Chapman & Hall/CRC
Computational Science Series

Process Algebra for Parallel and Distributed Processing

Edited by
Michael Alexander
William Gardner

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING

EDITED BY
MICHAEL ALEXANDER
WILLIAM GARDNER

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

Cover Image Credit: Intel Teraflops Research Chip Wafer from Intel, used with permission.

Chapman & Hall/CRC
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-6486-5 (Hardcover)

Library of Congress Cataloging-in-Publication Data

Process algebra for parallel and distrubuted processing / editors, Michael Alexander and William Gardner.
p. cm. -- (Chapman & Hall/CRC computational science series)
Includes bibliographical references and index.
ISBN 978-1-4200-6486-5 (alk. paper)
1. Parallel processing (Electronic computers) 2. Electronic data processing--Distributed processing. 3. Formal methods (Computer science) I. Alexander, Michael, 1970 Sept. 25- II. Gardner, William, 1952-

QA76.58.P7664 2009
004.01'51--dc22 2008029295

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING

Chapman & Hall/CRC
Computational Science Series

SERIES EDITOR

Horst Simon
Associate Laboratory Director, Computing Sciences
Lawrence Berkeley National Laboratory
Berkeley, California, U.S.A.

AIMS AND SCOPE

This series aims to capture new developments and applications in the field of computational science through the publication of a broad range of textbooks, reference works, and handbooks. Books in this series will provide introductory as well as advanced material on mathematical, statistical, and computational methods and techniques, and will present researchers with the latest theories and experimentation. The scope of the series includes, but is not limited to, titles in the areas of scientific computing, parallel and distributed computing, high performance computing, grid computing, cluster computing, heterogeneous computing, quantum computing, and their applications in scientific disciplines such as astrophysics, aeronautics, biology, chemistry, climate modeling, combustion, cosmology, earthquake prediction, imaging, materials, neuroscience, oil exploration, and weather forecasting.

PUBLISHED TITLES

PETASCALE COMPUTING: Algorithms and Applications
Edited by David A. Bader

PROCESS ALGEBRA FOR PARALLEL AND DISTRIBUTED PROCESSING
Edited by Michael Alexander and William Gardner

Foreword

This book brings together the state of the art in research on applications of process algebras to parallel and distributed processing.

Process algebras constitute a successful field of computer science. This field has existed for some 30 years and stands nowadays for an extensive body of theory of which much has been deeply absorbed by the researchers in computer science. Moreover, the theoretical achievements of the field are to a great extent justified by applications. The applications, in turn, strongly influence how the field evolves; some of the field's success may be attributed to frequently addressing needs that arose in practice.

Meanwhile, an explosion of complex systems of interacting components has been going on since the emergence of parallel and distributed processing. The complexity of the systems in question arises to a great extent from the many ways in which their components can interact. In developing a complex system of interacting components, it is important to be able to describe the behavior of the system in a precise way at various levels of detail, and to analyze it on the basis of the descriptions. Process algebras were and are developed for that purpose. Roughly speaking, a process algebra provides a collection of operators, a collection of equational laws for these operators, and a mathematical model of these laws. The latter allows for the behavior of a system to be described as composed of the emergent behaviors of several interacting components and for the described behavior to be analyzed by mere algebraic calculations.

The advent of process algebras was marked by the introduction of CCS in the seminal monograph *A Calculus of Communicating Systems* by Milner, published as volume 92 of Springer's Lecture Notes in Computer Science in 1980; the elaboration of CSP in the influential paper "*A theory of communicating sequential processes*" by Brookes, Hoare, and Roscoe, published in the *Journal of the ACM* in 1984; and the presentation of ACP as a strict algebraic theory in the paper "*Process algebra for synchronous communication*" by Bergstra and Klop, published in *Information and Control* in 1984.

The very first applications of process algebras were mostly concerned with the description and analysis of communication protocols. Later on, the applications became more and more advanced. Often, they were concerned with the description and analysis of embedded systems, and more recently with Internet-based distributed systems.

The applications of process algebras led to a number of developments. The very first applications brought about the development of basic algebraic verification techniques, i.e., basic techniques to establish—on the basis of algebraic

calculations—whether the actual behavior of a system is in agreement with its expected behavior. The construction of basic tools to facilitate description and analysis followed. Later applications led to extensions of existing process algebras and the development of more advanced algebraic verification techniques. Both make it easier to describe and analyze the behaviors of the systems often encountered in practice nowadays: systems that may change their communication topology dynamically, systems that must react within a certain amount of time under certain circumstances, systems that exhibit at certain stages behavior that is stochastic in nature, systems that in their behavior depend on continuously changing variables other than time, etc. Experience with the existing process algebras led also to the development of special process algebras for the definition of the semantics of programming languages that support parallel programming or the design of microprocessors that utilize parallelism to speed up instruction processing.

It is difficult to foresee future developments and applications, yet some tendencies are noticeable. One area that is gaining momentum is specialized process algebras that are tailored to a certain paradigm for parallel or distributed computing or even to a certain technology for parallel or distributed computing. This fits in with the tendency to apply process algebras to describe and analyze a prototype of a certain class of systems. Such applications can be useful in understanding certain aspects of the systems of an emerging class. However, the adequacy of the prototypes may be a major issue, because often drastic simplifications are needed to keep the description manageable. There is also a tendency to apply process algebras outside the realm of computing, which shows promise.

Many theoretical developments in the field of process algebras were collected by Bergstra, Ponse, and Smolka in the *Handbook of Process Algebra*, published in 2001. In 2005, a workshop was organized in Bertinoro, Italy, to celebrate the first 25 years of research in the field. Several special issues of the *Journal of Logic and Algebraic Programming* are devoted to this workshop. Despite the importance of applications of process algebras for the success of the field, both the handbook and these special issues concentrate strongly on the theoretical achievements. This shortcoming is compensated for in a splendid way by this book, which brings together the state of the art in research on applications of process algebras.

Kees Middelburg
Programming Research Group
University of Amsterdam, the Netherlands

Acknowledgments

The editors are very grateful to those listed below who served as peer reviewers for contributions to this book. Their diligent and conscientious efforts not only helped us to make the final selection of chapters but also provided numerous valuable suggestions to the authors, resulting in a high-quality collection.

Luca Aceto
School of Computer Science
Reykjavik University
Reykjavik, Iceland

Lorenzo Bettini
Dipartimento di Informatica
Universita' di Torino
Torino, Italy

Tommaso Bolognesi
CNR—Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo"
Pisa, Italy

Gerhard Chroust
Systems Engineering and Automation
Institute of Systems Sciences
Johannes Kepler University of Linz
Linz, Austria

Philippe Clauss
Scientific Parallel Computing and Imaging
Université Louis Pasteur
Strasbourg, France

Pedro R. D'Argenio
Department of Computer Science
Facultad de Matemática, Astronomia y Fisica
Universidad Nacional de Córdoba—CONICET
Córdoba, Argentina

John Derrick
Department of Computer Science
University of Sheffield
Sheffield, South Yorkshire, United Kingdom

Gaétan Hains
Laboratoire d'Algorithmique, Complexité et Logique
University of Paris-Est
Créteil, France

and

SAP Labs France
Mougins, France

Michael G. Hinchey
Lero—The Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland

Thomas John
Department of Information Systems
Vienna University of Economics and Business Administration
Vienna, Austria

Kenneth B. Kent
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada

Felix Mödritscher
Institute for Information Systems
and New Media
Vienna University of Economics and
Business Administration
Vienna, Austria

Raymond Nickson
School of Mathematics, Statistics,
and Computer Science
Victoria University of Wellington
Wellington, New Zealand

Frederic Peschanski
Laboratoire d'Informatique de Paris 6
UPMC Paris Universitas
Paris, France

Luigi Romano
Department of Technology
University of Naples Parthenope
Naples, Italy

Gwen Salaün
Department of Computer Science
University of Málaga
Málaga, Spain

Emil Sekerinski
Department of Computing
and Software
McMaster University
Hamilton, Ontario, Canada

Kenneth J. Turner
Department of Computing Science
and Mathematics
University of Stirling
Scotland, United Kingdom

Huibiao Zhu
Software Engineering Institute
East China Normal University
Shanghai, China

Introduction

Parallel processing is a rich and rapidly growing field of interest in computer science. Its spectrum ranges from small-scale, fine-grained multithreaded parallelism on single- and multicore processors to coarse-grained parallel execution on large, geographically dispersed distributed systems. With the rapid commoditization of multicore computers, developers are increasingly eager to exploit those multiple cores. Yet, this requires them to adopt parallel execution models that in turn need to explicitly address the underlying concurrency issues.

At the other end of the spectrum are applications that are spread out over a network of computers. They may be tightly coupled, such as for multiple computers cooperating on a single application in a high-performance cluster. Or, they may be loosely organized, as for mobile agents or service-oriented architectures (SOA). Regardless of the scale, a common requirement in parallel execution models is for carrying out interprocess communication and synchronization. Programmers know—or soon learn—that when this requirement is handled carelessly, a host of unpleasant failure modes tend to manifest: deadlocks, unrepeatable errors from race conditions, along with corruption of shared data.

One root cause is that the usual tools that programmers know best—mutexes, semaphores, condition variables, monitors, and message-passing protocols—are difficult to use correctly. Ad hoc design is common, and too often, success seems like a matter of good luck. In reality, many programmers new to parallel programming have an insufficient theoretical basis for what they are trying to do. Meanwhile, designers have learned that *modeling* is extremely helpful for all phases of system development—from understanding the requirements to expressing the specifications, and for systematically, even automatically, deriving an implementation "correct by construction" from the model. Yet, most popular modeling tools do not seem to give sufficient guidance for programming in concurrent environments.

Help is coming forth from computer scientists, who are fond of developing rigorous and logical ways of thinking and reasoning about computing. The purpose of this book is to show how one formal method of reasoning—that of *process algebra*—has become a powerful tool for solving design and implementation challenges of concurrent systems. Its power stems from providing a sound theoretical basis for concurrency, along with a formal notation that—in contrast to popular modeling techniques—is unambiguous. The price for obtaining the benefits is the necessity of coping with mathematical notation, symbols, and equations. Admittedly, this prospect may appear daunting to conventionally trained developers, who likely feel more comfortable drawing UML diagrams than puzzling out obscure equations. Fortunately, practitioners of process algebraic techniques are building domain-specific languages

and application-specific tools that can be utilized by developers who then need to know less about the underlying theoretical elements. In such tools, the process algebra may be largely kept under the hood.

This book is not intended as a tutorial in process algebras, but there is space here for a brief orientation: Just as with the "algebra" we all learned in grade school, its ingredients include symbols standing for constant values and for variables, and operators that act on the symbols. While elementary algebra is concerned with manipulating numbers, a process algebra—or the synonymous term *process calculus*—is concerned with the creation, life, and death of processes that carry out computations. The emphasis is on interactions of processes among each other and with their environment. Symbols are used to stand for individual actions or events that a process may engage in; the processes themselves; channels, an abstraction used for interprocess communication; and data transmitted over the channels. Operators specify a sequential ordering of events; a choice between several events; if/then decisions; looping or recursive invocation of processes; composition of processes so that they execute concurrently, either synchronizing cooperatively on specified events, or running independently; and syntax for parameterizing and renaming so that process definitions can be reused in a modular fashion. Practitioners can use one of the classical process algebras mentioned in the Foreword—CCS, CSP, and ACP—or a more recent one such as the pi-calculus. Established process algebras have the advantage of the availability of automated tools that can be used for exploring a specification's state space and proving properties such as absence of deadlock (see Section 9.1 for a good explanation of these provable properties). Alternatively, practitioners can extend any of the base algebras by adding new symbols and operators, or even invent a new process algebra from scratch. A key advantage of formal, mathematical notations with rigorous semantics is that one can prove that particular conditions such as deadlock states do or do not exist, not just hope for the best, or discover them at runtime.

While the notion of process algebras goes back over two decades, what is new is the rapid proliferation of parallel computing environments that need their help in transforming sequential programming models to new ones better suited to parallelism. The urgent necessity for reliably exploiting concurrent computing resources has caused researchers to press classical process algebras into practical service, along with their invention of new ones. This book is intended as a showcase for recent applications of process algebras by current researchers from diverse parts of the international computer science community. While these contributions may appear largely theoretical due to the quantities of symbols and formulas, they are, in reality, process algebras applied to specific problems. The formalism is needed to establish the soundness of the theoretical basis, and to prove that the resulting tools are properly derived.

This book will be of special interest to students of process algebras, to practitioners who are applying process algebras, and to developers who are looking for fresh approaches to software engineering in the face of concurrency. The chapters are worth studying from two perspectives: first, those who identify with the problem

domains (e.g., middleware systems or multicore programming) may ask, "Are the authors doing something that I could use, or can I adapt their approach?" and second, those who are interested in process algebras as a tool can ask, "How did the authors use a process algebra in their solution? What role did it play? How did they formally define it, and what did they prove in order to give it a sound basis?" A common pattern is to first create a process algebra whose symbols and operators are tailored to the problem domain (e.g., mobile agents). Next, a virtual (or abstract) machine (VM) is defined that executes specifications in the process algebra. Because both are formally defined, it is possible to prove that the VM is correct with respect to the algebra. The last step is to implement the VM, and this may be done by transferring it into a language that is semantically close, e.g., a functional programming language. Since the target language already possesses a compiler and/or runtime framework, the job of implementation is done, at least for prototyping and demonstration purposes.

The applied nature of the contributions is emphasized by organizing them into three target areas:

Part I Parallel Programming, Part II Distributed Systems, and Part III Embedded Systems.

We will now introduce each section and the chapters within it.

Part I Parallel Programming

The specific problem in view here is how to parallelize an algorithm, so as to take advantage of, say, multiple processor cores. In an ideal world, parallelization would be accomplished automatically, perhaps by compilers that are able to detect implicit parallelism in source code and generate instructions for concurrent threads on their own. Bearing in mind that modern processors already do this, in effect, with instruction streams—selecting independent instructions for out-of-order execution by multiple logic units—it may seem surprising that compilers have largely failed to match this at the source code level. But processors are detecting and exploiting implicit fine-grained parallelism. In contrast, fine-grained parallelism in software algorithms is often not worth exploiting, since there are significant overheads in spawning, managing, and collecting results from concurrent threads, plus potential bottlenecks for access to shared data. Furthermore, automatically identifying higher-level, coarse-grained parallelism implicit in source code, while desirable from an efficiency perspective, has proven to be a very challenging problem.

Therefore, in the real world, parallelization is still done on a best effort basis by hand, and then it becomes a question of ensuring that a parallel version is truly equivalent to the serial version. Our contributors have developed formal methodologies for achieving precisely that result.

In Chapter 1, Anand and Kahl address programming the Cell BE (Broadband Engine) processor from Sony, Toshiba, and IBM. Its heterogeneous multicore architecture features eight Synergistic Processor Units (SPUs) with their own local storage on a token ring under the control of a general-purpose Power Processor Element (PPE) core. The SPUs are intended to act as coprocessors for the PowerPC, being loaded on-the-fly with instructions and data, and coordinating via signals. The authors'

Coconut tool set provides the means to take an algorithm written in a domain-specific language (DSL) embedded in Haskell, and parallelize it for the Cell BE. The key to their approach is utilizing a graph-based internal representation of the program's data and control flows, which are then targeted to a VM that deals strictly with concurrency issues, e.g., data transfers and interprocessor signaling. The programmer manipulates the graph to create a high-performance schedule on the eight SPUs, with the authors' tool being used to verify that the scheduled version is correct, i.e., independent of a parallel execution order. The role of process algebra in this approach is to define the VM language, and then carry out correctness verification. While currently targeted to the Cell BE, their approach can potentially be ported to other multicore platforms.

In Chapter 2, Loidl et al. take an approach similar to that in Chapter 1 in that they also develop a runtime environment, Glasgow parallel Haskell (GpH), that can be ported to different parallel platforms, and they also focus on a functional programming language. Rather than attempting to automatically extract parallelism from GpH source code, they allow programmers to insert "par" and "seq" constructs into a program to give "semi-explicit" direction while carrying out successive steps of refinement to a parallel version. Their runtime environment, GUM, has been ported to a number of different parallel platforms.

These two chapters describe tools that are currently being used to program parallel systems. In comparison, Chapter 3 is more theoretical. Degerlund and Sere present an approach to taking algorithms described using another formal model called *action systems*, and developing an equivalent parallel version useful for scientific computing. An action system is specified using a process algebra called *refinement calculus*. The steps of refinement are used to introduce parallelism into the action system, with execution on a parallel target platform in view. The formal semantics of the refinement calculus ensure that the transformations are correct.

Part II Distributed Systems

Process algebras find their natural application in terms of formally modeling and verifying the behavior of distributed systems. Distributed systems are quite diverse, and this section also has the largest number of chapters.

We start with the work of Groote et al. in Chapter 4, who have developed a process algebra, mCRL2, which is specifically targeted at distributed applications. Its provision for local communication scope (i.e., restricted to a hierarchy of processes), as opposed to purely global scope, makes it useful for describing component-based architectures. mCRL2 also accommodates true concurrency "multiactions" distinct from interprocess communication, and supports the specification of abstract data types and action times. The authors have built tools capable of analyzing properties and simulating applications specified using mCRL2. This chapter has examples of visualizing the state space of a specification by means of a generated graphic.

One category of distributed systems that is currently gaining attention is the SOA that enables a business process to be automated by invoking software components located across a network. However, business processes and services described in words are subject to misinterpretations, leading to errors in integrating SOAs. A key

area for formalization is turning prose descriptions into unambiguous specifications. Chapters 5 and 6 make contributions in this area.

Chapter 5 shows Nestmann and Puhlmann using an existing process algebra, the pi-calculus, to formally specify business processes. Their approach allows business processes to be captured in the form of Business Process Modeling Notation (BPMN) process graphs—whose nodes specify interactions such as parallel split, synchronizing merge, exclusive choice, and others—and then converted to pi-calculus agents enhanced by the authors' "trio" construct. Interactions of business processes and services can be formally modeled, and the models analyzed for various soundness properties. The authors have created a tool to automate the property analysis.

In Chapter 6, Rosa uses an ISO-standard process algebra, LOTOS, to formalize the construction of middleware systems. Each architectural component is specified as a single LOTOS process, which defines the component's structure—the ports available to connect to other components—and behavior in process-algebraic terms. The use of a formal notation as an architecture description language, in contrast to ambiguous prose descriptions, aids both would-be service providers and service integrators, and makes it possible to prove temporal properties of an architecture. The author further employs this technique to create a library of abstract message-oriented components for use in defining middleware. A third demonstration formalizes middleware for wireless sensor networks.

Another manifestation of distributed systems is based on the notion of mobile agents moving around a network. The purpose is to send software to the data rather than pulling the data down to a computation node. As argued in Chapter 8, mobility helps to minimize the impact of two problems common to distributed systems: network latency and network failure. Systems based on mobile agents will also benefit from formal descriptions, especially when it comes to ensuring security. Chapters 7 and 8 deal with mobility. As in Chapters 1 and 2, the authors of Chapters 7 and 8 take the approach of formally defining a VM.

In Chapter 7, similar to Chapters 5 and 6, Paulino targets SOA and middleware. His VM, service-oriented mobility abstract machine, is based on an extension of the pi-calculus. By programming for the VM, programmers can be abstracted from the details of the network while still utilizing mobility. The author's strategy for deployment is to execute the machine on network nodes running an existing framework called DiTyCO.

In Chapter 8, Phillips presents his Channel Ambient (CA) calculus for specifying mobile applications. The textual notation also has a helpful graphical counterpart. Based on the abstract notion of an "ambient"—which may stand for a machine, a mobile agent, or a software module—the CA calculus provides operations whereby ambients may interact, and migrate in and out of each other, via channels. Security properties can be verified for a given specification. The execution target is called the Channel Ambient Machine (CAM), and its correctness with respect to the CA calculus is proven by the author. His implementation strategy is to map the CAM to a functional programming language, OCaml, that can then be executed on network nodes.

Part III Embedded Systems

While an exact definition of an embedded system is debatable, it is generally described as a product that contains within it some combination of software running on a general- or special-purpose processor, plus associated custom digital logic. The latter is commonly used to accelerate specific portions of calculations, which would otherwise require a more expensive processor in order to meet timing constraints. The term "embedded" refers to the fact that the end user is not necessarily aware that the product contains a computer, and, in any case, cannot utilize the computer for some other purpose. Embedded systems have special design constraints because, unlike software for desktop or server computers, the products have a significant recurring cost, such as parts, assembly, packaging, labor, etc., in addition to the nonrecurring engineering cost that goes into developing and interfacing the software and hardware components. Furthermore, they may have to meet rigorous requirements of power consumption (battery life), size, and weight. Cell phones and digital cameras are common examples. Robotic devices, such as autonomous vacuum cleaners, are embedded systems. Some are safety-critical, such as an antilock brake controller. The aim of choosing the best combination of hardware and software is to meet all the performance requirements at the lowest manufacturing cost, i.e., offering a sufficient set of features at a competitive price.

Parallelism in embedded systems comes in several forms: embedded devices are often designed in terms of concurrent threads, some monitoring sensor inputs, others computing outputs, or actuating control outputs. Some have multiple processors, which may be heterogeneous, such as a 16-bit microcontroller plus a DSP, and some have hardware/software concurrency. They are often designed and marketed in a family of related products with more/fewer features, or adding features over time, or that utilize differing HW technology more favorable to different production quantities. Formal methods are very attractive for ensuring reliability, especially for safety-critical products, and those in hard-to-access locations.

In Chapter 9, McEwan leads off with a contribution targeted at formally deriving a hardware implementation, although his technique is also applicable to software. As with Chapter 3, which combines the state-based formalism of action systems with the event-based refinement calculus, McEwan combines state-based Z with the event-based process algebra CSP, in a formal methodology called Circus. Z is used to provide a formal model for the data. Refinement toward an implementation proceeds by applying laws that safely inject parallelism into a sequential specification. The target language, Handel-C, used to synthesize digital logic, is close to CSP. Circus is flexible enough to allow engineering choices to guide the refinement, while still ensuring that the resulting implementation is correct.

Typical parallel systems leave the scheduling of multiple processes to the operating system under the assumption of adequate CPU time and memory. Accordingly, many process algebras used to specify parallel systems do not have visibility into process scheduling, yet real-time embedded systems must guarantee responses within certain time constraints, and do so in the context of limited resources (chiefly CPU time and memory). Formal notations to specify resource requirements such as timing constraints are therefore of great utility in the embedded domain.

In Chapter 10, Mousavi et al. address this problem by developing a pair of process algebras, together called PARS: one to specify processes, and another to specify scheduler behavior. The two, combined, produce a scheduled system.

The final chapter, Chapter 11, considers embedded systems through the lens of product lines, in which reuse of concurrent artifacts across different hardware platforms is emphasized. Yovine's solution is to use an algebraic language, FXML, to specify concurrent behavior, associated control and data dependencies, and timing constraints. FXML specifications are then processed by a software synthesis tool, Jahuel, to yield an implementation customized for a given platform. Platforms may differ in the means by which concurrency is supported and interprocess synchronization and communication are carried out. FXML can also be used in a design automation toolset as an intermediate specification with formal semantics. Moreover, one can define a translation of a nonformal language into FXML, thus giving the language a formal semantics and opening up the possibility of verification.

We would be remiss not to acknowledge a "dark cloud" in the picture: the challenge of scaling up some of these techniques for industrial-sized applications. Such specifications may have so many states that automated verification tools cannot reach them all in reasonable time. This drawback may not be evident from these chapters themselves, since the authors were forced to use small examples, both for the sake of clarity and due to space limitations. Yet some claim to have applied their techniques to larger-scale problems. For more details, consult the references to the authors' own related work, and they will be pleased to answer queries.

The process of collecting these chapters has highlighted for us, as educators, that colleges and universities have further to go in training undergraduates to be comfortable with concurrency and skilled in reliable methods of parallel programming. More exposure to formal methods is also needed, so that these approaches do not appear so foreign. As it is, in most software development curriculums, particularly in North America, formal methods are more "honored in the breach" than by systematic instruction. At the same time, researchers will do well to embed their process algebraic techniques into tools suitable for users without special training, in order to widen their prospects for adoption.

To conclude, we offer this book as an early collection of research fruits in what will undoubtedly become a burgeoning growth industry in the coming years, and to which the editors are eager to contribute their own research.

Editors

Michael Alexander holds degrees in electrical engineering (Technologisches Gewerbemuseum [TGM]), business administration (University of Southern California), and economics (University of Vienna). He is a supervisor in the Vienna, Austria KPMG IT Advisory practice. His experience includes teaching and research at Wirtschaftsuniversität Wien; as a lecturer and product management at IBM, Siemens, Nortel Networks, and Alcatel; as a product line manager for ADSL and Optical Access Networks. He has authored a textbook on networks and network security [1] published by Hüthig/Verlagsgruppe Süddeutsche, and has edited a special issue on mathematical methods in network management of the *Wiley International Journal of Network Management*. For the last four years, Dr. Alexander has served as the program committee chair for the Workshop on Virtualization in High-Performance Cluster and Grid Computing (VHPC). His current research interests include computer networking, formal methods, network management, concurrency, payment systems, machine learning methods in operations research, and e-learning.

William Gardner received his bachelor's degree in computer science from the Massachusetts Institute of Technology. Subsequently, he pursued his career in software engineering at Litton Systems (Canada) Ltd., Toronto, working primarily on embedded systems for defense customers. Returning to graduate school later in life, he did his doctoral research in the VLSI Design and Test Group in the Department of Computer Science at the University of Victoria, British Columbia, Canada. His interests are chiefly in design automation starting from formal specifications, particularly targeting embedded systems via hardware/software codesign. His software synthesis tool CSP++ [2] translates formal CSP specifications into executable C++. Its design flow features "selective formalism," where a system's control backbone is specified and verified in CSP, and then the principal functionality is provided by plugging C++ user-coded functions into the backbone, to be actuated by CSP events and channel communications. He was formerly on the faculty of Trinity Western University, British Columbia, Canada, and moved in 2002 to the University of Guelph, Ontario, Canada, where he is currently an associate professor in the Department of Computing and Information Science. With his graduate students, he has been participating in the R2D2C project at the NASA Goddard Space Flight Center, which involves automatic code generation from scenarios via under-the-hood formal specifications [3]. Dr. Gardner teaches courses on software development, operating systems, embedded systems, and hardware/software codesign.