PROGRAM DERIVATION

The Development of Programs From Specifications

Geoff Dromey

PROGRAM DERIVATION

The Development of Programs From Specifications

Geoff Dromey

Griffith University, Brisbane, Australia

Sydney · Wokingham, England · Reading, Massachusetts
Menlo Park, California · New York · Don Mills, Ontario
Amsterdam · Bonn · Singapore · Tokyo · Madrid · San Juan

Preface

The primary aim of this book is to make the principles of program derivation from specifications accessible to undergraduates early in their study of computing science.

The proliferation of personal computers in the home and in schools has meant that there are large numbers of people who have had exposure to using computers and even to 'writing' programs in languages like BASIC. This situation has left many people with the misconception that computing science education is focused upon the coding and debugging of computer programs, whereas this is far from the ideals and objectives of the science. For too long people have tried to learn how to build programs without the support of a rigorous mathematical and logical framework. As a consequence, the cost of developing high-quality software remains at a premium. There is but one chance of overcoming these problems, and that is to recognize computing science for what it really is, a mathematically-based discipline concerned with the application of rigorous methods for the specification, design, and implementation of computer systems.

It is one thing to be able to write down a few pages of program statements but it is an entirely different matter to produce correct programs that provably satisfy their specifications. That anything less is acceptable merely reflects the stage of development of the discipline.

What we have to offer in this book is not something that can cure all these problems. The intent has been to provide an introduction to the level of precision, habits of mind, and modes of reasoning, necessary for producing high-quality verifiable software at a reasonable cost.

There are plenty of computing science academics who have recognized the need for rigour and have introduced the ideas of formal program derivation into their courses. This in itself is not enough. Too often the formal aspects of program derivation are not introduced to students until long after they 'know' how to write programs. The implicit message conveyed by a belated introduction to formal program derivation is that it is not an essential part of the discipline – how could it be when they already know how to write programs! Essential things must always come early in a

course if they are to be taken seriously. Introducing program derivation right from the outset is a key challenge facing computing science educators. We have to face this challenge if we are to gain the advantages that formal program derivation can deliver.

There is no question with the argument that it is easier to teach the formal aspects of program derivation later in a course rather than near the beginning, but this sort of reasoning completely misses the point. It has been my repeated experience that very few students respond in the way we would hope to a belated introduction to formal program derivation. They learn the techniques but seldom attempt to apply them beyond the class exercises they are given. This is a sorry state of affairs that can only be rectified by an early exposure to formal program derivation.

The best time to do this is at the start of a university or college course although some will prefer to offer material of this level in the second or third years of a course.

There are two key ideas in this book. The first is a model for the stepwise development of programs. *It is shown that programs can be constructed by a sequence of refinements, each of which establishes the postcondition, or goal, of the computation for progressively weaker preconditions. The model makes a direct link between specifications and the stepwise refinement process.* Each refinement is guided by a transformation that weakens the precondition specification for which the postcondition is established. The state model for computation provides the foundation upon which this model for stepwise refinement is built. Interpreting this model for stepwise refinement directly, a program may be constructed by a sequence of refinements each of which expands the set of initial states for which the postcondition is established. This strategy realizes a limiting form of top-down design.

The second key idea in this book *establishes the connections between data structure, the refinement process, and the program control structure that the refinement process yields.* The reason for trying to make these connections is to ensure that the control structure of derived programs matches the structure of the data to be processed. Programs that possess this characteristic are usually much easier to understand and maintain.

Presenting these key ideas is not by itself enough to make formal program derivation palatable to novice programmers. The real key lies in the nature of the exercises and examples that are considered. Formal program derivation involves considerable notation and several difficult concepts. If these notations and concepts are simply presented to students and then the students are asked to work with and apply them, the chances of success are very small. Students must be eased into formal program derivation. One way of doing this is to start by focusing on *interpretation* rather than application. Only after students have had very considerable practice at interpreting formal specifications are they ready to start writing and using them in program derivation.

The logic and mathematical groundwork provided in Chapters 2, 3, and 4, together with their accompanying exercises, go a long way towards building students' confidence to tackle program derivation. Without paying serious attention to the exercises there is little hope of learning how to derive programs. It would have been easy to write this book and assume the requisite logic and mathematical background. This path has not been taken for several reasons. Firstly, most treatments of mathematics and logic have not been designed to support the writing of specifications and the derivation of programs. What is needed to support program derivation is a treatment of discrete mathematics that presents it as both an abstract reasoning calculus and as a notation suited for formal manipulation. As a consequence, the treatment must be slow and thorough if students are to acquire the level of mathematical maturity needed to apply effectively formal methods to program derivation.

Throughout the book Dijkstra's 'guarded commands' *program design language* has been used to express derived algorithms. The decision to opt for guarded commands rather than use a conventional *program implementation language* like Pascal or Modula-2 is not one that has been taken lightly. There are two principal reasons for making this choice. Firstly, deriving algorithms directly into an implemented programming language provides too much unnecessary distraction. During the derivation process we need to focus only on design considerations and not on implementation details. Subsequent translation from guarded commands to a standard programming language is essentially a mechanical process (automatic or otherwise) that is easy to do after the hard work of derivation is complete. The second important reason for opting to use guarded commands is that it provides a very simple and concise notation for expressing algorithms.

We suggest that there is strong pedagogical merit in the overall stance that we have taken.

There are several ways in which this book can be used to study program derivation:

Programme 1 Abridged introduction to program derivation

To obtain a focused but limited introduction to the main ideas of program derivation the following material would need to be covered along with the accompanying exercises:

Chapter 1 All sections
Chapter 2 Sections 2.3 and 2.4
Chapter 3 Section 3.2
Chapter 4 All sections
Chapter 5 All sections
Chapter 6 A selection of examples from each section

This treatment would be deficient particularly with regard to the background needed for advanced program specification and derivation.

Programme 2 Introduction to program derivation

For those who have previously had a course in discrete mathematics, Chapters 2 and 3 could be omitted from Programme 1 and selected examples from Chapters 7, 8 and 9 could be added.

Programme 3 A second course in program derivation

Those who have previously had an introduction to program specification and derivation would need to cover the following material:

Chapter 5	Sections 5.5, 5.6 and 5.7
Chapter 6	Sections 6.1 to 6.4
Chapters 7–12	All sections

The examples in the later chapters tend to be more advanced. Also, within chapters the later examples are usually more difficult.

Programme 4 Formal program derivation

Programmers, computer science graduates and others wishing to gain an understanding of formal methods of program derivation might be best advised first to read Chapter 1 and then study:

Chapter 4	All sections
Chapter 5	All sections
Chapters 6–12	Selected examples as required

In pursuing this programme it would probably be necessary to refer back to Chapters 2 and 3 from time to time.

As a final word on using this book, proficiency in specifying problems and formally deriving programs only comes with lots and lots of practice – there are no shortcuts.

The time for a more formal and more careful approach to programming has long since arrived! The words of Victor Hugo sum up the present situation:

> There is one thing stronger than all the armies of the world,
> and that is an idea whose time has come.

Acknowledgements

There have been many friends, colleagues, and students who have helped me with this book. I am particularly indebted to David Gries and Jifeng He

for a number of stimulating discussions in the early stages of this work. I would also like to thank Wlad Turski for helpful comments on some of the ideas in Chapter 5. I also appreciate the helpful suggestions for improving the manuscript made by David Billington, Trevor Chorvat, Andrew McGettrick, Tony Hoare, and a number of anonymous referees.

I would like to thank the following eminent Computer Scientists for permission to use their words of wisdom at the beginnings of Chapters:

R. W. Floyd, C. A. R. Hoare, F. P. Brooks, E. W. Dijkstra, W. Turski.

The professionalism of Stephen Troth in bringing this manuscript to publication has been of the highest order. His support and encouragement together with that of editor Andrew McGettrick and production editor Sheila Chatten is deeply appreciated. I would also like to thank Helen Whiter, Olwen Schubert, Kathryn Stanford and Lenore Olsen for their care and patience in preparing the manuscript.

Finally I would like to thank my wife, Aziza, and daughter, Tashen, for their support and understanding throughout this project.

R. Geoff Dromey

Brisbane, Australia
June 1988

Publisher's acknowledgements

The publisher would like to thank the following for giving their permission to reproduce some of their material:

Cambridge University Press for R.G. Dromey, 1987, Derivation of Sorting Algorithms from a Specification, *The Computer Journal*, **30**(6); R.G. Dromey and T. Chorvat, 1989, Structure Clashes – An Alternative to Program Inversion, *The Computer Journal*(in press).

The Institute of Electrical and Electronic Engineers, Inc. for R.G. Dromey, 1988, Systematic Program Development, *Transactions on Software Engineering*, **14**(1). (© 1988 IEEE).

John Wiley and Sons Limited for R.G. Dromey, 1985, Program Development by Inductive Stepwise Refinement *and* Forced Termination of Loops, *Software – Practice and Experience*, **15(1)**, pp.1–28 *and* 29–39. Reproduced by permission of John Wiley and Sons Limited.

Macmillan Publishing Company for a quotation from *Aims of Education and Other Essays* by Alfred North Whitehead. Copyright 1929 by Macmillan Publishing Company, renewed 1957 by Evelyn Whitehead. Reprinted with permission.

Contents

Part 1
TOOLS FOR PROGRAM DERIVATION

A number of tools are needed to derive programs effectively. Logic and discrete mathematics may be used to specify problems adequately and to support the sequences of constructive reasoning steps that must be made in deriving and proving programs correct.

In Chapter 1 of this book we examine the role of logic and mathematics in program derivation. We then go on to introduce a model for formal program derivation. The foundation on which program derivation is built is a precise specification. The predicate calculus, together with set-based notations and formalisms, provide a powerful set of tools for specifying problems. They allow us to write concise specifications that are readily amenable to the sort of transformations that are needed to derive programs.

To derive and characterize programs several specialized forms of specification are required. A precondition is used to describe the input or data that is to be processed, while the output or goal of a computation is described by a postcondition. A third form of specification, an invariant, plays a vital role not only in characterizing computations but also in guiding the derivation of programs. A final form of specification needed to model computations is the variant function. Variant functions characterize the termination properties of programs. All these different forms of specification are discussed in Chapter 4.

Chapter 1
Introduction

There is always a first step in a journey of ten thousand miles
Chinese Proverb

1.1 The Problem of Programming

The premium quality of a program is its *correctness*. In spite of this, there are very few complex programs in use today that do not contain some sorts of errors, anomalies, or peculiarities. Fortunately, most of these defects are extremely subtle and become apparent only in very unusual circumstances. The trouble is that very unusual circumstances *do* occur from time to time.

The root cause of the lack of quality in programs lies not simply with the people responsible for producing the software but rather with the inadequate methods that are employed to develop programs. The response of the programming community to this situation has been to develop methods for such things as:

- project management
- requirements analysis
- top-down problem decomposition
- structured programming
- abstract data types

- information hiding
- program walkthroughs
- rigorous testing.

These developments have led to considerable improvements in program quality. However, correctness remains a major problem. Programmers still regard it as inevitable that the programs that they write will contain errors (or 'bugs' as they are more usually called). In fact many practitioners claim that as much as one third to one half of all programming effort is expended in 'debugging' programs. Worse still, the act of 'removing a bug' more often than not introduces other bugs.

The present situation is not a happy one particularly because we are increasing our dependency on the use of computers in life-critical applications such as air-traffic control, nuclear reactor control and life-support systems.

The following random sample of program errors that have come to public attention in recent times gives a glimpse of the seriousness of the problem.†

- The space shuttle Discovery flew upside down during a laser-beam missile defence experiment because it expected information in nautical miles and was given it in feet.
- The first version of the F-16 navigation software inverted the aircraft whenever it crossed the equator.
- A version of the Apollo II software had the moon's gravity appearing repulsive rather than attractive.
- A computer error caused a US warship's gun to fire in the opposite direction from its intended target during an exercise off San Francisco in 1982.
- In February 1984, the cash machines of two UK banks had serious compatibility problems because one remembered the leap year and the other did not.
- In 1983 the Vancouver Stock Exchange index was found to be at 725 points rather than 960 points due to a cumulative error in the way the index was calculated.
- A series of accidental radiation overdoses was administered by cancer therapy machines in Georgia, USA in 1986 because of an error in the controlling computer program.

Such errors are just the tip of the iceberg.

How is it that software is so susceptible to errors? Much of the difficulty lies in the sheer complexity of the tasks being programmed. There

† Many other examples can be found in the *Software Engineering Notes* reference quoted in the Bibliography.

are usually a huge number of variables and relationships that need to be taken into account when building any substantial software system.

Errors can be introduced into a software system at three distinct stages. There are:

- defects in the requirements that result from the failure of the software to cope with all aspects of the environment in which it is used;
- design errors that result from the failure of the design to match the stated requirements;
- implementation defects that result from the failure of an implementation to satisfy its design details.

Requirements defects and design errors propagate right down to the implementation level. In fact these sorts of errors are usually much more insidious and harder to correct than implementation defects.

The question that needs to be asked is 'given the problems with program correctness, is there a way forward to greater reliability and higher productivity'? The answer at this stage has to be a qualified 'yes' but it will require a substantial change in our approach to programming.

1.2 The Role of Mathematics and Logic

What, above all else, needs to be recognized is that programming is a mathematical activity and programs are complex mathematical expressions. These perceptions, if taken to their logical conclusions, completely change our whole conception of programming.

A mathematical view of programming tells us that

- programs can be *proved* correct in much the same way as theorems in mathematics are proved correct;
- the 'intended meaning' of a program can be described using mathematics and logic;
- the assertion that a program satisfies a specification is a mathematical statement.

Probably the most important aspect of a mathematical view of programming is as follows:

> The development of a proof of correctness and the construction of a correct program that satisfies its specification can proceed hand in hand. A program can literally be derived from its specification.

It is this approach to program development that holds the key to greater reliability and higher productivity.

These facts about programming have been known for some time. Why haven't programmers, and the software industry in general, been rushing to adopt the mathematical approach to programming?

The reason usually given for rejecting the practicality of a mathematical approach to programming is that it can only be used for small programs. What this line of argument overlooks is that formal methods of specification are now being successfully applied to a growing number of quite large software problems. Once there is a formal specification base, a mathematical approach to deriving a program to satisfy the specification becomes much more within the realms of possibility. The problem of size remains daunting if we insist that manual formal program derivation in the large should be handled in exactly the same way as formal program derivation in the small.

There are two ways of confronting the 'size problem'. We can adopt an approach similar to that taken in the applied sciences and engineering where the mathematics applied in the field has been adapted to the size of the problem. Not all variables and relationships that might be considered in a laboratory experiment are taken into account. Instead, the mathematics is used to focus on critical relationships and crucial variables. In making these simplifications the problem is still solved with rigour within a supporting mathematical framework. Enough formal reasoning is retained to allow others to verify their reasoning and if necessary detect errors in correctness. Such an approach to program derivation is possible provided we have a formal specification as our starting point.

The other alternative for dealing with the size problem is to rely upon a semi-automated programming environment that can assist with technical steps such as proof checking, equivalence transformations, weakest precondition calculations, and so on. We may expect, as the discipline matures, that such tools will achieve greater prominence. With their assistance the program designer will be relieved of a lot of tedious detail and the need to perform mechanical tasks.

There is one other essential ingredient needed to support a mathematical approach to program derivation. It will require the support of all concerned in the program development process. This means changes in attitudes and practices on the part of the managers, the programmers, and the customers. Without strong support from all groups the task will be very difficult.

1.3 How Programs Are Derived

The principal steps in deriving a program are **specification**, **design** and **implementation**.