AN INTRODUCTION TO FORMAL PROGRAM VERIFICATION

Ali Mili

AN INTRODUCTION TO FORMAL PROGRAM VERIFICATION

Ali Mili
Laval University

VAN NOSTRAND REINHOLD COMPANY
New York

Library of Congress Catalog Card Number: 84-3528
ISBN: 0-442-26322-8

Manufactured in the United States of America

Published by Van Nostrand Reinhold Company Inc.
135 West 50th Street
New York, New York 10020

Van Nostrand Reinhold Company Limited
Molly Millars Lane
Wokingham, Berkshire RG11 2PY, England

Van Nostrand Reinhold
480 Latrobe Street
Melbourne, Victoria 3000, Australia

Macmillan of Canada
Division of Gage Publishing Limited
164 Commander Boulevard
Agincourt, Ontario M1S 3C7, Canada

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

Library of Congress Cataloging in Publication Data

Mili, Ali.
An introduction to formal program verification.

Includes index.
1. Computer programs—Verification. I. Title.
QA76.6.M5217 1984 001.64'2 84-3528
ISBN 0-442-26322-8

Preface

Programmer X is trying to write a computer program. He cannot explain precisely what function his program is computing or what function he wants it to compute, but he is writing it anyway. His outer loop tests for a variable b being non-zero ($b \neq 0$) but he cannot tell whether testing for b greater than zero ($b > 0$) would work as well for his purposes; in fact he is not quite sure ($b \neq 0$) is the right condition to test for (perhaps $b > 0$ is).

In the loop body of the outer loop is a **while** statement using a variable t; variable t is initialized before the outer loop, though the programmer suspects he could also initialize it in the loop body, before the inner **while** statement. The index i of the inner **while** statement is initialized to 2 but the programmer suspects that, in order to take into account boundary values, he must set it to 1 and change the loop condition and the loop body accordingly. Since he is not sure, he has chosen to try it as is on non-boundary values then eventually change it to make it work for boundary values.

Inside the inner loop is a statement $t := a/c$; variable c is supposed to be non-zero before execution of this statement, but the programmer cannot tell you why; using a **goto** statement he has made sure that whenever c is zero, the control jumps outside both loops and sends an error message. The test of the inner loop is ($i \leq n$), where n is the size of an array; the programmer suspects that, in order to take into account the boundary condition $n = 1$ he should test for ($i < n$) and move the incrementation $i := i + 1$ inside the loop body; realizing that by doing so he causes an array reference out of bounds, the programmer has fixed the problem by merely increasing the size of the array by 1.

Now the program behaves reasonably well for the test values that the programmer has submitted to it; but the programmer can convince nobody (not even himself, in fact) of the worth of his program.

The situation just described is typical of a programmer who lacks the background needed for a firm intellectual control of the programming process. This background is not gained through an extensive practical experience of programming (generally, programming experience is not thought of as being

cumulative); rather it is gained through the formal study of the mathematical processes underlying the programming process. Before one can undertake this endeavor, one must come to terms with two simple premises we accept as facts:

— First, the premise that programs are mathematical objects about which assertions can be made and proven, and that programming is essentially a mathematical discipline. Indeed, computers do not behave randomly; rather, they behave in predictable, predefined ways. Advances in the semantic definition of programming languages afford us the means to capture the behavior of computers in ways that are both rigorous and usable.
— Second, the premise that it is through a consistent and disciplined use of the mathematics of programming that one can gain the necessary latitude to intellectually manage the programming process. Applying mathematical formalisms to every statement, for every functional aspect is neither practical nor necessary; it is however necessary to understand the precise mechanics of the programming process and the precise semantics of each statement one writes, and be ready to use this understanding.

In this book, I have tried to collect a number of program verification methods and present them, as much as possible, with a common set of assumptions, notations and concerns. These methods can be seen both as complementary and as alternatives. Because each deals with one aspect of program complexity, each is more appropriate than the others for a *specific* class of programs, and this in a manner depending on where the burden of the complexity lies within the program. From this standpoint they can be seen as alternatives. On the other hand, because they give different perspectives on programs and their executions, they can be seen as complementary.

Even though it is derived from course notes I have developed for a university course, this book is meant to be of interest both to university students and to practicing programmers. Having used manuscripts of this book to teach short courses to practicing programmers, I have been repeatedly surprised by the positive response of participants, who seem to find the material relevant to their concerns. Whatever the reader's background may be, the author hopes this book will help him shape (or reshape) his view of programming for the better.

ALI MILI

Introduction

This is an outgrowth of notes I have prepared for courses I have offered in formal program verification at Texas A&M University in College Station and later, concurrently at McGill University in Montreal and Laval University (francophone) in Quebec City. Even though it has some similarities with CS14—Software Design and Development—and CS20—Formal Methods in Programming Languages—in the ACM-recommended curriculum (see: Recommendations for Master's Level Programs in Computer Science, *CACM* 24(3), March 1981), this course was most often taught as a graduate level Special Topics Course. Its prerequisites are fairly modest: at least one semester of programming, using preferably a Pascal-like programming language; and a semester of discrete mathematics, as covered, e.g. by C L Liu's *Elements of Discrete Mathematics*.

This book is based on the content developed for a fifteen-week course. Hence many topics are not covered even though they are both intrinsically interesting and quite relevant to program verification. These are, e.g.: specifying and verifying properties of data structures; verifying parallel programs; specifying and verifying cyclic programs; modal and temporal logic; program transformations; dynamic logic; weakest preconditions and guarded commands. These are available in the literature and in advanced texts (see bibliographic references).

In contrast with some earlier works on the subject, this book favors using functions and relations rather than predicates to explain program correctness. Also, no effort is made in the book to show how powerful program verification is; i.e. we seldom prove the correctness of large programs. Rather, the book concentrates on showing what program verification *is*. This choice of priorities is motivated by practical concerns (to improve the readability of the book), and to some extent by its introductory nature.

The book is made up of five parts. Part I contains Chapters 1, 2 and 3 and lays the background for the remainder of the book. Chapter 1 provides some motivation for program verification and presents the global perspective of the book. Chapter 2 presents some elements of discrete mathematics. Because most

of the material introduced is presumed known by the reader, more emphasis is placed on naming conventions and notational conventions than on the material itself. Chapter 3 presents elements of logical expression and logical reasoning.

Part II contains Chapters 4, 5 and 6 and presents the basic formulas of program correctness. Chapter 4 gives the mathematical definition of a specification, Chapter 5 gives the mathematical definition of a program and Chapter 6 deduces the formulas of correctness of a program with respect to a specification and introduces the symbolic execution method.

Part III contains Chapters 7 through 11 and presents inductive methods for the proof of programs. A program is a multi-dimensional entity, with many axes of complexity; each verification method is concerned with one particular axis. Chapter 7 proves the correctness of programs by induction on their control structure and Chapter 8 uses induction on their data structure. Chapter 9 proceeds by induction on the length of execution whereas Chapter 10 proceeds by induction on the trace of execution. Finally, Chapter 11 discusses the semantics of recursive programs for which two inductive proof methods are presented.

Part IV contains Chapter 12, 13 and 14 and addresses the logical conclusion of program verification: program design. Chapters 12, 13 and 14 present models of program design on the basis of (respectively): predicate decomposition, functional decomposition and relational decomposition. They differ solely by how they represent specifications.

Part V presents three appendices. Appendix A is a BNF description of the programming language adopted in this book: SM-Pascal (standing for simple Pascal). Appendix B and Appendix C give some results on while statements which are intended to give the reader some more insight into the richness of iteration.

Each chapter has its own bibliography and practice set of problems, which are an extension of the course material, and each chapter is divided into sections. Each section has its own set of exercises, which are applications of the course material. Problems and exercises are labelled depending on their difficulty: A is easy, B is medium and C is difficult. For some short chapters, a single set of exercises is given at the end of the chapter, along with the set of problems.

The words *he, his* and *him* are used to denote a person of either sex.

Notations

lub	least upper bound.
iff	if and only if.
wrt	with respect to.
:	belongs to.
$\subseteq$	is a subset of.
$\subset$	is a proper subset of.
$\neq$	is different from.
$\leqslant$	is less than or equal.
$\geqslant$	is greater than or equal.
S|t	the subset of S for which predicate t holds.
:A	the predicate true for elements of A and false for all others.
$\times$	cartesian product of sets.
$\times\times$	cartesian power of sets.
*	numerical product, or relative product of relations (context distinguishes).
**	numerical power, or relative power of relations (context distinguishes).
$\wedge$	logical conjunction ("and").
$\vee$	logical disjunction ("or").
$\sim$	logical negation ("not").
$\Rightarrow$	logical consequence ("implies").
=	logical equivalence ("is equivalent to").
$\forall$	for all.
$\exists$	there exists.
$\bullet$	composition of expressions.
ite	alternative expression.
it	conditional expression.
def	domain of definition of an expression.
+	binary: addition.

@	unary: transitive closure.
−	difference.
∪	union.
∩	intersection.
F	full relation.
I	identity relation.
Eq	equivalence class.
f^{-1}	inverse of f.
R^t	transpose of R.
L*(S)	lists on space S.
(i → j)	path from i to j.
[i → j]	path function.
PD	Predicate Decomposition.
FD	Functional Decomposition.
RD	Relational Decomposition.

Contents

Preface v
Introduction vii
Notations ix

I. Motivation and Background 1

1. Motivation and Perspective 3
2. Mathematics for Programming 7
3. Formal Logic: Languages and Methods 29

II. Specification, Abstraction and Verification 56

4. Specification 59
5. Execution and Functional Abstraction 68
6. Program Correctness and the Symbolic Execution Method 95

III. Verifying Programs by Induction 107

7. Induction on the Control Structure: The Invariant Assertion Method 109
8. Induction on the Data Structure: The Intermittent Assertion Method 146
9. Induction on the Length of Execution: The Subgoal Induction Method 163
10. Induction on the Trace of Execution: The Cutpoint Method 175
11. Proof of Recursive Functions 190

IV. Formal Program Design 213

12. Predicate Decomposition 215
13. Functional Decomposition 229
14. Relational Decomposition 245

V. Appendices 259

A. The BNF Syntax of SM-Pascal 261
B. Strongest Invariant Functions 267
C. The Self-Stabilizing Effect of While Statements 278

Index 285

Part I: Motivation and Background

> It is reasonable to hope that the relationship between Computation and Mathematical Logic will be as fruitful in the next century as that between Analysis and Physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.
>
> *John McCarthy, 1963.*

Chapter 1 provides some motivation for the verification of programs and presents the perspective and tone of this book. Chapter 2 introduces some elements of discrete mathematics. Chapter 3 presents elements of logical expression (propositions, predicates) and reasoning (induction and deduction).

Chapter 1

Motivation and Perspective

The purpose of this chapter is to highlight the mathematical nature of programs then discuss the perspective of this book with respect to the use of mathematics in program verification.

1 On the Richness of Programs

In this section we use a simple example to show how potentially rich (complex) programs are. We consider the following two programs on integer variables a, b and p:

$$M = (\textbf{while } b \neq 0 \textbf{ do begin } p := p+a;\ b := b-1 \textbf{ end}).$$
$$M' = (\textbf{while } b > 0 \textbf{ do begin } p := p+a;\ b := b-1 \textbf{ end}).$$

We wish to discuss the following questions:

a) Prove that if $(a = a0 \wedge b = b0 \wedge p = 0 \wedge b0 \geqslant 0)$ before execution of M then M terminates and $p = a0 * b0$ after.
b) Let a0, b0 and p0 be the values of variables a, b and p before execution of M. What are the values of a, b and p after execution of M?
c) Same question as (a), for program M′.
d) Same question as (b), for program M′.

Answers to these questions are briefly given below:

a) Claim: The assertion $(a0 * b0 = p + a * b)$ holds after any number (including zero) of iterations.
This claim can be proved by induction.
Basis of induction: $a = a0 \wedge b = b0 \wedge p = 0 \wedge b0 \geqslant 0 \Rightarrow a0 * b0 = p + a * b$.
Induction step: $a0 * b0 = p + a * b \Rightarrow (a0 * b0 = p + a + a * (b-1))$. Because $b0 \geqslant 0$ and because b decreases by decrements of 1, M terminates in a state such that $b = 0$. This, in conjunction with the assertion above yields the result sought.

b) If $b0 \geq 0$ then the final values of a, b and p are

$$a = a0,\ b = 0,\ p = p0 + a0 * b0$$

else the final values of a, b and p are undefined.

c) Claim: The assertion $A = (a0 * b0 = p + a * b \wedge b \geq 0)$ holds after any number (including zero) of iterations.
We prove this claim by induction:
Basis of induction: $a = a0 \wedge b = b0 \wedge p = 0 \wedge b0 \geq 0 \Rightarrow a0 * b0 = p + a * b \wedge b \geq 0$.
Induction step: If $A = (a0 * b0 = p + a * b \wedge b \geq 0)$ holds at some iteration and one more iteration is needed $(b > 0)$ then A holds after the iteration $(a0 * b0 = p + a + a * (b - 1) \wedge b - 1 \geq 0)$. Because $b0 \geq 0$ and b decreases at each step with a decrement of 1 the condition $b > 0$ eventually becomes false and the program exits; the conjunction of the two conditions $(a0 * b0 = p + a * b \wedge b \geq 0)$ and $(b \leq 0)$ yields $p = a0 * b0$.

d) If $b0 \geq 0$ then the final values of a, b and p are

$$a = a0,\ b = 0,\ p = p0 + a0 * b0$$

else

$$a = a0,\ b = b0,\ p = p0.$$

One may draw two lessons from the discussions above:

— Even simple programs are hard to analyze: The discussions above are not as straightforward and simple as the shortness and simplicity of programs M and M′ may lead one to believe.
— Even simple changes to a program ($b \neq 0$ vs. $b > 0$) can cause a deep impact on its functional properties; hence the need for programmers to be aware of the significance of every single symbol they write in their program.

2 Programming as a Mathematical Discipline

Programs are potentially complex objects. How does one tackle their complexity? A cogent answer is given by Dijkstra ([1]):

> As soon as programming emerges as a battle against unmastered complexity, it is quite natural that one turns to that mental discipline whose main pur-

pose has been for centuries to apply effective structuring to otherwise unmastered complexity. That mental discipline is more or less familiar to all of us, it is called Mathematics. If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed the most effective way to come to grips with complexity, we have no choice any longer: We should reshape our field of programming in such a way that, the mathematician's methods become equally applicable to our programming problems, for there are no other means.

Mathematics (discrete mathematics, in particular) plays a dual role in the functional analysis of programs, by enabling us to firmly grasp all the richness of programs and to harness the complexity of programs by means of proper structuring.

The mastery of the Mathematics underlying the programming activity is a key to effective program development. It is the basis for the *intellectual control* of the programming process (notion due to Mills, [2]); this intellectual control is virtually the only source of confidence that a programmer is entitled to have in his program. No amount of testing can certify the correctness of a program because in general the number of possible test values is virtually infinite.

Of course, the mathematical validation criteria that we use to prove programs correct have intrinsic limitations. The basic incompleteness results (Halting Problem, Church's Thesis, . . .) already tell us that no absolute validation criteria exist. But relative criteria do exist and some are quite effective. There is a strong analogy here with physical theories which can be absolutely refuted by counterexample but never absolutely validated through empirical confirmation (this did not prevent a relationship between mathematical analysis and physics to develop and be fruitful).

3 Perspective

In the following chapters, we shall discuss several aspects of the functional analysis of programs, along the various dimensions of complexity that a program presents. In doing so, we shall use mathematics to formalize the functional properties of programs, with an attempt to seek a proper balance between formality and intuition.

Through its wide range of program verification (analysis) methods, the book seeks to achieve the following goals:

— Present and discuss effective and reliable means to verify that a program is consistent with respect to the specification for which it was written.

— Shed light into the essence of automatic computations and the mathematical processes underlying the programming activity.
— Provide guidelines for the formal design of programs.

It is the second goal which is considered to be the most important one: It is both the most fundamental one—for it encompasses the others—and the most readily applicable one—for some theorems are so powerful that one cannot keep programming the same way after one has understood them. Fortunately, because of its logical nature, this understanding of programming scales up to programs of any size.

Bibliography

1: Dijkstra, E W. On a Methodology of Design. MC-25 Informatica Symposium, MC track 37. Mathematical Centrum, Amsterdam, 1971 pp 4.1-4.10.
2: Mills, H D. The Intellectual Control of Computers. Keynote address, The International Symposium on Current Issues of Requirements Engineering Environments. Kyoto, Japan, September 20-21 1982. Yutaka Ohno, editor. North-Holland, 1982.