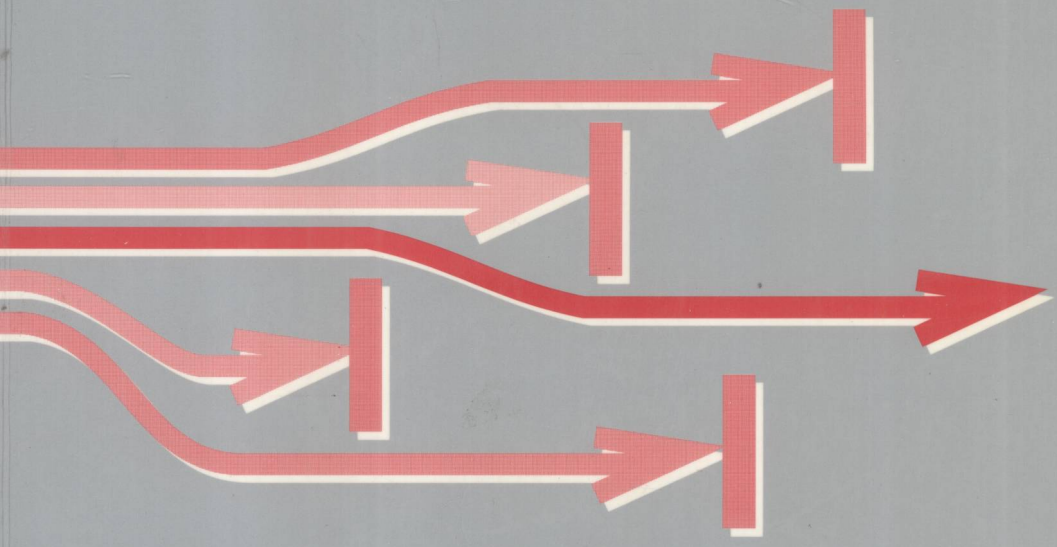
Derek Andrews

A Theory and Practice of Program Development



Springer

FACIT

Derek Andrews

A Theory and Practice of Program Development

Springer

Derek Andrews, BSc, MSc
Department of Mathematics and Computer Science,
Leicester University, University Road, Leicester LE1 7RH, UK

Series Editor

S.A. Schuman, BSc, DEA, CEng
Department of Mathematical and Computing Sciences
University of Surrey, Guildford, Surrey GU2 5XH, UK

ISBN 3-540-76162-4 Springer-Verlag Berlin Heidelberg New York

British Library Cataloguing in Publication Data
Andrews, Derek
A theory and practice of program development. - (Formal approaches to computing and information technology)
1.Computer software - Development
I.Title
005.1
ISBN 3540761624

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

Typesetting: Camera ready by editors
Printed and bound at the Athenæum Press Ltd., Gateshead, Tyne and Wear
34/3830-543210 Printed on acid-free paper

Formal Approaches to Computing and Information Technology

Springer
London
Berlin
Heidelberg
New York
Barcelona
Budapest
Hong Kong
Milan
Paris
Santa Clara
Singapore
Tokyo

Also in this series:

Proof in VDM: a Practitioner's Guide
J.C. Bicarregui, J.S. Fitzgerald, P.A. Lindsay, R. Moore and B. Ritchie
ISBN 3-540-19813-X

Systems, Models and Measures
A. Kaposi and M. Myers
ISBN 3-540-19753-2

Notations for Software Design
L.M.G. Feijs, H.B.M. Jonkers and C.A. Middelburg
ISBN 3-540-19902-0

Formal Object-Oriented Development
K. Lano
ISBN 3-540-19978-0

The B Language and Method
K. Lano
ISBN 3-540-76033-4

Formal Methods and Object Technology
S.J. Goldsack and S.J.H. Kent
ISBN 3-540-19977-2

Preface

All software development methods are based on the same basic idea: to provide a way of producing good software from a set of system requirements. A good software development method should provide a notation that can be used during the analysis of the system requirements to produce an abstract model (a specification) that captures the essence of the system and that can be used as the basis of the development. A good development method should try to encourage the specification of what the system does at the early stages of design and provides a series of strategies and transformations (usually described informally) on how to turn the specification into executable code. What is frequently missing from many development methods is a formal (in the mathematical sense) background and justification to both the notation and the transformations. A formal approach can provide an insight into the analysis, specification, design and development of computer systems.

An approach to software development is to focus on the separation of the abstract and concrete views of a system. A good development method should emphasize:

- abstract vs concrete;
- logical vs physical; and
- 'what' vs 'how'.

The abstract view should be easily understood by the system architect and should concentrate on describing what the system does. The concrete view of the system is a program that describes how the system works at a level of detail can be efficiently executed by a computer. Design can be seen as the bridge between these two views of a system. This bridge should have mathematical foundations.

The book provides a weakest pre-condition semantics for almost all of VDM-SL and uses the semantics to derive a software development method and to give the method a rigorous justification. The strategies and transformations that are used are both justified and explained mathematically.

The language described in the book is based on VDM-SL (perhaps it is closer to Meta-IV on which VDM-SL is based). It has been updated with a more modern syntax: it is a subset of VDM-SL with the following restrictions:

- no loose functions;

- restrictions on patterns (including no loose patterns); and
- parameters to functions and procedures must be read only.

Other changes are to the statement part of the language. The main change is the introduction of Dijkstra's guarded commands for the executable part of the language. This has been one for the following reasons:

1. guarded commands are used as one of the basic constructs of the language;
2. they are elegant; and
3. an excellent idea should be republicized!

Simple input and output statements have been introduced so that the executable language could be implemented and used. Some other minor changes to the syntax have been made to make programs more readable. The most important change is the introduction of a specification command for refinement purposes. The author should apologize for these changes, but the advantages of Dijkstra's guarded commands outweigh the need to conform to the ISO Standard for VDM-SL.

The semantics are used to derive appropriate program and data refinement rules. It is easier for a developer attempting a refinement, or proving properties of a specification or a program to derive new, useful, rules using weakest pre-conditions. In particular it has been found that proofs of properties of language constructs can be more easily discharged with this approach.

The book shows that it is now possible to derive both a specification language and development method based on a few simple concepts, and to give a formal explanation of all of the development steps used in producing code from a specification. The book introduces the idea that the purpose of analysis is to provide a specification that can easily be understood by the human mind before the expensive process of developing a program starts. Specifications are 'programs' that should be biased towards a style that allows efficient execution by an intellectual computer, and that programs are 'specifications' that can be executed efficiently by an electronic computer. Design is the process of transforming a specification into efficient code.

The approach provides the formal semantics of a simple, but powerful, wide-spectrum programming language (VDM-WSL) and gives a formal definition of both algorithmic and data refinement. The following topics are covered in detail using a formal approach:

- formal specification;
- proving properties of a specification;
- properties of programs;
- algorithmic refinement;
- data refinement; and
- the correctness of code.

The theory of algorithm and data refinement also provides insight into the design process and development strategies that can be used to produce efficient code. Issues such as why reusable code is difficult to achieve have also been addressed. A mathematical foundation and background to most, if not all, of the ideas of software specification, design and development are given.

A course developed at Leicester over a ten-year period now uses the approach put forward in this book. The course was initially based on the refinement style of C. B. Jones from [23]; it slowly evolved to the current style. In a third-year course at Leicester that uses the ideas in the book, students have carried out the specification and refinement of some small systems, for example a system to manage a book library and a car-hire system. Other examples that have been developed were an analysis of sorting algorithms and the correct implementation of both AVL and 2–3 trees. With teaching in mind, many of the easier lemmas, theorems, and corollaries have been left for the reader to complete; these are terminated with a hollow box.

The book would is suitable for a third- or fourth-year undergraduate or a post-graduate course in formal methods – with this in mind, a complete set of teaching material is available. The material consists of:

1. a complete set of over-head projector slides together with a set of teaching notes;
2. a student study guide consisting of weekly reading suggestions and exercises;
3. a set of worked examples together with model answers, these would be suitable to add to the study guide or for use as examination questions; and
4. a technical report that describes the kernel language, the full language and an executable subset.

This material is available in machine readable form (in LaTeX source) on the World-Wide-Web – contact the author at `derek@mcs.le.ac.uk`.

Acknowledgements

Weakest pre-conditions were used by Hehner and others to give the semantics of a (subset) of Pascal and by Dijkstra to give the semantics of his guarded command language [20]. The idea of giving the semantics of VDM-SL with relations and a termination condition was used by Cliff Jones in his Ph.D. thesis [24]. This approach extends naturally when using the weakest pre-conditions to give the semantics of the kernel language used in this book. In a paper by Nelson [30] that gave a slightly different semantics to Dijkstra's guarded command language: undefined ($\perp$) was added as a possible outcome of a command. In [3] Abrial defined the semantics of B in terms of a set of

basic commands. The formal semantics of VDM-SL have been given using relations ([18]) and based on this work, proof rules for VDM-SL have been given by Peter Gorm Larsen in his Ph.D. thesis [25]. Work at Oxford ([27, 26]) and work by Morris [28, 29] introduced the ideas of a programming calculus that dealt with both program and data refinement. Based on the Oxford work, in a series of examples, Andrews and Ince mixed program and data refinement to develop executable code from a specification [11, 10, 9]. This book pulls these ideas together to give a weakest pre-condition semantics for a close relative of VDM-SL, and uses these semantics to derive a set of refinement rules.

The author would also like to thank Professors Dr. H Klaeren and S. Schuman and an anonymous referee for their comments on an earlier draft and Rosie Kemp and the staff at Springer-Verlag for their patience and support.

Chapter Summaries

Chapter 1 – Writing Correct Programs Why refinement, the same simple example treated two different ways: starting with a specification, writing the code and then showing it to be correct and secondly deriving the correct code from the specification by a series of transformations – software development as a strategy game.

Chapter 2 – A Small Programming Language A small language for writing both specifications and programs. The semantics of the language are given using both relations and the weakest pre-condition approach. The concept of testing is discussed, leading to a meaning of program correctness.

Chapter 3 – Concepts and Properties The basic properties of the kernel language: simple program transformations. Setting and testing variables. The basic properties of a theory of refinement. A normal form and a framework for proving properties.

Chapter 4 – Building New Commands from Old Extending the language, principles and theory. Recursive definitions. (This is seen as a chapter for the advanced reader, the remainder of the book does not depend on it.)

Chapter 5 – Program Refinement Stepwise refinement. The concept of refinement. Replacing specifications: the meaning of program correctness and program refinement. Three different ways of looking at correctness and their equivalence. What is testing. Other refinement models.

Chapter 6 – The Basic Commands Refining the basic statements. Implementation issues; the language and the compiler writer.

Chapter 7 – Declarations and Blocks Introducing local variables, the idea of scope. Blocks and the refinement rules.

Chapter 8 – Command Sequences Introducing semicolon into a program. Concepts and properties. Refinement rules for semicolon.

Chapter 9 – The Alternative Command The if command and its properties. Using the command in refinement. Refinement rules for alternatives.

Chapter 10 – The Iterative Command The do command and its properties. Using the command in refinement. Refinement rules for iteration. The problem of termination and its proof. Loop parameters: initialization, guards, invariants and termination. Establishing the invariant. Refining loop bodies.

Chapter 11 – Functions and Procedures The definition of procedures. Introducing function procedures and proper procedures. Refinement rules for introducing and removing procedures and their calls. Refinement rules for using recursion.

Chapter 12 – Examples of Refinement at Work From a simple specification to executable code, the emphasis is on showing the ideas at work.

Chapter 13 – On Refinement and Loops How to refine a specification to use iteration. Hints on finding the guard, invariant and variant terms.

Chapter 14 – Functions and Procedures in Refinement Refining to procedures – introducing procedure declarations and procedure calls. Using recursion.

Chapter 15 – Refinement and Performance Using refinement to improve the performance of a program – a fast multiply and a fast integer divide algorithm.

Chapter 16 – Searching and Sorting Refining a search specification to executable code (linear and binary search). Refining a sort specification to executable code.

Chapter 17 – Data Refinement Replacing abstract data models with data that can be implemented by an executable programming language. The idea of data refinement. An informal explanation of the theory. A simple data refinement.

Chapter 18 – A Theory of Data Refinement The theory behind the rules and methods introduced in Chapter 17. Other approaches to data refinement. Dealing with assignment. This chapter is optional, but it does completes the theory of refinement and shows how everything fits together.

Chapter 19 – An Alternative Refinement of the security system Alternative ways of solving the specification used in Chapter 17.

Chapter 20 – Stacks and Queues Refining stacks and queues to executable code to illustrate data refinement in action. Different approaches to the same problem.

Chapter 21 – Dynamic Data Structures How to deal with pointers. Proving the correctness of a data representation that uses pointers.

Chapter 22 – Binary Trees A formal derivation of the standard algorithms for working with ordered binary trees.

Chapter 23 – Epilogue What next.

This book was produced using the software package 'Textures' (Blue Sky Research) and the LaTeX2$_\varepsilon$ and NPL VDM-SL macro packages on an Apple Macintosh computer.

Contents