Bertrand Meyer *Editor*

The French School of Programming

Springer

The French School of Programming

Bertrand Meyer
Editor

The French School of Programming

Springer

Editor
Bertrand Meyer
Constructor Institute
Schaffhausen, Switzerland

ISBN 978-3-031-34517-3   ISBN 978-3-031-34518-0 (eBook)
https://doi.org/10.1007/978-3-031-34518-0

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Paper in this product is recyclable.

We dedicate this book to the memory of

Gilles Kahn (1946–2006),

a pioneer of programming research in France and onetime director of INRIA, who exerted a profound influence on an entire generation of researchers.

Foreword

An email from Bertrand Meyer in November 2022 surprised me. He asked if I would write a foreword for this collective volume on programming written by French researchers. I was flattered, of course. But the surprise was what Bertrand said next. In spite of the book's title, *The French School of Programming*, Bertrand said, "As a matter of fact no such 'school' exists in the strict sense." But I believe that it does exist. I know this because I learned a lot of my computer science directly from its members.

I learned the foundations of distributed algorithms from Michel Raynal's books, particularly *Synchronization and Control of Distributed Systems and Programs* (with Jean-Michel Hélary). The books gave me the background for industrial distributed and concurrent systems, where I applied formal methods for assurance. These projects include nuclear secondary protection systems (the final line of defense with voting implemented using Laddic magnetic devices); railway signaling systems (with minimum headways between trains guaranteed by automatic synchronization control); and smart cards for financial transactions (a massively distributed system with no centralized control).

I learned abstract interpretation from Patrick and Radhia Cousot. I discovered the beauty and utility of Galois connections. This led me to appreciate the elegance of Hoare and He's *Unifying Theories of Programming*, which has formed the backbone of my research in the semantics of heterogeneous systems over the last two decades.

Marie-Claude Gaudel introduced me to her theoretical framework for understanding the relationship between testing and proof. She showed that testing could tell us more than whether an implementation conforms to a specification on a given test set. Her framework describes a landscape with two extremes. At one end, we prove nothing, but we need an infinite amount of testing. At the other, we test nothing, but we need a complete proof of a system and its context. We can move across this landscape, positioning ourselves between the two extremes. We need to make assumptions about the system under test that support selecting bounded test sets. Marie-Claude showed that testing can be based on strong principles. I am currently applying her ideas to a testing theory for probabilistic behaviors. Our systems under test are artificial neural networks, robotics, and cyber-physical systems. They solve decision and control tasks under uncertainty. A route to greater resilience for these systems is through a rigorous testing theory. As Marie-Claude might have said, *probabilistic* testing can be formal too.

In the late 1980s, I read Bertrand Meyer's book *Object-Oriented Software Construction*. I learned Eiffel and strengthened my knowledge of design by contract in a programming language. I used the book to teach a course on object-oriented programming twice a year to Oxford's part-time masters students from industry. The use of assertions as test oracles was well received and became standard practice in several partner companies.

This book contains contributions by other members of the French School. Jean-Marc Jézéquel has made significant contributions to the foundations of the theory of model-driven architecture. Giuseppe Castagna has established the foundations of session types. Pierre-Louis Curien has made fundamental contributions to research in theoretical computer science, including programming languages and proof theory. Jöelle Coutaz has been influential in human-computer interaction, including user interface plasticity and multimodal interaction. Jean-Pierre Jouannaud has helped establish the theory and practice of term rewriting. Jean-Jacques Ĺevy has made fundamental contributions to our understanding of concurrency and mobility, particularly through the join-calculus. Jean-Pierre Briot has helped us to understand the fundamentals of actors and agents. Thierry Coquand has made important contributions in constructive mathematics, especially the calculus of constructions.

Finally, the French computer scientist Jean-Raymond Abrial influenced me. (He has not made a contribution to this book.) Abrial's work on the Z notation inspired my entire career. I attended a course he gave at Wolfson College in Oxford in the early 1980s. I was working in industry and immediately saw how to apply his ideas to practical projects. I wrote specifications for the storage manager and the call processing subsystems for the operating system in GEC's System X telephone exchanges. I started teaching Z to others in the industry. I even wrote a couple of books on software specification using Z. Abrial's insights are responsible for my reputation as a computer scientist. He taught me how to use mathematics for software specification and development. I helped apply Z to IBM's Customer Information Control System. It is a family of mixed-language servers that provide online transaction processing. The project won a Queen's Award. I used a refinement theory based on Z to prove the correctness of the Mondex smart-card protocol. It was the first product certified to the very highest level of information security (ITSEC level 6). I taught Z to thousands of students around the world and in the global south in particular.

So yes, there is a French School of Programming. Rigor and mathematics underpins its research and has led to many remarkable breakthroughs in computer science. Somewhat fancifully, I see the school as a modern computer science version of the Bourbaki collective. Forty years ago, I made the trip from the Wolfson College workshop to look at the shelves in Oxford's Whitehead Library full of books by the prolific and elusive mathematician that never was. But the French School of Programming has real computer scientists who have made a huge contribution to our discipline and influenced all computer scientists everywhere over the last 50 years. I know, because I am one of them.

University of York Jim Woodcock
York, UK
December 2022

Preface: The French School of Programming

Is there one? Not in the physical sense of a school building featuring a big "*FRENCH SCHOOL OF PROGRAMMING*" sign above the door, classrooms, and students. (Not even in the *Rue des Écoles* in Paris, the Street of Schools, including among others the famous Collège de France where Gérard Berry, one of the authors of this book, teaches.) Also not in the virtual sense of a formal association of like-minded colleagues, such as the *School of Nancy* in glassmaking or the *School of Barbizon* in painting. The French computer science and software engineering community is in tune with the rest of the international science and technology world and has always participated enthusiastically in all its main currents, from the most academic and formal to the most industrial and practical. The contributions to this volume are representative of this vibrant diversity of interests and trends.

And yet there is a common spirit. A quest for elegance and simplicity; insistence on a sound mathematical basis, supported by the great tradition of French mathematics and its influence on the teaching of mathematics in *lycées*, universities, and *grandes écoles*; a focus on the truly important problems: these are some of the distinctive traits of the best work carried out by French researchers and by researchers in French institutions.

Starting from this observation, we contacted in December 2020 some of the most prestigious names in the field with a request to contribute. Most of those approached responded positively; the present volume, with its 13 contributions, is the result, produced after a mutual review process and many discussions. While no restriction had been stated regarding the possible involvement of coauthors, the contributions turned out all to be single-author, showing how seriously the contributors took the request to provide an original chapter reflecting some of their best work.

The book is divided into four parts, reflecting the diversity of interests in the French community and each of them corresponding to an area in which it has made major contributions over several decades:

- Software engineering (Part I)
- Programming language mechanism and type systems (Part II)

- Theory (Part III)
- Language design and programming methodology (Part IV)

They are preceded by a Preface and an Overview chapter. To reflect how intricately the community is bound to its international counterparts, it was important to start the volume by providing the perspective of a foreign colleague. Jim Woodcock, long involved in collaborations with some of the authors of this book and other members of the community, was kind enough to provide the insightful Foreword. The Overview chapter was written by Gérard Berry; it is not only a personal scientific history (of both the author and the French School, of which he has been a prominent member) but also an introduction to the rest of the volume.

Part I, **Software Engineering**, starts (Chap. 2) with an appraisal by Marie-Claude Gaudel of her own pioneering work on *software testing*, which played a major role in providing a sound theoretical basis for testing, now accepted as a full part of the verification process with its own mathematical basis, and extended here with new perspectives. Another critical area which today enjoys solid theoretical foundations is *distributed computing*, in no small part thanks to the books and articles by Michel Raynal; Chap. 3 gives an excellent overview of both the field and his work, with *simplicity* as its core concept. Jean-Marc Jézéquel was for many years Director of IRISA, the Brittany branch of Inria (the legendary national research center in computer science and digital technologies), the source of numerous major contributions; he is also a top researcher in software engineering and has been active in developing one of the principal trends in the field, *model-driven engineering*. Chapter 4 is, like several others in the book, a combination of a survey of this field and a description of the author's individual journey through it. Again in the same spirit of a personal appraisal of a field to which the author has made prime contributions, Joëlle Coutaz describes in her richly illustrated Chap. 5 the stunning evolution of *software engineering for human-computer interaction*.

Part II has three chapters devoted to **programming language mechanisms and type systems**, with a mix of theoretical and practical contributions, reflecting the unique richness of French work in this field and the lack of a strict separation between formal and informal approaches. In Chap. 6, Jean-Pierre Briot discusses an important abstraction, *agents*, providing a unifying generalization of classical programming language concepts of procedure, object, actor, component, and service. He shows the power of this concept and, as in other chapters, describes it in part by telling the story of his own personal itinerary. Chapter 7 by Pierre-Louis Curien is perhaps the most personal of all, describing his discovery of denotational (Scott-Strachey) semantics, all the way to sequential algorithms, "categorical abstract machines," and the Caml and OCaml languages—another set of widely influential French developments—as well as the Curry-Howard correspondence and other fundamental dualities in proofs, logic, and programming languages. Chapter 8 by Thierry Coquand (whose work has profoundly marked the field of logic for computer science, in particular through the Coq proof assistant, one of the most widely used frameworks for software verification) ponders *dependent system theory*, an opportunity to provide a sweeping review of ideas in the field since the days of the first AUTOMATH system.

Part III is devoted to **theory**. Chapter 9 continues the practice of telling the personal story of important discoveries; Patrick Cousot describes the initial insights that led him, together with Radhia Cousot, to invent the theory for which they are famous, *abstract interpretation*, and takes us through its advances and developments through the years. Abstract interpretation is an outstanding example of a theory that is both mathematically elegant and rich with practical applications to industrial software verification. In Chap. 10, Jean-Jacques Lévy, another prestigious representative of the French school of programming languages and logic, explores the notion of *redex* in lambda calculus and its many ramifications. Jean-Pierre Jouannaud, author of Chap. 11, is a pioneer in an important theoretical approach to computing and programming, rewriting systems; in his chapter, he provides a sweeping survey of the field and many insights into the nature of functional programming and functional languages.

The two chapters of Part IV, L**anguage Design and Programming Methodology**, are longer than the others and may be viewed as small monographs. Chapter 12 by Giuseppe Castagna is a comprehensive presentation of a wide range of topics in type theory and develops an extensive unifying theory. This chapter reflects the broad scope of the "French School" concept, as the authors of this book include both French researchers working abroad and foreigners having pursued their careers in France; the latter category has among others included Italians, particularly in type theory and semantics where the French and Italian schools have enjoyed close links. In the final Chap. 13, I take the reader through a number of fundamental decisions in the design of Eiffel and contrast them with the corresponding choices in such languages as Java, C++, and C#, which the chapter argues are based on a flawed understanding of object-oriented ideas, damaging to the quality of the resulting software.

Heartfelt thanks are due to the authors who responded to the call and provided not *pièces de circonstance* but, in all cases, highly personal, substantial, and insightful contributions. The preparation of the volume provided a pleasant experience of collaborative work.

In no way is this collection exhaustive; even as it was being completed, some participants already commented about who else could have participated and what a second volume might include. No such plans currently exist, but this book as it stands, with each chapter written by a major contributor to the field, often on a topic that he or she created, offers countless insights into both decisive past advances and promising ideas for the future. It provides a fascinating look into some of the most burning concepts of modern programming.

Zurich
April 2024

Bertrand Meyer

Contents

About the Authors[1]

Gérard Berry is Professor Emeritus à Collége de France. He worked from 1970 to 2001 at the École des Mines and Inria. His research and industrial activities concerned the mathematics and logic of computing, toward the design of semantically clean programming languages in close connection with novel computation models and application fields. From 1970 to 1982, he introduced the syntactic and semantic notions of stability and sequentiality, which were keys to the solution, with Pierre-Louis Curien, of the full abstraction problem (building semantics that exactly reflect the operational properties of the language) for lambda calculus-based languages.

In 1982, he turned to real-time and reactive programming, creating a mixed research group with control theorists and computer scientists. The key novelty was pure synchrony, in which concurrent processes are assumed to react instantaneously to incoming events and internal communication to produce their outputs. This led to the definition of the synchronous concurrent language Esterel and its mathematical semantics, with evolutions and implementations from 1984 to 2007, in strong cooperation with the Lustre group in Grenoble and the Signal group in Rennes. Esterel gained industrial applications in such areas as avionics, communication protocols, computer-aided design of digital circuits, hardware-software codesign, and human-machine interfaces.

From 2001 to 2009, he was the full-time Chief Scientist of the Esterel Technologies startup company, which improved and industrialized Esterel for hardware and software applications at Dassault Aviation, Intel, Texas Instruments, STMicroelectronics, NXP, etc. Esterel Technologies then bought the Lustre/SCADE team from Telelogic to create the SCADE 6 synchronous language which unified Esterel and Lustre; the company was bought by Ansys in 2012, and SCADE 6 became the world leader in certified safety-critical reactive applications with now 300+ industrial customers.

[1] *Institutions*: CNRS is the National Center for Scientific Research. Inria is the National Institute for Research in Digital Science and Technology. IRISA is a large Inria laboratory in Brittany (Rennes, Lannion and Vannes).

From 2009 to 2012, Gérard Berry came back to research as Director of Research and President of Inria's Evaluation Committee, also holding in parallel two yearly chairs at Collège de France in 2007–2008 and 2009–2010. In September 2012, he was appointed a Full Professor at Collège de France on the new Algorithms, Machines, and Languages chair, where he taught all the above and other subjects before becoming Emeritus Professor in September 2019.

Gérard Berry is a member of the French Academy of Sciences and Academy of Technologies and received the 2014 Gold Medal of CNRS.

Contact: gerard.berry@college-de-france.fr; Web: https://www.college-de-france.fr/chaire/gerard-berry-algorithmes-machines-et-langages-chaire-statutaire/biography

Jean-Pierre Briot is CNRS Research Director in informatics at CNRS. He is a member of LIP6, the joint informatics research lab of CNRS and Sorbonne Université in Paris. He is also an Honorary Visiting Professor at PUC-Rio (Pontifícia Universidade Católica do Rio de Janeiro) in Brazil.

His general research interests are in the design of intelligent adaptive and cooperative software, at the crossroads of artificial intelligence, distributed systems, programming languages, and software engineering. His current interest is in the use of AI techniques (notably deep learning) for music creation processes.

He has been a visiting professor or researcher in various institutions including UNIRIO, Kyoto University, PUC-Rio, Tokyo Institute of Technology, UIUC, USC, and University of Tokyo. He has advised or co-advised more than 30 PhD students and about 20 master students. He has edited 12 books or journal special issues. More details and publications are available at https://webia.lip6.fr/~briot/cv/.

Contact: Jean-Pierre.Briot@lip6.fr; Web: https://webia.lip6.fr/~briot/

Giuseppe Castagna received a PhD degree in Theoretical Computer Science from Université Paris 7 on January 1994. The same year he was appointed a research scientist at CNRS and joined the Computer Science Laboratory of the École Normale Supérieure de Paris.

In 2001, he started the "Programming Languages" group in École Normale Supérieure which he led till fall 2006, when he was appointed Senior Research Scientist of CNRS and joined the Institut de Recherche en Informatique Fondamentale (IRIF: Research Institute on the Foundations of Computer Science) of Université Paris Diderot. In 2011, he was peer-elected a member of the Academia Europæa. He is, since January 2023, the director of IREF, where he served as deputy director from 2018 to 2022.

His main research contributions are in the design and definition of typed programming languages and in the theory of subtyping.

Contact: Giuseppe.Castagna@irif.fr; Web: https://www.irif.fr/~gc/

Thierry Coquand is known for his work in type theory, proof theory, and constructive mathematics. His work on type theory has had a strong influence in the design of some interactive proof assistants, such as Agda, Coq, and Lean.

He was awarded the Kurt Gödel Society Centenary Research Prize, Senior Category, in 2008 for his work in proof theory and constructive mathematics, and he jointly got the ACM Software System Award in 2013 for his work in type theory.

Together with Vladimir Voevodsky (IAS Princeton) and Steve Awodey (CMU), he organized the special year 2012–2013 at the Institute for Advanced Study, Princeton, on the Univalent Foundations of Mathematics. Helped by a group of PhD students and postdocs, he was able to find the first computational interpretation of the axiom of univalence.

Contact: coquand@chalmers.se; Web: https://www.cse.chalmers.se/~coquand/

Patrick Cousot is the inventor, with Radhia Cousot, of abstract interpretation. He received the Doctor of Engineering degree in Computer Science and the Doctor ès Sciences degree in Mathematics from Université Grenoble-Alpes.

He is Silver Professor of Computer Science at the Courant Institute of Mathematical Sciences, New York University, USA. Previously, he was Professor at the École Normale Supérieure in Paris, the École Polytechnique, and the University of Lorraine and Research Scientist of CNRS at the Université Grenoble-Alpes.

He was awarded the Silver Medal of the CNRS (1999), the Grand Prix of Computer Science and Its Applications of the Airbus Foundation Group awarded by the French Academy of Sciences (2006), a Humboldt Research Award (2008), and the EATCS (European Association for Theoretical Computer Science) Distinguished Achievements Award (2022). With Radhia Cousot, he received the ACM-SIGPLAN Programming Languages Achievement Award (2013), the IEEE Harlan D. Mills Joint Award (2014), and the IEEE John Von Neumann Medal (2018). He received honorary doctorates from the Universität des Saarlandes (2001) and the Ca' Foscari University of Venice (2022). He is a fellow of the ACM and a member of the Academia Europaæ.

Contact: cousot@gmail.com; Web: https://cs.nyu.edu/~pcousot/

Joelle Coutaz is Honorary Professor at Université Grenoble-Alpes and Founder of the Engineering Human-Computer Interaction group within the Grenoble Informatics Laboratory (LIG). Her research interests include human-computer interaction, multimodal and tangible interaction, user interface plasticity, and end-user development for smart homes and ubiquitous computing.

Since 1989 and until she retired in 2013, she was continuously involved in European projects and active in the international scientific community. In particular, she has been involved in the ESPRIT-FP3 BRA/LTR project *AMODEUS 1&2*, the first in Europe to truly promote a multidisciplinary approach to HCI. She has served as Vice Chair of the *IFIP Working Group 2.7(13.4)* on "User Interface Engineering." She has been a member of the editorial board of *Interacting with Computer* (Oxford Academic) and of the *ACM Transactions on Computer-Human Interaction* (*TOCHI*). She has coordinated a working group on ambient intelligence for the French Ministry of Research to create a new trans-disciplinary field that brings together information and communications technologies and social and human sciences to address societal challenges in novel ways. From 2012 to 2020, she co-directed the *Amiqual4Home* innovation platform in the field of ambient intelligence funded by the EquipEx program of the "Programme d'Investissement d'Avenir" (investing in the future program) in collaboration with the *Inria center at the Université Grenoble Alpes*.

In 2013, she was named a Chevalier of the Légion d'Honneur (order of the Legion of Honor) for her pioneering contributions to human-computer interaction and has received several awards, including an honorary doctorate from the University of Glasgow, membership in the ACM SIGCHI Academy, as well as the titles of IFIP TC13 Pioneer and IFIP fellow for her substantial contributions and impact on the field of human-computer interaction. She is an honorary member of the Société Informatique de France.

Contact: joelle@crowley-coutaz.fr; Web: http://crowley-coutaz.fr/coutaz/joelle.html

Pierre-Louis Curien is an emeritus CNRS researcher at IRIF (Institut de Recherche en Informatique Fondamentale), a joint laboratory of CNRS and UPC (Université Paris Cité).

His research interests revolve around the theory (or semantics) of programming languages. His main works concern sequentiality, functional programming (design of the categorical abstract machine, which gave its name to the language OCaml), proof theory and type theory, and more recently homotopical algebra.

He received his PhD thesis in 1979 (under the guidance of Gérard Berry and the distant supervision of Maurice Nivat) and his Doctorat d'Etat in 1983 (both at Université Paris 7, ancestor of UPC). He was Cofounder and Director (1999–2009) of the Laboratory PPS (Preuves, Programmes et Systèmes—Proofs, Programs and Systems—now part of IRIF). He was Cofounder and Leader (2009–2019) of the Inria joint team **π**r2. He acted as Deputy Director of the FSMP (Fondation Sciences Mathématiques de Paris—Paris Mathematical Sciences Foundation) and of the federative structures that preceded its creation (2002–2011).

He is the editor-in-chief of the journal *Mathematical Structures in Computer Science* (since 2016).

He is the author of two influential books: *Categorical Combinators, Sequential Algorithms, and Functional Programming*, Pitman (1986), and *Domains and Lambda-Calculi*, Cambridge University Press (1998, with Roberto Amadio). He has supervised 20 PhD students. He won the 2020 Inria Grand Prize.

Contact: curien@irif.fr; Web: https://www.irif.fr/~curien/

Marie-Claude Gaudel's research interests are in the areas of software: formal methods, program robustness, testing, and certification.

Now retired, she was a professor at the University of Paris-Sud at Orsay (UPS), where for several years she was the director of the LRI (Computer Science Research Laboratory). Before joining UPS, she was a researcher at Inria and then in charge of the Software Engineering group at the industrial research center of Alcatel-Alsthom in Marcoussis.

She has been a member of the editorial boards of *Science of Computer Programming* and *Formal Aspects of Computing* since the founding of both journals.

She received an Outstanding Paper Award of the IEEE Chapter of Software Engineering for the results of her group on program robustness in Marcoussis, and she was awarded the CNRS Silver Medal for her pioneering work on software testing based on formal specifications. She is a Doctor Honoris Causa of EPFL (École Polytechnique Fédérale de Lausanne) and of the University of York (UK). She has been an invited professor and then "Pesquisador Visitante Especial" at the Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo. Currently, she is an Honorary Visiting Professor at the University of York. She is an active member of the French association "Femmes & Sciences" (women and sciences). She is a Chevalier of the Légion d'Honneur (order of the Legion of Honor) and an honorary member of the Société Informatique de France.

Contact: marieclaude.gaudel@gmail.com; Web: https://www.lri.fr/membre.php?mb=278

Jean-Marc Jézéquel is a Professor at the University of Rennes and a member of the DiverSE team at IRISA/Inria. Since 2021, he is Vice President of Informatics Europe. From 2012 to 2020, he was Director of IRISA.

His interests include model-driven software engineering for software product lines and specifically component-based, dynamically adaptable systems with quality of service constraints, including security, reliability, performance, timeliness, etc. He is the author of 4 books and of more than 300 publications in international journals and conferences. He was a member of the steering committees of the AOSD and MODELS conference series. He is currently Associate Editor-in-Chief of IEEE Computer Society and of the journal *Software and Systems Modeling*, as well as Member of the editorial boards of the *Journal of Systems and Software* and *the Journal of Object Technology*. He received an engineering degree from Telecom Bretagne in 1986 and a PhD degree in Computer Science from the University of Rennes in 1989.

In 2016, he received the Silver Medal from CNRS and in 2020 the IEEE/ACM MODELS career award. He was an invited professor at McGill University in 2022.

Contact: jean-marc.jezequel@irisa.fr; Web: https://people.irisa.fr/Jean-Marc.Jezequel/

Jean-Pierre Jouannaud graduated from École Polytechnique and obtained his doctorate from Université Paris 6 in 1978. He was then a professor successively at the universities of Nancy, Paris-Sud, and École Polytechnique, as well as an invited professor at SRI International and Stanford University, California (2 years); Universitat Politècnica de Catalunya, Spain (1 year); Keio University, Tokyo, Japan (6 months); National Taiwan University, Taipei, Taiwan (4 months); and Tsinghua University in Beijing, China (5 years). He is now Professor Emeritus at Université Paris-Saclay, Laboratoire de Méthodes Formelles (Formal Methods Laboratory).

Since the early 1980s, his research interests have been focusing on the interplay between deduction rules, rewrite rules, decision procedures, programming languages, and type theory; in the development of programming languages, rewriting tools, and proof assistants; and in the application of these tools to the formal proof of systems. He is in particular known as a member of the "OBJ group," a group of four

(Futatsugi, Goguen, Meseguer, Jouannaud) which developed the OBJ2 language in 1984 at SRI, under Joseph Goguen's direction. He has published over 100 theory-oriented papers which together totalize 9700 citations on Google Scholar.

During his career, he had a leading role in the creation of two conferences at the heart to his interests: *Rewriting Techniques and Applications* (RTA, which became FSCD, Formal Structures in Computation and Deduction) and Certified Programs and Proofs (CPP, an ACM conference).

He served on editorial boards and steering/prize/scientific committees including *Information and Computation*, *Journal of Symbolic Computation*, RTA, CSL, LICS, CPP, the EATCS Award, the Gödel Prize (including as chair in 2010), the Kleene Prize (as chair in 2010), the LICS "Test-of-Time Award," the Ackermann Prize, the Max Planck Institute at Saarbrücken, and Academia Sinica at Taipei.

Jean-Pierre Jouannaud received the SRI Award for the International Research Fellow of the Year (1984), the Monpetit Prize from the French Academy of Sciences (1997), and the Prize for International Cooperation between France and Taiwan (2000).

Contact: jeanpierre.jouannaud@gmail.com; Web: http://www.lix.polytechnique.fr/Labo/Jean-Pierre.Jouannaud/

Jean-Jacques Lévy is a senior researcher emeritus at Inria. He graduated from École Polytechnique in 1968 and received a PhD in Computer Science at the University of Paris 7 in 1978. He joined Inria in 1970 and served as a member of the research staff at Digital Equipment (DEC-PRL, 1987–1988), a professor of computer science at École Polytechnique (1992–2006), a director of the new Inria-Microsoft Research Joint Center (2006–2013), and a visiting professor at the Chinese Academy of Sciences (ISCAS, Beijing, 2013–2014).

He headed two Inria research teams (Para, Moscova) and two European projects (Confer-1, Confer-2); he was Scientific Chairman at Inria Rocquencourt (1994–1996) and Vice Chairman of Inria's Evaluation Committee (1997–2000). He supervised 20 PhD theses and was consultant at Xerox-PARC (1984) and DEC-SRC (1989). He received the CNRS Médaille de Bronze (1979).

Jean-Jacques Lévy worked on compilers, lambda calculus, term rewriting systems, CAD for VLSI, system programming, programming languages for distributed applications, and formal proofs of programs. He participated to the debugging of the embedded software for the Ariane 5 rocket (after its explosion), and he headed the on-board code review for the ISS European module Columbus. He is a (co-)author of 50 publications and 1 US patent.

Contact: jean-jacques.levy@inria.fr; Web: http://pauillac.inria.fr/~levy/

Bertrand Meyer's career has been partly in industry and partly in academia. He is currently Professor of Software Engineering and Provost at Constructor Institute in Schaffhausen, Switzerland, and CTO of Eiffel Software, a company he cofounded in Santa Barbara, California. He was previously a professor and chair of the Computer Science department at ETH Zurich. He has held visiting positions at the University of California, Santa Barbara, the University of Technology Sydney, Monash University, Politecnico di Milano, and the University of Toulouse. He holds degrees from École Polytechnique, Stanford (MSc), the Sorbonne, and the Université Henri Poincaré in Nancy (Dr. Sc.).

He has made contributions to a wide range of topics in software engineering, including design and programming methodology, with the introduction of the design by contract approach; object technology, through books and articles and the design of the Eiffel language; formal specification and verification; concurrent programming; computer science and software engineering education; software tools and development environments; software processes and agile methods; and requirements engineering. He has published numerous articles on these topics and supervised 29 PhD theses, including 25 at ETH Zurich from 2003 to 2016.

His 13 books include *Object-Oriented Software Construction* (2 editions, Jolt Award, translated into over 15 languages) and, more recently, *Touch of Class* (an introductory programming textbook borne out of teaching the "CS 1" class at ETH Zurich 13 times in a row); *Agile! The Good, the Hype and the Ugly* (a tutorial and critique of agile methods); and, from 2022, the *Handbook of Requirements and Business Analysis*, all 3 from Springer. In addition to his academic and industry activities, he is active as consultant, in particular for software-related legal issues.

He received the ACM Software System Award, the IEEE Harlan Mills Prize, the Dahl-Nygaard Object Technology Prize (as its first recipient), the Jolt Award, the ACM SIGSOFT Influential Software Engineering Award, and two honorary doctorates. He was also the recipient of an ERC Advanced Investigator Grant. He is an ACM Fellow and an IFIP Fellow and a member of the (French) Academy of Technologies and Academia Europaæ.

Contact: Bertrand.Meyer@inf.ethz.ch; Web: https://se.ethz.ch/~meyer, https://bertrandmeyer.com

Michel Raynal is an Emeritus Professor of Informatics, IRISA, University of Rennes. He is an established authority in the domain of concurrent and distributed algorithms and systems. The author of numerous papers on these topics, Michel Raynal is a senior member of Institut Universitaire de France and a member of Academia Europaæ. He is also Distinguished Chair Professor on Distributed Algorithms at the Polytechnic University (PolyU) of Hong Kong.

He chaired the program committees of the major conferences on distributed computing. He has also written 13 books on fault-tolerant concurrent (shared memory and message-passing) distributed systems, among which the following trilogy published by Springer: *Concurrent Programming: Algorithms, Principles, and Foundations* (515 pages, 2013), *Distributed Algorithms for Message-Passing Systems* (510 pages, 2013), and *Fault-Tolerant Message-Passing Distributed Systems: An Algorithmic Approach* (459 pages, 2018). His last book titled *Concurrent Crash-Prone Shared Memory Systems: A Few Theoretical Notions* (115 pages) has been published in 2022. Michel Raynal is also the Series Editor of the *Synthesis Lectures on Distributed Computing Theory* published by Morgan & Claypool.

Michel Raynal was the recipient of several Best Paper awards of major conferences (including ICDCS 1999, 2000, and 2001, SSS 2009 and 2011, Europar 2010, DISC 2010, and PODC 2014). He was the recipient of the 2015 Innovation in

Distributed Computing Award (also known as SIROCCO Prize), of the 2018 IEEE Outstanding Technical Achievement in Distributed Computing Award, and of an Outstanding Career Award from the French chapter of ACM SIGOPS.

Contact: michel.raynal@irisa.fr; Web: https://team.inria.fr/wide/team/michel-raynal/

Jim Woodcock is a Professor of Software Engineering at the University of York (UK) and an award-winning researcher and teacher. He is the Director of the York Centre for Autonomous Robotics for Laboratory Experiments and a member of the RoboStar research group. He is a Professor of Digital Twins at Aarhus University and Professor of Cyber-Physical Systems and a Distinguished Researcher at Southwest University, Chongqing, China.

He has dedicated his research career to searching for the mathematical principles that are essential to the practice of software engineering. He has over 40 years of experience in formal methods. His research interests are in unifying theories of programming (UTP), robotic digital twins, and industrial applications.

He leads the team developing extensive UTP theories and the Isabelle/UTP theorem prover. He worked on applying the Z notation to the IBM CICS project, helping to gain a Queen's Award for Technological Achievement. He created the theory and practical verification for NatWest Bank's Mondex smart-card system, the first commercial product to achieve ITSEC Level E6 (Common Criteria EAL 7).

For the last decade, he has researched the theory and practice of cyber-physical systems and robotics and recently their probabilistic semantics.

Jim Woodcock is Fellow of the Royal Academy of Engineering and a consulting chartered engineer. He received the Rudolf Christian Karl Diesel Prize for work on railway signaling. He is Editor-in-Chief of the ACM journal *Formal Aspects of Computing* and of the CUP journal *Research Directions: Cyber-Physical Systems*.

Contact: jim.woodcock@york.ac.uk; Web: https://www.cs.york.ac.uk/people/jim

Chapter 1
The French School of Programming: A Personal View

Gérard Berry

Abstract Although France has never been a world leader in the software industry, Computer science research has traditionally been at a worldwide level there, thanks to the Universities, the CNRS and Inria and to some researchers abroad. This book is devoted to important parts of the French field of programming languages, essential since program texts are the only way to drive computers. That research almost always tried to link mathematical rigor with practical concerns—an old French tradition. This has been particularly true for the development and linking of new and clean programming languages and formal verification systems, often created and linked together with the solid base of their mathematical semantics and their theorems. Such formal semantics served and still serve as a consistency guide during the design and implementation development, instead of being only addons by other people after the fact, as too often done with much less efficiency. This introductory chapter surveys the 12 subsequent chapters, each dedicated to a particular technical approach or language and written by their team leaders. It ends by myself telling in a nontechnical way how my own 50-years career dealt with the creation of a few original and mathematically well-studied theoretical frameworks and practical languages. Named Esterel, the last one has led to the creation in 2000 of a successful company that has become a world-leader in the field of certified software for safety-critical reactive systems (it also led to some success in industrial hardware design and verification, a successful application domain unfortunately killed by the 2008 financial crisis). Of course, this is not a single-man story, by far, and I also try to tell the associated social story with some humor because it was a lot of fun for me, and for my groups I hope.

I first thank Bertrand Meyer, who had the idea of this book about some important successes of French research in Computer Science, and more precisely on programming and reasoning about programs and applications: programming languages

G. Berry (✉)
Université de Paris-Sud and CNRS, Paris
e-mail: gerard.berry@college-de-france.fr

B. Meyer (ed.), *The French School of Programming*,
https://doi.org/10.1007/978-3-031-34518-0_1

and their semantics, type systems, program analysis and verification, software architecture and engineering, up to human-computer interaction. France has been at the forefront of research and at the highest level of creativity and relevance in this field, and I had the immense pleasure to be part of the game. Bertrand asked me to write an overview, and I feel honored and happy to do it here.

I will build this long overview in two parts. First, a quick survey of the book authors, chapters, and topics, which are very varied and quite complementary—I read them carefully. Then a bit of personal history, justified by the fact that It has involved many of the chapter authors, especially those with whom I worked for a long time.

Warning I must say I know very well several contributors to the book and their work, especially when I was part of it or directly connected to it. This may show in this overview, although I have no intention to consider other authors as less interesting because I know them less. But I am unable to resist following my heart.

1.1 Parts, Chapters, and Authors

The book is composed of four successive parts: Part I deals with Software Engineering, Part II with Programming language mechanisms and type systems, Part III with Theory, and Part IV with language design and programming methodology.

1.1.1 Part I: Software Engineering

In Chap. 2, **Marie-Claude Gaudel**, Professor Emeritus at Université de Paris-Sud and CNRS, Paris, discusses formal algebraic testing. Testing is of course a primary concern both in Computer Science and in Industry since nasty program errors are so easy to make even for careful scientists and engineers. But, in practice, testing is still often done in too limited and non-systematic ways. The chapter presents a formal framework aimed at making testing scientific and systematic, while keeping it much simpler and lighter than formal verification, the only real alternative that will not easily reach common practice. To make the testing phase implementation-independent, the system under test is supposed to give access to a given set of visible values but not to its internals (black box testing vs. white box or gray box testing). There, the tests are formally defined as sets of ground instances of algebraic equations whose satisfaction can be verified by feeding the program with the corresponding inputs and observing its outputs. The important question is how to design the test suite to apply, either in an exhaustive way if finiteness is possible or by determining a well-chosen subtest of test values if a full test set would be infinite or simply too big to be practical. All these questions and acting steps are rigorously defined in the chapter.

Then, the chapter studies successes in application cases by many researchers. Here, they concern three languages: LOTOS for telecom, with also deadlock detection that is of course essential for such a concurrent formalism; Hoare's CSP, also a concurrent language, assuming divergence freeness, and with a formal completeness result; and Cirrus, a complex specification language for which a test generator has been built using the HOL higher-order logic verification systems. The chapter then cites several successful experiments on real applications: an automatic subway, a nuclear plant safety system, a communication protocol, and a Transit Node that served as a comparison point between formal techniques. It contains no concrete examples but cites many other researchers and provides the user with a large number of useful references.

In Chap. 3, **Michel Raynal**, Honorary Professor at Rennes University, outlines his work on distributed algorithms, i.e., algorithms to coordinate the activities of separate programs run on geographically distinct locations. The problems are not new, but their importance has considerably grown in the last 20 years with the advent of the internet as a huge worldwide network that makes new kinds of applications possible and even mandatory: distributed data bases, cloud computing, blockchains, etc. The required distributed algorithms are more and more numerous and often (but not always) quite small in their description, but subtle and very prone to hardly visible errors that can have large consequences in terms of safety and security: they are employed to handle sensible data or dangerous machines, with computers and computation links that can fail.

In distributed computing, not everything is possible. For instance, a major result shows the impossibility of consensus between distant parties as soon as an arbitrary number of them may fail. This has led researchers to find solutions modulo a variety of limitations, for instance on the number and kinds of failures, as for blockchains techniques that are a way to implement consensus in a practically solid way. Michel Raynal is an expert of such algorithms. In the chapter, he presents a good selection of core distributed algorithms in a very understandable way. His numerous results are characterized by the two qualities he always did put forward, generality and simplicity, He has worked with almost all the field's specialists, including Leslie Lamport, Turing Award 2013, Maurice Herlihy, Rachid Guerraoui (who held the Inria yearly chair at Collège de France in 2018–2019), etc., and has improved many algorithms.

In Chap. 4, **Jean-Marc Jézéquel**, Professor at Rennes University, discusses several levels for the engineering road to large software endeavors, analyzing the pros and cons of the considered methods at each level. He starts with dedicated CASE tools, taking the examples of specialized ones in the telecom industry, namely SDL and LOTOS. The pros are rigor, higher level of abstraction, and existence of code generators and automatic verification systems. The cons are the difficulties of such abstraction levels for standard engineers and the sometimes-insufficient performance of generated codes that may have to be tweaked by hand. Then, Jézéquel studies Model Driven Architectures, as promoted by the Object Management Group (OMG). The pros are the higher abstraction levels and the clear separation between design logic and implementation, the cons the quality of the generated codes and again the often-difficult adoption by the engineers. Jézéquel then studies separations of concerns, with pros the more precise dedicated analyses that reduce confusions provoked by a single global one, but as cons the risk that separate analyses may become hard to reconcile, with the added difficulty to instrument the process. Finally, he analyzes the use of domain-specific languages whose specificity often makes programming simpler and clearer, but with the risk of too many limitations and of difficulties in maintaining the compilers and debugging environments in the long run.

In Chap. 5, **Joëlle Coutaz**, Professor Emeritus at Grenoble University, is a pioneer on HMI in France—HMI meaning Human Machine *Interaction,* not just Human-Machine Interface as many people still wrongly say and think. Interfaces are obviously important as shown by the evolution of computers inputs and outputs from input/outputs on punched cards to the modern finger/screen/sound-based interfaces and soon 3D ones. But an interface is only one way to physically achieve an interaction, while there are many other aspects involved in the interaction seen globally, often related to psychological questions. Personally, I learned a lot by working with Jean-Marie Hullot, later creator of the ergonomic interaction of the main Apple products, and I kept trying to make my colleagues and engineers stop to claim about their realizations "my ergonomics are nice because I like them". Interaction should be user-centric, and you absolutely need to check with many people from different horizons, not just yourself! Sadly enough, HMI is still a weakness of many open-source software applications, because it requires specific competencies that are not enough taught and put in practice in too many communities.

Joëlle Coutaz describes and analyzes many aspects of HMI in her chapter, illustrating then with many prototypes. She details Norman's "Seven stages of actions from perception to execution", a fundamental model of interaction involving both physiology and psychology. She then digs into human-centered interaction using a classification into interaction flexibility and interaction robustness, before detailing how to build software architectures and giving many examples of new tangible HMI devices while explaining how they make interaction more tangible, intuitive, and efficient.

1.1.2 Part II: Programming Language Mechanisms and Type Systems

In Chap. 6, **Jean-Pierre Briot**, Research Director at CNRS, discusses a quite recent software engineering question that one can place one step above modularity in programming languages, itself one step above the instruction level: the organization of large-scale and evolutive applications possibly made of heterogeneous components, some of which even dynamically discovered. The vocabulary becomes quite different from the usual one in computing, with often sources in social enterprise organization: multi-agent systems, proactive agents, cognitive agents, capability description languages, structural vs. non-structural, or temporal coupling, etc. For a basic scientist, these terms may look strange, but they do recover a reality in large systems design: for a similar three-level case, think of urbanism vs. building architecture vs. interior architecture. But how can one classify these terms and study their precise interaction? Briot proposes a tentative classification and organization structure based on three orthogonal axes: abstraction, way to select actions from early to late binding, and coupling flexibility of the considered concepts.

Chapter 7 is written by **Pierre-Louis Curien**, Honorary researcher at IRIF CRNS-University Paris Cité and former director of an ancestor of this lab. He was my first PhD student in 1977, when I moved to Sophia-Antipolis. His chapter has two parts: the first one about our common work when preparing his PhD, the second one when he turned to more logical questions, often following Jean-Yves Girard's creation of new logics—a domain where I am incompetent and which I won't comment here. The key question for the first part is called the Full Abstraction problem for PCF (Programming Computable Functions), a simply typed lambda-calculus plus recursion and basic Boolean and arithmetic functions. Plotkin had shown that Scott's model was not fully abstract, and Milner had proved that there exists a unique fully abstract model where all (finite) first-order and higher-order functions are definable by lambda-terms, building that model from syntax. The hot question was to uncover the semantic nature of the fully abstract model. In my thesis, I had made notable progress with my notion of stable functions invented before for other reasons. But we had to deal with sequentiality, a stricter constraint intrinsic to PCF's evaluation mechanism. The key was a nice definition of sequentiality by Kahn and Plotkin, which led us to a model of sequential algorithms, no more based on functions. Thus, we had to abandon the classical view of lambda-calculus models as necessarily functional and replace them by categorical ones, with adequate quotients to recover functions when needed. Our model was not yet fully abstract, but Curien made it so by adding a simple catch statement classical in functional languages. In parallel, Abramsky, Jagadeesan and Malacaria built the fully abstract model using sequential games instead of algorithms, but with a final quotient that loses the semantic intuition. Curien then switched to other problems linked to the various logics later introduced by Girard, see his chapter for details.

Chapter 8 is written by **Thierry Coquand**, Professor at Chalmers University in Göteborg, Sweden. With Gérard Huet, his PhD advisor, he developed the Calculus of Constructions or CoC, a landmark in formal logic that gathers and enriches previous work by the logicians Jean-Yves Girard on the second-order lambda-calculus and by Per Martin-Löf on higher-order intuitionistic logic, together with the fundamental Curry-Howard correspondence between proofs and computations. CoC is the mathematical basis of the Coq verification system, now well-known for its successes in mathematics and program verifications. Coq was used by Gonthier and his teams to formally machine-check important mathematical theorems such as the four-color theorem and the famous Feit-Thompson theorem on odd-order group classification, whose original proof occupies more than 250 pages of heavy mathematics. Coq has made possible the full formal verification of the CompCert industrial-class C compiler by Xavier Leroy and his team, the executable OCaml code of this compiler being automatically extracted from the proof. CompCert is a famous landmark in the field that triggered many other experiments, for instance in the DeepSpec collaborative program in the US. Coquand's paper nicely details the ideas behind CoC and how they were gathered and extended to build this beautiful and efficient theory.

1.1.3 Part III: Theory

Chapter 9 is due to **Patrick Cousot**, Professor Emeritus at Ecole Normale Supérieure Paris. He discusses the theory and practice of Abstract Interpretation, a major automatic program analysis technique he introduced in the 1970s with his late wife Radhia Cousot. It is based on smart finite computations to validate logical assertions attached to program instructions, according to the doctrine systematized in the 1960s by Dijkstra (the idea had been introduced by von Neumann and Goldstine in 1947 and applied by Turing in a beautiful note of 1949 where he showed how to prove correct a machine-language factorial program, despite having no multiplier in the hardware!). Abstract Interpretation performs symbolic execution using sets of values instead of single values at each program position. An automatically driven iterative bottom-up/top-down or chaotic iterations progressively refine these sets, until they become small enough to make assertions true or provide counterexamples (the process may also fail to conclude for some assertions, and the art is to avoid that case). It subsumes most ad-hoc static analysis techniques developed elsewhere.

The chapter details the evolution of theory and practice of Abstract Interpretation over time. Introduced in the early 1970s, it has been continuously developed by the Cousot's and their successive strong teams, first at Nancy University and then at Ecole Polytechnique and Ecole Normale Supérieure Paris. For practical use, it is well-served by very efficient verification software systems that have become industrial, all initially developed within the teams. The Polyspace older version is still commercialized and improved by Mathworks, with important use in the automotive industry for instance. The newer and much more powerful Astrée software, now commercialized by the AbsInt German company (guess where the name comes from), has an important use in the critical software industry. For example, Airbus uses it to prove the absence of run-time errors or other safety properties in actual flight-control software, and it is used for other large and critical industrial applications.

Chapter 10 is due to **Jean-Jacques Lévy**, with whom I worked at IRIA in the 1970s. He is Honorary Director of Research at Inria and a grandmaster in lambda-calculus. This calculus was created in 1926 by Alonzo Church to formalize logic and calculability in a simple but precise language, with which he proved the fundamental termination undecidability theorem. It is still the core of most higher-order functional languages such as Scheme, ML, Ocaml, F#, JavaScript, etc. (despite using the LAMBDA identifier for functions, LISP used dynamic binding instead of static binding, which is technically and practically quite different). Its definition fits in a few lines and looks simple at first glance, but its consequences are not, and each syntactic theorem about it is hard. It was common to build a particular proof for each of them (Church-Rosser, also called confluence, standardization, etc.). Lévy's creation, detailed in Chap. 3, is a term labelling system that completely captures the deep structure of the reduction space of a term, precisely unraveling the reduction and creation of redexes (reducible expressions) by categorizing them into cleverly labelled families along all possible reductions. This makes it possible to relate cousin redexes between different reductions, even if their syntactic form has changed. The main consequence is that an infinite reduction must reduce redexes with infinitely many distinct families, or, differently said, that reductions limited to a finite set of families are necessarily finite. By using a single ordinal instead of ad-hoc ones, this makes it possible to unify the proofs of most syntactic theorems. And most (but not all) typed systems that ensure reduction finiteness can be viewed as morphisms from Lévy's labels.

Chapter 11 is written by **Jean-Pierre Jouannaud**, whom I know from our scholarship at Ecole Polytechnique at the end of the 1960s. He is Professor Emeritus at Université Paris-Sud. His chapter is about the theory of term rewriting systems with pattern-matching, which come in two flavors: first-order, i.e., with function symbols never applied to other non-applied function symbols, or higher-order, where this operation is possible. While confluence is automatic for the lambda-calculus as seen just before, it is not for rewriting systems. Very interesting work has been devoted to this problem, first in the first-order case with the characterization of the key source of non-confluence: well-defined critical pairs of terms of subterms that can be efficiently analyzed to check for confluence. Later was added work for dealing with associativity and commutativity of infix operators, indispensable for many mathematical and programming structures. For higher-order rewriting systems, quite useful for formal mathematics and program verification, the techniques and results become more complex, but they are marvelously explained and illustrated in the text. Jouannaud played a major role in this endeavor.

1.1.4 Part IV: Language Design and Programming Methodology

Giuseppe Castagna discusses set-theoretic typing in Chap. 12. Polymorphic typing and type-checking really started with the Hindley-Longo type system introduced in 1973 for Milner's ML language and later enriched in many modern typed functional languages (Ocaml, Scala, F*, etc.). But there is another major track in the field, the set-theoretic typing analyzed here. It accepts the classical union, intersection, negation, and complementation operators on sets of values, not available with classic polymorphic typing. Such richer types have many advantages, since they may be used to encompass heterogenous sets of values. For instance, they can type heterogeneous lists where each element may be of a different type, say integer, strings, function, etc. Set-theoretic types also accept overloaded operators and functions, which makes it possible to use the same symbol or function name for different sets of argument types; think of $+$ for both integers and floats, disallowed for instance in OCaml. Another major practical advantage is that set-theoretic types can be added to originally untyped languages to make them safer by detecting type-related bugs at compile time instead of runtime. Good examples are TypeScript vs. JavaScript (now the language of the web and currently the second most popular language overall albeit a fragile one) and Typed Racket vs. Racket.

The first part of the chapter, well organized and easy to read, introduces set-theoretic types by many examples that show how to use them in practice and what are their advantages and difficulties. The mostly used language in examples is CDuce of the author's team, but there are also references to other set-theoretic languages. The second part, longer and much more technical but written with great care, analyzes the questions related to type-checking algorithms, not fully solved by now. It also explains the compromises used by existing languages when no perfect solutions exist.

Finally, **Bertrand Meyer**, the editor of this book, wrote the Chap. 13 about the design of the Eiffel object-oriented language, explicitly dedicated to high-safety applications where bugs cannot be tolerated, for instance in banking. He wrote it with a careful and detailed analysis of all the global and local decisions that had to be taken to keep the language solid and consistent. Meyer rightly notices that there are two ways to design a new object-oriented language.

The first and frequent way is to start from an already established base language such as C and to extend it by new constructs such as classes, inheritance, etc.: typical examples are C++ (1982), Objective C (1986), and Java (1996). This apparently natural process however most often led to having to keep some old design decisions that were thought acceptable at the time, for instance because speed and memory size limitations (C was first designed for a PDP-7 computer with 1.75 μs cycle time and 144 KB memory!). But, when programs became larger and more complex, and when bugs really started to matter, several of these decisions became problematic in terms of programming coherence and safety—I can testify, having developed large applications in C++ with my academic and industrial teams, which demanded us a very strong discipline to avoid pitfalls.

The second way, central to Eiffel and also to other language developments (including by myself, see below) is to start afresh with a careful analysis of what should be accepted or rejected, adopting the guideline to encourage to write good programs easy to validate instead of writing them fast and putting them into service after some testing—as still too often done. Note that the Synopsys company has evaluated the cost of poor-quality software in the USA to be $2.41 trillion in 2022![1]

[1] https://www.synopsys.com/software-integrity/resources/analyst-reports/cost-poor-quality-software.html?intcmp=sig-blog-cisq22

Such a process requires introducing explicit constructs that stress without ambiguity what is exactly meant instead of letting the task to dangerously implicit rules.

The chapter analyses in details the main Eiffel decisions and their reasons: uniform access to object properties; structured programming; the design of classes and multiple inheritance between them; explicit ways to solve conflicts by name redefinitions; design by contracts, one of the main components of Eiffel's programming safety, much richer and actually simpler that classical spread-out local assertions; the explicit handling of exceptions; elaborate mechanisms for thread concurrency; removing null pointer dereferencing, etc. Comparisons with other object-oriented languages are given on the fly. There is yet no real comparison with functional object-enriched languages such as OCaml (1996) and Scala (2004) that rely on quite different viewpoints and techniques to achieve the same goals. But this would require another full article, yet to be written!

1.2 A Bit of Personal History

I have been an active actor of several of the fields discussed in this book and I know well many of the Chapter's authors. From 1972 to 1977, I worked with Jean-Jacques Lévy and others on the syntactic aspects and semantic models of the lambda-calculus and of the recursive algebraic definitions introduced by Kleene in the 1950s. I introduced the theory of stable functions, a refinement of Scott's continuous functions that leads to new and more precise models of sequential programming languages, including the lambda-calculus based ones. Stable function then became unexpectedly important in formal logic after their independent reinvention by Jean-Yves Girard. In 1977, having just moved to Ecole des Mines Sophia-Antipolis on the French Riviera (= windsurfing + mountaineering), I hired Pierre-Louis Curien as my first PhD student to work on the full abstraction problem for sequential languages raised by Milner and Plotkin, which was considered as one of the most important open one in semantics; see Curien's Chap. 7 for the results we got.

From 1983 to 2001, when we could at last put our hands on real computers supporting Unix as well as on Z80-like embedded microprocessors, it was time for me to go for more applied work. I started working on real-time process control in a direct collaboration with control-theory and concurrency researchers, this within a new joint project between Ecole des Mines and the just created Inria Sophia-Antipolis lab. We invented a new notion of synchronous concurrency that reconciled concurrency and determinism, the latter being absolutely required in that field, and we made this notion alive in a concurrent, deterministic and control-flow oriented language called Esterel for which we developed an original semantics based on Plotkin's brand-new SOS semantics style. The key idea of Esterel was simple but iconoclastic: to avoid being bothered by the potentially complex interaction between environment time and program execution time, simply assume that your programs run on an infinitely fast processor! The same synchrony idea was independently invented in a different form by two French groups that also gathered control