

The material in this chapter is based on the following publication:

[BLT11b] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. "Reactive Turing Machines". In: *Proceedings of FCT 2011*. Ed. by O. Owe, M. Steffen, and J. Telle. LNCS 6914. Springer, 2011, pp. 348–359.

This is an abstract of the following full version technical report:

[BLT11c] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. *Reactive Turing Machines*. Tech. rep. arXiv:1104.1738v3. Cornell University Library, 2011.

6.1 Reactive Turing Machines

For an RTM we add to the finite set of data symbols $\mathcal{D}$ a special symbol $\Box$ to denote a blank tape cell, assuming that $\Box \notin \mathcal{D}$; we denote the set $\mathcal{D} \cup \{\Box\}$ of *tape symbols* by $\mathcal{D}_\Box$. In our definition, following the original definition of the Turing machine, we allow head movements to the left (L) and right (R); we use M to range over $\{L, R\}$.

DEFINITION 6.1. A *reactive Turing machine* (RTM) $\mathcal{M}$ is a six-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ where

1. $\mathcal{S}$ is a finite set of *states*;
2. $\mathcal{A}$ a finite set of actions;
3. $\mathcal{D}$ a finite set of data;
4. $\rightarrow \subseteq \mathcal{S} \times \mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\} \times \mathcal{S}$ is a $(\mathcal{D}_\Box \times \mathcal{A}_\tau \times \mathcal{D}_\Box \times \{L, R\})$-labelled relation on $\mathcal{S}$,
5. $\uparrow \in \mathcal{S}$ is the initial state, and
6. $\downarrow \subseteq \mathcal{S}$ is the set of final states.

An RTM is *deterministic* if $(s, d, a, e_1, M_1, t_1) \in \rightarrow$ and $(s, d, a, e_2, M_2, t_2) \in \rightarrow$ implies that $e_1 = e_2$, $t_1 = t_2$ and $M_1 = M_2$ for all $s, t_1, t_2 \in \mathcal{S}$, $d, e_1, e_2 \in \mathcal{D}_\Box$, $a \in \mathcal{A}_\tau$, $M_1, M_2 \in \{L, R\}$, and, moreover, $(s, d, \tau, e_1, M_1, t_1) \in \rightarrow$ implies that there do not exist $a \neq \tau$, e_2, M_2, t_2 such that $(s, d, a, e_2, M_2, t_2) \in \rightarrow$. $\triangle$

If $(s, d, a, e, M, t) \in \rightarrow$, we write $s \xrightarrow{a[d/e]M} t$. The intuitive meaning of such a transition is that whenever $\mathcal{M}$ 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 one position to the right on the tape (depending on whether $M = L$ or $M = R$), and then end up in state t. RTMs extend conventional Turing machines by associating with every transition an element $a \in \mathcal{A}_\tau$. The symbols in $\mathcal{A}$ are thought of as denoting observable activities; a transition labelled with an action symbol in $\mathcal{A}$ will semantically be treated as observable. Observable transitions are used to model interactions of an RTM with its environment or some other RTM, as will be explained more in detail below when we introduce a notion of parallel composition for RTMs. The symbol τ is used to declare that a transition is unobservable. A conventional Turing machine is an RTM in which all transitions are declared unobservable.

EXAMPLE 6.2. Assume that $\mathcal{A} = \{ c!d, c?d \mid c \in \{i, o\}, d \in \mathcal{D}_\square \}$. Intuitively, i and o are the input/output communication channels through which the RTM can interact with its environment. The action symbol $c!d$ ($c \in \{ i, o \}$) then denotes the event that a data element d is sent by the RTM along channel c, and the action symbol $c?d$ ($c \in \{ i, o \}$) denotes the event that a data element d is received by the RTM along channel c.

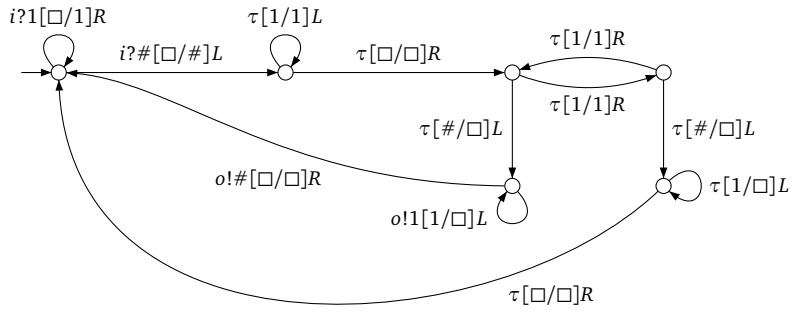

FIGURE 6.1: An example of a reactive Turing machine.

The state-transition diagram in Figure 6.1 concisely specifies an RTM that first inputs a string, consisting of an arbitrary number of 1s followed by the symbol #, stores the string on the tape, and returns to the beginning of the string. Then, it performs a computation to determine if the number of 1s is odd or even. In the first case, it simply removes the string from the tape and returns to the initial state. In the second case, it outputs the entire string, removes it from the tape, and returns to the initial state. ♢

The semantics of a conventional Turing machine is either the function on natural numbers that it computes, or the formal language that it accepts. The function or the formal language associated with a Turing machine is determined by its set of *computations*, i.e., sequences of configurations leading from some initial configuration to a final configuration. A computation is, by definition, terminating and abstracts from the moments of choice. For RTMs to serve as models of reactive

systems, it is important *not* to discard their infinite behaviours. Furthermore, we are going to model interaction by allowing the environment or other RTMs to influence choices during the operations of an RTM.

With every RTM $\mathcal{M}$ we are going to associate a transition system $\mathcal{T}(\mathcal{M})$. The states of $\mathcal{T}(\mathcal{M})$ are the configurations of the RTM, consisting of a state of the RTM, its tape contents, and the position of the read/write head on the tape. We represent the tape contents by an element of $\mathcal{D}_\Box^*$, replacing precisely one occurrence of a tape symbol d by a *marked* symbol $\check{d}$, indicating that the read/write head is on this symbol. We denote by $\check{\mathcal{D}}_\Box = \{\,\check{d} \mid d \in \mathcal{D}_\Box\,\}$ the set of *marked* tape symbols; a *tape instance* is a string $\delta \in (\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ such that δ contains exactly one element of $\check{\mathcal{D}}_\Box$. Note that we do not use δ exclusively for tape instances; we also use δ for sequences over $\mathcal{D}$. A tape instance thus is a finite sequence of symbols that represents the contents of a two-way infinite tape. Henceforth, we shall not distinguish between tape instances that are equal modulo the addition or removal of extra occurrences of the symbol $\Box$ at the left or right extremes of the string. That is, we shall not distinguish tape instances δ_1 and δ_2 if $\Box^\omega\delta_1\Box^\omega = \Box^\omega\delta_2\Box^\omega$. Note that a marked blank symbol $\check{\Box}$ is considered as a non-blank symbol with respect to adding or removing blanks, e.g. $\delta\Box\check{\Box}\Box = \delta\Box\check{\Box}$.

DEFINITION 6.3. A *configuration* of an RTM $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ is a pair (s, δ) consisting of a state $s \in \mathcal{S}$, and a tape instance δ. △

Our transition system semantics defines an $\mathcal{A}_\tau$-labelled transition relation on configurations such that an RTM-transition $s \xrightarrow{a[d/e]M} t$ corresponds with a-labelled transitions from configurations consisting of the RTM-state s and a tape instance in which some occurrence of d is marked. The transitions lead to configurations consisting of t and a tape instance in which the marked symbol d is replaced by e, and either the symbol to the left or to right of this occurrence of e is replaced by its marked version, according to whether $M = L$ or $M = R$. If e happens to be the first symbol and $M = L$, or the last symbol and $M = R$, then an additional blank symbol is appended at the left or right end of the tape instance, respectively, to model the movement of the head.

It is convenient to introduce some notation to be able to concisely denote the new placement of the tape head marker. Let δ be an element of $\mathcal{D}_\Box^*$. Then by $\delta^<$ 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, i.e.,

$$\delta^< = \begin{cases} \zeta\check{d} & \text{if } \delta = \zeta d \quad (d \in \mathcal{D}_\Box, \zeta \in \mathcal{D}_\Box^*)\,, \text{ and} \\ \check{\Box} & \text{if } \delta = \varepsilon\,. \end{cases}$$

Similarly, by ${}^>\delta$ we denote the element of $(\mathcal{D}_\Box \cup \check{\mathcal{D}}_\Box)^*$ obtained by placing the tape head marker on the left-most symbol of δ if it exists, and $\check{\Box}$ otherwise, i.e.,

$${}^>\delta = \begin{cases} \check{d}\zeta & \text{if } \delta = d\zeta \quad (d \in \mathcal{D}_\Box, \zeta \in \mathcal{D}_\Box^*)\,, \text{ and} \\ \check{\Box} & \text{if } \delta = \varepsilon\,. \end{cases}$$

We use this notation under the assumption that from δ extra occurrences of the symbol $\Box$ at the left and right extremes have been removed.

DEFINITION 6.4. Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ 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 of all configurations of $\mathcal{M}$;
2. its transition relation $\rightarrow$ is the least relation satisfying, for all $a \in \mathcal{A}_\tau$, $d, e \in \mathcal{D}_\Box$ and $\delta_L, \delta_R \in \mathcal{D}_\Box^*$:
   a) $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L {}^{\check{}}e \delta_R)$ iff $s \xrightarrow{a[d/e]_L} t$, and
   b) $(s, \delta_L \check{d} \delta_R) \xrightarrow{a} (t, \delta_L e^{\check{}} \delta_R)$ iff $s \xrightarrow{a[d/e]_R} t$;
3. its initial state is the configuration $(\uparrow, \check{\Box})$; and
4. its set of final states is the set of terminating configurations $\{(s, \delta) \mid s\downarrow\}$. $\triangle$

Turing introduced his machines to define the notion of *effectively computable function*. By analogy, our notion of RTM can be used to define a notion of *effectively executable behaviour*.

DEFINITION 6.5. A transition system is *executable* if it is associated with an RTM. $\triangle$

This definition automatically gives us the notion of an executable process.

DEFINITION 6.6. An *executable process* is a divergence-preserving branching bisimilarity class of labelled transition systems containing an executable transition system. $\triangle$

Parallel composition

To illustrate how RTMs are suitable to model a form of interaction, we shall now define on RTMs a notion of parallel composition, equipped with a simple form of communication. (We are not trying to define the most general or most suitable notion of parallel composition for RTMs here; the purpose of the notion of parallel composition defined here is just to illustrate how RTMs may run in parallel and interact.) Let $\mathcal{C}$ be a finite set of *channels* for the communication of data symbols between one RTM and another, and let $\mathcal{A}' = \{c!d, c?d \mid c \in \mathcal{C}, d \in \mathcal{D}_\Box\}$; it is assumed that $\mathcal{A}' \subseteq \mathcal{A}$.

First, we define a notion of parallel composition on transition systems.

DEFINITION 6.7. Let $T_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $T_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be transition systems, and let $\mathcal{C}' \subseteq \mathcal{C}$. The *parallel composition* of T_1 and T_2 is the transition system $[T_1 \parallel T_2]_{\mathcal{C}'} = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$, with $\mathcal{S}$, $\rightarrow$, $\uparrow$ and $\downarrow$ defined by

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

3. $\uparrow = (\uparrow_1, \uparrow_2)$; and
4. $\downarrow = \{(s_1, s_2) \mid s_1 \in \downarrow_1 \ \wedge \ s_2 \in \downarrow_2\}$. △

Then, we can define a similar notion of parallel composition on the associated transition systems with RTMs.

DEFINITION 6.8. Let $\mathcal{M}_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ and $\mathcal{M}_2 = (\mathcal{S}_2, \rightarrow_2, \uparrow_2, \downarrow_2)$ be RTMs, and let $\mathcal{C}' \subseteq \mathcal{C}$; by $[\mathcal{M}_1 \,\|\, \mathcal{M}_2]_{\mathcal{C}'}$ we denote the *parallel composition* of $\mathcal{M}_1$ and $\mathcal{M}_2$. The transition system $\mathcal{T}([\mathcal{M}_1 \,\|\, \mathcal{M}_2]_{\mathcal{C}'})$ associated with the parallel composition $[\mathcal{M}_1 \,\|_{\mathcal{C}}\, \mathcal{M}_2]_{\mathcal{C}'}$ of $\mathcal{M}_1$ and $\mathcal{M}_2$ is the parallel composition of the transition systems associated with $\mathcal{M}_1$ and $\mathcal{M}_2$, i.e., $\mathcal{T}([\mathcal{M}_1 \,\|\, \mathcal{M}_2]_{\mathcal{C}'}) = [\mathcal{T}(\mathcal{M}_1) \,\|\, \mathcal{T}(\mathcal{M}_2)]_{\mathcal{C}'}$. △

EXAMPLE 6.9. Let $\mathcal{M}$ denote the RTM in Figure 6.1. Let $\mathcal{A}$ be as in Example 6.2 and let $\mathcal{E}$ denote the RTM in Figure 6.2 below. Then, the parallel composition $[\mathcal{M} \,\|\, \mathcal{E}]_i$ exhibits the behaviour of outputting, along channel o, the string $11\#1111\#\cdots\#1^n\#\ldots$ ($n \geq 2$, n even). ♢

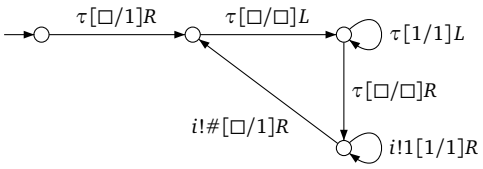

FIGURE 6.2: An RTM that enumerates and sends the string $1\#11\#111\#\ldots$.

An unobservable transition of an RTM, i.e., a transition labelled with τ, may be thought of as an internal computation step. Divergence-preserving branching bisimilarity allows us to abstract from internal computations as long as they do not discard the option to execute a certain behaviour. The following notion will be technically convenient in the remainder of this chapter.

DEFINITION 6.10. Given some transition system T, an *internal computation from state s to s'* is a sequence of states $s_1, \ldots, s_n$ in T such that $s = s_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} s_n = s'$. An internal computation is called *fully deterministic* iff, for every state s_i ($1 \leq i < n$), $s_i \xrightarrow{a} s_i{}'$ implies $a = \tau$ and $s_i{}' = s_{i+1}$. We shall write $s \twoheadrightarrow s'$ if there exists a fully deterministic computation from s to s'. △

It is easy to see that the following property holds for fully deterministic computations, as there is no branching.

LEMMA 6.11. *Let T be a transition system and let s and t be two states in T. If $s \twoheadrightarrow s'$, then s and s' are related by the maximal divergence-preserving branching bisimulation on T.* □

6.2 Expressiveness of RTMs

To confirm the expressiveness of RTMs, we shall establish in this section that every effective transition system can be simulated by an RTM up to branching bisimilarity, and that every boundedly branching computable transition system can be simulated by an RTM up to divergence-preserving branching bisimilarity. We use this as an auxiliary result to establish that a parallel composition of RTMs can be simulated by a single RTM, and we derive from it the existence of universal RTMs.

6.2.1 Effective & Computable Transition Systems

Let $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a transition system; the mapping $out : \mathcal{S} \rightarrow 2^{\mathcal{A}_\tau \times \mathcal{S}}$ associates with every state its set of outgoing transitions, i.e., for all $s \in \mathcal{S}$,

$$out(s) = \{(a,t) \mid s \xrightarrow{a} t\} \ ,$$

and $fin(\_)$ denotes the characteristic function of $\downarrow$.

DEFINITION 6.12. Let $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be an $\mathcal{A}_\tau$-labelled transition system. We say that T is *effective* if $\rightarrow$ and $\downarrow$ are recursively enumerable. We say that T is *computable* if both the functions $out(\_)$ and $fin(\_)$ are recursive. △

The notion of effective transition system originates with Boudol [Bou85]. For the notion of computable transition system we adopt the definition from [BBK87].

We shall not go into the details of explaining more carefully what are suitable codings into natural numbers of $\mathcal{A}_\tau$ and $\mathcal{S}$, and how they should be extended to codings of $\rightarrow$, $\downarrow$, $out(\_)$ and $fin(\_)$ so that the formal theory of recursiveness makes sense for arbitrary (countable) transition systems. (The reader may want to consult [Rog67, §1.10] for more explanations.) If $\rightarrow$ and $\downarrow$ are recursively enumerable, then this, intuitively, means that there exist algorithms that enumerate the transitions in $\rightarrow$ and the states in $\downarrow$. If the functions $out(\_)$ and $fin(\_)$ are recursive, then there exists an algorithm that, given a state s, yields the list of outgoing transitions of s and determines whether $s \in \downarrow$. Note that for an RTM the functions are given by definition.

PROPOSITION 6.13. *The transition system associated with an RTM is computable.* □

Hence, unsurprisingly, if a transition system is not computable, then it is not executable either. It is easy to define transition systems that are not computable (see the following example), so there exist behaviours that are not executable. The following example takes this a little further and illustrates that there exist behaviours that are not even executable up to branching bisimilarity.

EXAMPLE 6.14. (In this and later examples, we denote by φ_x the partial recursive function with index $x \in \mathbb{N}$ in some exhaustive enumeration of partial recursive

functions, see, e.g., [Rog67].) Assume that $\mathcal{A} = \{a, b, c\}$ and consider the $\mathcal{A}$-labelled transition system $T_0 = (\mathcal{S}_0, \rightarrow_0, \uparrow_0, \downarrow_0)$ with $\mathcal{S}_0$, $\rightarrow_0$, $\uparrow_0$ and $\downarrow_0$ defined by

$$\begin{aligned}
\mathcal{S}_0 &= \{s, t, u, v, w\} \cup \{s_x \mid x \in \mathbb{N}\} \ , \\
\rightarrow_0 &= \{(s,a,t),(t,a,t),(t,b,v),(s,a,u),(u,a,u),(u,c,w)\} \\
&\qquad \cup (s,a,s_0)\} \cup \{(s_x,a,s_{x+1}) \mid x \in \mathbb{N}\} \\
&\qquad \cup \{(s_x,a,t),(s_x,a,u) \mid \varphi_x \text{ is a total function}\}
\end{aligned}$$

The transition system is depicted in Figure 6.3.

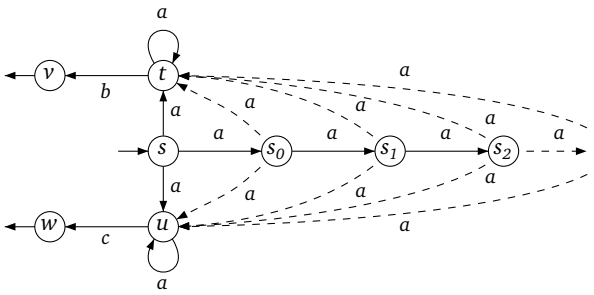

FIGURE 6.3: The transition system T_0.

To argue that T_0 is not executable up to branching bisimilarity, we prove by contradiction. Suppose that T_0 is executable up to branching bisimilarity. Then T_0 is branching bisimilar to a computable transition system T_0'. Then, in T_0', the set of states reachable by a path that contains exactly x a-transitions ($x \in \mathbb{N}$) and from which both a b- and a c-transition are still reachable, is recursively enumerable. It follows that the set of states in T_0' branching bisimilar to s_x ($x \in \mathbb{N}$) is recursively enumerable. But then, since the problem of deciding whether from some state in T_0' there is a path containing exactly one a-transition and one b-transition such that the a-transition precedes the b-transition, is also recursively enumerable, it follows that the problem of deciding whether φ_x is a total function must be recursively enumerable too, which it is not. We conclude that T_0 is not executable up to branching bisimilarity. Incidentally, note that the language associated with T_0 is $\{a^n b, a^n c \mid n \geq 1\}$, which *is* recursively enumerable (it is even context-free). ♢

Phillips associates, in [Phi93], with every effective transition system a *branching bisimilar* computable transition system of which, moreover, every state has a branching degree of at most 2. (Phillips actually establishes weak bisimilarity, but it is easy to see that branching bisimilarity holds.)

DEFINITION 6.15. Let $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ be a transition system, and let B be a natural number. We say that T has a branching degree *bounded by* B if, for every state $s \in \mathcal{S}$, $|out(s)| \leq B$. We say that T is *boundedly branching* if there exists $B \in \mathbb{N}$ such that the branching degree of T is bounded by B. △

PROPOSITION 6.16 (Phillips). *For every effective transition system T there exists a boundedly branching computable transition system T' such that $T \leftrightarrows_b T'$.* □

A crucial insight in Phillips' proof is that a divergence (i.e., an infinite sequence of τ-transitions) can be exploited to simulate a state of which the set of outgoing transitions is recursively enumerable, but not recursive. The following example, inspired by [Dar89], shows that introducing divergence is unavoidable.

EXAMPLE 6.17. Assume that $\mathcal{A} = \{a, b\}$, and consider the transition system $T_1 = (\mathcal{S}_1, \rightarrow_1, \uparrow_1, \downarrow_1)$ with $\mathcal{S}_1$, $\rightarrow_1$, $\uparrow_1$ and $\downarrow_1$ defined by

$$\begin{aligned}
&\mathcal{S}_1 = \{ s_{1,x}, t_{1,x} \mid x \in \mathbb{N} \} \ , \\
&\rightarrow_1 = \{ (s_{1,x}, a, s_{1,x+1}) \mid x \in \mathbb{N} \} \cup \{ (s_{1,x}, b, t_{1,x}) \mid x \in \mathbb{N} \} \ , \\
&\uparrow_1 = s_{1,0} \ , \text{ and} \\
&\downarrow_1 = \{ t_{1,x} \mid \varphi_x(x) \text{ converges} \} \ .
\end{aligned}$$

The transition system is depicted in Figure 6.4.

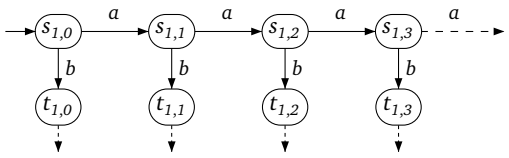

FIGURE 6.4: The transition system T_1.

Now suppose that T_2 is a transition system such that $T_1 \leftrightarrows^{\Delta}_{\mathrm{b}} T_2$, as witnessed by some divergence-preserving branching bisimulation relation $\mathcal{R}$; we argue that T_2 is not computable by deriving a contradiction from the assumption that it is.

Clearly, since T_1 does not admit infinite sequences of τ-transitions, if $\mathcal{R}$ is divergence-preserving, then T_2 does not admit infinite sequences of τ-transitions either. Let s_1 be some state in T_1 and s_2 in T_2. It follows that if $s_1 \mathcal{R} s_2$, then there exists a state s_2' in T_2 such that $s_2 \longrightarrow\!\!\!\!\!\rightarrow_2 s_2'$, $s_1 \mathcal{R} s_2'$, and $s_2' \xrightarrow{\tau}\!\!\!\!\!\!\rightarrow$. 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 $\varphi_x(x)$ converges:

1. it starts from the initial state $\uparrow_2$ of T_2;
2. it runs the algorithm to find a state without outgoing τ-transitions, and then it repeats the following steps x times:
   a) execute the a-transition enabled in the reached state;
   b) run the algorithm to find a state without outgoing τ-transitions again;

   since $\uparrow_1 \mathcal{R} \uparrow_2$, this 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 that must be enabled in $s_{2,x}$, followed, again, by the algorithm to find a state without outgoing τ-transitions; this yields a state $t_{2,x}$, without any outgoing transitions, such that $t_{1,x} \mathcal{R} t_{2,x}$.

From $t_{1,x} \mathrel{\mathcal{R}} t_{2,x}$ it follows that $t_{2,x} \in \downarrow_2$ iff $\varphi_x(x)$ converges, so the problem of deciding whether $\varphi_x(x)$ converges has been reduced to the problem of deciding whether $t_{2,x} \in \downarrow_2$. Since it is undecidable if $\varphi_x(x)$ converges, it follows that $\downarrow_2$ is not recursive, which contradicts our assumption that T_2 is computable. $\diamondsuit$

6.2.2 Simulating Boundedly Branching Computable Transition Systems

By Proposition 6.16, in order to prove that every effective transition system can be simulated up to branching bisimilarity by an RTM, it suffices to prove that every boundedly branching computable transition system can be simulated by an RTM.

Let $T = (\mathcal{S}_T, \rightarrow_T, \uparrow_T, \downarrow_T)$ be a boundedly branching computable transition system, say with branching degree bounded by B. We shall construct an RTM $\mathcal{M} = (\mathcal{S}_\mathcal{M}, \rightarrow_\mathcal{M}, \uparrow_\mathcal{M}, \downarrow_\mathcal{M})$, called the *simulator* for T, such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_{\mathrm{b}}^{\Delta} T$.

Tape contents

Let us assume encodings of the functions $\ulcorner \_ \urcorner : \mathit{out}(\_) \to \mathbb{N}$, $\ulcorner \_ \urcorner : \mathit{fin}(\_) \to \mathbb{N}$, and the sets $\ulcorner \_ \urcorner : \mathcal{A}_\tau \to \{1, \ldots, |\mathcal{A}_\tau|\}$ and $\ulcorner \_ \urcorner : \mathcal{S}_T \to \mathbb{N}$; the simulator RTM $\mathcal{M}$ stores these functions, actions, states and transitions on its tape as natural numbers. The existence of the encodings of the functions $\mathit{out}(\_)$ and $\mathit{fin}(\_)$ is due to the fact that they are recursive.

The way in which natural numbers are represented as sequences over some finite alphabet of tape symbols is largely irrelevant, but in our construction below it is sometimes convenient to have an explicit representation. In such cases, we assume that numbers are stored in unary notation using the symbol 1. That is, a natural number n is represented on the tape as the string 1^{n+1} of $n+1$ occurrences of the symbol 1. In addition to the symbol 1, we use the symbols $[\![$ and $]\!]$ to enclose the (static) codes of the two functions that steer the simulation of T on the tape, $|$ to separate the elements of a tuple of natural numbers, and $\#$ to separate tuples. The RTM $\mathcal{M}$ constructed below will incorporate the operation of some auxiliary Turing machines that may use some extra encoding and symbols; let $\mathcal{D}'$ be the collection of all these extra symbols. Then the tape alphabet $\mathcal{D}$ of $\mathcal{M}$ is

$$\mathcal{D} = \{1, [\![,]\!], |, \#\} \cup \mathcal{D}' \ .$$

We shall define $\mathcal{M}$ as the union of three fragments: an *initialisation fragment*, a *state fragment*, and a *step fragment*. The initialisation fragment prepares $\mathcal{M}$ for simulation, the state fragment calculates the possible transitions that can be taken from the current state and the step fragment actually simulates the step to the next state. See also the overview diagram in Figure 6.6 later on.

Instead of directly using (conventional) Turing machines computing $\mathit{out}(\_)$ and $\mathit{fin}(\_)$ we store their codes on the tape and use a Turing machine to interpret these codes. This is slightly more generic than necessary; the advantage of proceeding in this way is that we can easily adapt the simulator to obtain a universal RTM (in Section 6.2.4).

Initialisation fragment

The *initialisation fragment* Init prepares the tape for simulation of T by first writing the symbol $[\![$ on the tape, followed by (the codes of) the functions $out(\_)$ and $fin(\_)$ belonging to T which are separated by the symbol $|$. Then it writes the symbol $]\!]$ on the tape followed by the code of the initial state of T. Thereafter, it returns the tape head to the symbol $]\!]$. Let $\mathcal{M}_i$ be a (conventional) Turing machine that achieves precisely this; when started with an empty tape ($\check{\square}$), it halts with the tape instance $[\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner \uparrow_T \urcorner$.

The set of states of Init is defined as

$$\mathcal{S}_{\mathsf{Init}} = \mathcal{S}_{\mathcal{M}_i} \setminus \downarrow_{\mathcal{M}_i} \ ,$$

its initial state is defined as

$$\uparrow_{\mathsf{Init}} = \uparrow_{\mathcal{M}_i} \ ; \text{ and}$$

its set of transitions is defined as

$$\begin{aligned} \rightarrow_{\mathsf{Init}} = \{ (in, d, \tau, e, M, in') \mid (in, d, e, M, in') \in \rightarrow_{\mathcal{M}_i},\ in' \in \mathcal{S}_{\mathcal{M}_i} \setminus \downarrow_{\mathcal{M}_i} \} \\ \cup \{ (in, d, \tau, e, M, \uparrow_{\mathsf{State}}) \mid (in, d, e, M, in') \in \rightarrow_{\mathcal{M}_i},\ in' \in \downarrow_{\mathcal{M}_i} \} \ . \end{aligned}$$

LEMMA 6.18. *The fragment* Init *has a fully deterministic internal computation from* $(\uparrow_{\mathsf{Init}}, \check{\square})$ *to* $(\uparrow_{\mathsf{State}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner \uparrow_T \urcorner)$. □

State fragment

The *state fragment* State replaces the code of the current state on the tape by a sequence of codes that represents the behaviour of T in the current state. It is assumed that it starts with a tape instance of the form $[\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner s \urcorner$ with $s \in \mathcal{S}_T$.

Recall that the functions $out(\_)$ and $fin(\_)$ are both recursive. Hence, by [Rog67] there exists a (conventional) deterministic Turing machine $\mathcal{M}_s$ that, when it is started with a tape instance $[\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner s \urcorner$ terminates with the tape instance

$$[\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner (s \in \downarrow_T)? \urcorner | \ulcorner a_1 \urcorner | \cdots | \ulcorner a_k \urcorner \# \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \# \ ,$$

where $out(s) = \{ (a_i, s_i) \mid 1 \leq i \leq k \}$ and $\ulcorner (s \in \downarrow_T)? \urcorner$ is a special code denoting $fin(s)$, i.e. $\ulcorner true \urcorner$ or $\ulcorner false \urcorner$. Note that, since the branching degree of T is bounded by B, we have that $k \leq B$. We assume without loss of generality that the Turing machine $\mathcal{M}_s$ first copies the codes of $out(\_)$ and $fin(\_)$ to the right of the symbol $]\!]$ and thereafter never crosses this boundary symbol again for its computation. We refer to the sequence $(s \in \downarrow_T)?, a_1, \ldots, a_k$ that is generated and stored on the tape by $\mathcal{M}_s$ as the *menu* in s.

The set of states of State is defined as

$$\mathcal{S}_{\mathsf{State}} = \mathcal{S}_{\mathcal{M}_s} \setminus \downarrow_{\mathcal{M}_s} \ ;$$

its initial state is defined as

$$\uparrow_{\mathsf{State}} = \uparrow_{\mathcal{M}_s} \ ; \text{ and}$$

its set of transitions is defined as

$$\begin{aligned}\rightarrow_{\mathsf{State}} = \{\,(st,d,\tau,e,M,st') \mid (st,d,e,M,st') \in \rightarrow_{\mathcal{M}_s},\ st' \in \mathcal{S}_{\mathsf{State}} \setminus \downarrow_{\mathcal{M}_s}\,\} \\ \cup\, \{\,(st,d,\tau,e,M,\uparrow_{\mathsf{Step}}) \mid (st,d,e,M,st') \in \rightarrow_{\mathcal{M}_s},\ st' \in \downarrow_{\mathcal{M}_s}\,\}\ .\end{aligned}$$

(Note how we associate with $\mathcal{M}_s$ (a fragment of) an RTM by adding τ-labels to its transitions.)

LEMMA 6.19. *The fragment* State *has a fully deterministic internal computation from configuration* $(\uparrow_{\mathsf{State}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!]\ulcorner s \urcorner)$ *for each* $s \in \mathcal{S}_T$ *to*

$$(\uparrow_{\mathsf{Step}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!]\ulcorner (s \in \downarrow_T)? \urcorner | \ulcorner a_1 \urcorner | \cdots | \ulcorner a_k \urcorner \# \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#)\ ,$$

where the part at the right of the symbol $]\!]$ *on the tape represents the menu generated by applying the functions out*(_) *and fin*(_) *to s.* □

Step fragment

The purpose of the *step fragment* Step is to select an action a_i from the set of enabled actions in the current state, execute that action, and remove $\ulcorner (s \in \downarrow_T)? \urcorner$ and all (codes of) actions and states from the tape, except the code of the target state of the a_i-transition.

The state s in the simulated transition system T embodies a choice between its k outgoing transitions $s \xrightarrow{a_1} s_1$, …, $s \xrightarrow{a_k} s_k$, and is terminating if, and only if, $s \in \downarrow_T$. In order to get a branching bisimulation between T and the transition system associated with $\mathcal{M}$, the latter will necessarily have to include a configuration offering the same choice of outgoing transitions and the same termination behaviour. It is important to note that branching bisimilarity does not, e.g., allow the choice for one of the outgoing transitions to be made by a computation (resulting in a sequence of τ-transitions) that eliminates options one by one. The fragment Step will therefore have to include a special state $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$, for every potential menu. (Note that, since $k \leq B$, there will be at most $N = \sum_{k=0}^{B} 2 \cdot |\mathcal{A}_\tau|^k$ different menus in T.)

The functionality of the step fragment is split up in two parts: before and after the simulation of an a_i-transition. The first part uses the RTM $\mathcal{M}_{pd}$ to decode the menu on the tape ending up in the state $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$ from which termination, if enabled, or an a_i-transition can occur. In case the transition is performed, the second part finds the target state s_i of the a_i-transition. The RTM $\mathcal{M}_{pm}$ will move the code $\ulcorner s_i \urcorner$ to the right of the symbol $]\!]$ and the RTM $\mathcal{M}_{pc}$ will empty the remaining part of the tape.

The fragment Step starts from a tape instance of the form

$$[\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!]\ulcorner (s \in \downarrow_T)? \urcorner | \ulcorner a_1 \urcorner | \cdots | \ulcorner a_k \urcorner \# \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#$$

and then progresses to the state $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$, while removing from the tape the symbols $\ulcorner(s \in \downarrow_T)?\urcorner|\ulcorner a_1\urcorner|\cdots|\ulcorner a_k\urcorner$; this is a matter of decoding the information on the tape. For this decoding part we assume that $\mathcal{M}_{pd}$ is an RTM that halts with the tape instance $[\![\ulcorner out\urcorner|\ulcorner fin\urcorner]\!]\square\cdots\square\check{\#}\ulcorner s_1\urcorner|\cdots|\ulcorner s_k\urcorner\#$. Among the states of $\mathcal{M}_{pd}$ we have the previously mentioned special states $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$ for all $(s \in \downarrow_T?) \in \{true, false\}, a_1, \ldots, a_k \in \mathcal{A}_\tau, k \leq B$. A state $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$ is declared final if, and only if, $s \in \downarrow_T$, and it has an outgoing a_i-transitions to the states ne_i $(1 \leq i \leq k)$.

After the decoding part, the action a_i can be performed (while removing the symbol #) and the fragment ends up in the state ne_i. The goal of the states ne_i down to ne_1 is to find the code $\ulcorner s_i\urcorner$, replacing the symbols preceding $\ulcorner s_i\urcorner$ by $\square$, and to yield the tape instance $[\![\ulcorner out\urcorner|\ulcorner fin\urcorner]\!]\square\cdots\square\overset{\succ}{\ulcorner} s_i\urcorner|\ldots|\ulcorner s_k\urcorner\#$.

Let $\mathcal{M}_{pm}$ be an RTM that, when started with above tape instance, moves the found state code $\ulcorner s_i\urcorner$ to the right of the symbol $]\!]$ and halts with the tape instance $[\![\ulcorner out\urcorner|\ulcorner fin\urcorner]\!]\ulcorner s_i\urcorner\square\cdots\square\check{|}\ulcorner s_{i+1}\urcorner|\ldots|\ulcorner s_k\urcorner\#$.

Then, let $\mathcal{M}_{pc}$ be an RTM that, when started with the above tape instance, empties the remaining part of the tape, moves the tape head back to the symbol $]\!]$ and halts with the tape instance $[\![\ulcorner out\urcorner|\ulcorner fin\urcorner]\!]\check{\ulcorner} s_i\urcorner$.

The set of states of Step is defined as

$$\mathcal{S}_{\mathsf{Step}} = (\mathcal{S}_{\mathcal{M}_{pd}} \cup \{ne_1, \ldots, ne_B\} \cup \mathcal{S}_{\mathcal{M}_{pm}} \cup \mathcal{S}_{\mathcal{M}_{pc}}) \setminus (\downarrow_{\mathcal{M}_{pd}} \cup \downarrow_{\mathcal{M}_{pm}} \cup \downarrow_{\mathcal{M}_{pc}}) \ ;$$

its initial state is defined as

$$\uparrow_{\mathsf{Step}} = \uparrow_{\mathcal{M}_{pd}} \ ; \text{ and}$$

its set of transitions is defined as

$$\begin{aligned}
\rightarrow_{\mathsf{Step}} = {} & \{(sp, d, \tau, e, M, sp') \mid (sp, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pd}}\} \\
& \cup \{(sp_{(s\in\downarrow_T),a_1,\ldots,a_k}, \#, a_i, \square, R, ne_i) \\
& \qquad \mid (s \in \mathcal{S}_T)? \in \{true, false\}, a_1, \ldots, a_k \in \mathcal{A}_\tau, k \leq B, 1 \leq i \leq k\} \\
& \cup \{(ne_k, 1, \tau, \square, R, ne_k), (ne_k, |, \tau, \square, R, ne_{k-1}) \mid 1 < k \leq B\} \\
& \cup \{(ne_1, d, \tau, e, M, sp') \mid (\uparrow_{\mathcal{M}_{pm}}, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pm}}\} \\
& \cup \{(sp, d, \tau, e, M, sp') \mid (sp, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pm}},\ sp' \in \mathcal{S}_{\mathcal{M}_{pm}} \setminus \downarrow_{\mathcal{M}_{pm}}\} \\
& \cup \{(sp, d, \tau, e, M, \uparrow_{\mathcal{M}_{pc}}) \mid (sp, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pm}},\ sp' \in \downarrow_{\mathcal{M}_{pm}}\} \\
& \cup \{(sp, d, \tau, e, M, sp') \mid (sp, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pc}},\ sp' \in \mathcal{S}_{\mathcal{M}_{pc}} \setminus \downarrow_{\mathcal{M}_{pc}}\} \\
& \cup \{(sp, d, \tau, e, M, \uparrow_{\mathsf{State}}) \mid (sp, d, \tau, e, M, sp') \in \rightarrow_{\mathcal{M}_{pc}},\ sp' \in \downarrow_{\mathcal{M}_{pc}}\} \ .
\end{aligned}$$

See Figure 6.5 for a schematic overview of the fragment Step. Note that in this figure – for clarity reasons – only one of possibly many states $sp_{(s\in\downarrow)?,a_1,\ldots,a_k}$ and transitions thereto is drawn.

As mentioned before we can split the fragment up in two parts; we obtain the following two lemmas. First, a lemma for the internal computation up until the action a_i can be performed.

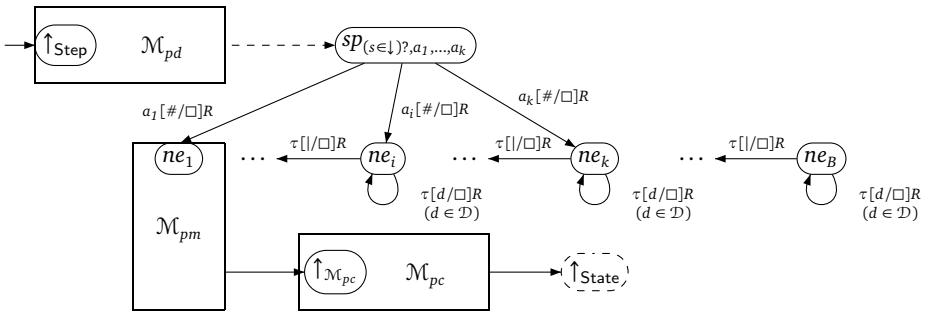

FIGURE 6.5: Diagram of the step fragment.

LEMMA 6.20. *The fragment* Step *(using the auxiliary RTM $\mathcal{M}_{pd}$) has a fully deterministic internal computation from*

$$(\uparrow_{\mathsf{Step}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner (s \in \downarrow_T)? \urcorner | \ulcorner a_1 \urcorner | \cdots | \ulcorner a_k \urcorner \# \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#)$$

to

$$(sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \square \cdots \square \# \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#) \,. \qquad \square$$

Second, a lemma for the internal computation after an action a_i is performed.

LEMMA 6.21. *The fragment* Step *(using the auxiliary RTMs $\mathcal{M}_{pm}$ and $\mathcal{M}_{pc}$) has a fully deterministic internal computation from* $(ne_i, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \square \cdots \square \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#)$ *to* $(\uparrow_{\mathsf{State}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner s_i \urcorner)$. $\square$

Simulator

The *simulator RTM* $\mathcal{M} = (\mathcal{S}_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, \uparrow_{\mathcal{M}}, \downarrow_{\mathcal{M}})$ is defined as the combination of the fragments Init, State and Step defined above. The set of states of $\mathcal{M}$ is defined as the union of the sets of states of all fragments:

$$\mathcal{S}_{\mathcal{M}} = \mathcal{S}_{\mathsf{Init}} \cup \mathcal{S}_{\mathsf{State}} \cup \mathcal{S}_{\mathsf{Step}} \;;$$

the transition relation of $\mathcal{M}$ is the union of the transition relations of all fragments:

$$\mathcal{S}_{\mathcal{M}} = \rightarrow_{\mathsf{Init}} \cup \rightarrow_{\mathsf{State}} \cup \rightarrow_{\mathsf{Step}} \;;$$

the initial state of $\mathcal{M}$ is the initial state of Init:

$$\uparrow_{\mathcal{M}} = \uparrow_{\mathsf{Init}} \;; \text{ and}$$

the set of final states of $\mathcal{M}$ consists of the states of Step $sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k}$ where s is a final state in T

$$\downarrow_{\mathcal{M}} = \{ sp_{(s\in\downarrow_T)?,a_1,\ldots,a_k} \mid s \in \downarrow_T \} \;.$$

Figure 6.6 schematically illustrates how the fragments are combined to constitute the simulator $\mathcal{M}$.

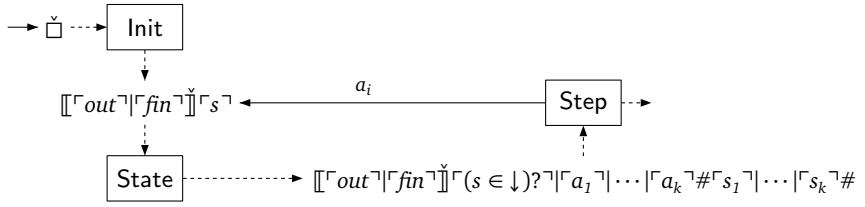

FIGURE 6.6: Diagram of the deterministic computable transition system simulator.

THEOREM 6.22. *For every boundedly branching computable transition system T there exists a reactive Turing machine $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^{\Delta} T$.* □

PROOF. Consider the RTM $\mathcal{M}$ of which the definition is sketched above. Using Lemma 6.18 we define the following relation:

$$\mathcal{R}_{\uparrow} = \{ (\uparrow_T, t) \mid t \in \{ (\uparrow_{\mathsf{Init}}, \check{\square}), \ldots, (\uparrow_{\mathsf{State}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \ulcorner \uparrow_T \urcorner) \} \} \ .$$

Using Lemmas 6.19, 6.20 and 6.21, we define the following relation for each $s \in \mathcal{S}_T$:

$$\mathcal{R}_s = \{ (s,t) \mid t \in \{ (ne_i, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \square \cdots \square \succ \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#), \ldots, (sp_{(s \in \downarrow_T)?, a_1, \ldots, a_k}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!] \square \cdots \square \check{\#} \ulcorner s_1 \urcorner | \cdots | \ulcorner s_k \urcorner \#) \} \} \ .$$

We can now define the relation

$$\mathcal{R} = \mathcal{R}_{\uparrow} \cup \bigcup_{s \in \mathcal{S}_T} \mathcal{R}_s \ .$$

The relation $\mathcal{R}$ is a divergence-preserving branching bisimulation between $\mathcal{T}(\mathcal{M})$ and T. ■

Combining the above theorem with Proposition 6.16 we can conclude that reactive Turing machines can simulate effective transition systems up to branching bisimilarity, but, in view of Example 6.17, not in a divergence-preserving manner.

COROLLARY 6.23. *For every effective transition system T there exists a reactive Turing machine $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b T$.* □

Note that all computations involved in the simulation of T are deterministic (see Lemmas 6.18–6.21). Therefore, if $\mathcal{M}$ is non-deterministic, then this is due to a state $sp_{(s \in \downarrow)?, a_1, \ldots, a_k}$ of which the menu includes some action a more than once. It follows that a deterministic computable transition system can be simulated up to divergence-preserving branching bisimilarity by a deterministic reactive Turing machine.

DEFINITION 6.24. A transition system $T = (\mathcal{S}, \rightarrow, \uparrow, \downarrow)$ is *deterministic* if, for every state $s \in \mathcal{S}$ and for every $a \in \mathcal{A}_\tau$, $s \xrightarrow{a} s_1$ and $s \xrightarrow{a} s_2$ implies $s_1 = s_2$. △

Clearly, if T is deterministic, then, for every state s in T, $|out(s)| \leq |\mathcal{A}_\tau|$. So a deterministic transition system is boundedly branching, and therefore we get the following corollary to Theorem 6.22.

COROLLARY 6.25. *For every deterministic computable transition system T there exists a deterministic reactive Turing machine $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^\Delta T$.* □

6.2.3 Parallel Composition

Using Theorem 6.22 we can now also establish that a parallel composition of RTMs can be simulated, up to divergence-preserving branching bisimilarity, by a single RTM. To this end, note that the transition systems associated with RTMs are boundedly branching and computable. Further note that the parallel composition of boundedly branching computable transition systems is again computable. It follows that the transition system associated with a parallel composition of RTMs is boundedly branching and computable, and hence, by Theorem 6.22, there exists an RTM that simulates this transition system up to divergence-preserving branching bisimilarity. Thus we get the following corollary.

COROLLARY 6.26. *For every pair of reactive Turing machines $\mathcal{M}_1$ and $\mathcal{M}_2$ and for every set of communication channels $\mathcal{C}$ there exists an RTM $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^\Delta \mathcal{T}(\llbracket \mathcal{M}_1 \parallel \mathcal{M}_2 \rrbracket_\mathcal{C})$.* □

6.2.4 Universality

A classical and central notion in the theory of computation is the *universal Turing machine*: a Turing machine that can simulate any arbitrary Turing machine on arbitrary input. Here, the (encoded) description of a Turing machine and the input are present on the tape beforehand. In this subsection we propose a notion of universal RTM and investigate to what extent such universal RTMs exist. Naturally, our notion of universal RTM should reflect our desiderata for introducing RTMs.

Firstly, since our main aim is to formalise communication explicitly, we want a universal RTM to first receive input via communication rather than finding it on its tape at the beginning of its operation (recall our assumption that the tape of our RTM is initially empty). To this end, we associate with the encoding $\ulcorner \mathcal{M} \urcorner$ of some RTM $\mathcal{M}$ (see [Rog67]) an RTM $\overline{\mathcal{M}}$ that sends $\ulcorner \mathcal{M} \urcorner$ along channel u, not used by $\mathcal{M}$ itself, and then terminates. This RTM $\overline{\mathcal{M}}$ will be put in parallel with the universal RTM to be defined, abstracting from communication over the channel u.

Secondly, the simulation of other Turing machines by a universal Turing machine is in the classical theory up to language equivalence. For example, Hopcroft, Motwani and Ullman define in [HMU06, Section 9.2.3] the universal Turing machine for the so-called universal language. Language equivalence is, however, too coarse if one is interested in the behaviour of an RTM rather than only the function it computes. Our notion of universal RTM should simulate every RTM up to divergence-preserving branching bisimulation instead of language equivalence.

An RTM $\mathcal{U}$ is *universal* (given some coding of RTMs) if for every RTM $\mathcal{M}$ it holds that $\mathcal{T}(\mathcal{M}) \leftrightarrows_{\mathrm{b}}^{\Delta} \left[\overline{\mathcal{M}} \,\|\, \mathcal{U}\right]_u$. However, we will show now that such a universal RTM $\mathcal{U}$ does not exist.

PROPOSITION 6.27. *There does not exist an RTM $\mathcal{U}$ such that for all RTM $\mathcal{M}$ it holds that $\left[\overline{\mathcal{M}} \,\|\, \mathcal{U}\right]_u \leftrightarrows_b^{\Delta} \mathcal{T}(\mathcal{M})$.* □

PROOF. Assume the existence of a universal RTM $\mathcal{U}$. Since $\mathcal{U}$ is an RTM, it has an associated transition system that has a branching degree bounded by, say, B. Now, assume an RTM $\mathcal{M}$ such that $\mathcal{T}(\mathcal{M})$ has no divergence and has a branching degree bounded by $B+1$. In particular, $\mathcal{T}(\mathcal{M})$ has a state s that realises the branching degree bound by having transitions $a_1, \ldots, a_{B+1}$ to $B+1$ pairwise non-bisimilar target states. If $\mathcal{U}$ were to simulate $\mathcal{M}$ up to divergence-preserving branching bisimulation, then there is a state s' in $\left[\overline{\mathcal{M}} \,\|\, \mathcal{U}\right]_u$ related to s that cannot do any (inert) τ-transitions, but still has to simulate all transitions of s. This means that s' must have a branching degree of $B+1$. This is a contradiction. ■

If we insist on simulation up to divergence-preserving branching bisimilarity, then we need to relax the notion of universality.

DEFINITION 6.28. An RTM $\mathcal{U}_B$ is *universal up to branching degree B* if for every RTM $\mathcal{M}$ with $\mathcal{T}(\mathcal{M})$ bounded by branching degree B it holds that $\mathcal{T}(\mathcal{M}) \leftrightarrows_{\mathrm{b}}^{\Delta} \left[\overline{\mathcal{M}} \,\|\, \mathcal{U}_B\right]_u$. △

We now present the construction of a collection of RTMs $\mathcal{U}_B$ for all branching degree bounds B. For the remainder of this section let $\mathcal{M} = (\mathcal{S}_M, \mathcal{A}_M, \mathcal{D}_M, \rightarrow_M, \uparrow_M, \downarrow_M)$ be an RTM such that the branching degree of $\mathcal{T}(\mathcal{M})$ is bounded by B. From our Definition 6.12, Proposition 6.13, the explanations in [Phi93], and by applying some standard recursion-theoretic techniques such as the enumeration theorem (see [Rog67]), it can be shown that the codes of the functions *out*(_) and *fin*(_) belonging to $\mathcal{T}(\mathcal{M})$ are recursively computable from $\ulcorner\mathcal{M}\urcorner$. Therefore, we can reuse the simulator RTM defined in Section 6.2.2; it suffices to adapt its initialisation fragment.

Instead of writing the codes of the functions *out*(_) and *fin*(_) and the initial state directly on the tape, the *initialisation fragment* InitU receives the code $\ulcorner\mathcal{M}\urcorner$ of an arbitrary $\mathcal{M}$ along some dedicated channel u, yielding the tape instance $\ulcorner\mathcal{M}\urcorner$. Let $\mathcal{M}_{ri}$ be an RTM that handles the receiving and storing of the code $\ulcorner\mathcal{M}\urcorner$ over channel u when started from an empty tape.

Then, it recursively computes, from $\ulcorner\mathcal{M}\urcorner$, the codes of the functions *out*(_) and *fin*(_), and the initial state $\uparrow_{\mathcal{M}}$ of $\mathcal{T}(\mathcal{M})$ and stores these on the tape. As mentioned before, these functions can be computed recursively, and let $\mathcal{M}_{ci}$ be the deterministic Turing machine that, when started from the tape instance $\ulcorner\mathcal{M}\urcorner$ halts with the tape instance $[\![\ulcorner out\urcorner | \ulcorner fin\urcorner]\!] \ulcorner\uparrow_{\mathcal{M}}\urcorner$.

The set of states of InitU is defined as

$$\mathcal{S}_{\mathsf{InitU}} = (\mathcal{S}_{\mathcal{M}_{ri}} \cup \mathcal{S}_{\mathcal{M}_{ci}}) \setminus (\downarrow_{\mathcal{M}_{ri}} \cup \downarrow_{\mathcal{M}_{ci}}) \ ,$$

its initial state is defined as

$$\uparrow_{\mathsf{InitU}} = \uparrow_{\mathcal{M}_{ri}} \ ; \text{ and}$$

its set of transitions is defined as

$$\begin{aligned}
\rightarrow_{\mathsf{InitU}} = \{\, &(in, d, \tau, e, M, in') \mid (in, d, \tau, e, M, in') \in \rightarrow_{\mathcal{M}_{ri}},\ in' \in \mathcal{S}_{\mathcal{M}_{ri}} \setminus \downarrow_{\mathcal{M}_{ri}} \} \\
&\cup \{\, (in, d, \tau, e, M, \uparrow_{\mathcal{M}_{ci}}) \mid (in, d, \tau, e, M, in') \in \rightarrow_{\mathcal{M}_{ri}},\ in' \in \downarrow_{\mathcal{M}_{ri}} \} \\
&\cup \{\, (in, d, \tau, e, M, in') \mid (in, d, e, M, in') \in \rightarrow_{\mathcal{M}_{ci}},\ in' \in \mathcal{S}_{\mathcal{M}_{ci}} \setminus \downarrow_{\mathcal{M}_{ci}} \} \\
&\cup \{\, (in, d, \tau, e, M, \uparrow_{\mathsf{State}}) \mid (in, d, e, M, in') \in \rightarrow_{\mathcal{M}_{ci}},\ in' \in \downarrow_{\mathcal{M}_{ci}} \}
\end{aligned}$$

Note that Lemma 6.18 holds for this fragment InitU as well, albeit that the path constitutes of a different set of configurations.

LEMMA 6.29. *The fragment* InitU *has a fully deterministic internal computation from* $(\uparrow_{\mathsf{InitU}}, \check{\square})$ *to* $(\uparrow_{\mathsf{State}}, [\![\ulcorner out \urcorner | \ulcorner fin \urcorner]\!]^{\check{}} \ulcorner \uparrow_{\mathcal{M}} \urcorner)$. □

Now, when the universal initialisation fragment sets up the simulation, the state and step fragments (that have already been defined in the previous section) can perform the simulation as before. We define the *universal RTM* $\mathcal{U}_B = (\mathcal{S}_{\mathcal{U}_B}, \mathcal{A}_{\mathcal{U}_B}, \mathcal{U}_{\mathcal{B}}, \rightarrow_{\mathcal{U}_B}, \uparrow_{\mathcal{U}_B}, \downarrow_{\mathcal{U}_B})$ for each branching degree B as the combination of the fragments InitU, State and Step defined above. Recall that the fragment Step contains states for every possible menu but that these menus have a branching degree that is bounded by B. Because of this we can reuse the step fragment; the definition of fragment is independent of the transition function it is simulating and only parametrized by the branching degree bound B.

The set of states of each particular $\mathcal{U}_B$ is defined as the union of the sets of states of the fragments:

$$\mathcal{S}_{\mathcal{U}_B} = \mathcal{S}_{\mathsf{InitU}} \cup \mathcal{S}_{\mathsf{State}} \cup \mathcal{S}_{\mathsf{Step}} \ ;$$

the transition relation of $\mathcal{U}_B$ is the union of the transition relations of all fragments:

$$\mathcal{S}_{\mathcal{U}_B} = \rightarrow_{\mathsf{InitU}} \cup \rightarrow_{\mathsf{State}} \cup \rightarrow_{\mathsf{Step}} \ ;$$

the initial state of $\mathcal{U}_B$ is the initial state of InitU:

$$\uparrow_{\mathcal{U}_B} = \uparrow_{\mathsf{InitU}} \ ; \text{ and}$$

the set of final states of $\mathcal{U}_B$ consists of the states of Step $sp_{(s \in \downarrow_T)?, a_1, \ldots, a_k}$ where s is a final configuration in $\mathcal{T}(\mathcal{M})$

$$\downarrow_{\mathcal{U}_B} = \{\, sp_{(s \in \downarrow_T)?, a_1, \ldots, a_k} \mid s \in \downarrow_T \} \ .$$

THEOREM 6.30. *For every B there exists an RTM* $\mathcal{U}_B$ *such that, for all RTMs* $\mathcal{M}$ *with a branching degree bounded by B, it holds that* $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^{\Delta} \left[\!\left[\overline{\mathcal{M}} \parallel \mathcal{U}_B \right]\!\right]_u$. □

If we drop the requirement that the simulation has to be divergence-preserving, we can find a single universal RTM. We replace the Turing machine $\mathcal{M}_{ci}$ in the fragment InitU by an adapted version that besides calculating *out*(_) and *fin*(_) also modifies *out*(_) to reduce the branching degree to at most 2 [BBK87]. This is, necessarily, at the cost of introducing divergence. The resulting universal RTM $\mathcal{U}$ is universal up to branching bisimulation.

THEOREM 6.31. *There exists an RTM $\mathcal{U}$ such that, for all RTMs $\mathcal{M}$, it holds that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b \left[\overline{\mathcal{M}} \,\|\, \mathcal{U}\right]_u$.* □

6.3 Explicit Interaction

In this section we show that, up to divergence-preserving branching bisimilarity, every executable transition system can be specified using the process theory TCP_τ [BBR09]. We do this by showing, for any given RTM, the construction of a *finite* recursive specification over TCP_τ that simulates its behaviour. Our specification will consist of a finite specification of a process that is a translated version of the finite control of the RTM, and a finite specification of tape memory. We shall prove that the parallel composition of these specifications specifies a transition system that is divergence-preserving branching bisimilar with the transition associated with the RTM. Further note that our specification deals explicitly with the interaction between the finite control and the tape of an RTM.

It follows from results obtained by Vaandrager in [Vaa92] that every TCP_τ-specification induces an effective transition system. Hence, by Corollary 6.23, we also get the converse: every transition system definable in TCP_τ is executable up to branching bisimilarity.

Since we will see that transition systems associated with TCP_τ-specifications can be simulated, up to branching bisimulation, by a finite control interacting with a queue (we will later see that we can obtain the tape process by supplementing a queue with some finite control), we can look upon the queue as the canonical TCP_τ-process.

We could argue that TCP_τ-specifications can be considered as the process-theoretic counterparts of unrestricted grammars. In automata and formal language theory a hierarchy of classes of languages with different expressivity is obtained by adding/dropping restrictions on the left-hand and right-hand side of grammars. In process theory, the stricter recursive specification format is used, and different classes of expressivity are obtained by allowing more/less operators (notably the parallel composition) in the right-hand sides. This we have also shown for regular expressions in [BLMT10]. For another study into the expressiveness of TCP_τ and the relation to different types of transition systems, we refer also to [Gla94].

We prove that for every reactive Turing machine $\mathcal{M}$ there exists a finite recursive TCP_τ-specification $E_{\mathcal{M}}$ and process expression p such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_{\mathrm{b}}^{\Delta} \mathcal{T}_{E_{\mathcal{M}}}(p)$. For clarity, we will present $E_{\mathcal{M}}$ in two steps. First, we will consider a finite recursive specification of the tape process E_T and show its correspondence with an infinite specification of the tape process. Then, we will present a fair translation of the finite

control of an RTM into a finite recursive specification E_{fc}. We conclude by showing that the correspondence of the combined finite specification $E_{\mathcal{M}}$ with the original RTM $\mathcal{M}$ holds.

The tape

The following infinite recursive specification E_T^∞ specifies the desired behaviour and interface of a tape process $T_{\delta_L \check{d} \delta_R}$ for every possible tape instance ($d \in \mathcal{D}_\Box, \delta_L, \delta_R \in \mathcal{D}_\Box^*$). Each name has an equation that expresses that the data element d under the head can be sent over channel r (read), a data element e can be received over channel w (write) to replace the data element under the head, and commands can be received over channel m (move) to move the head one position to the left (onto δ_L) or right (onto δ_R); each name has the following defining equation:

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

Note that this specification allows reading and writing and moving independently, as it was also originally defined by Turing in [Tur37].

The specification of the tape process above is clearly infinite, since we have a name for each possible tape instance. Our aim is, however, to have a finite specification. In earlier work by Baeten, Bergstra and Klop in [BBK87] a finite specification of a Turing machine is given in ACP_τ to simulate computable transition systems up to bisimilarity; the conventional Turing machine is simulated using finite control in parallel with two stacks. Their approach to model a tape as two stacks cannot be reused in our setting, which allows for states that can be terminating and have outgoing transitions at the same time. Their specification of the stack does not allow for intermediate termination, and it is not clear how to adapt it so that it does. Instead, we model the tape using a (first-in first-out) queue, which does allow for intermediate termination.

The following infinite linear recursive specification E_Q^∞ specifies the behaviour of the process Q_δ modelling a queue with contents δ that receives input over channel i and sends output over channel o (for every $d \in \mathcal{D}_\Box, \delta \in \mathcal{D}_\Box^*$):

$$\begin{aligned} Q_\varepsilon &\stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}} i?d.Q_d \ , \\ Q_{\delta d} &\stackrel{\text{def}}{=} \mathbf{1} + o!d.Q_\delta + \sum_{e \in \mathcal{D}} i?e.Q_{e \delta d} \ . \end{aligned}$$

Since we want the queue process to have a finite specification too, we use as a basis for the finite version the recursive specification originally given by Bergstra and Klop in [BK86], which uses six names, parallel composition, communication over an input channel i, output channel o and auxiliary channel ℓ, and abstraction. Bezem and Ponse have shown in [BP97] that this finite recursive specification is branching bisimilar (without the termination conditions 3 and 4 of Definition 2.5) with the

infinite recursive specification given above. In their proof, they also show that the finite recursive specification does not have infinite τ-paths, so in effect they show divergence-preserving branching bisimilarity.

An alternative finite recursive specification for the queue that we could have used is the one presented by Van Glabbeek and Vaandrager in [GV93]. Although this specification would be more in line with our specification of the stack and bag, it uses the renaming operator which is not in our specification language.

The following specification E_Q is an adaptation of the finite specification of Bergstra and Klop defining a version of the queue that always has the option to terminate.

$$Q_p^{j,k} \stackrel{\text{def}}{=} \mathbf{1} + \sum_{d \in \mathcal{D}_\square} j?d.\left[Q_k^{j,p} \parallel (\mathbf{1} + k!d.Q_j^{p,k})\right]_p \quad \text{for all } \{j,k,p\} = \{i,o,\ell\}.$$

Each name represents the process that receives data elements that are inserted over channel j, sends data elements that are removed over channel k, and uses the channel p internally. When we choose $Q_\ell^{i,o}$ as the initial name of this specification, it the same interface as the infinite queue specification E_Q^∞.

The first time the queue receives a data element, it splits into a two parallel components such that the first component is ready to receive new data elements and the second component retains the just received data element. From this moment on, every time a data element is received, a new parallel component is split off "to the right" to retain the received data element. See Figure 6.7 for a diagram of the queue process; depicted is the state when a data elements 0 and 1 have been inserted.

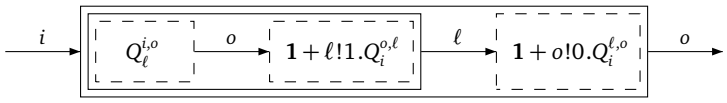

FIGURE 6.7: Diagram of the queue specification.

When a data element is removed, the parallel component becomes "empty" and the remaining data elements can be moved to the right by means of (encapsulated) communication over the internal channels, again resulting in splitting of the parallel components. See for example the following trace where data elements 0 and 1 are inserted and then removed:

$$\begin{aligned}
Q_\ell^{i,o} &\xrightarrow{i?0}\!\!\!\!\rightarrow \left[Q_o^{i,\ell} \parallel (\mathbf{1} + o!0.Q_i^{\ell,o})\right]_\ell \xrightarrow{i?1}\!\!\!\!\rightarrow \left[\left[Q_\ell^{i,o} \parallel (\mathbf{1} + \ell!1.Q_i^{o,\ell})\right]_o \parallel (\mathbf{1} + o!0.Q_i^{\ell,o})\right]_\ell \\
&\xrightarrow{o!0}\!\!\!\!\rightarrow \left[\left[Q_\ell^{i,o} \parallel Q_i^{o,\ell}\right]_o \parallel \left[Q_o^{\ell,i} \parallel (\mathbf{1} + o!1.Q_\ell^{i,o})\right]_i\right]_\ell \\
&\xrightarrow{o!1}\!\!\!\!\rightarrow \left[\left[Q_\ell^{i,o} \parallel Q_i^{o,\ell}\right]_o \parallel \left[Q_o^{\ell,i} \parallel Q_\ell^{i,o}\right]_i\right]_\ell
\end{aligned}$$

At the end, we are left with many empty cells. However, it can easily be shown that $\left[Q_k^{j,p} \parallel Q_j^{p,k}\right]_p \leftrightarrow_{\mathrm{b}}^{\Delta} Q_p^{j,k}$. Thus, the empty cells can be collapsed and removed.

The adaptation with respect to Bergstra and Klop's specification consists of the addition of a $\mathbf{1}$-summand to the defining equation of every name and to the right-most

component of the therein contained parallel composition. As a result, termination can occur in every state of the queue, and no other change in behaviour is incurred. Thus, similarly to [BP97] it can be proved that our infinite recursive specification is divergence-preserving branching bisimilar – this time with the termination conditions – with the finite recursive specification given above.

LEMMA 6.32. *We have that* $Q_\varepsilon \leftrightarrows_b^\Delta Q_\ell^{io}$. □

This lemma also allows us to use the more concise notation of the infinite specification, Q_δ for some $\delta \in \mathcal{D}_\square^*$, for a state of the queue process defined by the finite specification in the proofs below.

We can now define the finite recursive specification of the tape process E_T as the finite recursive specification of the queue E_Q and the following equations ($d \in \mathcal{D}_\square$)

$$
\begin{aligned}
H_d &\stackrel{\text{def}}{=} \mathbf{1} + r!d.H_d + \sum_{e\in\mathcal{D}_\square} w?e.H_e + m?L.H_d^L + m?R.H_d^R\,,\\
H_d^L &\stackrel{\text{def}}{=} i!d.\Big(\sum_{e\in\mathcal{D}_\square} o?e.H_e + o?\bot.i!\$.i!\bot.Back\Big)\,,\\
Back &\stackrel{\text{def}}{=} \sum_{d\in\mathcal{D}_\square} o?d.i!d.Back + o?\$.H_\square\,,\\
H_d^R &\stackrel{\text{def}}{=} i!\$.i!d.\Big(\sum_{e\in\mathcal{D}_\square} o?e.Fwd_e + o?\bot.Fwd_\bot\Big)\,,\\
Fwd_d &\stackrel{\text{def}}{=} \sum_{e\in\mathcal{D}_\square} o?e.i!d.Fwd_e + o?\bot.i!d.Fwd_\bot + o?\$.H_d\,,\\
Fwd_\bot &\stackrel{\text{def}}{=} \sum_{e\in\mathcal{D}_\square} o?e.i!\bot.Fwd_e + o?\$.i!\bot.H_\square\,.
\end{aligned}
$$

Unlike the stack, the queue allows us to reach any arbitrary data element contained within in a non-destructive way. We can repeatedly remove a data element from the queue over channel o and then insert it over channel i; we call this *shifting*. Doing this once is called a *shift operation*. Although shifting suggests the usage of a queue in a circular fashion, we have to map the (infinite and linear) data structure of the tape onto the queue. We use the queue to store only the part of the tape to the left of the head δ_L and to the right of the head δ_R and we keep the data element under the head d in a separate head process H_d. Additionally we use the marker $\bot$ as special queue data element to separate the left from the right part and also to indicate that the tape can be extended on the left or on the right, when needed, by inserting elements between $\bot$ and δ_L or between δ_R and $\bot$ respectively. Figure 6.8 illustrates the mapping of the tape instance $\delta_L \check{d} \delta_R$ and a shift operation.

In the recursive specification E_T above the main process H_d models the situation that the data element d is at the position of the head. This process H_d is put in parallel with the queue process $Q_{\delta_R\bot\delta_L}$ and provides the interface to the tape. Read and write operations for the tape are dealt with by the head process without accessing the queue; shifting only occurs when a move is requested. This is another reason to

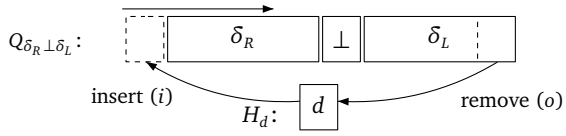

FIGURE 6.8: Diagram of the tape process.

have a separate head process that directly handles a read and write operation without touching the queue: if the data element at the position of the head would be on the queue as well, every read or write operation for the tape would cause shifting and require additional operations to get the queue in the right state again.

As mentioned above, moving the head to the left – handled by H_d^L – requires just one shift operation. However, we have to make sure not to remove the special marker $\perp$ after inserting data element d in the case that the string to the left of the head (δ_L) is empty. If this happens, we insert a search marker \$ followed by $\perp$ and cycle through the queue completely until \$ reappears. We get the following lemma that establishes that a move to the left behaves as expected using a fixed number of internal transitions.

LEMMA 6.33. *For every $d \in \mathcal{D}_\Box, \delta_L, \delta_R \in \mathcal{D}_\Box^*$ we have that*

$$\left[H_d^L \parallel Q_{\delta_R \perp \delta_L}\right]_{i,o} \leftrightarrows_b^\Delta \tau.\begin{cases} \left[H_{d_L} \parallel Q_{d\delta_R \perp \zeta_L}\right]_{i,o} & \text{if } \delta_L = \zeta_L d_L\,, \\ \left[H_\Box \parallel Q_{d\delta_R \perp}\right]_{i,o} & \text{if } \delta_L = \varepsilon\,. \end{cases}$$

□

PROOF. We prove the validity of the equation by means of an equational reasoning using the axioms of Table 2.3 (on page 18) and RSP. Then, the lemma follows by Proposition 2.18 (on page 2.18). We distinguish two cases for δ_L in $\left[H_d^L \parallel Q_{\delta_R \perp \delta_L}\right]_{i,o}$:

1. If $\delta_L = \zeta_L d_L$, then H_d^L moves the tape head to the left by performing one shift operation. So, first the data element under the head d is prefixed to the string to right of the head (δ_R), then the right-most date element (d_L) of the string to the left of the head (δ_L) is removed and put it under the head (see Figure 6.8).

$$\left[H_d^L \parallel Q_{\delta_R \perp \zeta_L d_L}\right]_{i,o} = \tau.\tau.\left[H_{d_L} \parallel Q_{d\delta_R \perp \zeta_L}\right]_{i,o} = \tau.\left[H_{d_L} \parallel Q_{d\delta_R \perp \zeta_L}\right]_{i,o}\,.$$

2. If $\delta_L = \varepsilon$, then H_d^L initially removes the special symbol $\perp$ from the queue, inserts the special search marker \$, reinserts $\perp$ and then switches to *Back*. This will shift through the queue contents until \$ is reached. At this point the queue is consistent again, so it removes the search marker and the blank symbol is put under the head.

$$\begin{aligned}\left[H_d^L \parallel Q_{\delta_R\perp}\right]_{i,o} &= \tau.\tau.\tau.\tau.\left[Back \parallel Q_{\perp\$d\delta_R}\right]_{i,o}\\ &= \tau.\tau.\tau.\tau.\tau^{2|d\delta_R|}.\left[Back \parallel Q_{d\delta_R\perp\$}\right]_{i,o}\\ &= \tau.\tau.\tau.\tau.\tau^{2|d\delta_R|}.\tau.\left[H_\Box \parallel Q_{d\delta_R\perp}\right]_{i,o}\\ &= \tau.\left[H_\Box \parallel Q_{d\delta_R\perp}\right]_{i,o}.\end{aligned}$$

We can observe that there is a fixed upper bound of $2|d\delta_R|+5$ to the number of τ-transitions (in the second case). Hence, there is no divergence. ■

Because shifting through the queue contents only goes in one direction, we have to use a different approach for moving the head to the right, which is handled by H_d^R. This time we need to have the left-most data element of the string to the right of the queue (δ_R) and we will have to shift through the entire queue contents to reach it. We do this by inserting a search marker \$ into the queue and shifting through it using a lookahead that remembers the data element that was previously removed from the queue. Once we encounter the search marker, we put this previously encountered data element under the head.

LEMMA 6.34. *For every $d \in \mathcal{D}_\Box, \delta_L, \delta_R \in \mathcal{D}_\Box^*$ we have that*

$$\left[H_d^R \parallel Q_{\delta_R\perp\delta_L}\right]_{i,o} \leftrightarrows_b^\Delta \tau.\begin{cases}\left[H_{d_R} \parallel Q_{\zeta_R\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = d_R\zeta_R\,,\\ \left[H_\Box \parallel Q_{\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = \varepsilon\,.\end{cases}$$ □

PROOF. We prove the validity of the equation by means of an equational reasoning using the axioms of Table 2.3 and RSP. Then, the lemma follows by Proposition 2.18.

$$\begin{aligned}\left[H_d^R \parallel Q_{\delta_R\perp\delta_L}\right]_{i,o} &= \tau.\tau.\tau^{2|\delta_L|+1}.\left[Fwd_\perp \parallel Q_{\delta_L d\$\delta_R}\right]_{i,o}\\ &= \begin{cases}\tau.\tau.\tau^{2|\delta_L|+1}.\tau^{2|d_R\delta_R|}.\tau.\left[H_{d_R} \parallel Q_{\delta_R\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = d_R\zeta_R\\ \tau.\tau.\tau^{2|\delta_L|+1}.\tau.\tau.\left[H_\Box \parallel Q_{\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = \varepsilon\end{cases}\\ &= \tau.\begin{cases}\left[H_{d_R} \parallel Q_{\delta_R\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = d_R\zeta_R\\ \left[H_\Box \parallel Q_{\perp\delta_L d}\right]_{i,o} & \text{if } \delta_R = \varepsilon\,.\end{cases}\end{aligned}$$

We can observe that there is a fixed upper bound of $2|\delta_L d_R\delta_R|+4$ to the number of τ-transitions. Hence, there is no divergence. ■

Putting everything together, we get the following result that shows that behavioural specification of the tape E_T^∞ is divergence-preserving branching bisimilar with the finite specification E_T.

LEMMA 6.35. *For each tape instance $\delta_L \check{d} \delta_R$ ($\delta_L, \delta_R \in \mathcal{D}_\Box^*, d \in \mathcal{D}_\Box$) we have that $T_{\delta_L \check{d} \delta_R} \leftrightarrows_b^\Delta \left[H_d \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o}$.* □

PROOF. We prove the validity of the equation by means of an equational reasoning using the axioms of Table 2.3 and RSP. Then, the lemma follows by Proposition 2.18.

$$T_{\delta_L \check{d} \delta_R} = \left[H_d \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o}$$

Now, expand the expression using axiom M and move the initial actions of H_d outside:

$$\begin{aligned} &= \mathbf{1} + r!d. \left[H_d \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} + \sum_{e \in \mathcal{D}_\Box} w?e. \left[H_e \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} \\ &\quad + m?L. \left[H_d^L \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} + m?R. \left[H_d^R \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} \end{aligned}$$

By applying Lemma 6.33 and 6.34 and axiom B we get:

$$\begin{aligned} &= \mathbf{1} + r!d. \left[H_d \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} + \sum_{e \in \mathcal{D}_\Box} w?e. \left[H_e \parallel Q_{\delta_R \perp \delta_L} \right]_{i,o} \\ &\quad + m?L.\tau. \begin{cases} \left[H_{d_L} \parallel Q_{d \delta_R \perp \zeta_L} \right]_{i,o} & \text{if } \delta_L = \zeta_L d_L \\ \left[H_\Box \parallel Q_{d \delta_R \perp} \right]_{i,o} & \text{if } \delta_L = \varepsilon \end{cases} \\ &\quad + m?R.\tau. \begin{cases} \left[H_{d_R} \parallel Q_{\zeta_R \perp \delta_L d} \right]_{i,o} & \text{if } \delta_R = d_R \zeta_L \\ \left[H_\Box \parallel Q_{\perp \delta_L d} \right]_{i,o} & \text{if } \delta_R = \varepsilon \end{cases} \\ &= \mathbf{1} + r!d.T_{\delta_L \check{d} \delta_R} + \sum_{e \in \mathcal{D}_\Box} w?e.T_{\delta_L \check{e} \delta_R} + m?L.T_{\delta_L{}^< d \delta_R} + m?R.T_{\delta_L d {}^> \delta_R} \,. \end{aligned}$$

We can observe that there are no τ-loops introduced by the specification. When moving left or right either one shift operation happens or we shift until the search marker is found, both yield a finite number of τ-transitions. Hence, no divergence is introduced. ■

Finite control

Let $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{D}, \rightarrow, \uparrow, \downarrow)$ be some RTM. We can write its associated transition system $\mathcal{T}(\mathcal{M})$ as a linear specification $E_{\mathcal{M}}^\infty$, which is infinite if $\mathcal{T}(\mathcal{M})$ is infinite.

This recursive specification $E^{\infty}_{\mathcal{M}}$ contains a name $S_{s,\delta_L \breve{d} \delta_R}$ for each reachable configuration $(s, \delta_L \breve{d} \delta_R)$ ($s \in \mathcal{S}, d \in \mathcal{D}_{\Box}, \delta_L, \delta_R \in {\mathcal{D}_{\Box}}^*$) from the initial configuration $(\uparrow, \breve{\Box})$. Each name $S_{s,\delta_L \breve{d} \delta_R}$ is defined by the following equation:

$$S_{s,\delta_L \breve{d} \delta_R} \stackrel{\text{def}}{=} \sum_{(s,d,a,e,L,t) \in \rightarrow} a.S_{t,\delta_L {}^{<}e \delta_R} + \sum_{(s,d,a,e,R,t) \in \rightarrow} a.S_{t,\delta_L e {}^{>} \delta_R} \; [+\,\mathbf{1}]_{s\downarrow} \; .$$

Here, $[+\,\mathbf{1}]_{s\downarrow}$ indicates that the $\mathbf{1}$-summand is only present if s is a final state. By construction the transition system $\mathcal{T}_{E^{\infty}_{\mathcal{M}}}(S_{\uparrow,\breve{\Box}})$ is isomorphic with $\mathcal{T}(\mathcal{M})$.

PROPOSITION 6.36. *The transition system $\mathcal{T}(\mathcal{M})$ is divergence-preserving branching bisimilar with $\mathcal{T}_{E^{\infty}_{\mathcal{M}}}(S_{\uparrow,\breve{\Box}})$.* □

Now that we have captured the behaviour of an RTM with an infinite recursive specification, it remains to construct a finite recursive specification and show that it is divergence-preserving branching bisimilar. We now present a finite recursive specification E_{fc} for the finite control of $\mathcal{M}$. For every state $s \in \mathcal{S}$ and data element $d \in \mathcal{D}_{\Box}$ we add the name $C_{s,d}$ to E_{fc} with the following equation ($s, t \in \mathcal{S}$, $a \in \mathcal{A}_{\tau}$, $d, e \in \mathcal{D}_{\Box}$, $M \in \{L, R\}$):

$$C_{s,d} \stackrel{\text{def}}{=} \sum_{(s,d,a,e,M,t) \in \rightarrow} \left(a.w!e.m!M. \sum_{f \in \mathcal{D}_{\Box}} r?f.C_{t,f} \right) \; [+\,\mathbf{1}]_{s\downarrow} \; .$$

In E_{fc} each name $C_{s,d}$ represents the part of the finite control of the RTM execution process where a transition can be chosen based on the current state and data element under the head. Once some action a is non-deterministically chosen, the tape – as explained above – is instructed over channel w to write data element e on the place under the head, then it is instructed over channel m to move the head to the left or right and finally over channel r to read the data element f under the new position of the head.

Now, if we put the finite control in parallel with the tape, we can obtain the following lemma.

LEMMA 6.37. *For each configuration $(s, \delta_L \breve{d} \delta_R)$ of a reactive Turing machine $\mathcal{M}$ we have that $S_{s,\delta_L \breve{d} \delta_R} \leftrightarrows^{\Delta}_{b} \left[C_{s,d} \parallel T_{\delta_L \breve{d} \delta_R} \right]_{r,w,m}$.* □

PROOF. In this proof we want to relate each reachable configuration, represented by the name $S_{s,\delta_L \breve{d} \delta_R}$, from the initial configuration of some RTM $\mathcal{M}$ to a name $C_{s,d}$ in the finite control specification E_{fc} put in parallel with a tape process with the corresponding contents, while encapsulating and abstracting from communication between the finite control and tape process. For example, if we have an RTM that has the configuration $(s, \delta_L \breve{d} \delta_R)$ and has the transition $s \xrightarrow{a[d/e]L} t$ in its transition relation, then the desired relation between a step in (a part of) the transition system associated with the RTM and the transitions in the specification are shown in Figure 6.9.

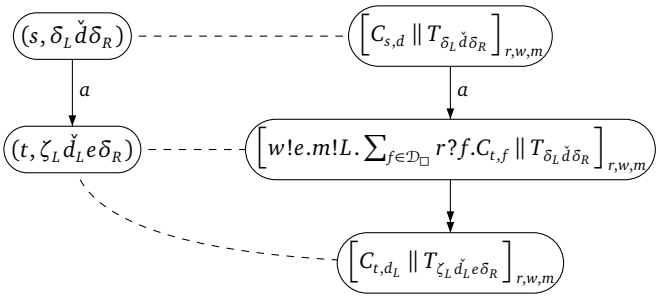

FIGURE 6.9: Relation between an RTM transition and specification transitions.

We now proceed to show that $E^\infty_{\mathcal{M}}$ is branching bisimilar with $E_{fc} \cup E^\infty_T$ by means of equational reasoning using the axioms of Table 2.3 and RSP. Then, the lemma follows by Proposition 2.18.

$$S_{s,\delta_L \check{d} \delta_R} = \left[C_{s,d} \parallel T_{\delta_L \check{d} \delta_R}\right]_{r,w,m}$$

Unfold $\left[C_{s,d} \parallel T_{\delta_L \check{d} \delta_R}\right]_{r,w,m}$ and, per transition, move the action outside (by applying almost all of the axioms).

$$= \sum_{(s,d,a,e,M,t) \in \rightarrow} a. \left[w!e.m!M. \sum_{f \in \mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L \check{d} \delta_R}\right]_{r,w,m} [+\mathbf{1}]_{s\downarrow}$$

Three communications with the tape follow by axiom CM5 and are moved outside by D1–D5 and TI1–TI5.

$$\begin{aligned}
&= \sum_{(s,d,a,e,M,t) \in \rightarrow} a.\tau. \left[m!M. \sum_{f \in \mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L \check{e} \delta_R}\right]_{r,w,m} [+\mathbf{1}]_{s\downarrow} \\
&= \sum_{(s,d,a,e,L,t) \in \rightarrow} a.\tau.\tau. \left[\sum_{f \in \mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L{}^< e \delta_R}\right]_{r,w,m} + \\
&\quad \sum_{(s,d,a,e,R,t) \in \rightarrow} a.\tau.\tau. \left[\sum_{f \in \mathcal{D}_\Box} r?f.C_{t,f} \parallel T_{\delta_L e^> \delta_R}\right]_{r,w,m} [+\mathbf{1}]_{s\downarrow} \\
&= \sum_{(s,d,a,e,L,t) \in \rightarrow} a.\tau.\tau.\tau. \left[C_{t,g} \parallel T_{\delta_L{}^< e \delta_R}\right]_{r,w,m} + \\
&\quad \sum_{(s,d,a,e,R,t) \in \rightarrow} a.\tau.\tau.\tau. \left[C_{t,g'} \parallel T_{\delta_L e^> \delta_R}\right]_{r,w,m} [+\mathbf{1}]_{s\downarrow}
\end{aligned}$$

We can remove the three τ-transitions by axiom B.

$$\begin{aligned}
&= \sum_{(s,d,a,e,L,t)\in\rightarrow} a.\Big[C_{t,g} \parallel T_{\delta_L{<}e\delta_R}\Big]_{r,w,m} + \\
&\quad \sum_{(s,d,a,e,R,t)\in\rightarrow} a.\Big[C_{t,g'} \parallel T_{\delta_L e{>}\delta_R}\Big]_{r,w,m} [+\mathbf{1}]_{s\downarrow} \\
&= \sum_{(s,d,a,e,L,t)\in\rightarrow} a.S_{t,\delta_L{<}e\delta_R} + \sum_{(s,d,a,e,R,t)\in\rightarrow} a.S_{t,\delta_L e{>}\delta_R} \quad [+\mathbf{1}]_{s\downarrow} .
\end{aligned}$$

We can observe that no τ-loops or infinite τ-paths are introduced by the specification, nor by the queue as is shown in Lemma 6.33 and 6.34. Hence, there is no divergence. ■

We have now established a finite version of the specifications for all three components of an RTM. This brings us to the following main result.

THEOREM 6.38. *For every reactive Turing machine $\mathcal{M}$ there exists a finite recursive TCP_τ-specification $E_\mathcal{M}$ and TCP_τ-process expression p such that $\mathcal{T}(\mathcal{M}) \leftrightarrows_b^\Delta \mathcal{T}_{E_\mathcal{M}}(p)$.* □

PROOF. Choose $E_\mathcal{M} = E_{fc} \cup E_T$ and $p = \Big[C_{\uparrow,\square} \parallel \big[H_\square \parallel Q_\perp\big]_{i,o}\Big]_{r,w,m}$. Then the theorem follows from Property 6.36 and Lemmas 6.32, 6.35, and 6.37. ■

As a corollary we find that every executable transition system is definable, up to divergence-preserving branching bisimilarity, by a recursive TCP_τ-specification. Since there exist recursive specifications with an unboundedly branching associated transition system (see, e.g., [BCLT10], for the converse of the aforementioned theorem), we have to give up divergence-preservation. Since the transition system associated with a finite recursive specification is clearly effective, we do get, by Corollary 6.23, the following result.

COROLLARY 6.39. *For every finite recursive TCP_τ-specification E and TCP_τ-process expression p, there exists an RTM $\mathcal{M}$ such that $\mathcal{T}_E(p) \leftrightarrows_b \mathcal{T}(\mathcal{M})$.* □

If we combine the above theorem with Theorem 6.22, Corollary 6.23 and Corollary 6.25 we get the following results.

COROLLARY 6.40. *Every boundedly branching computable transition system and every deterministic computable transition system is definable, up to to divergence-preserving branching bisimilarity, by a finite TCP_τ-specification.* □

COROLLARY 6.41. *Every effective transition system is definable, up to branching bisimilarity, by a finite TCP_τ-specification.* □

6.4 Conclusions

We have proposed a notion of reactive Turing machine and discussed its expressiveness in bisimulation semantics. Although it is not the aim of this work to contribute to the debate as to whether interactive computation is more powerful than traditional computation, our notion of RTM may nevertheless turn out to be a useful concept in the discussion. For instance, our result that the parallel composition of RTMs can be simulated by an RTM seems to contradict the conjecture implied in [GSAS04, Section 11] that concurrent interactive computation is more expressive than sequential interactive computation.

To be sure, however, we would need to firmly establish the robustness of our notion by showing that variations on its definition (e.g., multiple tracks or multiple tapes), and by showing that it can simulate the other proposals (persistent Turing machines [GSAS04], interactive Turing machines [LW00, WL08]). We also intend to consider interactive versions of other computational models. The λ-calculus would be an interesting candidate to consider, because of the well-known result that it is inherently sequential. This suggests that an interactive version of λ-calculus will be less expressive than RTMs. In particular, we conjecture that the evaluation of *parallel-or* or McCarthy's *amb* can be simulated with RTMs.

RTMs may also prove to be a useful tool in establishing the expressiveness of process theories. For instance, the transition system associated with a π-calculus expression is effective, so it can be simulated by an RTM, at least up to branching bisimilarity. The π-calculus can to some extent be seen as the interactive version of the λ-calculus. We conjecture that the converse – every executable transition system can be specified by a π-calculus expression – is also true, but leave the details for future work.

Petri showed already in his thesis [Pet62] that concurrency and interaction may serve to bridge the gap between the theoretically convenient Turing machine model of a sequential machine with unbounded memory, and the practically more realistic notion of extensible architecture of components with bounded memory. The specification we present in the proof of Theorem 6.38 is another illustration of this idea: the unbounded tape is modelled as an unbounded parallel composition. It would be interesting to further study the inherent trade-off between unbounded parallel composition and unbounded memory in the context of RTMs, considering unbounded parallel compositions of RTMs with bounded memory.

In this chapter we have established that the simulation of other RTMs by a universal RTM is not possible up to divergence-preserving branching bisimilarity. An RTM can at best simulate other RTMs with the same or a lower bound on their branching degree. But we have also shown that if we drop the divergence-preservation requirement, then universal RTMs do exist.

Finally, we have considered the correspondence between RTMs and the process theory TCP_τ. We have seen that every executable transition system is, up to divergence-preserving branching bisimilarity, definable by a finite recursive TCP_τ-specification. Interestingly, sequential composition is not used at all in the specifications. This means that BCP_τ is already sufficient and it can also be done with CCS.

Figure 6.10 presents a schematic overview of the main correspondence results of this chapter. If we consider these results, we can conclude that that bisimilarity gives a much finer perspective on the behaviour of Turing machines.

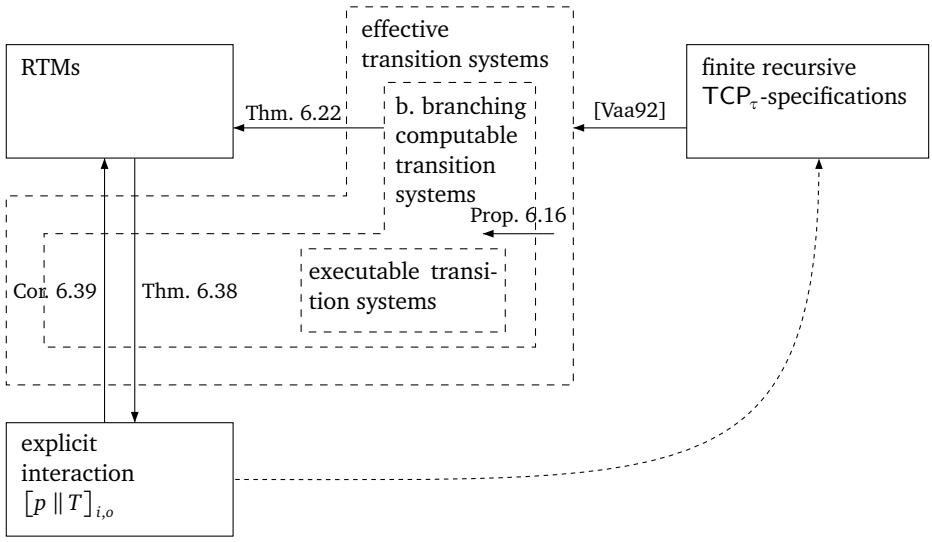

FIGURE 6.10: Correspondence results.

Note that there are a few indirect correspondences in the overview. Finite recursive TCP$_\tau$-specifications induce effective transition systems, which can be reduced to executable transition systems at the cost of losing divergence-preservation. Hence, there exists an RTM that simulates these specifications up to branching bisimilarity. Because subsequently this RTM can be defined by a finite recursive TCP$_\tau$-specification consisting of a finite-state process communicating with the queue, the queue can be considered as the canonical TCP$_\tau$-process. Note also that because RTMs can be defined by these finite recursive TCP$_\tau$- specifications that make the interaction explicit, we obtain an indirect correspondence between RTMs and finite recursive TCP$_\tau$-specifications in general.

Chapter 7

Conclusions

Following the Chomsky hierarchy, we have discussed several kinds of systems in the main chapters of this thesis (Chapters 3–6). For each class of systems we have first investigated the automata augmented with memory as a central notion, except for finite-state systems, which are memoryless. Then we have looked at a suitable specification language and investigated the correspondence of that language with the notion of automaton at hand. Finally, for each class of systems we have made the interaction within the automaton, between finite control and memory, explicit.

7.1 Automata

We have started with finite automata that can be used to represent memoryless, finite control. We have seen that finite transition systems are essentially finite automata and that, up to (divergence-preserving) branching bisimilarity, deterministic finite automata form a subclass of the (non-deterministic) finite automata.

When we augment finite automata with memory, we can associate with an automaton transition systems with a possibly infinite number of states. Based on the chosen memory and semantics, we get different classes of associated transition systems. If we augment finite automata with stack memory, we get pushdown automata; if we augment them with bag memory, we get parallel pushdown automata; and if we augment them with tape memory, we get reactive Turing machines. The different classes of automata yield different classes of transition systems. We have also seen that for (i) termination on final state, (ii) termination when the memory is empty, and (iii) termination on both final state and when the memory is empty, we get different classes of transition systems. In our definitions the stack of a PDA has an empty-test, while the bag of the PPDA does not include it. It would be interesting to see what transition system classes can be obtained if the situation is reversed. For RTMs we have investigated termination on final state only. In the future, other termination conditions could be considered.

The aforementioned differences in classes appear if we consider them up to (divergence-preserving) branching bisimilarity. If we consider the classes up to language equivalence, then all class differences collapse. Thus, we have seen that

from a process-theoretic point of view it matters how the definition of the automaton – of the memory and its interaction in particular – is chosen.

7.2 Specifications

We have seen that for each class of systems there exists a suitable specification language. For finite-state systems we have proposed the linear specifications, for pushdown systems the sequential specifications, and for parallel pushdown systems the basic parallel specifications. For computable and executable systems we have reused finite recursive TCP_τ-specifications. We have explored the correspondence between these specification languages and the automata that belong to the respective class.

For finite-state systems, the correspondence between finite automata and linear specifications holds up to isomorphism.

For pushdown systems the correspondence is deficient. We have seen that pop choice-free pushdown automata can be given, up to divergence-preserving branching bisimilarity, by a sequential specification. It is clear that not every non-pop choice-free pushdown automaton can be given by a sequential specification. However, it would be worth investigating whether the pop choice-freeness restriction is optimal. In the other direction, we have seen that due to the presence of the empty process in the specification language, we can get unbounded branching in the associated transition systems. We conjecture that a pushdown transition system cannot have unbounded branching. We applied the transparency-restrictedness restriction on sequential specifications and showed that they can be simulated, up to divergence-preserving branching bisimilarity, by a (pop choice-free) pushdown automaton. It is clear that the transparency-restricted requirement is too strict. There are non-transparency-restricted sequential specifications that do not have unbounded branching in their associated transition systems.

For parallel pushdown systems the correspondence results are different, but still deficient. We have shown that fully opaque, fully transparent, and mixed opaque/transparent recursive specifications can be simulated, up to divergence-preserving branching bisimilarity, by a parallel pushdown automaton. It is just a matter of choosing the right termination condition. For the mixed specification we have introduced an extra termination condition to obtain the correspondence result: termination on final state and transparent bag, i.e. a bag that only contains data elements which are designated as transparent. In the other direction, only single-state parallel pushdown automata can be given, up to divergence-preserving bisimilarity, by basic parallel specifications.

For computable and executable systems we have investigated the expressiveness of RTMs rather than the correspondence of RTMs with finite recursive TCP_τ-specifications. It follows from results in the literature that transition systems associated with finite recursive TCP_τ-specifications are effective transition systems, which can be reduced to executable transition systems at the cost of losing divergence-preservation. We have shown that executable transition systems can be simulated,

up to divergence-preserving branching bisimilarity, by an RTM, thus obtaining the correspondence from specifications to RTMs indirectly. In the other direction, we also obtain the result indirectly: by making the interaction in an RTM explicit, we obtain a finite recursive TCP_τ-specification.

7.3 Explicit Interaction

In the case of finite-state systems, we have discussed regular expressions rather than explicit interaction; we presented the correspondence between finite automata and extended regular expressions, i.e. regular expressions extended with communication and encapsulation. We could interpret this as making the interaction *within* a finite automaton explicit. Indeed, each state has a parallel component and control is handed over via communication.

The way the interaction within a pushdown automaton is made explicit depends on the termination condition. For termination on (final state and) empty stack we have shown that we can find a linear specification of the finite control of the pushdown automaton, put it in parallel with the sequential specification of the stack and obtain the correspondence up to divergence-preserving branching bisimilarity. For termination on final state we need an always-terminating stack. We have shown that there exists no such sequential specification. Putting the linear specification mentioned above in parallel with a finite recursive TCP_τ-specification of an always-terminating stack, we are able to obtain the correspondence.

For basic parallel pushdown automata we considered termination on final state and on (final state and) empty bag. We also considered termination on final state and transparent bag. It turned out we could find a single linear specification of the finite control of the parallel pushdown automaton. When we put it in parallel with different basic parallel specifications of the bag, we could obtain the correspondence results, up to branching bisimilarity, for parallel pushdown automata with respective termination conditions: the bag for termination on (final state and) empty bag, the transparent bag for termination on final state and the partially transparent bag for termination on final state and transparent bag. It remains an open question whether divergence-preservation can be included as well.

In the case of computable and executable systems we have made the interaction within the RTM explicit. We could find a linear specification for the finite control of the RTM. To obtain a tape we add some linear specification and put it in parallel with a finite recursive TCP_τ-specification of a queue. When the tape is put in parallel with the linear specification of the finite control, we obtained the correspondence, up to divergence-preserving branching bisimilarity, with the RTM.

7.4 Future Directions

In this thesis we have been mainly concerned with classical results from automata and formal language theory. We have chosen our definitions as close as possible to automata theory to get the tightest correspondences. In the future, variations of

definitions of the PDA, PPDA and RTM could be explored. For example, the aforementioned different termination conditions for the RTM or the absence/presence of the empty-test in the PDA and PPDA.

We have seen that the correspondence, up to (divergence-preserving) branching bisimilarity, between specification languages and automata with memory are not complete. In future work we could explore up until which equivalences the correspondences do hold. A step in this direction was already made in [BCT08] by giving the correspondence between the full class of sequential specifications and pushdown automata by stepping down to contrasimulation.

We have omitted in this thesis the Petri nets and context-sensitive languages. It would be interesting to study how these notions fit in the framework that we have presented in this thesis. From the specification language side, this also holds for the specifications language that is the combination of the sequential and basic parallel specifications.

In general, it would be worthwhile to compare the models of computation (or execution) to other notions with interaction found in literature. For example, the comparison of RTMs with persistent Turing machines. The π-calculus can to some extent be seen as the interactive version of the λ-calculus. The investigation of the π-calculus and our RTM could prove to be interesting.

Bibliography

[Bae05] J. C. M. Baeten. „A brief history of process algebra". In: *Theoretical Computer Science* 335.2–3 (2005), pp. 131–146 (cit. on p. 2).

[Bae11] J. C. M. Baeten. *Models of Computation: Automata and Processes*. Lecture notes, available at `http://se.wtb.tue.nl/sewiki/2it15/`. 2011 (cit. on pp. 22, 39).

[BBR09] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra – Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science 50. Cambridge University Press, 2009 (cit. on pp. xiii, 3, 5, 10, 13, 17, 128).

[BB88] J. C. M. Baeten and J. A. Bergstra. „Global Renaming Operators in Concrete Process Algebras". In: *Information and Computation* 78.3 (1988), pp. 205–245 (cit. on p. 72).

[BBK87] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. „On the Consistency of Koomen's Fair Abstraction Rule". In: *Theoretical Computer Science* 51.1–2 (1987), pp. 129–176 (cit. on pp. 19, 116, 128, 129).

[BBK93] J. C. M. Baeten, J. A. Bergstra, and J. W. Klop. „Decidability of Bisimulation Equivalence for Processes Generating Context-Free Languages". In: *Journal of the ACM* 40.3 (1993), pp. 653–682 (cit. on pp. 4, 51, 53, 63).

[BCG07] J. C. M. Baeten, F. Corradini, and C. A. Grabmayer. „A Characterization of Regular Expressions under Bisimulation". In: *Journal of the ACM* 54.2 (2007), (6)1–28 (cit. on pp. 31, 35).

[BCLT10] J. C. M. Baeten, P. J. L. Cuijpers, B. Luttik, and P. J. A. van Tilburg. „A Process-Theoretic Look at Automata". In: *Proceedings of FSEN 2009*. Ed. by F. Arbab and M. Sirjani. Lecture Notes in Computer Science 5961. Springer, 2010, pp. 1–33 (cit. on pp. 13, 22, 39, 57, 137).

[BCT08] J. C. M. Baeten, P. J. L. Cuijpers, and P. J. A. van Tilburg. „A Context-Free Process as a Pushdown Automaton". In: *Proceedings of CONCUR 2008*. Ed. by F. van Breugel and M. Chechik. Lecture Notes in Computer Science 5201. Springer, 2008, pp. 98–113 (cit. on pp. 39, 63, 69, 72, 76, 144).

[BCT09] J. C. M. Baeten, P. J. L. Cuijpers, and P. J. A. van Tilburg. „A Basic Parallel Process as a Parallel Pushdown Automaton". In: *Proceedings of EXPRESS 2008*. Ed. by D. Gorla and T. Hildebrandt. Electronic Notes in Theoretical Computer Science 242.1. Elsevier, 2009, pp. 35–48 (cit. on pp. 80, 98, 106).

[BLMT10] J. C. M. Baeten, B. Luttik, T. Muller, and P. J. A. van Tilburg. „Expressiveness modulo Bisimilarity of Regular Expressions with Parallel Composition (Extended Abstract)". In: *Proceedings of EXPRESS 2010*. Ed. by S. B. Fröschle and F. D. Valencia. Electronic Proceedings in Theoretical Computer Science 41. Open Publishing Association, 2010, pp. 1–15 (cit. on pp. 22, 31, 64, 128).

[BLT11a] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. „Computations and Interaction". In: *Proceedings of ICDCIT 2011*. Ed. by R. Natarajan and A. Ojo. Lecture Notes in Computer Science 6536. Springer, 2011, pp. 35–54 (cit. on pp. 22, 39).

[BLT11b] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. „Reactive Turing Machines". In: *Proceedings of FCT 2011*. Ed. by O. Owe, M. Steffen, and J. Telle. Lecture Notes in Computer Science 6914. Springer, 2011, pp. 348–359 (cit. on p. 111).

[BLT11c] J. C. M. Baeten, B. Luttik, and P. J. A. van Tilburg. *Reactive Turing Machines*. Tech. rep. arXiv:1104.1738v3. Cornell University Library, 2011 (cit. on p. 111).

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990 (cit. on p. 92).

[Bas96] T. Basten. „Branching bisimilarity is an equivalence indeed!" In: *Information Processing Letters* 58.3 (1996), pp. 141–147 (cit. on p. 11).

[BBP94] J. A. Bergstra, I. Bethke, and A. Ponse. „Process Algebra with Iteration and Nesting". In: *The Computer Journal* 37.4 (1994), pp. 243–258 (cit. on pp. 16, 17).

[BK84] J. A. Bergstra and J. W. Klop. „Process Algebra for Synchronous Communication". In: *Information and Control* 60.1–3 (1984), pp. 109–137 (cit. on pp. xiii, 3, 13).

[BK85] J. A. Bergstra and J. W. Klop. „Algebra of Communicating Processes with Abstraction". In: *Theoretical Computer Science* 37 (1985), pp. 77–121 (cit. on p. 90).

[BK86] J. A. Bergstra and J. W. Klop. „Process Algebra: Specification and Verification in Bisimulation Semantics". In: *Mathematics and Computer Science II*. Ed. by M. Hazewinkel, J. K. Lenstra, and L. G. L. T. Meertens. CWI Monographs 4. North-Holland, 1986, pp. 61–94 (cit. on p. 129).

[BP97] M. Bezem and A Ponse. „Two finite specifications of a queue”. In: *Theoretical Computer Science* 177.2 (1997), pp. 487–507 (cit. on pp. 129, 131).

[BGRR07] A. Blass, Y. Gurevich, D. Rosenzweig, and B. Rossman. „Interactive Small-Step Algorithms I: Axiomatization”. In: *Logical Methods in Computer Science* 3.4 (2007) (cit. on pp. 4, 111).

[Bos97] D. Bosscher. „Grammars Modulo Bisimulation”. PhD thesis. Centrum Wiskunde & Informatica, University of Amsterdam, 1997 (cit. on pp. 51, 64).

[Bou85] G. Boudol. „Notes on algebraic calculi of processes”. In: *Logics and Models of Concurrent Systems*. Ed. by K. R. Apt. NATO-ASI Series F13. Springer-Verlag, 1985, pp. 261–303 (cit. on p. 116).

[Cau86] D. Caucal. „Décidabilité de l’egalité des langages algébriques infinitaires simples”. In: *Proceedings of STACS 1986*. Ed. by B. Monien and G. Vidal-Naquet. Lecture Notes in Computer Science 210. Springer, 1986, pp. 37–48 (cit. on p. 63).

[Cho56] N. Chomsky. „Three models for the description of language”. In: *IRE Transactions on Information Theory* 2.3 (1956), pp. 113–124 (cit. on pp. 2, 25).

[Cho59] N. Chomsky. „On certain formal properties of grammars*”. In: *Information and control* 2.2 (1959), pp. 137–167 (cit. on p. 2).

[Chr93] S. Christensen. „Decidability and Decomposition in Process Algebras”. PhD thesis. University of Edinburgh, 1993 (cit. on pp. xiii, 80, 90, 91, 96).

[CHM93] S. Christensen, Y. Hirshfeld, and F. Moller. „Bisimulation Equivalence is Decidable for Basic Parallel Processes”. In: *Proceedings of CONCUR 1993*. Ed. by E. Best. Lecture Notes in Computer Science 715. Springer, 1993, pp. 143–157 (cit. on pp. 80, 97, 98, 101).

[CHS95] S. Christensen, H. Hüttel, and C. Stirling. „Bisimulation Equivalence is Decidable for all Context-Free Processes”. In: *Information and Computation* 121.2 (1995), pp. 143–148 (cit. on pp. 4, 63, 64).

[Chu32] A. Church. „A Set of Postulates for the Foundation of Logic”. In: *The Annals of Mathematics* 33.2 (1932), pp. 346–366 (cit. on p. 3).

[Chu36] A. Church. „An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363 (cit. on pp. 1, 109).

[Dar89] P. Darondeau. „Bisimulation and Effectiveness”. In: *Information Processing Letters* 30.1 (1989), pp. 19–20 (cit. on p. 118).

[Dic13] L. E. Dickson. „Finiteness of the Odd Perfect and Primitive Abundant Numbers with *n* Distinct Prime Factors”. In: *American Journal of Mathematics* 35.4 (1913), pp. 413–422 (cit. on p. 88).

[Fre79] F. L. G. Frege. *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. L. Nebert, 1879 (cit. on p. 2).

[Gla93] R. J. van Glabbeek. „The Linear Time – Branching Time Spectrum II". In: *Proceedings of CONCUR 1993*. Ed. by E. Best. Lecture Notes in Computer Science 715. Springer, 1993, pp. 66–81 (cit. on pp. 3, 5, 10, 63).

[Gla94] R. J. van Glabbeek. „On the Expressiveness of ACP (extended abstract)". In: *Proceedings of ACP 1994*. Ed. by A. Ponse, C. Verhoef, and S. F. M. van Vlijmen. Workshops in Computing. Springer, 1994, pp. 188–217 (cit. on p. 128).

[GLT09] R. J. van Glabbeek, B. Luttik, and N. Trčka. „Branching Bisimilarity with Explicit Divergence". In: *Fundamenta Informaticae* 93.4 (2009), pp. 371–392 (cit. on p. 11).

[GV93] R. J. van Glabbeek and F. W. Vaandrager. „Modular Specifications of Process Algebras". In: *Theoretical Computer Science* 113.2 (1993), pp. 293–348 (cit. on p. 130).

[GW96] R. J. van Glabbeek and W. P. Weijland. „Branching Time and Abstraction in Bisimulation Semantics". In: *Journal of the ACM* 43.3 (1996), pp. 555–600 (cit. on pp. 3, 10).

[GSAS04] D. Q. Goldin, S. A. Smolka, P. C. Attie, and E. L. Sonderegger. „Turing machines, transition systems, and interaction". In: *Information and Computation* 194.2 (2004), pp. 101–128 (cit. on pp. 4, 111, 138).

[GSW06] D. Q. Goldin, S. A. Smolka, and P. Wegner, eds. *Interactive computation: The new paradigm*. Springer, 2006 (cit. on pp. 4, 111).

[Gre65] S. A. Greibach. „A New Normal Form Theorem for Context-Free Phrase Structure Grammars". In: *Journal of the ACM* 12.1 (1965), pp. 42–54 (cit. on p. 19).

[Gro92] J. F. Groote. „A Short Proof of the Decidability of Bisimulation for Normed BPA-Processes". In: *Information Processing Letters* 42.3 (1992), pp. 167–171 (cit. on pp. 4, 63).

[HP89] D. Harel and A. Pnueli. „On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Ed. by K. R. Apt. NATO-ASI Series F13. Springer-Verlag, 1989, pp. 477–498 (cit. on p. 109).

[HM01] Y. Hirshfeld and F. Moller. „Pushdown automata, multiset automata, and Petri nets". In: *Theoretical Computer Science* 256.1–2 (2001), pp. 3–21 (cit. on p. 108).

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985 (cit. on pp. xiii, 3, 13, 72).

[HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed. Pearson, 2006 (cit. on pp. 2, 4, 24, 34, 37, 39, 40, 47, 51, 53, 62, 74, 109, 125).

[HS91] H. Hüttel and C. Stirling. „Actions Speak Louder than Words: Proving Bisimilarity for Context-Free Processes". In: *Proceedings of LICS 1991*. Ed. by G. Kahn. IEEE Computer Society Press, 1991, pp. 376–386 (cit. on pp. 4, 63).

[Kle36] S. C. Kleene. „General recursive functions of natural numbers". In: *Mathematische Annalen* 112.1 (1936), pp. 727–742 (cit. on pp. 1, 109).

[Kle56] S. C. Kleene. „Representation of Events in Nerve Nets and Finite Automata". In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Annals of Mathematical Studies 34. Princeton University Press, 1956, pp. 3–42 (cit. on pp. 2, 16, 30, 31).

[LW00] J. van Leeuwen and J. Wiedermann. „On Algorithms and Interaction". In: *Proceedings of MFCS 2000*. Ed. by M. Nielsen and B. Rovan. Lecture Notes in Computer Science 1893. Springer, 2000, pp. 99–113 (cit. on pp. 4, 111, 138).

[Lin01] P. Linz. *An Introduction to Formal Languages and Automata*. 3rd ed. Jones and Bartlett Publishers, 2001 (cit. on pp. 2, 4, 24, 26).

[MP43] W. S. McCullough and W. Pitts. „A logical calculus of ideas immanent in nervous activity". In: *Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 2).

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science 92. Springer-Verlag, 1980 (cit. on pp. xiii, 2, 3, 13).

[Mil84] R. Milner. „A Complete Inference System for a Class of Regular Behaviours". In: *Journal of Computer and System Sciences* 28.3 (1984), pp. 439–466 (cit. on pp. 16, 22, 31).

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989 (cit. on pp. 3, 13).

[Mil93] R. Milner. „Elements of Interaction – Turing Award Lecture". In: *Communications of the ACM* 36.1 (1993), pp. 78–89 (cit. on pp. 2, 109).

[Mil99] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999 (cit. on p. 3).

[Mol96] F. Moller. „Infinite results". In: *Proceedings CONCUR 1996*. Ed. by U. Montanari and V. Sassone. Lecture Notes in Computer Science 1119. Springer, 1996, pp. 195–216 (cit. on pp. 4, 5, 60, 79, 81, 108).

[Neu56] J. von Neumann. „Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components". In: *Automata Studies*. Ed. by C. E. Shannon and J. McCarthy. Annals of Mathematical Studies 34. Princeton University Press, 1956, pp. 43–98 (cit. on p. 2).

[Par81] D. M. R. Park. „Concurrency and automata on infinite sequences". In: *Proceedings of TCS 1981*. Ed. by P. Deussen. Lecture Notes in Computer Science 104. Springer, 1981, pp. 167–183 (cit. on p. 10).

[Pet62] C. A. Petri. „Kommunikation mit Automaten." Schriften des IIM Nr. 2. PhD thesis. Bonn: Institut für Instrumentelle Mathematik, 1962 (cit. on pp. 2, 138).

[Phi93] I. C. C. Phillips. „A Note on Expressiveness of Process Algebra". In: *Proceedings of the First Imperial College Department of Computing Workshop on Theory and Formal Methods*. Ed. by G. L. Burn, S. Gay, and M. D. Ryan. Workshops in Computing. Springer-Verlag, 1993, pp. 260–264 (cit. on pp. 117, 126).

[Plo04] G. D. Plotkin. „A structural approach to operational semantics". In: *Journal of Logic and Algebraic Programming* 60–61 (2004), pp. 17–139 (cit. on p. 14).

[Rog67] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill Book Company, 1967 (cit. on pp. 116, 117, 120, 125, 126).

[Sip97] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997 (cit. on pp. 2, 4, 24, 34, 37, 39, 51, 74, 109).

[Srb01] J. Srba. „Basic Process Algebra with Deadlocking States". In: *Theoretical Computer Science* 266.1–2 (2001), pp. 605–630 (cit. on pp. 4, 51, 64–68, 90).

[Sti03] C. Stirling. „Bisimulation and Language Equivalence". In: *Logic for Concurrency and Synchronisation* 15 (2003), pp. 269–284 (cit. on p. 4).

[Sud88] T. A. Sudkamp. *Languages and Machines*. 2nd ed. Addison-Wesley Publishing Company, 1988 (cit. on pp. 2, 4, 24, 34, 37, 39, 51, 52, 74, 109).

[Trč07] N. Trčka. „Silent Steps in Transition Systems and Markov Chains". PhD thesis. Eindhoven University of Technology, 2007 (cit. on p. 18).

[Tur37] A. M. Turing. „On Computable Numbers, With an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265 (cit. on pp. 1, 109, 129).

[Tur39] A. M. Turing. „Systems of Logic Based on Ordinals". In: *Proceedings of the London Mathematical Society* s2-45.1 (1939), pp. 161–228 (cit. on p. 111).

[Vaa92] F. W. Vaandrager. „Expressiveness Results for Process Algebras". In: *Proceedings of the REX Workshop 1992*. Ed. by J. W. de Bakker, W. P. de Roever, and G. Rozenberg. Lecture Notes in Computer Science 666. Springer, 1992, pp. 609–638 (cit. on pp. 128, 139).

[VM01] M. Voorhoeve and S. Mauw. „Impossible Futures and Determinism". In: *Information Processing Letters* 80.1 (2001), pp. 51–58 (cit. on p. 63).

[WL08] J. Wiedermann and J. van Leeuwen. „How We Think of Computing Today". In: *Proceeding of CiE 2008*. Ed. by A. Beckmann, C. Dimitracopoulos, and B. Löwe. Lecture Notes in Computer Science 5028. Springer, 2008, pp. 579–593 (cit. on pp. 111, 138).

Index

For a complete overview of the symbols and acronyms used in this thesis, please refer to the Glossary on page xi.

Summary

From Computability to Executability
A process-theoretic view on automata theory

The theory of automata and formal languages was devised in the 1930s to provide models for and to reason about computation. Here we mean by *computation* a procedure that transforms input into output, which was the sole mode of operation of computers at the time. Nowadays, computers are systems that *interact* with us and also with each other; they are non-deterministic, reactive systems. Concurrency theory, split off from classical automata theory in the seventies, provides a model of computation similar to the model given by the theory of automata and formal languages, but focuses on concurrent, reactive and interactive systems.

This thesis investigates the integration of the two theories, exposing the differences and similarities between them. Where automata and formal language theory focuses on computations and languages, concurrency theory focuses on *behaviour*. To achieve integration, we look for process-theoretic analogies of classic results from automata theory. The most prominent difference is that we use an interpretation of automata as *labelled transition systems* modulo (divergence-preserving) branching bisimilarity instead of treating automata as language acceptors. We also consider similarities such as grammars as recursive specifications and finite automata as labelled finite transition systems. We investigate whether the classical results still hold and, if not, what extra conditions are sufficient to make them hold.

We especially look into three levels of Chomsky's hierarchy: we study the notions of finite-state systems, pushdown systems, and computable systems. Additionally we investigate the notion of parallel pushdown systems. For each class we define the central notion of automaton and its behaviour by associating a transition system with the automaton. Then we introduce a suitable specification language and investigate the correspondence with the respective automaton (via its associated transition system). Because we not only want to study interaction with the environment, but also the interaction within the automaton, we make the interaction explicit by means of communicating parallel components, with one component representing the finite control of the automaton and one component representing the memory.

First, we study *finite-state systems* by reinvestigating the relation between finite-state automata, left- and right-linear grammars, and regular expressions, but now up to (divergence-preserving) branching bisimilarity.

For *pushdown systems* we augment the finite-state systems with *stack memory* to obtain the pushdown automata and consider different *termination styles*: termination on empty stack, on final state, and on final state and empty stack. Unlike for language equivalence, up to (divergence-preserving) branching bisimilarity the associated transition systems for the different termination styles fall into different classes. We obtain (under some restrictions) the correspondence between context-free grammars and pushdown automata for termination on final state and empty stack. Finally, we make the interaction within a pushdown automaton explicit, but in a different way depending on the termination style.

By analogy with pushdown systems we investigate the *parallel pushdown systems*, obtained by augmenting finite-state systems with *bag memory*, and consider analogous termination styles. We investigate the correspondence between context-free grammars that use parallel composition instead of sequential composition and parallel pushdown automata. While the correspondence itself is rather tight, it unfortunately only covers a small subset of the parallel pushdown automata, i.e. the single-state parallel pushdown automata. When making the interaction within parallel pushdown automata explicit, we obtain a rather uniform result for all termination styles.

Finally, we study *computable systems* and the relation with effective and computable transition systems and Turing machines. For this we present the reactive Turing machine, a classical Turing machine augmented with capabilities for interaction. Again, we make the interaction in the reactive Turing machine between its finite control and the *tape memory* explicit.

Samenvatting

Van berekenbaarheid naar uitvoerbaarheid
Een procestheoretische kijk op de automatentheorie

De theorie van automaten en formele talen heeft zijn oorsprong in de jaren dertig. In die tijd werden er modellen opgesteld, zoals bijvoorbeeld de Turingmachine, om te kunnen beredeneren wat berekenbaar is en wat niet. Met een '*berekening*' bedoelen we hier de transformatie van invoer naar uitvoer. Destijds was het herhaaldelijk uitvoeren van de bijbehorende operatie het enige wat computers konden. Tegenwoordig zijn computers echter systemen die *interactief* zijn; ze wisselen continu informatie uit, niet alleen met de gebruiker maar ook met elkaar. De procestheorie, afgesplitst van de automatentheorie in de jaren zeventig, gebruikt berekeningsmodellen die erg lijken op die van de theorie van automaten en formele talen, maar meer zijn gericht op parallelle, reactieve en interactieve systemen.

Dit proefschrift onderzoekt de integratie van deze twee theorieën met als doel de verschillen en overeenkomsten bloot te leggen. Waar de theorie van automaten en formele talen de nadruk legt op berekeningen en talen, legt de procestheorie de nadruk op gedrag. Om tot integratie te komen, zoeken we naar procestheoretische analogieën van klassieke resultaten uit de automatentheorie. Het prominentste verschil is dat we hierbij automaten interpreteren als gelabelde transitiesystemen modulo vertakkende bisimulatie, in plaats van automaten te beschouwen als acceptanten van een taal. (Wanneer mogelijk, proberen we te zorgen dat de vertakkende bisimulatierelatie ook divergentiebehoudend is.) We bekijken daarnaast klassieke overeenkomsten zoals die tussen grammatica's en recursieve specificaties en tussen eindige automaten en transitiesystemen.

We volgen in dit proefschrift drie niveaus van Chomsky's hiërarchie, die de volgende klassen van systemen omschrijven: eindige systemen, pushdownsystemen en berekenbare systemen. Daarnaast verkennen we de notie van parallelle pushdownsystemen. Voor iedere klasse definiëren we een bijbehorende automaat en leggen we het gedrag vast door er een transitiesysteem mee te associëren. Vervolgens introduceren we een geschikte specificatietaal en onderzoeken we de overeenstemming met de respectievelijke automaat, via het geassocieerde transitiesysteem. Omdat we niet alleen de interactie van het systeem met de omgeving willen bestuderen, maar ook de

interactie die plaatsvindt binnen de automaat, maken we deze laatste expliciet door de introductie van communicerende parallelle componenten: een component die de eindige besturing van de automaat representeert en een component die het geheugen representeert.

Eerst bestuderen we *eindige systemen* (zonder geheugen) door de relaties tussen eindige automaten, links- en rechtslineaire grammatica's, en reguliere expressies opnieuw te bekijken, maar nu met behulp van (divergentiebehoudende) vertakkende bisimulatie.

Voor *pushdownsystemen* verkrijgen we pushdownautomaten door eindige systemen uit te breiden met *stack*geheugen. We beschouwen verschillende terminatiestijlen: terminatie bij lege stack; in een eindtoestand; bij lege stack én in een eindtoestand. Voor vertakkende bisimulatie vallen de geassocieerde transitiesystemen voor de verschillende terminatiestijlen uiteen in verschillende klassen, wat niet het geval is voor taalgelijkheid. We verkrijgen, onder enkele restricties, de overeenkomst tussen contextvrije grammatica's en pushdownautomaten voor terminatie bij lege stack én in een eindtoestand. Ten slotte maken we interactie binnen de pushdownautomaat expliciet. De manier waarop dit gebeurt, wordt echter bepaald door de terminatiestijl.

Op vergelijkbare wijze als met pushdownsystemen onderzoeken we de *parallelle pushdownsystemen*, verkregen door eindige systemen uit te breiden met een *bag*geheugen. We beschouwen wederom de verschillende terminatiestijlen zoals eerder genoemd. We onderzoeken de overeenkomst tussen commutatieve contextvrije grammatica's, die parallelle compositie gebruiken in plaats van sequentiële compositie, en parallelle pushdownautomaten. Hoewel de overeenkomst relatief sterk is, dekt de relatie maar een kleine deel van alle parallelle pushdownautomaten af, namelijke die met slechts één toestand. Door het expliciet maken van de interactie binnen de parallelle pushdownautomaat krijgen we echter wel een mooi en uniform resultaat voor alle terminatiestijlen.

Ten slotte bestuderen we *berekenbare systemen* en de relatie met effectieve en berekenbare transitiesystemen en Turingmachines. Hiertoe introduceren we de reactieve Turingmachine: een klassieke Turingmachine uitgerust met mogelijkheden om interactie aan te gaan met zijn omgeving. Wederom maken we ook de interactie binnen de reactieve Turingmachine expliciet, dat wil zeggen tussen de eindige besturing en *tape*geheugen.

Curriculum Vitae

Paulus Johannes Adrianus (Paul) van Tilburg was born on 17 April 1980 in Breda, The Netherlands. After finishing his high school education VWO (pre-academic secondary education) cum laude in 1998 at the Onze Lieve Vrouwe Lyceum in Breda, The Netherlands, he first studied Electrical Engineering from 1998 until 2001 and then Computer Science at the Eindhoven University of Technology. In 2006, he obtained his Bachelor of Science degree in Computer Science, followed in 2007 by graduating cum laude as Master of Science in Computer Science & Engineering within the group of Formal Methods on the axiomatisability of the process algebra CCS. From August 2007 until September 2011 he worked as a Ph.D. student on the project "Models of Computation: Automata and Processes", funded by the NWO (Dutch Organisation for Scientific Research), at the Eindhoven University of Technology of which the results are presented in this dissertation.

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11