The Correctness-by-Construction Approach to Programming

Derrick G. Kourie • Bruce W. Watson

The Correctness-by-Construction Approach to Programming

Springer

Derrick G. Kourie
University of Pretoria
Department of Computer Science
Pretoria
South Africa

Bruce W. Watson
Stellenbosch University
FASTAR Group, Information Science
Stellenbosch
South Africa

ISBN 978-3-642-27918-8 e-ISBN 978-3-642-27919-5
DOI 10.1007/978-3-642-27919-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012935686

ACM Codes: D.2, F.3, F.4

Printed on acid-free paper

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

Preface

Software correctness is a perennial concern. It can and should be addressed as well as possible at various levels and in various complimentary ways. This book is devoted to software correctness at the programming-in-the-small level—down at the point where the developer is in the process of giving birth, so to speak, to an algorithm or part of an algorithm. Its concern lies with that which Brooks [15] calls the *essentials*:

> Those aspects of the programming task which are inescapably error-prone; which confound and confuse our minds, even though individual programming commands in isolation seem quite simple; those uncomfortable points in the code where we are inclined to behave intuitively, guessing at or making little leaps in logic—just tiny little leaps—in an effort to speed up the coding effort. And when we are done with coding and everything compiles nicely, then we hold our breath and live in faith and hope that if there were small misjudgments, they will be exposed during testing.

The book advocates a development style known as *correctness by construction*. The idea is to start with a succinct specification of the problem, which is progressively evolved into code in small, tractable refinement steps. Experience has shown that the resulting algorithms are invariably simpler and more efficient than solutions that have been hacked into correctness. Furthermore, such solutions are guaranteed to be correct (i.e. they are guaranteed to comply with their specifications) in the same sense that the proof of a mathematical theorem is guaranteed to be correct.

The idea is not new. It emerged from earlier attempts in computer science to prove programs to be correct *after* the code had been written. By the mid-eighties, hopes that such ex post facto correctness proofs could contribute practically to software correctness more or less reached a dead end. Without imposing some restraint on how code is to be produced, proofs rapidly become too complex—both for the human mind, and for computers. Instead, a tradition built up, starting with some of the most prominent founding personalities in computer science (Dijkstra, Hoare, Knuth, Wirth) of methodically evolving correct code from specifications in a disciplined step-wise fashion.

Dijkstra was arguably the most vociferous proponent of these ideas. He positioned himself as a prophetic voice crying out in the wilderness that the only path to creating enlightened software developers was through "the cruelty of really teaching computer science" [13]. He contrasted this approach with software engineering, whose charter he disparagingly characterised as "How to program if you cannot." This kind of polarising language has led to unfortunate caricatures around two computer science stereotypes: industry-based developers who supposedly hack around in the real world producing lots of flakey code; and head-in-the-clouds academia engaging in impractical esoteric scientific research. In this caricatured world, the former call themselves software engineers and the latter call their research "formal methods".

We vigourously contest these polarised stereotypes and we hope that this book will contribute to their erosion. We aim to convince the reader that the kind of methodical formal approach that Dijkstra and others have advocated is well within the reach of the average computer scientist and software engineer. Not only that: we hope that the reader will discover that, when confronted with algorithmic problems whose logic is unusually complicated or confusing, it is both satisfying and profitable to develop the code by engaging in a correctness by construction style of programming. We have therefore pitched this text at those who actually develop code, rather than at the formal method purists. At the risk of being accused of being insufficiently formal, we have avoided the kind of presentation style which has given formal methods the reputation of being the domain of an elite few.

The way in which we set about achieving our purpose is by a series of graded examples, rather than by an over-emphasis on the theory that drives the correctness by construction development method. However, a modicum of theoretical and notational background is unavoidable, and this we provide in Chap. 2. After rapidly reviewing first-order predicate logic in this chapter, we relate it to the idea of (total) correctness of Hoare pre-post formulae. This allows us to define the notion of the weakest precondition which, in turn, allows for precisely defining the semantics of the commands used in Dijkstra's Guarded Command Language (GCL)—the notation used throughout the book. Initially we rely on Hoare pre-post notation for expressing the refinement laws of Morgan's refinement calculus [32], but later also introduce Morgan's somewhat more concise notation. We restrict ourselves to a small but useful set of the refinement laws, thus shielding the average computer scientist from the more obscure refinement rules which will only interest theoretical computer scientists.

Chapter 3 illustrates the correctness by construction development method on a number of simple algorithms, many of which might have already been seen in the first or second year of study. Chapter 4 looks at a variety of intermediate range algorithms across a broad spectrum of application domains: analysing array properties (such as finding the longest segment of different elements); raster graphics applications; computational geometry; the majority voting problem; etc. Chapter 5 considers the development method in the context of procedure calls, including recursive procedure calls.

Chapters 6 and 7 are intended as *case studies*. Chapter 6 shows how the correctness by construction method was used to derive an elegant recursive algorithm for constructing the cover graph of a so-called set intersection closed lattice. The formal concept analysis (FCA) research community are discovering how variants of these lattices can be used in a numerous applications such as machine learning and data clustering. The derived algorithm turns out to be significantly superior to many other competing algorithms in the domain. Although a version of the algorithm had been intuitively discovered in the nineties, its articulation was so obscure that even domain specialists found it difficult to understand and verify. As a result, there were niggling doubts about its correctness, despite thorough testing. The case study highlights the fact that the correctness by construction derivation leads to a clear, comprehensible version of the algorithm. Its correctness can thus be readily apprehended and accepted by the user community.

The Chap. 7 case study illustrates yet another useful feature of correctness by construction: it offers a rational basis for articulating algorithm taxonomies. The chapter shows how, when a number of *related* algorithms are developed in this style, their commonalities are clearly exposed, thus offering a basis for taxonomising the related algorithms. The resulting taxonomies are not only useful from a pedagogical perspective; they also tend to expose algorithmic "gaps" in the derived taxonomy, thus suggesting further areas of algorithmic research. In this text we have chosen to illustrate the idea in respect of algorithms to construct minimal acyclic finite automata. Such automata are widely used for in domains such as natural language processing, voice recognition and intrusion detection. This is but one of several other studies which have relied on correctness by construction as a basis for taxonomising.

Although these last two chapter are specialist in nature, we consider them important in that they dispel the myth that correctness by construction should be positioned in the domain of dilettante formal methods theoreticians. On the contrary, we think that any respectable computer science/software engineering university curriculum ought to cover the basic material to be found in this book and that every well-educated computer scientist/software engineering graduate should know something about its major themes. It is becoming increasingly apparent that in universities where such material is casually bypassed under the pretext of focussing the curriculum on industry needs, the better-informed students feel cheated by what they perceive as a dumbing down of courses—and they would be right! Such a viewpoint directly contradicts IEEE's Guide to the Software Engineering Body of Knowledge (SWEBOK)[1] which identifies themes covered in this book as part of the software engineer's armory of tools and methods. Similarly, this book's material will be seen to be consonant with the aspirations of the Software Engineering Method and Theory (SEMAT) initiative which, in its call to action[2], somewhat controversially aims to "refound software engineering based on a solid theory".

The first four chapters of the book, as well as Chap. 6 has formed the core of a fourth year course (involving about 30 contact hours) that we have presented for more than a decade. More of the book can be covered in this time if the instructor

[1] See Chap. 10 of the SWEBOK specifications available from http://www.computer.org/portal/web/swebok/home.

[2] See http://www.semat.org/bin/view.

selectively omits and/or assigns as self-study, some of the material in Chaps. 3, 4 and 6. We have found that students are well-able to cope with self-studying many of the examples in Chaps. 3 and 4, provided that the instructor has initiated them into the approach by walking through a representative number of examples. Such self-study-based fast tracking through Chaps. 3, 4 and 6 enables one to cover the main ideas in Chap. 7 as well—something that is well worth doing.

Students who wish to take the course are advised that they should have a basic background in logic. Subject to this proviso, we believe that much of the material can be taught at third year level and probably even earlier. Indeed, because of Dijkstra's influence, this approach to programming was taught at an introductory level at Eindhoven University of Technology.

Many people have contributed to this book in many different ways. They all deserve our sincerest thanks:

- Numerous students whose feedback over the years has helped improve the quality of text.
- Loek Cleophas, who has read and critiqued earlier drafts of the book.
- Alexander Skelton, who wrote the first draft of Chap. 5 as a student project.
- Our many colleagues and friends who have constantly inspired and encouraged us in various ways to produce this book.
- Last, but not least, our respective families who have been a constant source of support and encouragement to us.

Pretoria, South Africa *Derrick G. Kourie*
Eindhoven, Netherlands *Bruce W. Watson*

Contents

1 Introduction 1
- 1.1 Invariance Examples 2
  - 1.1.1 A Chess Board Problem 2
  - 1.1.2 A Black and White Balls Game 4
- 1.2 The Way Ahead 5

2 Background 7
- 2.1 Predicates 8
  - 2.1.1 Propositional Calculus 8
  - 2.1.2 Predicate Calculus 10
  - 2.1.3 Predicates Define Sets of States 10
  - 2.1.4 Strong and Weak Predicates 11
- 2.2 Specifying Pre- and Postconditions 14
  - 2.2.1 Hoare Triples as Specifications of Total Correctness 14
  - 2.2.2 Weakest Preconditions and Semantics 16
- 2.3 Guarded Command Language 17
  - 2.3.1 Empty Command 18
  - 2.3.2 Diversion: Some Extreme Cases 18
  - 2.3.3 Assignment 24
  - 2.3.4 Composition 27
  - 2.3.5 Selection 30
  - 2.3.6 Repetition 32
- 2.4 Refinement Rules 34
  - 2.4.1 Strengthen Postcondition Rule 35
  - 2.4.2 Weaken Precondition Rule 36
  - 2.4.3 Skip Rule 36
  - 2.4.4 Sequences of Refinements 37
  - 2.4.5 Refinement and Weakest Preconditions 37
  - 2.4.6 Assignment Rule 37
  - 2.4.7 Composition Rule 38
  - 2.4.8 Following Assignment Rule 40

2.4.9 Selection Rule ... 41
2.4.10 Repetition Rule ... 42
2.4.11 Procedures and Procedure Calls ... 45
2.5 Object Orientation ... 45
2.6 Supplementary Notation ... 48
2.6.1 Morgan's Refinement Calculus ... 48
2.6.2 Arrays and Sequences ... 49
2.6.3 Additional GCL Commands ... 50
2.7 Revision Exercises ... 51

3 Simple Examples ... 55
3.1 Linear Search ... 56
3.1.1 Formulating the Problem ... 56
3.1.2 Choosing the Invariant ... 56
3.1.3 Establishing the Invariant ... 57
3.1.4 Refining to Create a Loop ... 58
3.1.5 Putting it All Together ... 60
3.2 Finding the Maximal Element ... 60
3.2.1 Formulating the Problem ... 60
3.2.2 Choosing the Invariant ... 61
3.2.3 Establishing the Invariant ... 62
3.2.4 Refining to Create a Loop ... 63
3.2.5 Putting it All Together ... 66
3.3 Binary Search ... 66
3.3.1 Formulating the Problem ... 66
3.3.2 Decomposing the Problem ... 67
3.3.3 Generating the Binary Search Code ... 68
3.3.4 After the Binary Search ... 72
3.3.5 Putting it All Together ... 73
3.4 Pattern Matching ... 74
3.4.1 Formulating the Problem ... 75
3.4.2 Developing the Loop ... 75
3.4.3 Putting it All Together ... 77
3.5 Exponentiation ... 77
3.5.1 Formulating the Problem ... 78
3.5.2 Establishing the Invariant ... 78
3.5.3 Refining to Create a Loop ... 79
3.5.4 Discussion ... 83
3.6 Integer Logarithm Approximation ... 84
3.6.1 Problem Statement and Invariant ... 84
3.6.2 Refinement Steps ... 85
3.6.3 Justifying the Assignment ... 85
3.6.4 Strengthening Predicates by Decreasing Ranges ... 86
3.6.5 Discussion ... 87
3.7 Revision Exercise ... 88

4 Intermediary Examples ... 95
4.1 Dutch National Flag ... 95
4.1.1 Formulating the Problem ... 96
4.1.2 Choosing the Invariant ... 98
4.1.3 Refining the Specification ... 99
4.1.4 Proving the Third Guard Command ... 100
4.1.5 Putting it All Together ... 102
4.1.6 Discussion ... 102
4.2 Longest Segment ... 103
4.2.1 Formulating the Problem ... 104
4.2.2 A First Attempt at Refinement ... 105
4.2.3 A Revised Attempt at Refinement ... 107
4.2.4 Putting it All Together ... 111
4.2.5 Discussion ... 112
4.3 Palindromes ... 112
4.3.1 The Outer Loop ... 113
4.3.2 Formulating the Problem ... 113
4.3.3 Refining the Specification ... 115
4.3.4 Putting it All Together ... 116
4.3.5 Discussion ... 117
4.4 Raster Lines ... 117
4.4.1 Formulating the Problem ... 118
4.4.2 Deriving the Loop ... 121
4.4.3 Developing the Loop's Body ... 122
4.4.4 Putting it All Together ... 125
4.4.5 Discussion ... 126
4.5 Raster Circle ... 127
4.5.1 Problem Statement ... 127
4.5.2 From Invariant to Loop ... 129
4.5.3 Refining the Loop's Body ... 129
4.5.4 Determining the Guards ... 132
4.5.5 Deriving the Guards ... 133
4.5.6 Putting it All Together ... 134
4.6 Majority Voting ... 136
4.6.1 Formulating the Problem ... 137
4.6.2 Arriving at an Invariant and Developing the Loop ... 138
4.6.3 Developing the Guards ... 139
4.6.4 Discussion ... 143
4.7 Computational Geometry ... 144
4.7.1 Background and Notation ... 144
4.7.2 The Approach to Solving the Problem ... 146
4.7.3 Deriving the Solution Constructively ... 147
4.7.4 Discussion ... 150
4.8 Revision Exercises ... 151

5 Procedures and Recursion ... 161
5.1 Introduction ... 161
5.2 Procedures ... 162
5.2.1 Parameterless Procedures ... 162
5.2.2 Pass by Value ... 164
5.2.3 Pass by Result ... 167
5.2.4 Pass by Value Result ... 168
5.2.5 Functions ... 169
5.3 Procedure Refinement Strategy ... 170
5.4 Recursive Procedures ... 171
5.5 Terminating Recursive Programs ... 173
5.6 Recursive Examples ... 177
5.6.1 Factorial ... 177
5.6.2 Searching a List ... 181
5.6.3 Evaluating an Expression Tree ... 185
5.6.4 MergeSort ... 191
5.7 Conclusion ... 194

6 Case Study: Lattice Cover Graph Construction ... 197
6.1 Introduction ... 197
6.2 Preliminaries ... 198
6.2.1 Lattices ... 198
6.2.2 Set Intersection-Closed Lattices ... 201
6.3 The Algorithm ... 205
6.3.1 The Basic Structure ... 206
6.3.2 Articulating and Attaining *inv1*(i) ... 207
6.3.3 Articulating and Attaining $inv2(i)$ ... 208
6.3.4 Filling in S_1 ... 210
6.3.5 Completing the Select Command ... 211
6.3.6 The Completed Algorithm ... 214
6.3.7 The Operational Implications ... 215
6.4 Refactorings ... 218
6.4.1 Efficiently Inserting $C_i \cap X$ ... 218
6.4.2 Finding the Parent of X ... 219
6.4.3 Discussion ... 221
6.5 A Gentle Introduction to Formal Concept Analysis ... 222

7 Case Study 2: Classifying MADFA Construction Algorithms ... 227
7.1 Introduction ... 227
7.2 From DFAs to MADFAs ... 228
7.2.1 Deterministic Finite Automata—DFAs ... 228
7.2.2 Acyclic Deterministic Finite Automata—ADFAs ... 230
7.2.3 Minimum Acyclic Deterministic Finite Automata—MADFAs ... 231
7.2.4 Concepts for MADFA Construction Algorithms ... 232

7.3 An Abstract MADFA Construction Algorithm 237
7.3.1 Structural Invariant Instantiations 239
7.3.2 The Procedures to be Instantiated 241
7.3.3 The Importance of the Skeleton-Based Taxonomy 241
7.4 Trie Intermediate ADFA 242
7.4.1 Procedure add_word_T 242
7.4.2 Adding Only Prefix Words 244
7.4.3 Adding a Non-prefix Word in a Trie 244
7.4.4 Procedure $cleanup_T$ 246
7.4.5 An Example 249
7.5 Arbitrary Intermediate ADFA 250
7.5.1 Procedure add_word_N 251
7.5.2 Procedure $cleanup_N$ 255
7.5.3 Commentary 255
7.6 Word Adding Based on a Partial Order 255

References 259

Index 263