Reasoned Programming

Krysia Broda Susan Eisenbach Hessam Khoshnevisan
Steve Vickers

Contents

Foreword

How do you describe what a computer program does without getting bogged down in how it does it? If the program hasn't been written yet, we can ask the same question using a different tense and slightly different wording: How do you *specify* what a program should do without determining exactly how it should do it? Then we can add the question: When the program is written, how do you judge that it satisfies its specification?

In civil engineering, one can ask a very similar pair of questions: How can you specify what a bridge should do without determining its design? And, when it has been designed, how can you judge whether it does indeed do what it should?

This book is about these questions for software engineering, and its answers can usefully be compared with what happens in civil engineering. First, a specification is a different *kind* of thing from a design; the specification of a bridge may talk about load-bearing capacity, deflection under high winds and resistance of piers to water erosion, while the design talks about quite different things such as structural components and their assembly. For software, too, specifications talk about external matters and programs talk about internal matters.

The second of the two questions is about judging that one thing satisfies another. The main message of the book, and a vitally important one, is that *judgement relies upon understanding.* This is obviously true in the case of the bridge; the judgement that the bridge can bear the specified load rests on structural properties of components, enshrined in engineering principles, which in turn rest upon the science of materials. Thus the judgement rests upon a tower of understanding.

This tower is well-established for the older engineering disciplines; for software engineering, it is still being built. (We may call it 'software science'.) The authors have undertaken to tell students in their first or second year about the tower as it now stands, rather than dictate principles to them. This

is refreshing; in software engineering there has been a tendency to substitute formality for understanding. Since a program is written in a very formal language, and the specification is also often written in formal logical terms, it is natural to emphasize formality in making the judgement that one satisfies the other. But in teaching it is stultifying to formalize before understanding, and software science is no exception — even if the industrial significance of a formal verification is increasingly being recognized.

This book is therefore very approachable. It makes the interplay between specification and programming into a human and flexible one, albeit guided by rigour. After a gentle introduction, it treats three or four good-sized examples, big enough to give confidence that the approach will scale up to industrial software; at the same time, there is a spirit of scientific enquiry. The authors have made the book self-contained by including an introduction to logic written in the same spirit. They have tempered their care for accuracy with a light style of writing and an enthusiasm which I believe will endear the book to students.

Robin Milner
University of Edinburgh
January 1994

Preface

Can we ever be sure that our computer programs will work reliably? One approach to this problem is to attempt a mathematical proof of reliability, and this has led to the idea of Formal Methods: if you have a formal, logical *specification* of the properties meant by 'working reliably', then perhaps you can give a formal mathematical proof that the program (presented as a formal text) satisfies them.

Of course, this is by no means trivial. Before we can even get started on a formal proof we must turn the informal ideas intended by 'working reliably' into a formal specification, and we also need a formal account of what it means to say that a program satisfies a specification (this amounts to a *semantics* of the programming language, an account of the meaning of programs). None the less, Formal Methods are now routinely practised by a number of software producers.

However, a tremendous overhead derives from the stress on *formality*, that is to say, working by the manipulation of symbolic forms. A *formal* mathematical proof is a very different beast from the kind of proof that you will see in mathematical text books. It includes the minutest possible detail, both in proof steps and in background assumptions, and is not for human consumption — sophisticated software support tools are needed to handle it. For this reason, Formal Methods are often considered justifiable only in 'safety critical' systems, for which reliability is an overriding priority.

The aim of this book is to present *informal* formal methods, showing the benefits of the approach even without strict formality: although we use logic as a notation for the specifications, we rely on informal semantics — a programmer's ordinary intuitions about what small, linear stretches of code actually do — and we use proofs to the level of rigour of ordinary mathematics.

This can, of course, serve as a first introduction to strict Formal Methods, but it should really be seen much more broadly. The benefits of Formal

Methods do not accrue just from the formality. The very effort of writing a specification prior to the coding focuses attention on what the user wants to get out of the program, as opposed to what the computer has to do, and the satisfaction proof, even if informal, expresses our idea of how the algorithm works. This does not require support tools, and the method — which amounts really to methodical commenting — is practicable in *all* programming tasks.

Moreover, the logic plays a key role in modularization, because it bundles the code up into small, self-contained chunks, each with its specific task defined by the logic.

Although most of the techniques presented are not new (and can be found, for instance, in the classic texts of Gries and Reynolds), we believe that many aspects of our approach are of some novelty. In particular

Functional programming: Functional programming is presented as a programming language in its own right (and we include a description of the main features of Miranda); but we also use it as a reasoning tool in imperative programming. This is useful because the language of functional programming is very often much clearer and more concise than that of imperative programming (the reason being that functional programs contain less detail about *how* to solve a problem than do imperative programs).

Procedures: It is difficult to give a semantics that covers procedures, and many treatments (though not Reynolds') ignore them. This is reinforced by the standard list of ingredients of structured programming (sequence, decision and iteration), which are indeed all that is structurally necessary; but in fact procedures are the single most effective structure in making large programs tractable to human minds and this is because they are the basic unit of interface between specification and code — both inwards, between the specification and implementing code, and outwards, between the specification and calling code. The role of the logical specification in promoting modularity is crucial, and we have paid unusual attention to showing not only how specifications may be satisfied but also to how they may be used.

Loop invariants: We have tried hard to show loop invariants as an expression of initial intuitions about the computation, rather than as either a *post hoc* justification or as something that appears by magic by playing with the post-condition. Often they arise naturally out of diagrams; nearly always they can function as statements of intent for what sections of the code are to do. We have never shirked the duty of providing them. Experience even with machine code shows that the destinations of jumps are critical places at which comments are vital, and this covers the case of loop invariants.

Real programming languages: We have done our best to address real programming problems by facing up to the complexities of real imperative

languages, saying what can be said rather than restricting ourselves to artificial simplicities. Thus, while pointing out that reasoning is simpler if features such as side-effects are avoided, we have tried to show how the more complicated features might be attacked, at least informally.

The book is divided into two complementary parts, the first on Programming and the second on Logic. Though they are both about logical reasoning, the first half concerns the ideas about programs that the reasoning is intended to capture, while the second half is more about the formal machinery. The distinction is somewhat analogous to that often seen in books about programming languages: a first part is an introduction to programming using the language, and a second part is a formal report on it.

To read our book from scratch, one would most likely read the two parts in parallel, and this is in fact how we teach the material for our main computer science course at Imperial. However, the division into two reasonably disjoint parts means that people who already have some background in logic can see the programming story told without interruption.

The approach to the logic section has been strongly influenced by our experience in teaching the subject as part of a computer science course. We put great stress right from the start on the use of the full predicate logic as a means of expression, and our formal treatment of logical proof is based on natural deduction because it *is* natural — its formal structure does reflect the way informal mathematical reasoning is carried out (in the first part, for instance). We have taken the opportunity to use the two parts to enrich each other, so, for instance, some of the proofs about programs in the first part are presented as illustrations of the box proof techniques of the second part, and many of the logic examples in the second part are programs.

Part I *Programming*

1 Introduction
2 Functions and expressions
3 Specifications
4 Functional programming in Miranda
5 Recursion and induction
6 Lists
7 Types
8 Higher-order functions
9 Specification for Modula-2 programs
10 Loops
11 Binary chop
12 Quick sort
13 Warshall's algorithm
14 Tail recursion

Part II *Logic*

15 An introduction to logic
16 Natural deduction
17 Natural deduction for predicate logic
18 Models

The preceding contents list shows the order in which we cover the material in the first year of our undergraduate computer science course. The gap in Part I, between Chapters 8 and 9, is where we teach Modula-2 as a language. Students who have already been taught an imperative programming language would be able to carry straight on from Chapter 8 to Chapter 9.

There are other courses that could be based on this book. Either part makes a course without the other, and indeed in a different class we successfully teach the Part II material separately with Part I following. For the more mathematically minded who find imperative program reasoning inelegant, Chapters 9 through 14 could be omitted and this would then enable the material to be taught in a single semester course.

Acknowledgements

We would like our first acknowledgement to be to David Turner and Research Associates for the elegant Miranda language and the robust Miranda system. Much of the written material has been handed out as course notes over the years and we thank those students and academic staff who attempted the exercises and read, puzzled over and commented on one section or another.

We would also like to thank Paul Taylor for his box proof macros; Lee McLoughlin for helping with the diagrams; Kevin Twidle for keeping our production system healthy; Mark Ryan for helping to turn Word files into LaTeX; Peter Cutler, Iain Stewart and Ian Moor for designing and testing many of the programs; special thanks must go to Roger Bailey whose Hope course turned into our Miranda lectures.

Lastly, we would like to credit those who have inspired us. Courses evolve rather than emerge complete. *Reasoned Programming* could not have existed in its current form without the ideas of Samson Abramsky and Dov Gabbay, for which we are most grateful.

Krysia Broda,
Susan Eisenbach
Hessam Khoshnevisan,
Steve Vickers
Imperial College,
January 1994

Chapter 1

Introduction

1.1 How do you know a program does what you want it to?

You write a computer program in order to get the computer to do something for you, so it is not difficult to understand that when you have written a program you want to be reasonably confident that it does what you intended. A common approach is simply to run it and see. If it does something unexpected, then you can try to correct the errors. (It is common to call these 'bugs', as though the program had blamelessly caught some disabling infection. Let us instead be ruthlessly frank and call them 'errors', or 'mistakes'.) Unfortunately, as the computer scientist Edgser Dijkstra has pointed out, testing can only establish the *presence* of errors, not the absence, and it is common to regard programs as hopelessly error-prone. It would be easy to say that the answer is simple: Don't write any errors! Get the program right first time! Novice programmers quickly see the fatuity of this, but then fall into the opposite trap of not taking care to keep errors out.

In practical programming there are various techniques designed to combat errors. Some help you to write error-free programs in the first place, while others aim to catch errors early when they are easier to correct. This book explains one particular and fundamental idea: *the better you understand what it is that the program is supposed to do, the easier it is to write it correctly.*

1.2 Why bother?

Here is the warranty on a well-known and perfectly reputable operating system:

> The Supplier makes no warranty or representation, either express or implied, with respect to this software, its quality, performance, merchantability, or fitness for a particular purpose. As a result, this software is sold 'as is'

> and you the purchaser are assuming the entire risk as to its quality and performance.

Fortunately, the programmers engaged to write the software did not treat this legal disclaimer as the definitive statement of what the program was supposed to do. They worked hard and conscientiously to produce a well-thought-out and useful product of which they could be proud. None the less, the potential is for even the tiniest of software errors to produce catastrophic failures. This worries the legal department, and, for the sake at least of legal consequences, they do their best to dissociate the company from uses to which the software is put in the real world.

There are other contexts where litigation is not even a theoretical factor. For instance, if you work in a software house, your colleagues may need to use your software, and they will want to be confident that it works. If something goes wrong, blanket disclaimers are quite beside the point. The management will want to know what went wrong and what you are doing about it.

More subtly, you are often *your own customer* when you write different parts of the program at different times or reuse parts of other programs. This is because, by the time you come to reuse the code, it is easy to forget what it did.

All in all, therefore, we see that the quality you are trying to achieve in your software, and the responsibility for avoiding errors, goes beyond what can be defined by legal or contractual obligations.

1.3 What *did* you want your program to do?

Your finished software will contain lots of *code* — instructions for the computer written in some programming language or other. It is important to recognize that the activity that this describes is essentially meaningless from the point of view of the users because they do not need to know what is happening inside the computer. This remains true even for users who are able to read and understand the code. Users are interested in such questions as:

- What is the program's overall effect?
- Is it easy to understand what it does?
- Is it easy to use?
- Does it help you detect and correct your mistakes, or does it cover them up and punish you for them?
- How fast is it?
- How much memory does it use?
- Does it contain any errors?

None of these is expressed directly by the code. Generally speaking, the collection of computer instructions in itself tells you nothing about what the

program achieves when run in the real world.

It follows that in progressing from your first vague intention to the completed software you have done two distinct things: first, you have turned the vague ideas into something precise enough for the computer to execute; and second, you have converted the users' needs and requirements into something quite different — instructions for the computer.

The sole purpose of this book is to show how to divide this progression into two parts: first, to turn the vagueness into a precise account *of the users' needs and wants*; and then to turn that into computer instructions.

This 'precise account of the users' needs and wants' is called a *specification*, and the crucial point to understand is that it is expressing something quite different from the code, that is, the users' interests instead of the computer's. If the specification and code end up saying the same thing in different ways — and this can easily happen if you think too much from the computer's point of view when you specify — then doing both of them is largely a waste of time.

1.4 Local versus global behaviour

One distinction between the code and the specification is that whereas the code describes individual execution steps — *local* behaviour — the specification is often about the overall, *global* behaviour. The following is an example (though it does not use an orthodox programming language).

WALKIES: Walking According to Local Kommands In Easy Steps. The following is an example WALKIES program:

```
GO 3 METRES; TURN LEFT 90 DEGREES;
GO 3 METRES; TURN LEFT 90 DEGREES;
GO 3 METRES; TURN LEFT 90 DEGREES;
GO 3 METRES; TURN LEFT 90 DEGREES
```

The *local* behaviour of this is that it does four walks with right angles in between; a *global* property is that it ends up at the starting position. The program does not explicitly describe this global property; we need some geometry to deduce it. This is not trivial because, with the wrong geometry, the global property can fail! Therefore, the geometric reasoning must be deep enough to resolve this.

Consider this program:

```
GO 10000 km; TURN LEFT 90 DEGREES;
GO 10000 km; TURN LEFT 90 DEGREES;
GO 10000 km; TURN LEFT 90 DEGREES;
GO 10000 km; TURN LEFT 90 DEGREES
```

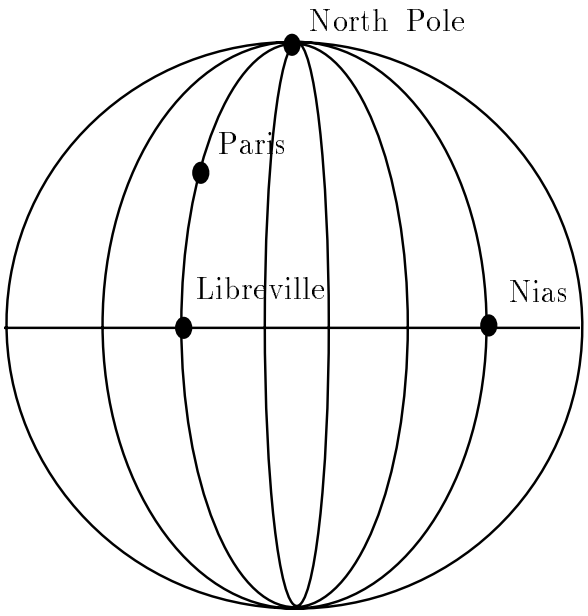

Figure 1.1

You would not end up where you started, if you started at the North Pole and walked round the Earth (Figure 1.1).

The metre was originally defined as one ten millionth part of the Earth's circumference from the North Pole to the Equator via Paris, so the WALKIES trip here goes from the North Pole to Libreville (via Paris), then near to a little island called Nias, then back to the North Pole, and then *on to Libreville again.* You don't get back to where you started. Thus the global properties of a program can depend very much on hidden geometrical assumptions: is our world flat or round? They are not explicit in the program.

WALKIES is not a typical programming language, and properties of programs do not typically depend on geometry, although like WALKIES, their behaviour may depend on environmental factors. But it is nevertheless true that program code usually describes just the individual execution steps and how they are strung together, not their overall effect.

1.5 Reasoned programs

Once we have made both the code and the specification precise, then it is a valid and useful exercise to try and compare them as precisely as possible. In later chapters we shall see specific mathematical techniques for making this comparison. What they amount to is that we try to give mathematical precision not only to the vague overall intention (obtaining a specification),

but also to all the comments in the program. They can be written logically, and whether they fit the code can be analyzed precisely.

When code is supported by this kind of careful specification and reasoning, it is a much more stable product. When you have written it, you have greater confidence that it works. When you reuse it, you know exactly what it is supposed to do. When you modify it, you have a clearer idea of how your changes fit into its structure.

Here, then, is our overall goal:

specification + reasoning + code $\longrightarrow$ a *Reasoned Program*

1.6 Reasoned program*ming*

We have presented the Reasoned Program as the desired software end product, but there is also something important to say about the process of developing it, in other words about *Reasoned Programming.*

It is possible to see the purpose of the specification as being to say what the code does; but that is the wrong way round. Really, the purpose of the code is to achieve what the specification sets out. This means that it is much better to specify first and then to code. In everyday terms, you can perform a task more effectively if you can understand first what it is that you are trying to achieve.

In the words of Hamming,

'Typing is no substitute for thinking.'

This means make your ideas precise before you type them in — work out what you want before you tell the computer how to do it. In the thirty years since Hamming formulated his ideas, our ideas of how to set about this have advanced greatly, and this book is written to teach you, in practical terms, how the modern ideas work.

It is always tempting to start straight off on the code. You gain your initial experience in very short programs and find that this method works. It gets something into the computer, and you get feedback that is gratifyingly quick, even if it often shows that mistakes are present. Many programmers continue to work in this way for the whole of their careers. They find that even if they accept the idea of aiming for Reasoned Programs, it all seems an impossible dream 'Yes, all right in theory, but ...'. They write the code, and *then* — perhaps just before a deadline — try to clarify the reasons.

This is a mistake. There are two essential aspects to the final product that distinguish it from the initial vague intentions, namely precision and local execution steps, and if you go for the code first, you are trying to obtain both aspects at once. On the other hand, if you first think about the specification, then you are just looking for precision. After all, it is the specification that