

Cambridge University Press  
0521021758 - The B-Book: Assigning Programs to Meanings  
J.-R. Abrial  
Frontmatter  
[More information](#)

---

## The B-Book

Cambridge University Press  
0521021758 - The B-Book: Assigning Programs to Meanings  
J.-R. Abrial  
Frontmatter  
[More information](#)

---

# The B-Book

Assigning Programs to Meanings

J.-R. Abrial



Cambridge University Press  
0521021758 - The B-Book: Assigning Programs to Meanings  
J.-R. Abrial  
Frontmatter  
[More information](#)

---

CAMBRIDGE UNIVERSITY PRESS  
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press  
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

[www.cambridge.org](http://www.cambridge.org)  
Information on this title: [www.cambridge.org/9780521496193](http://www.cambridge.org/9780521496193)

© Cambridge University Press 1996

This publication is in copyright. Subject to statutory exception and to the provisions of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of Cambridge University Press.

First published 1996  
This digitally printed first paperback version 2005

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

ISBN-13 978-0-521-49619-3 hardback  
ISBN-10 0-521-49619-5 hardback

ISBN-13 978-0-521-02175-3 paperback  
ISBN-10 0-521-02175-8 paperback

Cambridge University Press has no responsibility for the persistence or accuracy of URLs for external or third-party Internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Cambridge University Press  
0521021758 - The B-Book: Assigning Programs to Meanings  
J.-R. Abrial  
Frontmatter  
[More information](#)

---

to Hélène Villers

## Tribute

---

Those who have the privilege of friendship with Jean-Raymond Abrial have long been aware of the great work in which he has been engaged. It is no less than a complete understanding of the nature of software engineering, from the capture and analysis of requirements, the formalization of specifications, the evolution of designs, the generation of programs and their implementation on computers. The publication of this book is the culmination of his work, and the complete fulfilment of our fondest hopes.

There will now be a much wider class of readers, for whom the book will come as a revelation, their first introduction to the power of its author's innovative intellect, their first appreciation of the clarity and masterful simplicity of his writing. His achievement is to reconcile the concepts of mathematics with the promptings of intuition, and harness both to solve the problems of modern programming practice. There is much to enjoy learning from the text, and even more to be learnt by putting its lessons into practice. Read, learn, enjoy and prosper!

C.A.R. Hoare

## Foreword

---

This book is much more than a new programming manual. It introduces a method in which the program design is included in the global process that goes from understanding the problem to the validation of its solution.

The mathematical basis of the method provides the exactness while the proposed notation eliminates the ambiguities of the vernacular language. At the same time, the process is simple enough for an industrial use. “Industrial” is in fact the key word.

The general aim of formal methods is to provide correctness of the problem specification. Here we can see how the solution can be found, step by step, by a continuously monitored process. The mathematical verification of each step is so closely bound to the refinement activity that it is no longer possible to separate the design choices from the checking process. Imagination is helped by exactness!

But how about the efficiency? Isn't the design too long? Are the design people able to do this work? Are the machines powerful enough to implement the method? The answers are easy to give. Let me tell you.

My company has been involved, since the sixties, in the realisation of train control systems, which must meet stringent safety requirements. As soon as we began to use programmed logic (end of the seventies) we had to solve the problem of software correctness. Together with other methods, we chose to use the program proving method proposed by C.A.R. Hoare. In 1986, J.-R. Abrial introduced us to the B method. We decided to learn it and to use it. The tools did not exist at the time. We contributed to their elaboration by offering a real-world benchmark with our applications, and proposed some improvements. Now the tools can be found on the market, and the method can be used with its full efficiency. What did we learn?

- First, understanding the principles of the method is quite easy and expertise comes in less than a year.
- Then the method encourages and facilitates re-usability, based on use of a growing library of already proven abstract machines.
- The time saved during test and validation phases is very important, resulting in a global economic balance that is quite positive.
- The produced programs are efficient in spite of their structure being organised in layers of increasing abstraction.
- The tools can be implemented on simple workstations.

The use of the method has been a decisive element by increasing our confidence when using software for safety related applications. Moreover, the new international standards recommend the use of formal methods for the specification and design of safety-related software.

Thanks to J.-R. Abrial, we now have an industrial method to build correct programs. We hope that this book will convince the readers to save their money by using this method.

Pierre Chapront  
Technical Director  
GEC-ALSTHOM Transport

## Introduction

---

This book is a very long discourse explaining how, in my opinion, the task of programming (in the small as well as in the large) can be accomplished *by returning to mathematics*.

By this, I first mean that the precise mathematical definition of what a program does must be present at the *origin* of its construction. If such a definition is lacking, or if it is too complicated, we might wonder whether our future program will mean anything at all. My belief is that a program, in the absolute, means absolutely nothing. A program only means something *relative* to a certain intention, that must predate it, in one form or another. At this point, I have no objection with people feeling more comfortable with the word “English” replacing the word “mathematics”. I just wonder whether such people are not assigning themselves a more difficult task.

I also think that this “return to mathematics” should be present in the very process of program construction. Here the task is to assign a program to a well-defined meaning. The idea is to accompany the technical process of *program* construction by a similar process of *proof* construction, which guarantees that the proposed program agrees with its intended meaning.

Simultaneous concerns about the architecture of a program and that of its proof are surprisingly efficient. For instance, when the proof is cumbersome, there are serious chances that the program will be too; and ingredients for structuring proofs (abstraction, instantiation, decomposition) are very similar to those for structuring programs. Ideally, the relationship between the construction of a program and its proof of correctness should be so intimate as to make it impossible to detect which of the two is driving the other. It might then be reasonable to say that constructing a program is just constructing a proof.

Today, very few programs are specified and constructed in this way. Does this correlate with the fact that, today, so many programs are fragile?

Jean-Raymond Abrial



## Acknowledgements

---

The writing of this book spreads over a period of almost fifteen years. During that period, I have met many people, among which certain have had a positive influence on the work presented in this book. I would like to thank them all.

Clearly, the main source of influence, without which this book could not have been brought into existence, lies in the ideas conveyed by C.A.R. Hoare and E.W. Dijkstra. The view of a program as a mathematical object, the concepts of pre- and post-conditions, of non-determinism, of weakest pre-condition, all these ideas are obviously central to what is presented in this book.

The B method, being a “model oriented” method of software construction, is thus close to VDM and to Z. Obviously, many ideas of both these methods can be recognized in B. This is reasonable for Z, since I was one of its originators before and during my visit at the Programming Research Group in Oxford from 1979 to 1981. This is also reasonable for VDM since I shared an office with C.B. Jones during that same period. From him, I learned the idea of program development and the concept of refinement and its practical application, under the form of proof obligations.

Discussions with C.C. Morgan on specification and refinement have had a significant influence on the material of this book. His idea of enlarging the concept of program to embody that of specification has had a seminal effect on this work.

The collective work done at the Programming Research Group during the eighties on the notion of refinement has been directly borrowed in my presentation of refinement. To the best of my knowledge, the people concerned were P. Gardiner, J. He, C.A.R. Hoare, C.C. Morgan, K.A. Robinson, and J.W. Sanders.

During the practical elaboration of the method, certain people have had a significant influence on this work. Belonging to that category are G. Laffitte, F. Mejia,

I. McNeal, P. Behm, J.-M. Meynadier and L. Dufour, whom I thank very warmly.

G. Laffitte influenced this work by his careful reviews, his accurate criticisms, and the sometimes very serious rearrangements he proposed for some of the mathematical developments of this book.

F. Mejia proposed some important improvements in the area of structuring large software constructions. Together with B. Dehbonei, he developed a complete tool set for B, now commercialized as *Atelier B*.

I. McNeal has made various contributions to the early development of the method. This has had some beneficial influence on the mechanization of proofs.

P. Behm, J.-M. Meynadier and L. Dufour made very interesting suggestions and constructed a prototype prover whose mechanisms are extremely useful.

The magnificent team of DIGILOG, which is industrializing and commercializing *Atelier B*, and developing software systems with it, deserves special congratulations. Their competence, enthusiasm, and kindness make it a real pleasure to work with them. I would like to thank F. Badeau, F. Bustany, E. Buvat, P. Lartigue, J.-Ph. Pitzalis, C. Roques, D. Sabatier, T. Servat, C. Tognetty, and C. Zagoury.

A number of other people have been working indirectly on the B project by reviewing this book, by teaching this work, by applying it, or by promoting it. I would like to thank them all, particularly an anonymous reviewer and also P. Bieber, P. Chartier, J.-Y. Chauvet, C. Da Silva, T. Denvir, P. Desforges, R. Docherty, M. Ducassé, M. Elkoursi, Ph. Facon, H. Habrias, N. Lopez, I. Mackie, L. Mussat, P. Ozello, J.-P. Rubaux, P. Ryan, S. Schuman, M. Simonot, and H. Waeselynk.

Casual meetings and discussions with B. Meyer and M. Sintzoff have had an indirect influence on this work. Meeting them is always an intellectual pleasure, which, to my regret, does not happen often enough.

In the industrial world, a number of institutions have made possible, in one way or another, the writing of this book. I am particularly indebted to ADI, BP, DIGILOG/groupe STERIA, DIGITAL, GEC-ALSTHOM Transport, GIXI, INRETS, INSEE, MATRA Transport, RATP and SNCF. These institutions, at various stages of the many years of the development of this project, supported it in various ways. I would like to thank particularly the following persons: P. Barrier, P. Beaudelaire, J. Betteridge, P. Chapront, A. Gazet, A. Guillon, C. Hennebert, J.-L. Lapeyre, J.-C. Rault, and O. Sebilliau.

xiv      *Acknowledgements*

The publishing of this book has been a long and sometimes painful process, especially at the end of it, where a number of unusual difficulties emerged. Bertrand Meyer, Cliff Jones, and Tony Hoare played a significant contribution in trying to solve these difficulties. May they be very warmly thanked for their help.

In conclusion, I would like to give many thanks to David Tranah from Cambridge University Press. I am particularly indebted to him for making possible the publication of my book while respecting the independence within which this scientific work has been performed.

## What is B?

---

**B** is a method for specifying, designing, and coding software systems.

### *Coverage*

The method essentially deals with the central aspects of the software life cycle, namely: the technical specification, the design by successive refinement steps, the layered architecture, and the executable code generation.

### *Proof*

Each of the previous items is envisaged as an activity that involves writing mathematical proofs in order to justify its results. It is, precisely, the collection of such proofs that makes one convinced that the software system in question is indeed correct.

### *Abstract Machine*

The basic mechanism of this approach is that of the abstract machine. This is a concept that is very close to certain notions well-known in programming, under the names of modules, classes or abstract data types.

### *Data and Operations*

A software system conceived with that method is composed of several abstract machines. Each machine contains some data and offers some operations. The data cannot be reached directly; they are always reached through the operations of the machine. They are said to be encapsulated in the machine.

### *Specification of Data*

The data of an abstract machine are specified by means of a number of mathematical concepts such as sets, relations, functions, sequences and trees. The static laws that the data must follow are defined by means of certain conditions, called the invariant.

### *Specification of Operations*

The specification of the operations of an abstract machine is expressed as a non-executable pseudo-code that does not contain any sequencing or loop. In this pseudo-code one describes each operation as a pre-condition and an atomic action. The pre-condition expresses the indispensable condition without which the operation cannot be invoked. The atomic action is formalized by means of a generalization of the notion of substitution. Among these generalized substitutions is the non-deterministic choice that leaves room for some later decision to be taken in the refinement phase. The formal definition of the pseudo-code allows one to prove that the invariant of an abstract machine is always preserved by the operations it offers.

### *Refinement towards an implementation*

The initial model of an abstract machine (its specification) may be refined in an executable module (its code). This passage from specification to code is carried out entirely under the control of the method. It is thus necessarily concluded by some proofs, whose goal is to show that the final code of a machine indeed satisfies its initial specification.

### *Using refinement as a technique of specification*

Besides the previous (classical) one, there exists another practical use of refinement. It consists in using refinement as a means of including more details of the problem into the formal development. Thus the formal translation of the initial problem statement is performed gradually rather than all at once.

### *Refinement Techniques*

Refinement is conducted in three different ways: the removal of the non-executable elements of the pseudo-code (pre-condition and choice), the introduction of the classical control structures of programming (sequencing and loop), and the transformation of the mathematical data structures (sets, relations, functions, sequences and trees) into other structures that might be programmable (simple variables, arrays, or files).

### *Refinement Steps*

In order to carefully control the previous transformations, the refinement of an abstract machine is performed in various steps. During each such step, the initial abstract machine is entirely reconstructed. It keeps, however, the same operations, as viewed by its users, although the corresponding pseudo-code is certainly modified. In the intermediate refinement steps, we have a hybrid construct, which is not a mathematical model any more, but certainly not yet a programming module.

### *Layered Architecture*

Experience shows that it is preferable to have a small number of refinement steps. As soon as its level of complexity becomes too high, it is recommended to

decompose a refinement into smaller pieces. The last refinement of a machine is thus implemented using the specification of one, or more, abstract machines that are, themselves, refinable. This is done by means of calls to the operations offered by the machines in question. As you can see, the “user” of an abstract machine is, thus, always the ultimate refinement of another abstract machine. In this way, the layered architecture of our software system (or of its translated informal specification) is constructed piece by piece.

### *Library*

The machines on which the last refinement of a given machine is implemented may exist prior to that refinement. In fact, together with the method, a series of pre-defined abstract machines are proposed, which constitutes a library of machines, whose purpose is to encapsulate the most classical data structures.

### *Re-use*

For a given project, it is advisable to extend that library so as to organize the basis on which the future abstract machines of higher level will be implemented. As you can see, the method allows one to choose either a purely top down design, or a bottom up one, or, better, a mixed approach integrating the re-use of specification and that of code.

### *Code Generation*

The ultimate refinement of a machine may be easily translated into one or several imperative programming languages. By doing so, the method provides a solution to the problem of porting an application from one language to another.

### *B User Group*

There exists a user group, called the BUG, for discussions and exchange of information on **B**. Here is its electronic address: `bug.@estas1.inrets.fr`. A mailing list for this book is also available at `bbook.@estas1.inrets.fr`.

## What is the B-Book?

---

The **B-Book** is the standard reference for the **B** method and its notations.

It contains the mathematical basis on which the method is founded and the precise definition of the notations used. It also contains a large number of examples illustrating how to use the method in practice. The book comprises four parts and a collection of appendices:

Part I	Mathematics
Part II	Abstract Machines
Part III	Programming
Part IV	Refinement

### *Part I*

Part I contains a systematic construction of predicate logic and set theory. It also contains the definition of various mathematical structures that are needed to formalize software systems. A special emphasis is put on the notion of proof. Part I consists of the following chapters:

Chapter 1	Mathematical Reasoning
Chapter 2	Set Notation
Chapter 3	Mathematical Objects

*Part II*

Part II contains a presentation of the Generalized Substitution Language (GSL) and the Abstract Machine Notation (AMN). These notations are the ones we use in order to specify software systems. They are presented together with a number of examples showing how large specifications can be built systematically. A set-theoretical foundation of GSL and AMN is also presented. Part II consists of the following chapters:

Chapter 4	Introduction to Abstract Machines
Chapter 5	Formal Definition of Abstract Machines
Chapter 6	Theory of Abstract Machines
Chapter 7	Constructing Large Abstract Machines
Chapter 8	Examples of Abstract Machines

*Part III*

Part III introduces the two basic programming features, namely sequencing and loop. After a theoretical presentation, an important chapter is devoted to the study of the systematic construction of a variety of examples of algorithm developments. Part III consists of the following chapters:

Chapter 9	Sequencing and Loops
Chapter 10	Programming Examples



*Part IV*

Part IV presents a notion of refinement for both generalized substitutions and abstract machines. Refinement is given a mathematical foundation within set theory. The construction of large software systems by means of layered architectures of modules is also explained. Finally, a number of large examples of complete development are studied with a special emphasis on the methodological approach. Part IV consists of the following chapters:

Chapter 11	Refinement
Chapter 12	Constructing Large Software Systems
Chapter 13	Examples of Refinement

*Appendices*

A collection of appendices contains a summary of all the logical and mathematical definitions. It also contains a summary of all the rules and proof obligations:

Appendix A	Summary of Notations
Appendix B	Syntax
Appendix C	Definitions
Appendix D	Visibility Rules
Appendix E	Rules and Axioms
Appendix F	Proof Obligations

## How to use this book

---

This book can be used by people having very different concerns.

For instance, you might intend to learn the method *as a formal method practitioner*. In this case, you are probably not (although you might be) interested in the detailed mathematics presented in the book. It is then recommended to read the book as follows:

Appendix A	Summary of Notations
Chapter 2	Set Notation (section 2.7)
Chapter 4	Introduction to Abstract Machines
Chapter 7	Constructing Large Abstract Machines (sections 7.2 and 7.3)
Chapter 8	Examples of Abstract Machines
Chapter 11	Refinement (sections 11.1.1, 11.2.1, 11.2.5, 11.2.7 and 11.2.8)
Chapter 12	Constructing Large Software Systems (sections 12.1 and 12.2)
Chapter 13	Examples of Refinement

At the other extreme of the spectrum, you are a *computer scientist* and you are interested in the mathematical foundation of the method. In that case, you might be reading the book as follows:

Appendix A	Summary of Notations
Chapter 1	Mathematical Reasoning
Chapter 2	Set Notation
Chapter 3	Mathematical Objects
Chapter 6	Theory of Abstract Machines
Chapter 9	Sequencing and Loops
Chapter 11	Refinement
Appendix C	Definitions
Appendix E	Rules and Axioms

In between, there might be people interested in looking at how the method can be used in order to *structure large specifications and large designs*. The following reading can then be recommended:

Appendix A	Summary of Notations
Chapter 4	Introduction to Abstract Machines
Chapter 6	Theory of Abstract Machines
Chapter 7	Constructing Large Abstract Machines

Chapter 11	Refinement
Chapter 12	Constructing Large Software Systems
Chapter 13	Examples of Refinement

People interested in *developing small programs* in a systematic fashion can read the book as follows:

Appendix A	Summary of Notations
Chapter 4	Introduction to Abstract Machines
Chapter 10	Programming Examples

For people interested in the *formal details of the notations*, it is recommended to read the book as follows:

Chapter 5	Formal Definition of Abstract Machines
Chapter 7	Constructing Large Abstract Machines (section 7.4)
Chapter 11	Refinement (section 11.3)
Chapter 12	Constructing Large Software Systems (section 12.6)
Appendix D	Visibility Rules
Appendix F	Proof Obligations

# Contents

---

<b>I</b>	<b>Mathematics</b>	1
<b>1</b>	<b>Mathematical Reasoning</b>	3
1.1	Formal Reasoning	4
1.1.1	Sequent and Predicate	4
1.1.2	Rule of Inference	5
1.1.3	Proofs	6
1.1.4	Basic Rules	6
1.2	Propositional Calculus	9
1.2.1	The Notation of Elementary Assertions	9
1.2.2	Inference Rules for Propositional Calculus	11
1.2.3	Some Proofs	14
1.2.4	A Proof Procedure	22
1.2.5	Extending the Notation	26
1.2.6	Some Classical Results	27
1.3	Predicate Calculus	29
1.3.1	The Notation of Quantified Predicates and Substitutions	29
1.3.2	Universal Quantification	32
1.3.3	Non-freeness	32
1.3.4	Substitution	34
1.3.5	Inference Rules for Predicate Calculus	35
1.3.6	Some Proofs	36
1.3.7	Extending the Proof Procedure	38
1.3.8	Existential Quantification	39
1.3.9	Some Classical Results	41
1.4	Equality	42
1.5	Ordered Pairs	47
1.6	Exercises	51

<b>2</b>	<b>Set Notation</b>	55
2.1	Basic Set Constructs	56
2.1.1	Syntax	57
2.1.2	Axioms	60
2.1.3	Properties	62
2.2	Type-checking	64
2.3	Derived Constructs	72
2.3.1	Definitions	72
2.3.2	Examples	72
2.3.3	Type-checking	73
2.3.4	Properties	75
2.4	Binary Relations	77
2.4.1	Binary Relation Constructs: First Series	77
2.4.2	Binary Relation Constructs: Second Series	79
2.4.3	Examples of Binary Relation Constructs	82
2.4.4	Type-checking of Binary Relation Constructs	84
2.5	Functions	85
2.5.1	Function Constructs: First Series	86
2.5.2	Function Constructs: Second Series	89
2.5.3	Examples of Function Constructs	90
2.5.4	Properties of Function Evaluation	90
2.5.5	Type-checking of Function Constructs	93
2.6	Catalogue of Properties	94
2.6.1	Membership Laws	95
2.6.2	Monotonicity Laws	96
2.6.3	Inclusion Laws	97
2.6.4	Equality Laws	99
2.7	Example	115
2.8	Exercises	120
<b>3</b>	<b>Mathematical Objects</b>	123
3.1	Generalized Intersection and Union	123
3.2	Constructing Mathematical Objects	130
3.2.1	Informal Introduction	130
3.2.2	Fixpoints	131
3.2.3	Induction Principle	136
3.3	The Set of Finite Subsets of a Set	141
3.4	Finite and Infinite Sets	144
3.5	Natural Numbers	145
3.5.1	Definition	145
3.5.2	Peano's "Axioms"	148
3.5.3	Minimum	153
3.5.4	Strong Induction Principle	156
3.5.5	Maximum	158

<i>Contents</i>	xxvii
3.5.6 Recursive Functions on Natural Numbers	158
3.5.7 Arithmetic	161
3.5.8 Iterate of a Relation	166
3.5.9 Cardinal of a Finite Set	167
3.5.10 Transitive Closures of a Relation	168
<b>3.6 The Integers</b>	<b>170</b>
<b>3.7 Finite Sequences</b>	<b>174</b>
3.7.1 Inductive Construction	174
3.7.2 Direct Construction	176
3.7.3 Operations on Sequences	177
3.7.4 Sorting and Related Topics	182
3.7.5 Lexicographical Order on Sequences of Integers	187
<b>3.8 Finite Trees</b>	<b>188</b>
3.8.1 Informal Introduction	188
3.8.2 Formal Construction	190
3.8.3 Induction	192
3.8.4 Recursion	194
3.8.5 Operations	197
3.8.6 Representing Trees	199
<b>3.9 Labelled Trees</b>	<b>202</b>
3.9.1 Direct Definition	203
3.9.2 Inductive Definition	203
3.9.3 Induction	205
3.9.4 Recursion	206
3.9.5 Operations Defined Recursively	206
3.9.6 Operations Defined Directly	208
<b>3.10 Binary Trees</b>	<b>208</b>
3.10.1 Direct Operations	209
3.10.2 Induction	209
3.10.3 Recursion	210
3.10.4 Operations Defined Recursively	210
<b>3.11 Well-founded Relations</b>	<b>211</b>
3.11.1 Definition	212
3.11.2 Proof by Induction on a Well-founded Set	213
3.11.3 Recursion on a Well-founded Set	214
3.11.4 Proving Well-foundedness	217
3.11.5 An Example of a Well-founded Relation	219
3.11.6 Other Examples of Non-classical Recursions	219
<b>3.12 Exercises</b>	<b>221</b>
<b>II Abstract Machines</b>	<b>225</b>
<b>4 Introduction to Abstract Machines</b>	<b>227</b>
4.1 Abstract Machines	228
4.2 The Statics: Specifying the State	229

4.3	The Dynamics: Specifying the Operations	230
4.4	Before-after Predicates as Specifications	231
4.5	Proof Obligation	232
4.6	Substitutions as Specifications	232
4.7	Pre-conditioned Substitution (Termination)	234
4.8	Parameterization and Initialization	236
4.9	Operations with Input Parameters	238
4.10	Operations with Output Parameters	240
4.11	Generous versus Defensive Style of Specification	241
4.12	Multiple Simple Substitution	243
4.13	Conditional Substitution	244
4.14	Bounded Choice Substitution	244
4.15	Guarded Substitution (Feasibility)	246
4.16	A Substitution with no Effect	247
4.17	Contextual Information: Sets and Constants	248
4.18	Unbounded Choice Substitution	252
4.19	Explicit Definitions	256
4.20	Assertions	260
4.21	Concrete Variables and Abstract Constants	261
4.22	Exercises	262
<b>5</b>	<b>Formal Definition of Abstract Machines</b>	<b>265</b>
5.1	Generalized Substitution	265
5.1.1	Syntax	265
5.1.2	Type-checking	270
5.1.3	Axioms	271
5.2	Abstract Machines	272
5.2.1	Syntax	272
5.2.2	Visibility Rules	274
5.2.3	Type-checking	275
5.2.4	On the Constants	278
5.2.5	Proof Obligations	278
5.2.6	About the Given Sets and the Pre-defined Constants	280
<b>6</b>	<b>Theory of Abstract Machines</b>	<b>283</b>
6.1	Normalized Form	283
6.2	Two Useful Properties	287
6.3	Termination, Feasibility and Before-after Predicate	288
6.3.1	Termination	289
6.3.2	Feasibility	290
6.3.3	Before-after Predicate	292
6.4	Set-Theoretic Models	295
6.4.1	First Model: a Set and a Relation	295
6.4.2	Second Model: Set Transformer	298
6.4.3	Set-theoretic Interpretations of the Constructs	301



<i>Contents</i>	xxix
6.5 Exercises	303
<b>7 Constructing Large Abstract Machines</b>	<b>307</b>
7.1 Multiple Generalized Substitution	307
7.1.1 Definition	308
7.1.2 Basic Properties	308
7.1.3 The Main Result	311
7.2 Incremental Specification	312
7.2.1 Informal Introduction	312
7.2.2 Operation Call	314
7.2.3 The INCLUDES Clause	316
7.2.4 Visibility Rules	318
7.2.5 Transitivity	319
7.2.6 Machine Renaming	320
7.2.7 The PROMOTES and the EXTENDS Clauses	320
7.2.8 Example	320
7.3 Incremental Specification and Sharing	322
7.3.1 Informal Introduction	322
7.3.2 The USES Clause	323
7.3.3 Visibility Rules	324
7.3.4 Transitivity	324
7.3.5 Machine Renaming	325
7.4 Formal Definition	325
7.4.1 Syntax	325
7.4.2 Type-checking	326
7.4.3 Proof Obligations for the INCLUDES Clause	331
7.4.4 Proof Obligations for the USES Clause	334
7.5 Exercises	336
<b>8 Examples of Abstract Machines</b>	<b>337</b>
8.1 An Invoice System	338
8.1.1 Informal Specification	338
8.1.2 The <i>Client</i> Machine	339
8.1.3 The <i>Product</i> Machine	341
8.1.4 The <i>Invoice</i> Machine	343
8.1.5 The <i>Invoice_System</i> Machine	348
8.2 A Telephone Exchange	349
8.2.1 Informal specification	349
8.2.2 The <i>Simple_Exchange</i> Machine	352
8.2.3 The <i>Exchange</i> Machine	355
8.3 A Lift Control System	358
8.3.1 Informal Specification	358
8.3.2 The <i>Lift</i> Machine	358
8.3.3 Liveness Proof	364
8.3.4 Expressing Liveness Proof Obligations	366

8.4	Exercises	369
<b>III</b>	<b>Programming</b>	371
<b>9</b>	<b>Sequencing and Loop</b>	373
9.1	Sequencing	374
9.1.1	Syntax	374
9.1.2	Axiom	374
9.1.3	Basic Properties	374
9.2	Loop	377
9.2.1	Introduction	377
9.2.2	Definition	378
9.2.3	Interpretation of Loop Termination	382
9.2.4	Interpretation of the Before-after Relation of the Loop	385
9.2.5	Examples of Loop Termination	386
9.2.6	The Invariant Theorem	387
9.2.7	The Variant Theorem	388
9.2.8	Making the Variant and Invariant Theorem Practical	390
9.2.9	The Traditional Loop	392
9.3	Exercises	398
<b>10</b>	<b>Programming Examples</b>	403
10.0	Methodology	403
10.0.1	Re-use of Previous Algorithms	403
10.0.2	Loop Proof Rules	406
10.0.3	Sequencing Proof Rule	407
10.1	Unbounded Search	408
10.1.1	Introduction	408
10.1.2	Comparing two Sequences	411
10.1.3	Computing the Natural Number Inverse of a Function	416
10.1.4	Natural Number Division	420
10.1.5	The Special Case of Recursive Functions	422
10.1.6	Logarithm in a Given Base	424
10.1.7	Integer Square Root	425
10.2	Bounded Search	427
10.2.1	Introduction	427
10.2.2	Linear Search	430
10.2.3	Linear Search in an Array	431
10.2.4	Linear Search in a Matrix	433
10.2.5	Binary Search	435
10.2.6	Monotonic Functions Revisited	437
10.2.7	Binary Search in an Array	442
10.3	Natural Number	446
10.3.1	Basic Scheme	446
10.3.2	Natural Number Exponentiation	447

<i>Contents</i>	xxxi
<b>10.3.3</b> Extending the Basic Scheme	448
<b>10.3.4</b> Summing a Sequence	450
<b>10.3.5</b> Shifting a Sub-sequence	451
<b>10.3.6</b> Insertion into a Sorted Array	453
<b>10.4</b> Sequences	455
<b>10.4.1</b> Introduction	455
<b>10.4.2</b> Accumulating the Elements of a Sequence	458
<b>10.4.3</b> Decoding the Based Representation of a Number	461
<b>10.4.4</b> Transforming a Natural Number into its Based Representation	462
<b>10.4.5</b> Fast Binary Operation Computations	465
<b>10.4.6</b> Left and Right Recursion	469
<b>10.4.7</b> Filters	473
<b>10.5</b> Trees	482
<b>10.5.1</b> The Notion of Formula	483
<b>10.5.2</b> Transforming a Tree into a Formula	484
<b>10.5.3</b> Transforming a Tree into a Polish String	487
<b>10.5.4</b> Transforming a Formula into a Polish String	488
<b>10.6</b> Exercises	496
<b>IV Refinement</b>	499
<b>11 Refinement</b>	501
<b>11.1</b> Refinement of Generalized Substitutions	501
<b>11.1.1</b> Informal Approach	501
<b>11.1.2</b> Definition	503
<b>11.1.3</b> Equality of Generalized Substitution	503
<b>11.1.4</b> Monotonicity	504
<b>11.1.5</b> Refining a Generalized Assignment	506
<b>11.2</b> Refinement of Abstract Machines	507
<b>11.2.1</b> Informal Approach	507
<b>11.2.2</b> Formal Definition	511
<b>11.2.3</b> Sufficient Conditions	512
<b>11.2.4</b> Monotonicity	516
<b>11.2.5</b> Example Revisited	522
<b>11.2.6</b> The Final Touch	523
<b>11.2.7</b> An Intuitive Explanation of the Refinement Condition	530
<b>11.2.8</b> Application to the Little Example	532
<b>11.3</b> Formal Definition	533
<b>11.3.1</b> Syntax	533
<b>11.3.2</b> Type-checking	534
<b>11.3.3</b> Proof Obligations	537
<b>11.4</b> Exercises	540
<b>12 Constructing Large Software Systems</b>	551
<b>12.1</b> Implementing a Refinement	551

<b>12.1.1</b>	Introduction	551
<b>12.1.2</b>	The Practice of Importation	556
<b>12.1.3</b>	The IMPLEMENTATION Construct	559
<b>12.1.4</b>	The IMPORTS Clause	561
<b>12.1.5</b>	Visibility Rules	561
<b>12.1.6</b>	Machine Renaming	563
<b>12.1.7</b>	The VALUES Clause	563
<b>12.1.8</b>	Comparing the IMPORTS and the INCLUDES Clauses	565
<b>12.1.9</b>	The PROMOTES and EXTENDS Clauses	565
<b>12.1.10</b>	Concrete Constants and Concrete Variables Revisited	566
<b>12.1.11</b>	Allowed Constructs in an Implementation	566
<b>12.2</b>	Sharing	574
<b>12.2.1</b>	Introduction	574
<b>12.2.2</b>	The SEES Clause	579
<b>12.2.3</b>	Visibility Rules	579
<b>12.2.4</b>	Transitivity and Circularity	583
<b>12.2.5</b>	Machine Renaming	583
<b>12.2.6</b>	Comparing the USES and the SEES Clauses	583
<b>12.3</b>	Loops Revisited	584
<b>12.4</b>	Multiple Refinement and Implementation	584
<b>12.5</b>	Recursively Defined Operations	587
<b>12.5.1</b>	Introduction	588
<b>12.5.2</b>	Syntax	591
<b>12.5.3</b>	Proof Rule	591
<b>12.6</b>	Formal Definition	594
<b>12.6.1</b>	Syntax of an IMPLEMENTATION	594
<b>12.6.2</b>	Type-checking with an IMPORTS Clause	595
<b>12.6.3</b>	Type-checking with a SEES Clause	596
<b>12.6.4</b>	Proof Obligations of an IMPLEMENTATION	597
<b>12.6.5</b>	Proof Obligation for a SEES Clause	601
<b>13</b>	<b>Examples of Refinements</b>	<b>603</b>
<b>13.1</b>	A Library of Basic Machines	603
<b>13.1.1</b>	The <i>BASIC_CONSTANTS</i> Machine	604
<b>13.1.2</b>	The <i>BASIC_IO</i> Machine	604
<b>13.1.3</b>	The <i>BASIC_BOOL</i> Machine	605
<b>13.1.4</b>	The <i>BASIC_enum</i> Machine for Enumerated Sets	606
<b>13.1.5</b>	The <i>BASIC_FILE_VAR</i> Machine	607
<b>13.2</b>	Case Study: Data-base System	608
<b>13.2.1</b>	Machines for Files	611
<b>13.2.2</b>	Machines for Objects	623
<b>13.2.3</b>	A Data-base	630
<b>13.2.4</b>	Interfaces	637
<b>13.3</b>	A Library of Useful Abstract Machines	647
<b>13.3.1</b>	The <i>ARRAY_VAR</i> Machine	647

*Contents*

xxxiii

<b>13.3.2</b> The <i>SEQUENCE_VAR</i> Machine	647
<b>13.3.3</b> The <i>SET_VAR</i> Machine	647
<b>13.3.4</b> The <i>ARRAY_COLLECTION</i> Machine	648
<b>13.3.5</b> The <i>SEQUENCE_COLLECTION</i> Machine	648
<b>13.3.6</b> The <i>SET_COLLECTION</i> Machine	650
<b>13.3.7</b> The <i>TREE_VAR</i> Machine	650
<b>13.4</b> Case Study: Boiler Control System	655
<b>13.4.1</b> Introduction	655
<b>13.4.2</b> Informal Specification	656
<b>13.4.3</b> System Analysis	661
<b>13.4.4</b> System Synthesis	673
<b>13.4.5</b> Formal Specification and Design	676
<b>13.4.6</b> Final Architecture	693
<b>13.4.7</b> Modifying the Initial Specification	694
<i>Appendix A</i> <b>Summary of Notations</b>	701
A.1 Propositional Calculus	701
A.2 Predicate Calculus	702
A.3 Equality and Ordered Pairs	702
A.4 Basic and Derived Set Constructs	702
A.5 Binary Relations	703
A.6 Functions	705
A.7 Generalized Intersection and Union	706
A.8 Finiteness	706
A.9 Natural Numbers	707
A.10 Integers	709
A.11 Finite Sequences	711
A.12 Finite Trees	713
<i>Appendix B</i> <b>Syntax</b>	715
B.1 Predicate	715
B.2 Expression	716
B.3 Substitution	716
B.4 Machine	717
B.5 Refinement	719
B.6 Implementation	720
B.7 Statement	721
<i>Appendix C</i> <b>Definitions</b>	725
C.1 Logic Definitions	725
C.2 Basic Set-theoretic Definitions	726
C.3 Binary Relation Definitions	726
C.4 Function Definitions	728
C.5 Fixpoint Definitions	728
C.6 Finiteness Definitions	729
C.7 Natural Number Definitions	730
C.8 Integer Extensions	732
C.9 Finite Sequence Definitions	734

xxxiv	<i>Contents</i>	
	C.10 Finite Tree Definitions	736
	C.11 Well-founded Relation Definition	738
	C.12 Generalized Substitution Definitions	738
	C.13 Set-theoretic Models	741
	C.14 Refinement Conditions	742
	<i>Appendix D Visibility Rules</i>	743
	D.1 Visibility of a Machine	743
	D.2 Visibility of a Refinement	747
	D.3 Visibility of an Implementation	750
	<i>Appendix E Rules and Axioms</i>	753
	E.1 Non-freeness Rules	753
	E.2 Substitution Rules	754
	E.3 Basic Inference Rules	756
	E.4 Derived Inference Rules	758
	E.5 Set Axioms	760
	E.6 Generalized Substitution Axioms	761
	E.7 Loop Proof Rules	761
	E.8 Sequencing Proof Rule	762
	<i>Appendix F Proof Obligations</i>	763
	F.1 Machine Proof Obligations	763
	F.2 INCLUDES Proof Obligations	765
	F.3 USES Proof Obligations	767
	F.4 Refinement Proof Obligations	769
	F.5 Implementation Proof Obligations	771
	Index	775