Texts in Theoretical Computer Science.
An EATCS Series

Markus Roggenbach · Antonio Cerone ·
Bernd-Holger Schlingloff ·
Gerardo Schneider · Siraj Ahmed Shaikh

Formal Methods for Software Engineering

Languages, Methods, Application Domains

Springer

Texts in Theoretical Computer Science. An EATCS Series

Markus Roggenbach · Antonio Cerone ·
Bernd-Holger Schlingloff · Gerardo Schneider ·
Siraj Ahmed Shaikh

Formal Methods for Software Engineering

Languages, Methods, Application Domains

With a foreword by *Manfred Broy*
and a contribution on the origins and
development of Formal Methods by *John V. Tucker*

Springer

Markus Roggenbach
Department of Computer Science
Swansea University
Swansea, UK

Antonio Cerone
Department of Computer Science
Nazarbayev University
Astana, Kazakhstan

Bernd-Holger Schlingloff
Institut für Informatik
Humboldt-Universität zu Berlin
Berlin, Germany

Gerardo Schneider
Department of Computer Science
and Engineering
University of Gothenburg
Gothenburg, Sweden

Siraj Ahmed Shaikh
Institute for Future Transport and Cities
Coventry University
Coventry, UK

ISSN 1862-4499 ISSN 1862-4502 (electronic)
Texts in Theoretical Computer Science. An EATCS Series
ISBN 978-3-030-38799-0 ISBN 978-3-030-38800-3 (eBook)
https://doi.org/10.1007/978-3-030-38800-3

The title page shows detail from the cover of the book “Rechnung auff der Linihen und Federn” by Adam Ries, Sachsse, Erffurdt, 1529.[1] It depicts the following scene: “A table-abacus competes against longhand calculations using ‘Arabic’ numerals, which were still new in Europe. Either one could trounce calculating by hand with Roman numerals—but which was faster? Are the coins a wager on the outcome?”[2]
In his book Ries describes two practices: working with the calculation board (established practice) and numerical calculations with digits (new practice). Historically, as we all know, the new practice as the superior one took over.
In the same sense, the authors of this book hope that their advocated approach of utilising Formal Methods in software engineering will prove to be of advantage and become the new standard.

[1] Cover of the book “Rechnung auff der Linihen und Federn” by Adam Ries, Sachsse, Erffurdt, 1529. Digitized by SLUB Dresden.
Link to the image: http://digital.slub-dresden.de/id267529368/9.
Link to the rights notice: https://creativecommons.org/publicdomain/mark/1.0/.

[2] See: https://www.computerhistory.org/revolution/calculators/1/38/139.

Foreword by Manfred Broy

The development of programs and software engineering are fascinating technical challenges. If software runs on a piece of hardware, if the hardware is embedded into a cyber-physical system, and as soon as the system is started, a process is initiated and the system shows some behaviour and—if designed and programmed in a careful way—it performs a certain task and it generates a behaviour which fulfills specific expectations.

As we have painfully experienced, software systems show a complexity, especially if they are large and used in complicated applications that are often beyond the imagination of engineers. As a result, we all have learned to work and live with imperfect software systems that often show behaviours and properties which are different from what we expect and—in the worst case—do not perform the task software was written for. This is unacceptable—not only in safety critical applications.

As a result, there is a lot of research to find better ways to engineer software systems such that they become reliable and show high quality. High quality means that they provide adequate user interfaces, guarantee the expected functionality, or, even more, over-fulfill the purposes they are built for and that they behave never in an incorrect way. For the engineering of such systems, a large number of proposals have been published and also experimented with, in practice. Some of them being quite useful and successful, others did not deliver what they promised.

An important observation is that computer programs and software in general are formal objects. They are written in a formal language, they are executed on a machine with a formal operational semantics, and each statement of the programming language results in precisely defined behaviours of the machine (state changes, input, and output) exactly determined by the software. In the end, strictly speaking, software is just a huge formula—however, usually not written in the classical style of mathematical formulas, but in the style of algorithmic languages. But, after all, it is a formal object. This means that we are and should be able to provide a kind of a formal theory that describes the elements of the programming languages and the behaviour of programs that is expressed and generated by these elements.

This underlines that there is a difference between writing a text in a natural language and writing a program. Soon, we have learned that writing a program is error-prone. Too many things have to be kept in mind and thought of when writing a line of program text such that it is very likely that what we are writing is sometimes not what we want.

Here formal theories can help a lot to give a precise meaning and some deep understanding, not only for programs and the behaviours they generate, but also for specifications which formally describe certain aspects of program behaviour. The main contribution of formalization is precision, abstraction, and helpful redundancy. Redundancy means that we work out different—if possible formal—more adequate formulations of specific aspects that support the concentration onto specific properties. This way, relevant aspects are represented in isolation to be able to study them independently which may reduce complexity. This has led to a number of formal theories addressing quite different aspects of programs including their functional behaviour, quality issues, and questions of robustness.

This shows that theories providing formal foundations for formalisms, languages, and also for methods in software construction are indispensable artifacts to support software development.

In the academic community, having all this in mind, soon the term "Formal Methods" has been become popular. This term is carefully defined and explained in this book. It is illustrated both by examples and use cases as well as by careful discussion and proper definitions.

For Formal Methods, the challenge is to keep the balance between being formal and providing methods. In this book, numerous examples are given for such a line of attack, but we have to always keep in mind that it is dangerous to define a formalism and to believe that this formalism is already a development method. However, here is another challenge: in the details of the definitions of formalisms, we have to decide about concepts that are critical and difficult. A simple example would be the use of partial functions in specifications: as long as all functions are total, expressions written with these functions have well-defined values. For partial functions, it gets much trickier: what is the value of an expression when certain subexpressions are built of partial functions which happen not to provide a result for this particular application? What is the value of the overall expression then? Is it always undefined? What are the rules to deal with this and to reason about it? Of course, there are many different ways to provide a theory for expressions with partial functions, but obviously not all of them are equally well-behaving and well-suited for engineering. Therefore, when defining formal theories, a rich number of delicate questions have to be solved—many of them related to the goal to use the formalism as an element of a Formal Method.

Another example is how to represent concurrency. Concurrency is a fact of everyday life. We are living in a concurrent world. Our software systems are connected and run concurrently. There are a number of constructs that have been invented to describe concurrent structures and concurrent processes of software systems by formal theories, and again there are challenges—first of all, to come up with a sound theory and a formal model and second to deal with the question whether the theory is exactly addressing the structures and behaviours which are typical for practical systems on one side and are easy to deal with on the other side.

Therefore, it is a very valuable contribution of this book to present an interesting selection of formal theories and to explain how they can be used in the context of methods for software engineering. Certainly, this is a book highly relevant for people interested in formal theories for software engineering usable as elements of methods. It also addresses students in informatics who want to learn about this subject and, even more, scientists who work on formal theories and methods.

I hope this book will also find interest by practical engineers to give them some clue how formal foundations and rigorous methods could be combined to formal methods to help them in their everyday development tasks.

July 2021 Manfred Broy

Preface

De Omnibus Dubitandum—"Doubt Everything"

R. Descartes

This book is about Formal Methods in software engineering. Although software engineering is nowadays a largely empirical science, its foundations rely on mathematics and logic. Ultimately, the task of a software engineer is to transform ideas into programs. Ideas are by nature informal, and they are often vague and subjective. In contrast, a program is a formal entity with a precise meaning, and this meaning is independent of the programmer. Therefore, the transition from ideas to programs necessarily involves a formalisation at some point. An early formalisation has several benefits:

- It allows to formulate concepts on an abstract level;
- it is a means for unambiguous communication of ideas;
- it helps to resolve misunderstandings, thus preventing errors at a later stage; and
- it enables to gain insights by transformation, simulation, and proof.

Formal Methods are a way to realize these advantages in a rigorous process.

This book elaborates on several views of how to do this. In Chap. 1, we approach a definition of what actually constitutes a Formal Method. The rest of the book is structured into three parts: languages, methods, and application domains. These parts represent different dimensions of the views:

1. A *language* is a means to formally describe ideas;
2. a *method* is a set of procedures for manipulating such descriptions; and
3. an *application domain* represents a concrete way in which real-life problems drive the different views.

Each part consists of several chapters which are more or less independent.

In the *languages part*, we present "classical" views on elements of computation.

Chapter 2: Logics are formal languages to describe reasoning.
Chapter 3: The process algebra Csp is a formal language to describe behaviours.

In the *methods part*, we discuss a variety of procedures.

Chapter 4: Casl is a computer supported method for the specification of software, which is based on classical logic as discussed in the language part.
Chapter 5: Specification-based testing is a computer supported method for the validation of software.

Finally, the *application part* provides three contributions to apply Formal Methods to real-world problems.

Chapter 6: In the chapter on specification and verification of normative documents, we discuss a way to reason about legal contracts with logic.
Chapter 7: In the chapter on Formal Methods for human-computer interaction, we discuss how to capture cognitive theories with logic and CSP.
Chapter 8: In the chapter on formal verification of security protocols, we discuss how to verify authentication properties with CSP.

These three chapters have in common that they present solutions to general challenges in software engineering. These solutions are based on the application of one specific Formal Method. It should be noted, though, that other Formal Methods would be applicable to these challenges as well.

We conclude our book by providing a historical perspective on Formal Methods for software engineering:

Chapter 9: In the chapter on the history of Formal Methods, John V. Tucker surveys some of the problems and solution methods that have shaped and become the theoretical understanding and practical capability for making software.

This is followed by some summarizing and reflecting remarks from the book authors.

Audience, Prerequisites, and Chapter Dependencies

This book addresses final year B.Sc. students, M.Sc. students, and Ph.D. students in the early phases of their research. It is mainly intended as a underlying textbook for a university course. Formal Methods are one means in software engineering that can help ensure that a computer system meets its requirements. They can make descriptions precise and offer different possibilities for analysis. This improves software development processes, leading to better, more cost-effective, and less-error-prone systems.

Due to their ubiquity, software failures are overlooked by society as they tend to result in nothing more serious than delays and frustrations. We accept it as mere inconvenience when a software failure results in a delayed train or an out-of-order cash machine or a need to repeatedly enter details into a website. However, the problems of systems failures become more serious (costly, invasive, and even deadly) as automatic control systems find their way into virtually every aspect of our daily lives. This increasing reliance on computer systems makes it essential to develop and maintain software in which the possibility and probability of hazardous errors are minimised. Formal Methods offer cost-efficient means to achieve a high degree of software quality.

However, in computer science and software engineering education, Formal Methods usually play a minor role only.[3] Often, this is due to the lack of suitable textbooks. Typical questions an academic teacher faces when preparing such a course include the following: Which of the many Formal Methods shall be taught? Will the topics be relevant to mainstream students? Which examples and case studies should be used? This book offers constructive answers to such questions. It does not focus on one specific Formal Method, but rather provides a wider selection of them. For each method, material from basic to a more advanced level is presented. Thus, the teacher can choose to what depth a specific method shall be studied. All material is illustrated by examples accessible to the target audience.

Moreover, for individual students, this book can serve as a starting point for their own scientific work, e.g., in a thesis. Even if the reader does not plan to work directly in one of the addressed areas, the book offers solid background knowledge of Formal Methods as a whole.

We assume some basic knowledge of mathematical notation as taught in the first two years of typical B.Sc. curricula in computer science or software engineering. However, we will introduce all formal concepts from scratch, whenever they are used. For the casual reader, the book contains an index, where one can look up the defining page for each technical term.

The material in Part I is foundational for the subsequent parts, whereas the chapters in Parts II and III can be read in any order. General dependencies are depicted in Fig. 1.

More specifically, dependency on the introduction is only from a motivational, but not from a technical point of view. In reading the book, Part I can serve as a "reference", Part II and Part III depend on Part I only in some technical aspects. The reader interested just in specific topics of these parts can safely start there and refer to Part I only when needed.

Although the linear order of reading the chapters would be preferred, for readers who want to focus on specific aspects, the authors suggest two possible alternative paths through the book. Chapter 1 provides a common start to both.

The first path is for those who wish to stay with logic: Chapter 2 leads on to Chap. 4 to provide a grounding in logic and the use in algebraic specifications. Chapter 6 follows on as an area of application for modal logics.

[3] See, e.g., Cerone et al., *Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering*, 2020, https://arxiv.org/abs/2010.05708.

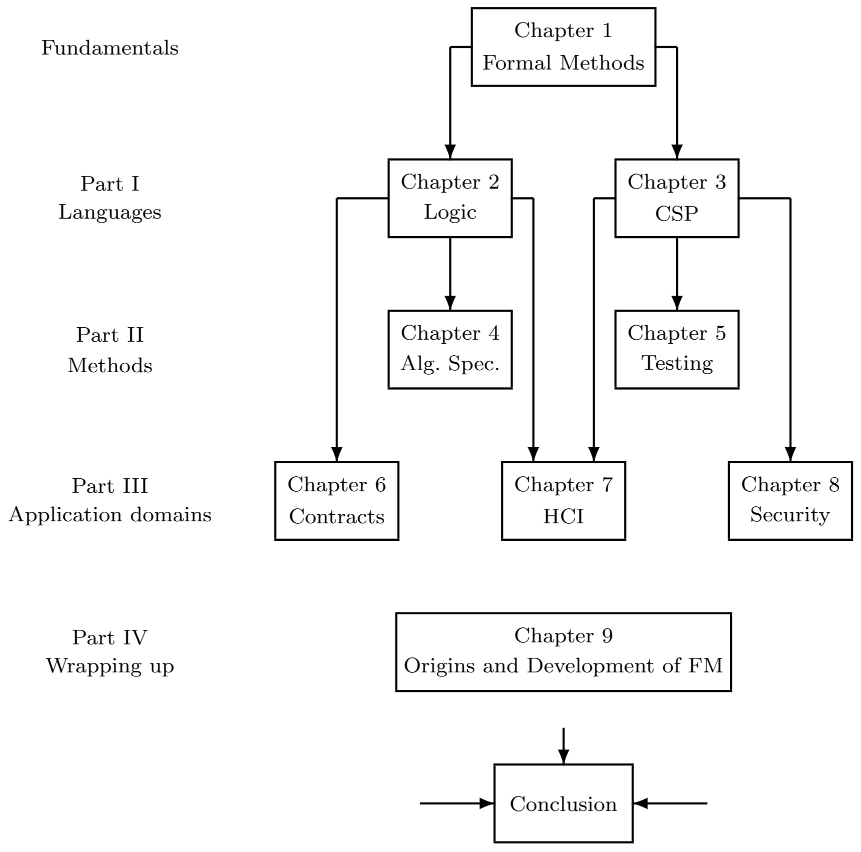

Fig. 1 Structure of the book

An alternative path starts with Chap. 3, thoroughly covering CSP both in theory and practice. Chapter 5 offers a formal perspective on testing. Chapters 7 and 8 provide case studies both using CSP to demonstrate how the process algebra is applied. Only the last part of Chap. 7 depends on logic, limited to temporal logic.

Chapter 9, written by our colleague John V. Tucker, puts the contents of the previous chapters into the historical context. It can be read at any time and it is independent of any of the other chapters.

The conclusion serves to summarise and remind the reader of the final message of the book. It is the natural ending to any reading path.

Book Use and Online Supporting Materials

This is not a typical software engineering book. Nor is it promoting a particular formal approach as many books on the subject do. Formal methods are increasingly *acknowledged* amongst the wider software community. However, there is no evidence to suggest that they are widely *adopted*. It is this gap that this book is designed to address. The use of tools is emphasised and supported; the expectation is that certain parts are to be *done* rather than just *read*. Therefore, the authors have set up a website for the book which contains exercises and links to tools. Currently, this website can be accessed at

https://sefm-book.github.io.

Book History

The inception of this book is due to the first International School on Software Engineering and Formal Methods held in Cape Town, South Africa, from late October to early November of 2008, organised by Antonio on behalf of the United Nations University International Institute for Software Engineering (UNU-IIST), which was located in Macau, SAR China. The two-week school consisted of five courses on the application of Formal Methods to software design and verification delivered to an audience of graduate and research tudents from several African countries, who were hosted by UNU-IIST. In line with the UNU-IIST mandate, the authors were encouraged to find young minds taking up the challenge of Formal Methods and demonstrating commitment to it. The book draws upon the topics of the school with a similar audience in mind and a strong desire to make the subject more widely accessible. Hence learning is promoted through examples running across the book. The pedagogic style is largely owed to the instructional setting offered by the school.

Two more schools followed, in Hanoi, Vietnam, in November 2009, and in Thessaloniki, Greece, in September 2012, are also hosted by UNU-IIST. These events provided additional opportunities for feedback and reflection from school participants. UNU-IIST hosted Markus for one week in 2009. During that meeting, Antonio and Markus sketched the first structure of the book. UNU-IIST organised a one-week workshop in August 2012 in Mezzana (Val di Sole), Italy. During this workshop, the authors decided the final structure and content of the book. After the closing of UNU-IIST in 2013, the authors continued the collaboration through regular virtual meetings and some physical meetings in Coventry and Swansea, UK. Since January 2020, Antonio Cerone, School of Engineering and Digital Sciences, Nazarbayev University, Nur-Sultan, Kazakhstan, has been partly funded to work on the book by the Project SEDS2020004 “Analysis of cognitive properties of interactive systems using model checking”, Nazarbayev University, Kazakhstan (Award number: 240919FD3916).

During the years since the Mezzana workshop, the book content has been updated and widely tested in undergraduate and postgraduate courses by the authors and a number of their colleagues at various universities around the world. The intense cycle of collaborative writing, internal reviewing, and in-class testing was followed by an external reviewing process, in which the reviewers offered their reflections on individual chapters and then incorporated in the final revision by the authors.

Author Team

The book's content, organisation, and writing style were curated by the five book authors. The author team reached out to John V. Tucker, who kindly accepted our invitation to contribute a chapter on the origins and development of Formal Methods. For the writing of some individual chapters, the author team invited Liam O'Reilly for the chapter on algebraic specification in CASL and Hoang Nga Nguyen for the chapter on formal verification of security protocols. The book authors are grateful for their contributions, which made it possible for the book to appear in its current form.

Swansea, UK	Markus Roggenbach
Nur-Sultan, Kazakhstan	Antonio Cerone
Berlin, Germany	Bernd-Holger Schlingloff
Gothenburg, Sweden	Gerardo Schneider
Coventry, UK	Siraj Ahmed Shaikh
September 2021	

Acknowledgments

The authors would like to express their gratitude to a number of people. The following colleagues, listed in alphabetical order, were so kind to contribute in different ways, including reading (parts of) the book and commenting on it: Ulrich Berger, Mihai Codescu, Alan Dix, Stephen Fenech, Marie-Claude Gaudel, Michael Harrison, Magne Haveraaen, Yoshinao Isobe, Alexander Knapp, Ranko Lazić, Antónia Lopes, Faron Moller, Till Mossakowski, Gordon Pace, Jan Peleska, Cristian Prisacariu, Jörn Müller-Quade, Fernando Schapachnik, Steve Schneider, and David Williams. We are grateful for their input and feedback that helped greatly shaping and improving our book. Manfred Broy provided a foreword to our book, which we very much appreciate. We also would like to thank Alfred Hoffman, Ronan Nugent, Wayne Wheeler, and Francesca Ferrari from Springer Verlag for their help, support, and patience.

Swansea, UK — Markus Roggenbach
Nur-Sultan, Kazakhstan — Antonio Cerone
Berlin, Germany — Bernd-Holger Schlingloff
Gothenburg, Gothenburg — Gerardo Schneider
Coventry, UK — Siraj Ahmed Shaikh
September 2021

Contents