Cambridge Tracts in Theoretical Computer Science 50

Process Algebra: Equational Theories of Communicating Processes

Process algebra is a widely accepted and much used technique in the specification and verification of parallel and distributed software systems. This book sets the standard for the field. It assembles the relevant results of most process algebras currently in use, and presents them in a unified framework and notation.

The authors describe the theory underlying the development, realization and maintenance of software that occurs in parallel or distributed systems. A system can be specified in the syntax provided, and the axioms can be used to verify that a composed system has the required external behavior. As examples, two protocols are completely specified and verified in the text: the Alternating-Bit communication Protocol, and Fischer's Protocol of mutual exclusion.

The book serves as a reference text for researchers and graduate students in computer science, offering a complete overview of the field and referring to further literature where appropriate.

J. C. M. BAETEN is Professor of formal methods in the Division of Computer Science at Eindhoven University of Technology, Netherlands.

T. BASTEN is Professor of computational models in the Faculty of Electrical Engineering at Eindhoven University of Technology, Netherlands, and Research Fellow at the Embedded Systems Institute, Eindhoven.

M. A. RENIERS is Assistant Professor in the Division of Computer Science at Eindhoven University of Technology, Netherlands.

Cambridge Tracts in Theoretical Computer Science 50

Editorial Board

Titles in the series

A complete list of books in the series can be found at
http://www.cambridge.org/uk/series/sSeries.asp?code=CTTC.
Recent titles include the following:

21. D. A. Wolfram *The Clausal Theory of Types*
22. V. Stoltenberg-Hansen, Lindström & E. R. Griffor *Mathematical Theory of Domains*
23. E.-R. Olderog *Nets, Terms and Formulas*
26. P. D. Mosses *Action Semantics*
27. W. H. Hesselink *Programs, Recursion and Unbounded Choice*
28. P. Padawitz *Deductive and Declarative Programming*
29. P. Gärdenfors (ed.) *Belief Revision*
30. M. Anthony & N. Biggs *Computational Learning Theory*
31. T. F. Melham *Higher Order Logic and Hardware Verification*
32. R. L. Carpenter *The Logic of Typed Feature Structures*
33. E. G. Manes *Predicate Transformer Semantics*
34. F. Nielson & H. R. Nielson *Two-Level Functional Languages*
35. L. M. G. Feijs & H. B. M. Jonkers *Formal Specification and Design*
36. S. Mauw & G. J. Veltink (eds.) *Algebraic Specification of Communication Protocols*
37. V. Stavridou *Formal Methods in Circuit Design*
38. N. Shankar *Metamathematics, Machines and Gödel's Proof*
39. J. B. Paris *The Uncertain Reasoner's Companion*
40. J. Desel & J. Esparza *Free Choice Petri Nets*
41. J.-J. Ch. Meyer & W. van der Hoek *Epistemic Logic for AI and Computer Science*
42. J. R. Hindley *Basic Simple Type Theory*
43. A. S. Troelstra & H. Schwichtenberg *Basic Proof Theory*
44. J. Barwise & J. Seligman *Information Flow*
45. A. Asperti & S. Guerrini *The Optimal Implementation of Functional Programming Languages*
46. R. M. Amadio & P.-L. Curien *Domains and Lambda-Calculi*
47. W.-P. de Roever & K. Engelhardt *Data Refinement*
48. H. Kleine Büning & T. Lettman *Propositional Logic*
49. L. Novak & A. Gibbons *Hybrid Graph Theory and Network Analysis*
51. H. Simmons *Derivation and Computation*
52. P. Blackburn, M. de Rijke & Y. Venema *Modal Logic*
53. W.-P. de Roever et al *Concurrency Verification*
54. Terese *Term Rewriting Systems*
55. A. Bundy et al *Rippling: Meta-Level Guidance for Mathematical Reasoning*

Process Algebra: Equational Theories of Communicating Processes

J. C. M. BAETEN
T. BASTEN
M. A. RENIERS
Eindhoven University of Technology, Netherlands

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo, Delhi

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521820493

First published 2010

Printed in the United Kingdom at the University Press, Cambridge

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-82049-3 Hardback

Additional resources for this publication at www.processalgebra.org

Contents

Forewords

Tony Hoare
Cambridge, United Kingdom, February 2009

Algebra is the simplest of all branches of mathematics. After the study of numerical calculation and arithmetic, algebra is the first school subject which gives the student an introduction to the generality and power of mathematical abstraction, and a taste of mathematical proof by symbolic reasoning. Only the simplest reasoning principle is required: the substitution of equals for equals. (Even computers are now quite good at it.) Nevertheless, the search for algebraic proof still presents a fascinating puzzle for the human mathematician, and yields results of surprising brevity and pleasing elegance.

A more systematic study of algebra provides a family tree that unifies the study of many of the other branches of mathematics. It identifies the basic mathematical axioms that are common to a whole sub-family of branches. The basic theorems that are proved from these axioms will be true in every branch of mathematics which shares them. At each branching point in the tree, the differences between the branches are succinctly highlighted by their choice between a pair of mutually contradictory axioms. In this way, algebra is both cumulative in its progress along the branches, and modular at its branching points.

It is a surprise to many computer programmers that computer programs, with all their astronomical complexity of structure and behavior, are as amenable to the axioms of algebra as simple numbers were at school. Indeed, algebra scales well from the small to the large. It applies to the large-scale behavior of systems evolving concurrently in parallel; and it underlies the manipulation of the minute detail of the individual instructions of code. At the highest level, algebraic reasoning provides the essential basis for program transformations that match the structure of a complete system to that of the available hardware configuration; and at the lowest level, it provides the justification for optimization

of the code of a program so as to achieve the full potential of the processor instruction set.

This book exploits the power of algebra to explore the properties of concurrent programs, particularly those that control the behavior of distributed systems, communicating with neighbors on identified channels. It starts with the simplest theories at the base of the tree, and gradually extends them in a modular way by additional axioms to deal with sequential programming as well as concurrent, and with communication as well as assignment. In the later chapters, it exploits the modularity of algebra to describe priorities, probabilities, and mobility.

The technical approach of the book is grounded in Computer Science. It emphasizes term models based on the syntax of the operators used in the algebra, and it justifies the axioms by appeal to the execution of the terms as programs. Its crowning achievement is to exploit the unifying power of algebra to cover a range of historic theories, developed for various purposes up to 20 years ago. Earlier these theories were thought to be irreconcilable rivals; but as a result of research by the authors of this book and others, the rivals are now seen to be close family members, each superbly adapted to the particular range of problems that it was designed to tackle.

Robin Milner
Cambridge, United Kingdom, February 2009

Nowadays, much of what is still called 'computing' involves the behavior of composite systems whose members interact continually with their environment. A better word is 'informatics', because we are concerned not just with calculation, but rather with autonomous agents that interact with – or inform – one another. This interactivity bursts the bounds of the sequential calculation that still dominates many programming languages. Does it enjoy a theory as firm and complete as the theory of sequential computation? Not yet, but we are getting there.

What is an informatic process? The answer must involve phenomena foreign to sequential calculation. For example can an informatic system, with many interacting components, achieve deterministic behavior? If it can, that is a special case; non-determinism is the norm, not the exception. Does a probability distribution, perhaps based upon the uncertainty of timing, replace determinism? Again, how exactly do these components interact; do they send each other messages, like email, to be picked up when convenient? – or is each interaction a kind of synchronized handshake?

Over the last few decades many models for interactive behavior have been

of the code of a program so as to achieve the full potential of the processor instruction set.

This book exploits the power of algebra to explore the properties of concurrent programs, particularly those that control the behavior of distributed systems, communicating with neighbors on identified channels. It starts with the simplest theories at the base of the tree, and gradually extends them in a modular way by additional axioms to deal with sequential programming as well as concurrent, and with communication as well as assignment. In the later chapters, it exploits the modularity of algebra to describe priorities, probabilities, and mobility.

The technical approach of the book is grounded in Computer Science. It emphasizes term models based on the syntax of the operators used in the algebra, and it justifies the axioms by appeal to the execution of the terms as programs. Its crowning achievement is to exploit the unifying power of algebra to cover a range of historic theories, developed for various purposes up to 20 years ago. Earlier these theories were thought to be irreconcilable rivals; but as a result of research by the authors of this book and others, the rivals are now seen to be close family members, each superbly adapted to the particular range of problems that it was designed to tackle.

Robin Milner
Cambridge, United Kingdom, February 2009

Nowadays, much of what is still called 'computing' involves the behavior of composite systems whose members interact continually with their environment. A better word is 'informatics', because we are concerned not just with calculation, but rather with autonomous agents that interact with – or inform – one another. This interactivity bursts the bounds of the sequential calculation that still dominates many programming languages. Does it enjoy a theory as firm and complete as the theory of sequential computation? Not yet, but we are getting there.

What is an informatic process? The answer must involve phenomena foreign to sequential calculation. For example can an informatic system, with many interacting components, achieve deterministic behavior? If it can, that is a special case; non-determinism is the norm, not the exception. Does a probability distribution, perhaps based upon the uncertainty of timing, replace determinism? Again, how exactly do these components interact; do they send each other messages, like email, to be picked up when convenient? – or is each interaction a kind of synchronized handshake?

Over the last few decades many models for interactive behavior have been

proposed. This book is the fruit of 25 years of experience with an algebraic approach, in which the constructors by which an informatic system is assembled are characterized by their algebraic properties. The characteristics are temporal, in the same way that sequential processes are temporal; they are also spatial, describing how agents are interconnected. And their marriage is complex.

The authors have teased out primitive elements of this structure. In doing so they have applied strict mathematical criteria. For example, to what extent can the dynamic characteristics of a set of process constructors be reflected, soundly or completely, by a collection of algebraic axioms? And to what extent can the authors' calculus ACP (Algebra of Communicating Processes) be harmonized with other leading calculi, especially Hoare's CSP (Communicating Sequential Processes) and Milner's CCS (Calculus of Communicating Systems)? Consideration of these questions gives the book a firm and appreciable structure. In particular, it shows up a shortcoming of what (for some 50 years) has been called automata theory: that theory never seriously attempted to model the ways in which two automata might interact.

So it may seem that the book is mainly an account of the frontiers of research into process theory. It is much more, and I hope that syllabus designers will take note: the presentation is lucid and careful, enriched with exercises, to the extent that many parts of it can be used as a basis for university courses, both undergraduate and postgraduate. If in such courses we expose students to theories that are still advancing, then they share the excitement of that progress.

Jan Bergstra
Amsterdam, the Netherlands, February 2009

This book about process algebra improves on its predecessor, written by Jos Baeten and Peter Weijland almost 20 years ago, by being more comprehensive and by providing far more mathematical detail. In addition the syntax of ACP has been extended by a constant 1 for termination. This modification not only makes the syntax more expressive, it also facilitates a uniform reconstruction of key aspects of CCS, CSP as well as ACP, within a single framework.

After renaming the empty process (ϵ) into 1 and the inactive process (δ) into 0, the axiom system ACP is redesigned as BCP. This change is both pragmatically justified and conceptually convincing. By using a different acronym instead of ACP, the latter can still be used as a reference to its original meaning, which is both useful and consistent.

Curiously these notational changes may be considered marginal and significant at the same time. In terms of theorems and proofs, or in terms of case

proposed. This book is the fruit of 25 years of experience with an algebraic approach, in which the constructors by which an informatic system is assembled are characterized by their algebraic properties. The characteristics are temporal, in the same way that sequential processes are temporal; they are also spatial, describing how agents are interconnected. And their marriage is complex.

The authors have teased out primitive elements of this structure. In doing so they have applied strict mathematical criteria. For example, to what extent can the dynamic characteristics of a set of process constructors be reflected, soundly or completely, by a collection of algebraic axioms? And to what extent can the authors' calculus ACP (Algebra of Communicating Processes) be harmonized with other leading calculi, especially Hoare's CSP (Communicating Sequential Processes) and Milner's CCS (Calculus of Communicating Systems)? Consideration of these questions gives the book a firm and appreciable structure. In particular, it shows up a shortcoming of what (for some 50 years) has been called automata theory: that theory never seriously attempted to model the ways in which two automata might interact.

So it may seem that the book is mainly an account of the frontiers of research into process theory. It is much more, and I hope that syllabus designers will take note: the presentation is lucid and careful, enriched with exercises, to the extent that many parts of it can be used as a basis for university courses, both undergraduate and postgraduate. If in such courses we expose students to theories that are still advancing, then they share the excitement of that progress.

Jan Bergstra
Amsterdam, the Netherlands, February 2009

This book about process algebra improves on its predecessor, written by Jos Baeten and Peter Weijland almost 20 years ago, by being more comprehensive and by providing far more mathematical detail. In addition the syntax of ACP has been extended by a constant 1 for termination. This modification not only makes the syntax more expressive, it also facilitates a uniform reconstruction of key aspects of CCS, CSP as well as ACP, within a single framework.

After renaming the empty process (ϵ) into 1 and the inactive process (δ) into 0, the axiom system ACP is redesigned as BCP. This change is both pragmatically justified and conceptually convincing. By using a different acronym instead of ACP, the latter can still be used as a reference to its original meaning, which is both useful and consistent.

Curiously these notational changes may be considered marginal and significant at the same time. In terms of theorems and proofs, or in terms of case

studies, protocol formalizations and the design of verification tools, the specific details of notation make no real difference at all. But by providing a fairly definitive and uncompromising typescript a major impact is obtained on what might be called 'nonfunctional qualities' of the notational framework. I have no doubt that these nonfunctional qualities are positive and merit being exploited in full detail as has been done by Baeten and his co-authors. Unavoidably, the notational evolution produces a change of perspective. While, for instance, the empty process is merely an add on feature for ACP, it constitutes a conceptual cornerstone for BCP.

Group theory provides different notational conventions (additive and multiplicative) for the same underlying structure, and in a similar fashion, the format of this book might be viewed as a comprehensive and consistent notational convention for a theory of process algebra. But the same theory might instead be captured, when used in another context, with a different preferred notation.

Process algebra as presented here is equational logic of processes viewed as a family of first order theories. Each theory is provided with its Tarski semantics, making use of many-sorted algebras with total operations and nonempty sorts. Although this may sound already quite technical and may be even prohibitive for a dedicated computer scientist, these are the most stable and clear-cut semantic principles that mathematical logic has developed thus far. When developing axioms for process algebras the authors do not deviate from that path for ad hoc reasons of any kind. Thus there is a very clear separation between the logical metatheory, consisting of first order equational logic and its model-theory on the one hand and the many design decisions concerning the subject matter that is process theory on the other hand. A prominent methodological principle underlying the work is that the significant ramification of design decisions concerning various process formalisms can be made systematic in a way comparable to the development of say ring theory in mathematics.

In addition to making use of first order logic, the equational form of most axioms allows a systematic exploitation of technical results from term rewriting. This is not a matter of process theory per se but it is proving helpful throughout the book.

One may ask why process algebra should be considered a topic in computer science and not just in applied mathematics. While some parts of process theory are now moving in the direction of systems biology, the process algebras covered in this book may sooner or later show up in physics. Quantum process algebras have not been covered here but various forms already exist and that kind of development is likely to continue for many more years.

Just like its predecessor has been for many years, this book will definitely be useful as a reference work for research in process theory.

Preface

What is this book about?

This book sets the standard for process algebra. It assembles the relevant results of most process algebras currently in use, and presents them in a unified framework and notation. It addresses important extensions of the basic theories, like timing, data parameters, probabilities, priorities, and mobility. It systematically presents a hierarchy of algebras that are increasingly expressive, proving the major properties each time.

For researchers and graduate students in computer science, the book will serve as a reference, offering a complete overview of what is known to date, and referring to further literature where appropriate.

Someone familiar with CCS, the Calculus of Communicating Systems, will recognize the minimal process theory MPT as basic CCS, to which a constant expressing successful termination is added, enabling sequential composition as a basic operator, and will then find a more general parallel-composition operator. Someone familiar with ACP, the Algebra of Communicating Processes, will see that termination is made explicit, leading to a replacement of action constants by action prefixing, but will recognize many other things. The approaches to recursion of CCS and ACP are both explained and integrated. Someone familiar with CSP, Communicating Sequential Processes, will have to cope with more changes, but will see the familiar operators of internal and external choice and parallel composition explained in the present setting.

The book is a complete revision of another (Baeten & Weijland, 1990). Moreover, as the unification theme has become more important, it can also be seen as a successor to (Milner, 1989) and (Hoare, 1985).

Process algebra has become a widely accepted and used technique in the specification and verification of parallel and distributed software systems. A system can be specified in the syntax provided, and the axioms can be used

to verify that a composed system has the required external behavior. As examples, a couple of protocols are completely specified and verified in the text: the Alternating-Bit communication Protocol, and Fischer's protocol of mutual exclusion. The book explains how such a task can be undertaken for any given parallel or distributed system. As the system gets bigger though, tool support in this endeavor, using for example the mCRL2 tool set, will become necessary.

Despite the breadth of the book, some aspects of process algebra are not covered. Foremost, relationships with logic, important when discussing satisfaction of requirements, are not addressed (except briefly in Chapter 10). This was left out as it does not pertain directly to the algebraic treatment and would make the book too voluminous. Readers interested in this subject may refer to (Bradfield & Stirling, 2001). Some of the extensions (probabilities, priorities, and mobility) are only briefly touched upon; references are given to literature that provides a more in-depth treatment.

How to use this book?

The book can be used in teaching a course to students at advanced undergraduate or graduate level in computer science or a related field. It contains numerous exercises varying in difficulty. Such a course should cover at least Chapters 1 through 8, together comprising a complete theory usable in applications. From the remaining chapters, different choices can be made, since the chapters can be read independently. The text is also suitable for self-study.

Chapter 1 presents an introduction that delineates more precisely the field of process algebra. It also gives a historic overview of the development of the field. In Chapter 2, some notions are explained that are used in the remainder of the book. The material concerns equational theories, algebras, and term rewriting systems. Chapter 3 covers the semantic domain of transition systems. The notion of bisimilarity is explained, and the concept of a structural operational semantics, a standard way to assign transition systems to terms in a language, is introduced. Some relevant theorems about structural operational semantics are presented and are used in the remainder of the book when providing semantics to equational theories.

Chapter 4 starts with process algebra. A minimal process theory is presented, that illustrates the basic steps involved in establishing a set of laws and a model for an equational theory. Then, two basic extensions are considered: first, the extension with the successful-termination constant 1, and second, the extension with projection operators. Differences between the two types of extensions are explained. As a prequel to the succeeding chapter, the extension

with the iteration operator is considered. By means of this operator, some infinite processes can be defined.

Chapter 5 is devoted to recursion, which is the main means of specifying non-terminating processes. It is shown how to define such processes, and how to reason with them. The unbounded stack is considered as a first example.

Chapter 6 adds sequential composition. Furthermore, it looks at renaming operators, and operators that can block or skip part of a process. Chapter 7 adds parallel composition and communication. Buffers and bags are considered as examples. As a larger example, the specification of the Alternating-Bit Protocol is presented.

Chapter 8 considers abstraction, which enables one to hide some of the behavior of a system in order to focus on the rest. It is the central ingredient for verification. As an example, a verification is presented of the Alternating-Bit Protocol. The notions of divergence and fairness are explained and treated.

Chapter 9 gives a short introduction to the extension with explicit timing. Fischer's protocol is presented as an example. Chapter 10 considers the interplay between data and processes, and at the same time takes a closer look at the notion of a state, and what can be observed from it. Chapter 11 briefly covers some extensions to and variants of the basic theories, namely priorities, probabilities, mobility, and different forms of parallel composition. The book concludes with Chapter 12, which considers other semantics, besides bisimulation semantics, and their interrelations.

The book is supported through the website **www.processalgebra.org**. This website contains supplementary material, such as solutions to selected exercises, slides presenting the book content, additional exercises and exam problems, a tool to support algebraic reasoning, an up-to-date process algebra bibliography, and much more. Lecturers, students, and researchers are invited to have a look at the website, to get the most out of using the book.

Acknowledgements

A book like this can only be written thanks to the contributions and support of many individuals. The authors wish to acknowledge Luca Aceto, Suzana Andova, Jan Bergstra, Maarten Boote, Victor Bos, Beverley Clarke, Clare Dennison, Henk Goeman, Jan Friso Groote, Evelien van der Heiden, Leszek Holenderski, Abigail Jones, Hugo Jonker, Gerben de Keijzer, Uzma Khadim, Fabian Kratz, Kim Larsen, Bas Luttik, Jasen Markovski, Sjouke Mauw, Kees Middelburg, Mohammad Mousavi, Dennis van Opzeeland, Ralph Otten, Alban Ponse, Isabelle Reymen, Maria van Rooijen, Eugen Schindler, Natalia

Sidorova, David Tranah, Pieter Verduin, Jeroen Voeten, Marc Voorhoeve, Peter Weijland, and Tim Willemse.

1
Process algebra

1.1 Definition

This book is about *process algebra*. The term 'process algebra' refers to a loosely defined field of study, but it also has a more precise, technical meaning. The latter is considered first, as a basis to delineate the field of process algebra.

Consider the word 'process'. It refers to *behavior* of a *system*. A system is anything showing behavior, in particular the execution of a software system, the actions of a machine, or even the actions of a human being. Behavior is the total of events or actions that a system can perform, the order in which they can be executed and maybe other aspects of this execution such as timing or probabilities. Always, the focus is on certain aspects of behavior, disregarding other aspects, so an abstraction or idealization of the 'real' behavior is considered. Rather, it can be said that there is an *observation* of behavior, and an action is the chosen unit of observation. Usually, the actions are thought to be discrete: occurrence is at some moment in time, and different actions can be distinguished in time. This is why a process is sometimes also called a *discrete event system*.

The term 'algebra' refers to the fact that the approach taken to reason about behavior is algebraic and axiomatic. That is, operations on processes are defined, and their equational laws are investigated. In other words, methods and techniques of universal algebra are used (see e.g., (MacLane & Birkhoff, 1967)). To allow for a comparison, consider the definition of a *group* in universal algebra.

Definition 1.1.1 (Group) A *group* is a structure $(G, *, ^{-1}, u)$, with G the universe of elements, binary operator $*$ on G, unary operator $^{-1}$, and constant $u \in G$. For any $a, b, c \in G$, the following laws, or axioms, hold:

- $a * (b * c) = (a * b) * c$;

https://doi.org/10.1017/CBO9781139195003.005 Published online by Cambridge University Press

- $u * a = a = a * u$;
- $a * a^{-1} = a^{-1} * a = u$.

So, a group is any mathematical structure consisting of a single universe of elements, with operators on this universe of elements that satisfy the group axioms. Stated differently, a group is any *model* of the *equational theory of groups*. Likewise, it is possible to define operations on the universe of processes. A *process algebra* is then any mathematical structure satisfying the axioms given for the defined operators, and a process is then an element of the universe of this process algebra. The axioms allow *calculations* with processes, often referred to as *equational reasoning*.

Process algebra thus has its roots in universal algebra. The field of study nowadays referred to as process algebra, however, often, goes beyond the strict bounds of universal algebra. Sometimes the restriction to a single universe of elements is relaxed and different types of elements, different sorts, are used, and sometimes binding operators are considered. Also this book goes sometimes beyond the bounds of universal algebra.

The simplest model of system behavior is to see behavior as an input/output function. A value or input is given at the beginning of a process, and at some moment there is a(nother) value as outcome or output. This behavioral model was used to advantage as the simplest model of the behavior of a computer program in computer science, from the start of the subject in the middle of the twentieth century. It was instrumental in the development of (finite-state) *automata theory*. In automata theory, a process is modeled as an automaton. An automaton has a number of *states* and a number of *transitions*, going from state to state. A transition denotes the execution of an (elementary) action, the basic unit of behavior. Besides, an automaton has an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e., an execution path of actions that lead from the initial state to a final state. Given this basic behavioral abstraction, an important aspect is when to consider two automata equal, expressed by a notion of equivalence, the *semantic equivalence*. On automata, the basic notion of semantic equivalence is language equivalence: an automaton is characterized by the set of runs, and two automata are equal when they have the same set of runs. An algebra that allows equational reasoning about automata is the algebra of regular expressions (see e.g., (Linz, 2001)).

Later on, the automata model was found to be lacking in certain situations. Basically, what is missing is the notion of *interaction*: during the execution from initial state to final state, a system may interact with another system. This is needed in order to describe parallel or distributed systems, or so-called

reactive systems. When dealing with models of and reasoning about interacting systems, the phrase *concurrency theory* is used. Concurrency theory is the theory of interacting, parallel and/or distributed systems. Process algebra is usually considered to be an approach to concurrency theory, so a process algebra will usually (but not necessarily) have *parallel composition* as a basic operator. In this context, automata are mostly called *transition systems*. The notion of equivalence studied is usually not language equivalence. Prominent among the equivalences studied is the notion of *bisimilarity*, which considers two transition systems equal if and only if they can mimic each other's behavior in any state they may reach.

Thus, a usable definition of *the field of process algebra* is the field that studies the behavior of parallel or distributed systems by algebraic means. It offers means to describe or *specify* such systems, and thus it has means to talk about parallel composition. Besides this, it can usually also talk about alternative composition (choice between alternatives) and sequential composition (sequencing). Moreover, it is possible to reason about such systems using algebra, i.e., equational reasoning. By means of this equational reasoning, *verification* becomes possible, i.e., it can be established that a system satisfies a certain property. Often, the study of transition systems, ways to define them, and equivalences on them are also considered part of process algebra, even when no equational theory is present.

1.2 Calculation

Systems with distributed or parallel, interacting components abound in modern life: mobile phones, personal computers interacting across networks (like the web), and machines with embedded software interacting with the environment or users are but a few examples. In our mind, or with the use of natural language, it is very difficult to describe these systems exactly, and to keep track of all possible executions. A formalism to describe such systems precisely, allowing reasoning about such systems, is very useful. Process algebra is such a formalism.

It is already very useful to have a formalism to describe, to specify interacting systems, e.g., to have a compact term specifying a communication protocol. It is even more useful to be able to reason about interacting systems, to verify properties of such systems. Such verification is possible on transition systems: there are automated methods, called *model checking* (see e.g., (Clarke *et al.*, 2000)), that traverse all states of a transition system and check that a certain property is true in each state. The drawback is that transition systems grow very large very quickly, often even becoming infinite. For instance, a system

having 10 interacting components, each of which has 10 states, has a total number of 10 000 000 000 states. It is said that model-checking techniques suffer from the *state-explosion problem*. At the other end, reasoning can take place in logic, using a form of deduction. Also here, progress is made, and many *theorem-proving tools* exist (see e.g., (Bundy, 1999)). The drawback here is that finding a proof needs user assistance, as the general problem is undecidable, and this necessitates a lot of knowledge about the system.

Equational reasoning on the basis of an algebraic theory takes the middle ground, in an attempt to combine the strengths of both model checking and theorem proving. Usually, the next step in the procedure is clear. In that sense, it is more rewriting than equational reasoning. Consequently, automation, which is the main strength of model checking, can be done straightforwardly. On the other hand, representations are compact and allow the presence of parameters, so that an infinite set of instances can be verified at the same time, which are strong points of theorem proving.

As an example, Chapter 8 presents a complete verification of a simple communication protocol: it is verified that the external behavior of the protocol coincides with the behavior of a one-place buffer. This is the desired result, because it proves that every message sent arrives at the receiving end.

1.3 History

Process algebra started in the 1970s. At that point, the only part of concurrency theory that existed was the theory of Petri nets, conceived by Petri starting from his thesis in 1962 (Petri, 1962). In 1970, three main styles of formal reasoning about computer programs could be distinguished, focusing on giving semantics (meaning) to programming languages.

(i) *Operational semantics*: A computer program is modeled as an execution of an abstract machine. A state of such a machine is a valuation of variables; a transition between states is an elementary program instruction. The pioneer of this field is McCarthy (McCarthy, 1963).

(ii) *Denotational semantics*: In a denotational semantics, which is typically more abstract than an operational semantics, computer programs are usually modeled by a function transforming input into output. The most well-known pioneers are Scott and Strachey (Scott & Strachey, 1971).

(iii) *Axiomatic semantics*: An axiomatic semantics emphasizes proof methods proving programs correct. Central notions are program assertions, proof triples consisting of precondition, program statement,

and postcondition, and invariants. Pioneers are Floyd (Floyd, 1967) and Hoare (Hoare, 1969).

Then, the question was raised how to give semantics to programs containing a parallel-composition operator. It was found that this is difficult using the methods of denotational, operational, or axiomatic semantics as they existed at that time, although several attempts were made. (Later on, it became clear how to extend the different types of semantics to parallel programming, see e.g., (Owicki & Gries, 1976) or (Plotkin, 1976).) Process algebra developed as an answer to this question.

There are two paradigm shifts that need to be made before a theory of parallel programs in terms of a process algebra can be developed. First of all, the idea of a behavior as an input/output function needs to be abandoned. The relation between input and output is more complicated and may involve *nondeterminism*. This is because the interactions a process has between input and output may influence the outcome, disrupting functional behavior. A program can still be modeled as an automaton, but the notion of language equivalence is no longer appropriate. Secondly, the notion of *global* variables needs to be overcome. Using global variables, a state of a modeling automaton is given as a valuation of the program variables, that is, a state is determined by the values of the variables. The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at any given moment. It turns out to be simpler to let each process have its own local variables, and to denote exchange of information explicitly via *message passing*.

Bekič

One of the first people studying the semantics of parallel programs was Hans Bekič. He was born in 1936, and died due to a mountain accident in 1982. In the early seventies, he worked at the IBM lab in Vienna, Austria. The lab was well-known in the sixties and seventies for its work on the definition and semantics of programming languages, and Bekič played a part in this, working on the denotational semantics of ALGOL and PL/I. Growing out of his work on PL/I, the problem arose how to give a denotational semantics for parallel composition. Bekič tackled this problem in (Bekič, 1971). This internal report, and indeed all the work of Bekič, is made accessible through the book edited by Cliff Jones (Bekič, 1984). The following remarks are based on this book.

In (Bekič, 1971), Bekič addresses the semantics of what he calls 'quasi-parallel execution of processes'. From the introduction:

> Our plan to develop an *algebra of processes* may be viewed as a *high-level* approach: we are interested in how to compose complex processes from simpler (still arbitrarily complex) ones.

Bekič uses global variables, so a state is a valuation of variables, and a program determines an action, which gives in a state (non-deterministically) either *null* if and only if it is an end-state, or an elementary step, giving a new state and rest-action. Further, Bekič has operators for alternative composition, sequential composition, and (quasi-)parallel composition. He gives a law for quasi-parallel composition, called the 'unspecified merging' of the elementary steps of two processes. That law is definitely a precursor of what later would be called the expansion law of process algebra. It also makes explicit that Bekič has made the first paradigm shift: the next step in a merge is not determined, so the idea of a program as a function has been abandoned.

Concluding, Bekič contributed a number of basic ingredients to the emergence of process algebra, but he does not yet provide a coherent comprehensive theory.

CCS

The central person in the history of process algebra without a doubt is Robin Milner. A.J.R.G. Milner, born in 1934, developed his process theory CCS, the *Calculus of Communicating Systems*, over the years 1973 to 1980, culminating in the publication of the book (Milner, 1980) in 1980.

Milner's oldest publications concerning the semantics of parallel composition are (Milner, 1973; Milner, 1975), formulated within the framework of denotational semantics, using so-called transducers. He considers the problems caused by non-terminating programs, with side effects, and non-determinism. He uses operators for sequential composition, for alternative composition, and for parallel composition. He refers to (Bekič, 1971) as related work.

Next, in terms of the development of CCS, are the articles (Milner, 1979) and (Milne & Milner, 1979). In that work, Milner introduces *flow graphs*, with ports, where a named port synchronizes with the port with its co-name. Operators are parallel composition, restriction (to prevent certain specified actions), and relabeling (for renaming ports). Some laws are stated for these operators.

The two papers that put in place most of CCS as it is known to date, (Milner, 1978a) and (Milner, 1978b), conceptually built upon this work, but appeared in 1978. The operators prefixing and alternative composition are added and provided with laws. Synchronization trees are used as a model. The prefix τ occurs as a *communication trace*, i.e., what remains of a synchronization of a name and a co-name. Such a remains is typically unobservable, and later,

τ developed into what is now usually called the silent step. The paradigm of message passing, the second paradigm shift, is taken over from (Hoare, 1978). Interleaving is introduced as the observation of a single observer of a communicating system, and the expansion law is stated. Sequential composition is not a basic operator, but a derived one, using communication, abstraction, and restriction.

The paper (Hennessy & Milner, 1980), with Matthew Hennessy, formulates basic CCS, with two important semantic equivalence relations, observational equivalence and strong equivalence, defined inductively. Also, so-called Hennessy-Milner logic is introduced, which provides a logical characterization of process equivalence. Next, the book (Milner, 1980) was published, which is by now a standard process algebra reference. For the first time in history, the book presents a complete process algebra, with a set of equations and a semantic model. In fact, Milner talks about *process calculus* everywhere in his work, emphasizing the calculational aspect. He presents the equational laws as truths about his chosen semantic domain, transition systems, rather than considering the laws as primary, and investigating the range of models that they have. The book (Milner, 1980) was later updated in (Milner, 1989).

CSP

A very important contributor to the development of process algebra is Tony Hoare. C.A.R. Hoare, born in 1934, published his influential paper (Hoare, 1978) as a technical report in 1976. The important step is that he does away completely with global variables, and adopts the message-passing paradigm of communication, thus realizing the second paradigm shift. The language CSP, *Communicating Sequential Processes*, described in (Hoare, 1978) has synchronous communication and is a guarded-command language (based on (Dijkstra, 1975)). No model or semantics is provided. This paper inspired Milner to treat message passing in CCS in the same way.

A model for CSP was elaborated in (Hoare, 1980). This is a model based on trace theory, i.e., on the sequences of actions a process can perform. Later on, it was found that this model was lacking, for instance because deadlock behavior is not preserved. For this reason, a new model based on so-called failure pairs was presented in (Brookes *et al.*, 1984), for the language that was then called TCSP, *Theoretical* CSP. Later, TCSP was called CSP again. In the language, due to the less discriminating semantics when compared to the equivalence adopted by Milner and the presence of two alternative composition operators, it is possible to do without a silent step like τ altogether. The book (Hoare, 1985) gives a good overview of CSP.

Between CCS and CSP, there is some debate concerning the nature of alternative composition. Some say the + of CCS is difficult to understand (exemplified by the philosophical discussion on 'the weather of Milner'), and CSP proposes to distinguish between internal and external non-determinism, using two separate operators; see also (Hennessy, 1988a).

Some other process theories

Around 1980, concurrency theory and in particular process theory is a vibrant field with a lot of activity world wide. There is research on Petri nets, partially ordered traces, and temporal logic, among others. Other process theories are trace theory and the invariants calculus. In particular, there is the metric approach by De Bakker and Zucker (De Bakker & Zucker, 1982a; De Bakker & Zucker, 1982b). It has a notion of distance between processes: processes that do not differ in behavior before the n-th step have a distance of at most 2^{-n}. This turns the domain of processes into a metric space, that can be completed. Recursive equations allow to specify unbounded process behavior. In the metric approach by De Bakker and Zucker, solutions to an important class of recursive equations, so-called *guarded* recursive equations, exist by application of Banach's fixed point theorem. This result later influenced the development of process algebra, in particular the development of ACP.

ACP

Jan Bergstra and Jan Willem Klop started to work in 1982 on a question of De Bakker's as to what can be said about solutions of *unguarded* recursive equations. As a result, they wrote the paper (Bergstra & Klop, 1982). In this paper, the phrase 'process algebra' is used for the first time, with exactly the two meanings given in the first part of this chapter. The paper defines a process algebra with alternative, sequential, and parallel composition, but without communication. A model was established based on projective sequences, meaning that a process is given by a sequence of approximations by finite terms, and in this model, it is established that all recursive equations, both guarded and unguarded, have a solution. In adapted form, this paper was later published as (Bergstra & Klop, 1992). In (Bergstra & Klop, 1984a), this process algebra, called PA, for Process Algebra, was extended with communication to yield the theory ACP, the *Algebra of Communicating Processes*. Textbooks on ACP are (Baeten & Weijland, 1990; Fokkink, 2000).

Comparing the three most well-known process algebras to date, CCS, CSP, and ACP, it can be concluded that there is a considerable amount of work and

applications realized in all three of them. In that sense, there seem to be no fundamental differences between the theories with respect to the range of applications. Historically, CCS was the first with a complete theory. Compared to the other two, CSP has the least distinguishing equational theory. More than the other two, ACP emphasizes the algebraic aspect: there is an equational theory with a range of semantic models. Also, ACP has the most general communication scheme: in CCS, communication is combined with abstraction, and also CSP has a restricted communication scheme.

Further developments

The development of CCS, CSP, and ACP was followed by the development of other process algebras, such as SCCS (Milner, 1983), CIRCAL (Milne, 1983), MEIJE (Austry & Boudol, 1984), and the process algebra of Hennessy (Hennessy, 1988a). Moreover, many process algebras were extended with extra features, such as timing or probabilities. A number of these extensions are also addressed in this book.

Over the years, many process algebras have been developed, each making its own set of choices in the different possibilities. The reader may wonder whether this is something to be lamented. In (Baeten *et al.*, 1991), it is argued that this is actually a good thing, as long as there is a good exchange of information between the different research groups, as each different process algebra has its own set of advantages and disadvantages. The theoretical framework developed in this book is generic, in the sense that most features found in other process algebras can be defined in it. Throughout the book, it is indicated how this can be achieved.

This book

This book follows the ACP approach in its emphasis on algebra. The main difference with the theory set out in (Bergstra & Klop, 1984a; Baeten & Weijland, 1990) is that successful termination is integrated in the theory almost from the beginning. As set out in (Baeten, 2003), this leads to some other changes in the theory. The basic theory starts with a prefixing operator as in CCS and CSP, and adds the sequential-composition operator, which is a basic operator in ACP, in a later chapter.

This book arose as a complete and thorough revision of the book (Baeten & Weijland, 1990). Although many changes have occurred, the approach and methodology remain the same. Also some parts of the text have remained almost unchanged. The book has been updated in many places to reflect the

latest developments, making it the most complete and in-depth account of the state-of-the-art in process algebra at the time of writing.

Bibliographical remark

The historic overview of this section first appeared in (Baeten, 2005).