

Preface to Second Edition

Why a New Edition?

This edition has essentially the same content as the first. We have resisted the temptation to ‘soup up’ the content and to deviate from our original aim of providing a *basic* introduction to formally based software construction. Rather than drastically change the content, we have tried to address the educational conflict between providing lots of detail and giving a very general global overview. Many readers feel the need to have much of the work described in great detail, but others find such detail overwhelming and a distraction from the broader picture. However, ‘the devil is in the detail’ — all the details must be correct, otherwise all is lost and we gain nothing.

This edition makes greater use of footnotes to qualify the main text and add detail to the exposition. This is done in an attempt to avoid too many distractions whilst trying to be as technically correct as possible.

So, we keep to basics rather than allow ourselves to be tempted to include more advanced material — even though the topics included may not be those chosen by others as their starting point.

Readers should still read the preface to the first edition — it is all still relevant. If you have not already done so, perhaps now is a good time to read it — before continuing here.

What’s New?

Although the approach is still constructive rather than retrospective (as is the case with testing or even verification), we do allow for inspiration to be employed within our formal framework. This not only encompasses ‘checking’, as in the first edition, but also facilitates the use of ‘eureka’ steps. These effectively start ‘in the middle’ and requires that we work both backwards, to the problem, and forwards, to an implementation. At a first glance, this ‘sideways’ technique looks like more work than simply working in a single direction, however, providing that our inspiration works, this approach is very specific to the given problem and moves quickly to a correct design. But here is not the place to tell the full story.

A proper description of the structure of the book, and how the (rest of the) chapters relate to one another, is given in Chapter 0. However, we ought to say something here about the more important changes, which the reader can appreciate without having to delve into the technical material.

In this edition there are more chapters even though some are very small; there is slightly less detailed working in the text and hence more for the reader to do between (printed) steps. The technical appendices have been extended, and more detailed expositions will be placed on the web (www.springeronline.com/uk/1-85233-820-2) where extra, supplementary, material will be added as and when requested / required / available.

There are many small changes and a few major structural ones. At the risk of glossing over some necessary details we aim to ‘get to the plot’ quickly by including, in Chapter 0, a few brief ‘sketches’ of the entire synthesis process applied to some simple numerical calculations. However, in order that we can do the job properly, this must be counter-balanced by a very large Part A (Preliminaries), which consists of Chapters 1 and 2 and includes details of the mathematical notions with which the reader should be familiar, but perhaps not the notations — well, almost, there are a few novel twists — and matters related to programming.

Chapter 1 is distilled from four original chapters. This can be skimmed at a first reading, so that the reader is acquainted with its contents, and then relevant sections studied more fully as and when needed in the sequel. (The experience of each reader will be different and hence the need for in-depth study will also be different.)

Chapter 2 is the only one which is completely new. It discusses, in outline, how specifications have been used by others in the construction of software. Although this too can be skipped on the first reading, it is included (in this position) so that the reader is aware of some of the aspirations and problems of earlier work on Formal Methods. This is mainly to provide a basis for comparison with our constructive approach, but the techniques can also be used in a mutually supportive fashion.

In programming we certainly support the idea of using sensible (helpful, indicative, ‘meaningful’) names when referring to data items, functions etc. but we have serious doubts about the particularly undisciplined use of comments. The use of comments and assertions and their relationship to program correctness is discussed here and provides more motivation for adopting Formal Methods.

The language Pascal was used to illustrate certain points in the initial chapter of the first edition. For those who are unfamiliar with that language — or perhaps did not

appreciate its relative unimportance in the overall scheme of the book and felt threatened by its unfamiliarity — we have chosen, in this edition, to use a generic block-structured procedural language. Any similarity with a language known to the reader should provide useful reinforcement. But we shall resist the temptation to wander away from our central topic of study and stray into a formal definition of its semantics. Concepts needed within the language will be discussed in appropriate detail when encountered.

We shall use the same language in the programs constructed as part of our ‘methodology’ (although that is not a term whose use we would encourage).

Chapters 1 and 2 can be speed read by an experienced and ‘mathematically aware’ programmer. Chapter 3, in Part B (Fundamentals), is where the real work starts.

Elsewhere, original chapters have been split up so as to permit clearer focussing on important topics that warrant individual discussion and further study. Overall, we aim to compute (correctly derive) programs that perform calculations. Throughout, we have tried to place more emphasis on the relationship between problem breakdown and program assembly. Once mastered, the approach can be applied to ‘larger’ problems using bigger building blocks; it is not only, as often perceived, for ‘programming in the small’.

What this Book is Not About?

This is not a book about Requirements Engineering or about Programming Languages, even though both of these subjects impinge on what we do here. Those topics are closely related to software development and are certainly necessary, but they are not our main concern. And, as already noted, this book is not about ‘Programming’ — coding — or Data Structures.

Although the book has evolved from taught courses (selected chapters), it is not really a textbook *per se*. (Certainly some exercises are included, and there is a lot of detailed working; but this is to reinforce and emphasize the necessity of paying attention to detail, both in the theory and as the basis of mathematically based software engineering tools.) Neither is it a monograph; it is more of an explication — an extended explanation. We shall try to react to questions received and add extra material in response to readers’ needs via the web.

Acknowledgements

The bulk of the material within this book has been distilled from courses presented by the author and his colleagues over a period of some 20 years. During this period of time, researchers and teachers have all been influenced (taught!) by the work of others — sometimes consciously, but often not. Notwithstanding the inevitable omissions from any list, we include, within the bibliography at the end of the book, the more obvious textbooks which have helped form and transform our understanding of Formal Methods. Collectively we share in the ongoing search for better ways of presenting, explaining, and teaching the most recent developments in Formal Methods that have matured sufficiently to warrant wider exposure. (Of course, there are also very many research papers, but to cite any of them would not be appropriate in a basic introduction such as this.)

Regrettably, many of these books are no longer in print, a tragedy of the need for publishers to pander to popular needs for trendy IT books rather than support the Science of Computing or the Engineering of Software. But they all have something to offer, even if you have to borrow a copy from a library.

Interactions, formal and informal, direct and electronic, with colleagues within *BCS-FACS* (the BCS specialist group in Formal Aspects of Computing Science) and fellow members of the editorial board of *Formal Aspects of Computing* are gratefully acknowledged.

Again thanks (and apologies) are due the students who have suffered our attempts to present a topic whilst it was still in its academic infancy. Particular thanks go to my friend and colleague Roger Stone.

The first edition was written largely during a period of study leave from Loughborough University.

I am also indebted to Steve Schuman and Tim Denvir who assisted in honing the first edition, to Mark Withall who reported more errors (typos?) than anyone else, and to Rosie Kemp, Helen Callaghan, Jenny Wolkowicki (and others whose names I do not know) at Springer for their support and patience.

What Next?

At the risk of being unconventional, we mention here some of the more advanced aspects of Formal Methods, which follow on naturally from concepts introduced here and which the reader might pursue next.

The way in which we specify *types* can also be used to derive and present new, composite types and, object-oriented, *classes*. Extra mechanisms need to be introduced so as to facilitate inheritance between hierarchically related classes; but the basic framework for reuse is already in place, and the O-O notion of pattern is merely a generalisation of the tactics introduced here.

Within this text, we meet genuine, non-interfering, parallelism. Other kinds of parallelism are possible and relate naturally to distributed systems that work in a non-deterministic fashion (and can be characterised by non-deterministic eureka rules). Such systems may need to be specified using temporal logic (in which properties change with time). They therefore provide instances of situations where we need to distinguish between (and link) requirements and specifications. And they may well be implemented by multi-processor systems.

As you will see, program transformation plays an important role in our constructions since it allows us to move from recursive functions to iterative statements. But that is all we use it for. When we know more about the target implementation systems (hardware and software) we can study the complexity of the designs we produce and further transform these, correct, programs and systems so as to improve their efficiency.

So there is certainly plenty of scope for development and application of the basic material to be put to use, once it has been fully mastered. Now to work.

John Cooke
Loughborough University
May 2004



<http://www.springer.com/978-1-85233-820-6>

Constructing Correct Software

Cooke, D.J.

2005, XXI, 509 p. 100 illus., Softcover

ISBN: 978-1-85233-820-6