The B-Book

The B-Book

Assigning Programs to Meanings

J.-R. Abrial

CAMBRIDGE
UNIVERSITY PRESS

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
Information on this title: www.cambridge.org/9780521496193

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

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 acccurate 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 magnificient 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. Sebilleau.

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.@estasl.inrets.fr`. A mailing list for this book is also available at `bbook.@estasl.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