Formal Methods, Informally

Learn to program more effectively, faster, with better results… and enjoy both the learning experience and the benefits it ultimately brings.

While this undergraduate-level textbook is *motivated* by formal methods, so encouraging habits that lead to correct and concise computer programs, its informal presentation sidesteps any rigid reliance on formal logic which programmers are sometimes led to believe is required. Instead, a straightforward and intuitive use of simple "What's true here?" comments encourages precision of thought without prescription of notation.

Drawing on decades of the author's experience in teaching/industry, the text's careful presentation concentrates on key principles of structuring and reasoning about programs, applying them first to small, understandable algorithms. Then students can concentrate on turning those reliably into their corresponding – and correct – program source codes. The text includes over 200 exercises, for many of which full solutions are provided. A set of all solutions is available for instructors' use.

Carroll Morgan has been an innovator, educator and researcher in computer science for his whole career: first in industry, then as Lecturer and Fellow at the University of Oxford, and finally as Professor at the University of New South Wales. He is best known for his pioneering work in systematic- and correctness-oriented methods of writing computer programs and systems, and especially for his text *Programming from Specifications*. He is a member of IFIP Working Groups 1.3, 1.7, 2.1 and 2.3 and received (jointly) the "Best Cybersecurity Paper of the Year" award from the National Security Agency in 2015.

"This accessible and compellingly written book will deepen your understanding of how code works and why it works correctly. It is full of practical insights for both students and experienced programmers, as well as university educators looking for a new – and better – way to teach programming."

Graeme Smith, University of Queensland

"Carroll Morgan's *Formal Methods, Informally* is a timely guide to checking everyday code by asking the right questions. Building on distilled logic and math mechanisms, rigorous thinking is promoted as a most valuable tool for developing verifiable software. This book is an insightful must-read for students, educators and practitioners alike."

Luigia Petre, Åbo Akademi University

Formal Methods, Informally

How to Write Programs That Work

CARROLL MORGAN
University of New South Wales, Sydney

CAMBRIDGE
UNIVERSITY PRESS

Shaftesbury Road, Cambridge CB2 8EA, United Kingdom

One Liberty Plaza, 20th Floor, New York, NY 10006, USA

477 Williamstown Road, Port Melbourne, VIC 3207, Australia

314–321, 3rd Floor, Plot 3, Splendor Forum, Jasola District Centre,
New Delhi – 110025, India

103 Penang Road, #05–06/07, Visioncrest Commercial, Singapore 238467

Cambridge University Press is part of Cambridge University Press & Assessment,
a department of the University of Cambridge.

We share the University's mission to contribute to society through the pursuit of
education, learning and research at the highest international levels of excellence.

www.cambridge.org
Information on this title: www.cambridge.org/highereducation/isbn/9781009420990

DOI: 10.1017/9781009421003

When citing this work, please include a reference to the DOI 10.1017/9781009421003

First published 2026

Cover image: Michael Phillips / Moment Open / Getty Images

A catalogue record for this publication is available from the British Library

A Cataloging-in-Publication data record for this book is available from the Library of Congress

ISBN 978-1-009-42099-0 Hardback
ISBN 978-1-009-42102-7 Paperback

Additional resources for this publication at www.cambridge.org/Morgan.

Contents

Preface

What is this text about, and who is it for?

This text is about how to program more effectively, faster, with better results... and how to enjoy doing so.

And it's intended for two kinds of people: those who have some experience of programming, but are not yet experts; and those who *are* experts, but might like to see a possible explanation of *why* they are. Thus its purpose is to introduce (to the first group) and make explicit (to the second group) some design- and coding techniques that can be used to organise, check and maintain large programs, especially those whose correct functioning is crucial (i.e. they must have as few bugs as possible).

But –in spite of that emphasis on "large"– the examples used in this text will be very small ones, in many cases programs that readers will have written already. In fact the more experienced programmers might have written them dozens of times.

Our use of familiar, simple examples to introduce formal methods *informally* is a deliberate, although possibly unusual, choice; and it is quite different from what is usually done.

THE USUAL APPROACH to teaching programming is to introduce a general-purpose, widely used programming language as a first step, and then to apply it initially to very simple problems. The idea there is to instil familiarity with *some* programming language –it does really not matter so much which one– and then, by choosing more and more ambitious examples, to move the students to a level where they realise that what they are learning has two main components. One of them is "how to find algorithms".

An *algorithm* describes steps that if followed will fulfil a task. Computers are used to follow algorithms' steps so that they can be carried out automatically, quickly and reliably. An algorithm is the "essence" of a computer program.

The other component of this "usual" approach is expressing algorithms (the essence) in computer-program *code* (i.e. in a programming language), so that the computer can carry them out, can "execute" them on its own.

But when programs are large and complex, it is not easy to be sure they will work as desired, or even at all. First, the algorithm itself might be complex, and not well understood. It could even be wrong. And second, even if the algorithm is not complex (but especially if it is), translating it into code can be tricky too, and sometimes very error prone. It's a bit like "typos" in ordinary writing — but actually it's much worse. A typo (think "misspelling") in a computer program is usually picked up automatically: the rules for programs, their "spelling and grammar", are very exact. Computers detect "syntax errors" in programs very easily.

What computers *don't* detect is grammatically correct programs that –actually– don't do what their programmers expect them to. Maybe the algorithm is wrong in the first place; maybe it was coded up incorrectly; maybe both. But –again "usually"– people simply accept all that as just a "fact of life" when writing computer programs.

BUT ANOTHER APPROACH is often taken by people who *don't* accept "all that ... as a fact of life": it's called "formal methods".

Formal methods uses rigorous logic, sometimes in a very sophisticated form, to *prove* mathematically that a program does what it should. Obviously, to prove that, it must be known in the first place what "should" means, that is, what the program is actually supposed to do — and that knowledge must be exact, at least if the proof is to have any rigour at all. So to use formal methods properly, the program's *specification* must be written in mathematical language too, not just the proof. And it must be possible to translate the program's code into a mathematical form as well; and then after that more mathematics must be deployed to connect the (mathematised) program to the (mathematical) specification. So it's "mathematics all the way down".

Formal methods is therefore a very specialised part of computer science, and how it works –and especially *why* it works– are fascinating topics in their own right. This "second, alternative" approach to teaching programming, which we are describing in these paragraphs, presents formal methods first, though perhaps not at its highest intensity, as a sort of "limbering-up exercise" before students are even allowed to attempt and submit answers to real programming exercises, to "touch the computer at all" so to speak. After that, they are allowed to use the just-learned formal methods on real programs, first small ones and then steadily larger... and they proceed further as in the first approach above.

THE PROBLEM WITH *BOTH* OF THOSE TWO APPROACHES...
is that *neither* of them gets us where we want to go: not the programmers, and not the people who depend on the programs they write. And that's why we don't adopt either of them in this text.

The first approach leads to enormous amounts of time wasted, as running programs are tested, debugged, tested again, debugged further... Nights and weekends are lost doing that, and –even then– there are further losses when users suffer because the programs *still* don't work, and their unexpected failures prevent ordinary people from doing their own jobs — or worse.

And the second approach simply puts off many people right from the start. You don't *have* to be a mathematician to be a programmer: "everyone knows that" — and actually they are right. Many students therefore reject the formal-methods-first approach: they develop scepticism towards the whole idea, and we all end up worse off than we would have been if we just went straight into programming, that is, without any "method" at all.

WHAT'S LEFT is what you will find in this text: neither of those two approaches. Here, we use simple programs, programs that people can write already; and at the same time, we use *general* ideas "lifted" from formal methods and made accessible and understandable in an everyday way. This approach really does deserve our title:[1]

Formal Methods, Informally .

"Informal" formal methods shows you how to combine the *ideas* of formal methods, a way of thinking about programs, and the *ideas* built into designing algorithms you want to program, into an approach that

[1] The course this text accompanies is called *(In-)Formal Methods: The Lost Art*. The "Lost Art" is explained in the Afterword.

- Does not require you to learn mathematical logic first, and yet...
- Still reduces the number of mistakes you make when coding.
- Helps you to divide a large programming job into smaller pieces, share them with colleagues, and then put the results back together again.
- Leaves behind a description of the essential ideas that were used during the coding (the "documentation"), so that the program can be easily understood and modified by others, later on.
- Helps you to understand where an error occurred in the design process if, after all, a bug turns up in the running code, so that the programming *process* (e.g. in your organisation, or just your daily life), can be tuned and improved. If a program ends up wrong, you need to fix not only the program *but also the way it was made*. Sometimes that involves "tweaking" the *social* processes involved. Formal methods, even informally, helps you find out which processes need attention.[2]
- Helps you to produce modular components that can be reused in many programs — not just the ones they were originally written for.
- Suggests ways in which the program (components) can be tested, both during coding and after it is complete.

The simple programming examples used in this text –the ones you have probably done already in a more traditional way– can be found listed under PROGRAM EXAMPLES in the index. The programming language used for those examples is `Python`, but the approach can easily be adapted to other, similar languages. (See also App. G.2.) More than 200 drills and exercises are included, targeted not only on writing small programs but also on how to think about the *process* of doing so.

The general algorithmic topics covered include sorting, searching, order of growth (complexity), data structures and encapsulation, loop correctness and –as a more advanced topic– concurrency.

But none of those topics is covered exhaustively: it would be more accurate to say that in *this* text they are only "visited". Our purpose here is to introduce and then encourage further interest in more specialised programming topics and techniques –which readers will later discover to have their own authoritative texts– and *at the same time* to give some idea of how to think effectively about those topics and how to use them. We don't tell the whole story here; rather we help people to find and better understand the more comprehensive stories, told by others, that will deepen and complete their knowledge.

Because of the simple examples chosen, the programs we study here are connected to what people already know: they can concentrate on the new *process* we are following, without having to wonder how the example program actually works. Introducing a new topic *and* how to think about it *and* why it works *and* the really difficult problems that it solves, all at the same time... for a general audience is often a bad idea...

...because most people learn best in layers, each layer building on the one before, each one benefiting from the proper assimilation of earlier material.

[2] An example of the importance of process in general (i.e. not necessarily computer-based), is the *Kansas City walkway collapse* where in 1981 a suspended bridge fell onto some 2,000 guests attending a gala dance-party below. The engineering-design "bug" (by analogy with the computing term), which led to the failure, could have been spotted by a first-year engineering student ... if she had ever been given the chance to look. But the human, i.e. the "social" processes followed by the designers and builders did not expose the design to proper scrutiny by anyone.

So our aim in this text, in this approach, is that when people get a bit further into their programming careers, meet new topics, stretch their capabilities, solve challenging problems, they will say

> *Oh yes! I met that in "Formal Methods, Informally" . . .*
> *and I learned there how to think about problems like this.*

And then they will get off to a running start — in the right direction.

Organisation and use

The material is presented in four major parts, and the first is the most important: "Everyday programs". It could be used on its own for an introductory university course, or for a short "summer school" of students meeting programming for the first time.

The text's more than 200 drills and exercises are cross-referenced to the topics that gave rise to them, and to each other.

The downward-pointing arrows within certain drills and exercises indicate that an answer is given in Appendix H or I, respectively. Marginal indications ("rubrics", like "Ex. 1.1" on p. 3) refer to exercises that are directly relevant to that point in the text: they do not *need* to be followed then and there, but they do offer the opportunity for a closer look at the material they annotate. That is, to follow or not is the reader's choice. In any case, the exercises in turn refer back to the rubric, so it is easy to return to the text after having a look. (A complete set of answers is available, but many of them are reserved as a teaching resource.)

The material is based substantially on an "informal" formal methods course that has been given at The University of New South Wales and at Macquarie University, both in Sydney; and there are mini-project-sized assignments associated with the course (but not given here) that each can take several weeks to complete, depending on how they are presented. It's not hard to fit three of them into, say, a nine-week term.

Acknowledgments

I'm grateful for the support of the School of Computer Science and Engineering at the University of New South Wales, allowing me the flexibility to try out this approach on undergraduate students, over almost ten years so far, and similarly the Trustworthy Systems Group –at the time, part of the CSIRO's Data61 but now fully at UNSW– which has allowed me time to teach the course and to write this text and which provided the right intellectual environment for an experiment of this kind. It provided many of the students who have attended and helped to improve the course.

Indeed *very* many improvements and corrections to the text have been suggested by those using early drafts: the students at UNSW in 2021–3, Annabelle McIver (who has used it for teaching at Macquarie University) and Thomas Kunc, Enzo Lee Solano and Paula Tennent (all three veterans of it, and later invaluable as teaching assistants).

Tom was essential in holding together the delivery and presentation of the course in 2022, especially as we made the post-COVID transition from a small class size to a much larger one: he structured an approach to our (partly) automated assessment of the students' homework. Paula and Enzo were indispensable in 2023 for suggesting further improvements to the course presentation and organisation, and for helping to carry them out.

Many more people have been kind enough to read through evolving versions of this material, offering comments –*and* corrections– concerning technical, historical and grammatical matters. Among them are Cliff Jones, Rustan Leino, Sebastian Sequoia-Grayson, Graeme Smith and Trudy Weibel. Miki Tanaka and Junming Zhao organised an "informal methods" study group at Trustworthy Systems — which was not only fun and rewarding (I hope for *everyone* involved), but also led to significant changes in the way some of the material was explained.

David Tranah, from Cambridge University Press, encouraged vast improvements with his meticulous, tireless reading of every chapter — and his precise and always pertinent observations and suggestions. Very thorough copyediting by Laura Emsden and Susan Parkinson improved the presentation significantly.

Dedication

This presentation of how to *reason* about programs, rather than just writing them down, is dedicated to all the academics and computer-science professionals whose insights into how to employ both simplicity and power, at the most fundamental level, will eventually be recognised by *everyone* as the true determinant of how computers should be designed, programmed and deployed. A small number of them are cited here directly, indicated by bold-faced entries in the index.

And it is dedicated to my family, all my boys and girls.

In memoriam Geoffrey Seward Smith *and* Jean-Raymond Abrial, *mentors both.*