CURRENT TRENDS IN PROGRAMMING METHODOLOGY

VOLUME I

Software Specification and Design

RAYMOND T. YEH, *Editor*

CURRENT TRENDS IN PROGRAMMING METHODOLOGY

VOLUME I

Software Specification and Design

RAYMOND T. YEH, *Editor*

Department of Computer Sciences
The University of Texas at Austin

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Main entry under title

Current trends in programming methodology

Bibliography v 1
Includes index.
CONTENTS v 1 Software specifications and design —
1 Programming (Electronic computers) I Yeh, Raymond Tzuu-Yau (date)
QA76.6.C87 001 6'42 76-46467
ISBN 0-13-195701-5 (v 1)

10 9 8 7 6 5 4 3

Printed in the United States of America

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

PREFACE

Programming is a problem-solving activity. As such, it must deal with a whole spectrum of activities concerned with specifications, design, validation, modeling, and structuring of programs and data. The purpose of this series in programming methodology is intended to bring together a collection of tutorial papers in each volume which are representative of the current trends in one of the above mentioned specific subject areas of the programming activities.

The intention of this first volume is to survey recent developments of programming principles and techniques for the systematic design of well-structured and reliable software architecture.

Large programs are clearly among the most complex creations of human intellect. Our human inability in coping with the complexity has been a major factor contributing to software unreliability. In the past few years there has been much effort devoted to the development of techniques for the systematic design of well-structured software architecture. All of these techniques seem to be pointing to the concept of *modularity*. Several principles have emerged for the construction of useful modules.

The notion of *abstraction* is a powerful tool to combat this complexity. The principle of abstraction is concerned with the selection of essential properties and omitting inessential ones. In most programming languages, there appear many implicit abstractions which are built into the language, for example, the notions of procedure, array, stack, etc. The recent research trend is for the development of explicit abstractions, i.e., those introduced by the programmer. Through explicit abstraction, a program can be decomposed into hierarchical structures and built up from "modules" or units of abstraction, and is easier to design, implement, and maintain. The principle of abstraction used alone does not always yield useful decomposition. However, together with two other principles, they form a set of powerful tools for the modularization of programs.

Another principle is that of *information hiding*. The purpose of hiding not only requires making visible certain essential properties of a module, but making inaccessible certain nonessential details that should not affect other parts of a system.

In Chapters 1 and 2, the concept of *data abstraction* is introduced. A data abstraction is comprised of a group of related operations that act on a particular class of objects with the constraint that the behavior of the objects can be observed only by the application of the operations. In Chapter 1, the role of formal specification in the program construction process is explained. It surveys some formal specification techniques for describing data abstractions, and possible future research is also discussed.

Chapter 2 is concerned with the relation between programming languages and the "structure" of programs. The thesis of this chapter is that programming languages, being the vehicle of expressing a programmer's thought, have strong influence over the way of a programmer's thinking when constructing a program. The chapter surveys the features of existing programming languages which aid the programmer in his ability to abstract. Finally, it introduces a powerful language feature called "form" as a way of representing an abstract concept.

Chapter 3 presents a general methodology for the design, implementation, and proof of large software systems. The basis of this methodology lies in the use of a formal specification technique to describe a software system as a hierarchy of abstract machines such that each abstract machine can be formally specified as a module (in the sense described in Chapters 1 and 2).

Chapter 4 deals with general observations on the improvement of software reliability in the design process assuming the software structure is characterized by the interfaces between modules. It suggests some techniques by considering certain unpleasant facts of life in the specification in the early stages of the design process for the improvement of reliability.

The first four chapters thus represent an integrated treatment of software design with the principles of abstraction (especially data abstraction) and hiding as its foundation.

Another principle is that of *localization*. This principle is concerned with textual locality. Thus, subroutines and arrays are examples of localizations. This principle applied to control structure results in the so-called structured program. A structured program is one in which the text can be grouped into a set of vested regions such that each region has a single entry and single exit. Note that the single-entry/single-exit regions of a structured program need to be in correspondence with abstractions to be useful because the computational abstractions realized in these ways are close textually. Such a programming technique attempts to systematically place restrictions on the developmental process of programs. The purpose of these restrictions is to allow the programmer to control the complexity in the program development, and hence construct well-structured programs.

In Chapter 5, an operational procedure is described for the development of large reliable programs based on the top-down programming technique. It introduces a systematic procedure to restructure programs by decomposing programming work into logical and clerical components as a basis for systematic program development and quality control. Such a procedure turns out to be an excellent management tool for large software projects.

Chapter 6 consists of a lengthy discussion of the true nature of structured programming. It brings out opposing points of view about whether or not goto statements should be abolished. In the flow of this discussion, a methodology of program design by systematic transformation is introduced. It also focuses on improved syntax for iteration and error exit as an aid for writing a large class of programs with understandability and efficiency without the use of goto statements.

Chapter 7 presents a design methodology for fault-tolerant software. The main idea of the method consists of structuring a program into *recovery blocks*, each consisting of *primary* and *several alternatives*. The alternatives are built-in "redundancy" for the prevention of and recovery from failures. Each recovery block has an associated *acceptance test* which specifies what is expected of a successful execution of this block. When invoked, the primary block is executed, followed by an acceptance test. If the test is passed, control is returned to the invoking sequence. Otherwise, the system state is restored to a state just prior to the execution of the primary block and an alternative is executed, etc. If no alternative satisfies the acceptance test, then an error message is passed to the invoking sequence and perhaps an alternative of this sequence will be initiated, etc. This chapter provides an engineering tool in software system construction so that tolerance is built in at the design phase.

Chapter 8 discusses a programming technique in which the central agent of process control is a table of control records. It is shown that control-record-driven processing is based on a top-down approach to the design, and that it is particularly applicable as a design tool for programming that leads to jumping about in an irregular manner during the course of action.

In Chapter 9, a constructive approach to nondeterministic programming is introduced. "Guarded commands" are introduced as building blocks for alternative and repetitive constructs. Program "semantics" is formally defined by means of a "predicate transformer" which transforms a set of program states after the execution of a program to the set of all possible states before the execution of the same program. Thus, the distinction between determinism and nondeterminism is no longer significant in this semantic context. The focus is on the nature of computation and hence the concept of iteration, and not how the program iterates. A calculus is introduced for the formal derivation of programs expressed in terms of these constructs.

Finally, a comprehensive annotated bibliography is included which provides necessary pointers to literature.

This book can be used as a textbook by upper-division undergraduate or first year graduate students in computer science or computer engineering as a course in programming methodology or software systems design. It should, however, be supplemented by exercises and projects. It should serve as an excellent reference book for professional software engineers.

I would like to take this opportunity to thank all contributing authors, and Ann Marmor-Squires for annotating the bibliography. Thanks are also due to John Wang who laboriously combined all bibliographic entries into an integrated guide.

RAYMOND T. YEH

Austin, Texas

CONTENTS

CHAPTER 1

AN INTRODUCTION TO FORMAL SPECIFICATIONS OF DATA ABSTRACTIONS

BARBARA LISKOV
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

STEPHEN ZILLES
IBM Research
San Jose, California

Abstract

The main purposes of this chapter are to discuss the importance of formal specifications and to survey a number of promising specification techniques. The role of formal specifications both in proofs of program correctness, and in programming methodologies leading to programs which are correct by construction, is explained. Some criteria are established for evaluating the practical potential of specification techniques. The importance of providing specifications at the right level of abstraction is discussed, and a particularly interesting class of specification techniques, those used to construct specifications of data abstractions, is identified. A number of specification techniques for describing data abstractions are surveyed and evaluated with respect to the criteria. Finally, directions for future research are indicated.

Key Terms

specifications; specification techniques; data abstractions; proofs of correctness; programming methodology.

This work was supported in part by the IBM funds for research in computer science.

1.1. INTRODUCTION

In the past, the advantages of formal specifications have been outweighed by the difficulty of constructing them for practical programs. However, recent work in programming methodology has identified a program unit, supporting a data abstraction, that is both widely useful and for which it is practical to write formal specifications. Some formal specification techiques have already been developed for describing data abstractions. It is the promise of these techniques, some of which are described later in the chapter, which leads us to believe that formal specifications can soon become an intrinsic feature of the program construction process. By writing this material, we hope to encourage research in the development of formal specification techniques and their application to practical program construction.

In the remainder of the introduction we discuss what is meant by formal specifications and then explain some advantages arising from their use. In Section 1.2, a number of criteria are presented that permit us to judge techniques for constructing formal specifications. Section 1.3 identifies the kind of program unit, supporting a data abstraction, to which the specification techniques described later in the chapter apply. Section 1.4 discusses properties of specification techniques for data abstractions, and, in Section 1.5, some existing techniques for providing specifications for data abstractions are surveyed and compared. Finally, we conclude by pointing out areas for future research.[1]

Proofs of Correctness

Of serious concern in software construction are techniques that permit us to recognize whether a given program is correct, i.e., does what it is supposed to do. Although we are coming to realize that correctness is not the only desirable property of reliable software, it is surely the most fundamental: If a program is not correct, then its other properties (e.g., efficiency, fault tolerance) have no meaning since we cannot depend on them.

Techniques for establishing the correctness of programs may be classified as formal or informal. All techniques in common use today (debugging, testing, program reading) are informal techniques; either the investigation of the properties of the program is incomplete or the steps in the reasoning place too much dependence on human ingenuity and intuition. The continued existence of errors in software to which such techniques have been applied attests to their inadequacy. Formal techniques, such as the verification condition (Floyd [1967] and Hoare [1969]) and fixed-point (Manna [1973]) methods, attempt to establish properties of a program with respect to all legitimate inputs by means of a process of reasoning in which each step is formally justified by appeal to rules of inference, axioms and theorems. Unfortunately, these techniques

[1]This chapter is an expanded version of earlier material published by the authors Liskov and Zilles [1975]. It differs from the earlier publication primarily in Section 1.5, which has been substantially enlarged and contains additional examples.

have been very difficult to apply and have therefore not yet been of much practical interest. However, interest in formal techniques can be expected to increase in the future; economic pressure for reliable software is growing (Boehm [May 1973]), and the domain of applicability of formal techniques is also growing because of the development of programming methodologies leading to programs to which formal techniques are more readily applied. Indeed, application of proof techniques to practical programs is being attempted in the area of operating system security (Schroeder [1975] and Price [1973] and Neumann [1974]), where the need for absolute certainty about the correct functioning of software is very great.

To study techniques that establish program correctness, it is interesting to examine a model of what the correctness of a program means. What we are looking for is a process that establishes that a program correctly implements a *concept* that exists in someone's mind. The concept can usually be implemented by many programs—an infinite number, in general—but of these only a small, finite number are of practical interest. This situation is shown in Figure 1.1. In current practice, the concept is stated informally and, regardless of the technique used to demonstrate the correctness of a program (usually testing), the result of applying the technique can be stated only in informal terms.

$$\begin{tikzcd}
& \text{Concept} \arrow[dl] \arrow[dr] & \\
P_1 & \cdots & P_n
\end{tikzcd}$$

Figure 1.1. A concept and all programs which implement the concept correctly.

With formal techniques, a *specification* is interposed between the concept and the programs. Its purpose is to provide a mathematical description of the concept, and the correctness of a program is established by proving that it is equivalent to the specification. The specification will be provably satisfied by a class of programs (again, often an infinite number of which only a small, finite number are of interest). This situation is shown in Figure 1.2.

$$\begin{tikzcd}
& \text{Concept} \arrow[d] & \\
& \text{Specification} \arrow[dl] \arrow[dr] & \\
Q_1 & & Q_m
\end{tikzcd}$$

Figure 1.2. A concept, its formal specification, and all programs which can be proved equivalent to the specification.

Proofs of large programs do not consist of a single monolithic proof with no interior structure. Instead, the overall proof is divided into a hierarchy of many smaller proofs which establish the correctness of separate program units. For each program unit, a proof is given that it satisfies its specification; this proof makes use of the specifications of other program units, and rests on the assumption that those program units will be proved consistent with their specifications.[2] Thus a specification is used in two ways: as a description against which a program is proved correct, and as a set of axioms in the proof of other programs. At the top of the proof hierarchy is a program unit which corresponds to the entire program. At the bottom is the programming language, and the hierarchy is based on the axioms for the programming language and its primitives.

The proof methodology can fail in two ways. First, a proof may incorrectly establish some program (or program unit) P as equivalent to the specification when, in fact, it is not. This is a problem which can be eliminated by using a computer as, at least, a proof checker. (Observe that one advantage of using formal specifications is that such specifications can be processed by a computer.)

The second way the methodology can fail is if the specification does not correctly capture the meaning of a concept. We will say a specification *captures* a concept if every Q_i in Figure 1.2 is some P_j in Figure 1.1. There is no formal way of establishing that a specification captures a concept, but we expect to have gained from using the proof methodology because (hopefully) a specification is easier to understand than a program, so that "convincing oneself" that a specification captures a concept is less error-prone than a similar process applied to a program. Furthermore, any distinction between concept and specification may be irrelevant because of the hierarchical nature of the proof process. If a program P is proved equivalent to its specification, and every program using P is proved correct using that specification, then the concept that P was intended to implement can safely be ignored.

Advantages of Formal Specifications

Proving the correctness of programs as described above is a two-step process: first, a formal specification is provided to describe the concept, and second, the program is proved equivalent to the specification by formal, analytic means. Formal techniques are not necessarily limited to axiomatic methods. For example, it may also be possible to develop testing methodologies that are based on a comparison of the formal specification and the implementation. The output of a methodology would be a set of critical test cases which, if successfully executed, establish that the program correctly implements the specification. The formality of the specification means that the computer can aid in the proof process, for example, by checking the steps of a program proof, or by automatically generating test cases.

Clearly, the specification must be present before a proof can be given. However, formal specifications are of interest even if not followed by a formal proof. Formal

[2]Special techniques (Manna [1973]) must be used if the program units are mutually recursive.

specifications are very valuable in conjunction with the idea of making code "public" (Baker [January 1972]) in order to encourage programmers to read one another's codes. In the absence of a formal specification, a programmer can only compare a program he is reading with his intuitive understanding of what the program is supposed to do. A formal specification would be better, since intuition is often unreliable. With the addition of formal specifications, code reading becomes an informal proof technique; each step in the proof process now rests on understanding a formal description rather than manipulating the description in a formal way.[3] As such, it can be a powerful aid in establishing program correctness.

Formal specifications can also play a major role while a program is being constructed. It is widely recognized that a specification of what a program is intended to do should be given before the program is actually coded, both to aid understanding of the concept involved, and to increase the likelihood that the program, when implemented, will perform the intended function. However, because it is difficult to construct specifications using informal techniques, such as English, specifications are often omitted, or are given in a sketchy and incomplete manner. Formal specification techniques, like the ones to be described later in this chapter, provide a concise and well-understood specification or design language, which should reduce the difficulty of constructing specifications.

Formal specifications are superior to informal ones as a communication medium. The specifications developed during the design process serve to communicate the intentions of the designer of a program to its implementors, or to communicate between two programmers: the programmer implementing the program being specified, and the programmer who wishes to use that program. Problems arise if the specification is ambiguous: that is, if it fails for some reason to capture the concept so that two programs with different conceptual properties both satisfy the specification. Ambiguities can be resolved by mutual agreement, provided those using the specification realize that an ambiguity exists. Often this is not realized, and instead the ambiguity is resolved in different ways by different people. Formal specifications are less likely to be ambiguous than informal ones because they are written in an unambiguous language. Also, the meaning of a formal specification is understood in a formal way, and therefore ambiguities are more likely to be recognized.

The above paragraphs have sketched a program construction methodology that could lead to programs which are correct by construction. Formal specifications play a major role in this methodology, which differs from standard descriptions of structured programming (Dijkstra [1972]) primarily in the emphasis it places on specifications.[4] Specifications are first introduced by the designer to describe the concepts he develops in a precise and unambiguous way. Each concept will be supported by a

[3]The relationship between proofs and understanding is a major motivating factor in structured programming. For example, the "go to" statement is eliminated because the remaining control structures are each associated with a well-known proof technique, and therefore the programs are intellectually manageable (Dijkstra [1972]).

[4]See the publication by Hoare [1971] for a structured programming example in which specifications are emphasized.

program module. The specifications are used as a communication medium among the designers and the implementors to insure both that an implementor understands the designer's intentions about a program module he is coding, and that two implementors agree about the interface between their modules. Finally, the correctness of the program is proved in the hierarchical fashion described earlier. The method of proof may be either formal or informal, and the proofs can be carried out as the modules are developed, rather than waiting for the entire program to be coded. Progress in developing formal specification techniques will enhance the practicality of applying this methodology to the construction of large programs.

1.2. CRITERIA FOR EVALUATING SPECIFICATION METHODS

An approach to specification must satisfy a number of requirements if it is to be useful. Since one of the most important goals of specification techniques is to permit the writing of specifications for practical programs, the criteria described below include practical as well as theoretical considerations.

We consider that the first criterion must be satisfied by any specification technique:

1. *Formality.* A specification method should be formal, that is, specifications should be written in a notation which is mathematically sound. This criterion is mandatory if the specifications are to be used in conjunction with proofs of program correctness. In addition, formal specification techniques can be studied mathematically, so that other interesting questions, such as the equivalence of two specifications, may be posed and answered. Finally, formal specifications are capable of being understood by computers, and automatic processing of specifications should be of increasing importance in the future.

The next two criteria address the fundamental problem with specification techniques—the difficulty encountered in using them.

2. *Constructibility.* It must be possible to construct specifications without undue difficulty. We assume that the writer of the specification understands both the specification technique and the concept to be specified. Two facets of the construction process are of interest here: the difficulty of constructing a specification in the first place, and the difficulty in knowing that the specification captures the concept.
3. *Comprehensibility.* A person trained in the notation being used should be able to read a specification and then, with a minimum of difficulty, reconstruct the concept which the specification is intended to describe. Here (and in criterion 2) we have a subjective measure in mind in which the difficulty

encountered in constructing or reading a specification is compared with the inherent complexity (as intuitively felt) of the concept being specified. Properties of specifications which determine comprehensibility are size and lucidity. Clearly, small specifications are good since they are (usually) easier to understand than larger ones. For example, it would be nice if a specification were substantially smaller than the program it specifies. However, even if the specification is large, it may still be easier to understand than the program because its description of the concept is more lucid.

The final three criteria address the flexibility and generality of the specification technique. It is likely that techniques satisfying these criteria will meet criteria 2 and 3 as well.

4. *Minimality.* It should be possible using the specification method to construct specifications which describe the interesting properties of the concept and *nothing more.* The properties which are of interest must be described precisely and unambiguously but in a way which adds as little extraneous information as possible. In particular, a specification must say *what* function(s) a program should perform, but little, if anything, about how the function is performed. One reason this criterion is desirable is because it minimizes correctness proofs by reducing the number of properties to be proved.
5. *Wide Range of Applicability.* Associated with each specification technique there is a class of concepts which the technique can describe in a natural and straightforward fashion, leading to specifications satisfying criteria 2 and 3. Concepts outside of the class can only be defined with difficulty, if they can be defined at all (for example, concepts involving parallelism will not be describable by any of the techniques discussed later in the chapter). Clearly, the larger the class of concepts which may be easily described by a technique, the more useful the technique.
6. *Extensibility.* It is desirable that a minimal change in a concept result in a similar small change in its specification. This criterion especially impacts the constructibility of specifications.

1.3. THE SPECIFICATION UNIT

The quality of a specification (the extent to which it satisfies the criteria of the preceding section) is dependent in large part on the program unit being specified. If a specification is attached to too small a unit—for example, a single statement—what the specification says may be uninteresting, and furthermore there will be more specifications than can conveniently be handled. The specification could express no more than the following comment, sometimes seen in programs:

```
x := x + 1,  "increase x by 1"
```