Undergraduate Topics in Computer Science

Faron Moller • Georg Struth

Modelling Computing Systems

Mathematics for Computer Science

Springer

Faron Moller
Department of Computer Science
Swansea University
Swansea, UK

Georg Struth
Dept. Computer Science
University of Sheffield
Sheffield, UK

ISSN 1863-7310 Undergraduate Topics in Computer Science
ISBN 978-1-84800-321-7 ISBN 978-1-84800-322-4 (eBook)
DOI 10.1007/978-1-84800-322-4
Springer London Heidelberg New York Dordrecht

Library of Congress Control Number: 2013943907

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Starred sections are optional and often represent advanced material.

List of Figures

Preface

The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.
- Ted Nelson.

Computer Science is a relatively young discipline. University Computer Science Departments are rarely more than a few decades old. They will typically have emerged either from a Mathematics Department or an Engineering Department, and until recently a Computer Science degree was predominantly about writing computer programs (the mathematical software) and building computers (the engineering hardware). Textbooks typically referred to programming as an "art" or a "craft" with little scientific basis compared to traditional engineering subjects, and many computer programmers still like to see themselves as part of a pop culture of geeks and hackers rather than as academically-trained professionals.

However, the nature of Computer Science is changing rapidly, reflecting the increasing ubiquity and importance of its subject matter. In the last decades, computational methods and tools have revolutionised the sciences, engineering and technology. Computational concepts and techniques are starting to influence the way we think, reason and tackle problems; and computing systems have become an integral part of our professional, economic and social lives. The more we depend on these systems – particularly for safety-critical or economically-critical applications – the more we must ensure that they are safe, reliable and well designed, and the less forgiving we can be of failures, delays or inconveniences caused by the notorious "computer glitch."

Unlike traditional engineering disciplines which are solidly rooted on centuries-old mathematical theories, the mathematical foundations underlying Computer Science are younger, and Computer Scientists have yet to agree on how best to approach the fundamental concepts and tasks in the design of computing systems. The Civil Engineer knows exactly how to define and analyse a mathematical model of the components of a bridge design so that it can be relied on not to fall down, and the Aeronautical Engineer knows exactly how to define and analyse a mathematical model of an aeroplane wing for the same purpose. However, Software Engineers have few universally-accepted mathematical modelling tools at their disposal. In the words of the eminent Computer Scientist Alan Kay, "most undergraduate degrees in computer science these days are basically Java vocational training." But computing systems can be at least as complex as bridges or aeroplanes, and a canon of mathematical methods for modelling computing systems is therefore very much needed. "Software's Chronic Crisis" was the title of a popular and widely-cited Scientific American article from 1994, and, unfortunately, its message remains valid today.

University Computer Science Departments face a sociological challenge posed by the fact that computers have become everyday, deceptively easy-to-use objects. A single generation ago, new Computer Science students typically had teenage backgrounds spent writing Basic and/or Assembly Language programs for their early hobbyist computers. A passion for this activity is what drove these students into University Computer Science programmes, and they were not disappointed with the education they received. Their modern-day successors on the other hand – born directly into the heart of the computer era – have grown up with the internet, a billion dollar computer games industry, and mobile phones with more computing power than the space shuttle. They often choose to study Computer Science on the basis of having a passion for using computing devices throughout their everyday lives, for everything from socialising with their friends to downloading the latest films, and they often have less regard than they might to the considerations of what a University Computer Science programme entails, that it is far more than just using computers.

There is a universal trend of large numbers of first-year students transferring out of Computer Science programmes and into related programmes such as Media Studies or Information Studies. This trend, we feel, is often unjustified, and can be reversed by a more considered approach to modelling and the mathematical foundations of system design, one which the students can connect and feel at home with right from the beginning of their University education. This has been our motivation in writing this textbook aimed at teaching first-year undergraduate students the essential mathematics and modelling techniques for computing systems in a novel and relatively lightweight way.

The book is divided into two parts. Part I, subtitled *Mathematics for Computer Science*, introduces concepts from Discrete Mathematics which are in the curriculum of any University Computer Science programme, as well as much which often is not. This material is typically taught in service modules by mathematicians, and new Computer Science students often find it difficult to engage with the material presented in a purely mathematical context. We attempt here to present the material in an engaging and motivating fashion as the basis of computational thinking.

Part II of the book – *Modelling Computing Systems* – develops a particular approach to modelling based on state transition systems. State transition systems have always featured in the Computer Science curriculum, but traditionally (and increasingly historically) only within the study of formal languages. Here we introduce them as general modelling devices, and explore languages and techniques for expressing and reasoning about system specifications and (concurrent) implementations. Although Part I covers twice as many pages as Part II, the title of the book is nonetheless justified: much of the Mathematics presented in Part I itself is used directly for modelling systems, and forms the basis on which the approach developed in Part II is based.

The main benefit of mathematical formalisation is that systems can be modelled and analysed in precise and unambiguous ways; but formal precision can also be a major pitfall in modelling since it can compromise simplicity and intuition. In this book, therefore, we always try to start from intuition and examples, and we aim at developing precise concepts from that basis. How and when to be precise is certainly not less important to learn than precision itself: the ability to give mathematical proofs often does not depend on knowing precise formal definitions and foundations. One can, for example, write down recursive functions without having a precise formal concept in mind.

There is a long standing tradition in disciplines like Physics to teach modelling through little artifacts. The fundamental ideas of computational modelling and thinking as well can better be learned from idealised examples and exercises than from many real world computer applications. This book builds on a large collection of logical puzzles and mathematical games that require no prior knowledge about computers and computing systems; these can be much more fun and sometimes much more challenging than analysing a device driver or a criminal record database. Also, computational modelling and thinking is about much more than just computers!

In fact, games play a far more important role in the book: they provide a novel approach to understanding computer software and systems which is proving to be very successful both in theory and practice. When a computer runs a program, for example, it is in a sense playing a game against the user who is providing the input to the program. The program represents a strategy which the computer is using in this game, and the computer wins the game if it correctly computes the result. In this game, the user is the adversary of the computer and is naturally trying to confound the computer, which itself is attempting to defend its claim that it is computing correctly, that is, that the program it is running is a winning strategy. (In Software Engineering, this game appears in the guise of *testing*.) Similarly, the controller of a software system that interacts with its environment plays a game against the environment: the controller tries to maintain the system's properties, while the environment tries to confound them.

This view suggests an approach to addressing three basic problems in the design of computing systems:

1. ***Specification*** refers to the problem of precisely identifying the task to be solved, as well as what exactly constitutes a solution. This problem corresponds to the problem of defining a winning strategy.
2. ***Implementation*** or ***Synthesis*** refers to the problem of devising a solution to the task which respects the specification. This problem corresponds to the problem of implementing a winning strategy.
3. ***Verification*** refers to the problem of demonstrating that the devised solution does indeed respect the specification. This problem corresponds to the problem of proving that a given strategy is in fact a winning strategy.

This analogy between the fundamental concepts in Software Engineering on the one hand, and games and strategies on the other, provides a mode of computational thinking which comes naturally to the human mind, and can be readily exploited to explain and understand Software Engineering concepts and their applications. It also motivates our thesis that Game Theory provides a paradigm for understanding the nature of computation.

There are over 200 exercises presented throughout each chapter, all of which have complete solutions at the back of the book, as well as over 200 futher exercises at the end of each chapter whose solutions are not provided. The exercises within the chapters are often used to explore subtleties or side-issues, or simply to put lengthy arguments into an appendix, and as such should all be attempted; their solutions at the back of the book should be looked at as well, as they often explain the issues which the exercises are attempting to highlight.

Most of the material in this book has been used successfully for over a decade in first-year Discrete Mathematics and Systems Modelling modules. Countless eyes have passed over the text, and a thousand students have solved its exercises. Nonetheless there will inevitably be a (hopefully small) flurry of errors in the text for which we accept full responsibility and offer our sincere apologies.

Faron Moller
Swansea

Georg Struth
Sheffield