other relevant fundamental results such as Salqvist’s correspondence theorem, which allows characterizing classes of models using certain temporal formulae, and the connection between expressive completeness and Ehrenfeucht games. One exception to the general breadth of this book is the theory of finite automata over infinite words. This is only briefly covered and the reader is referred for results to relevant research papers. This theory is one of the most important tools used today in the research and applications of linear temporal logic, and does not receive here the coverage it deserves.

The book is not an easy-to-read introduction to temporal logic, and some existing knowledge of temporal logics is useful; it is geared more towards experts in temporal logic who want to broaden their knowledge. It would be hard to use this book as a textbook for an undergraduate level course. However, it can be used as a teacher’s reference for an advanced graduate level course on temporal logic.

The style of writing in this book is similar to that of a research paper. It contains proofs for most of the theorems presented. The proofs are comprehensible, but are not always trivial to follow, and some maturity in understanding proofs in logic can be very helpful. It is natural for an exposition of this length to have a number of typos, but it is hoped they will be corrected in later printings. In the typesetting, the same fonts are used for temporal operators and propositions, making such expressions sometimes difficult to parse. The bibliography deserves some more attention in order to be more useful for locating the relevant referenced papers.

In general, this new book is a very comprehensive study of temporal logics, their expressiveness and axiomatization. it is not an easy text for beginners in this area; it is very thorough and presents the subjects of its focus in great depth. It can point practitioners of this field to existing results and thus is recommended as a good research reference.

D. Peled
AT&T Bell Laboratories

Chris Casey
262pp. softbound

Development of large software is complex, time consuming and error prone. Many software engineering methodologies and techniques have been proposed to lessen these problems. But most of these methods are informal: except for the final code produced, the initial specifications and the intermediate designs are stated using imprecise, ambiguous and verbose natural language and/or diagrams and can lead to inconsistent descriptions whose completeness is uncertain. With an informal notation, we cannot talk about consistency and completeness of descriptions, quite apart from the problem of checking for these properties. Testing, used as a technique for ensuring that the produced software is free of errors, only enables one to uncover bugs. Except in some special classes of programs, exhaustive testing to certify that the developed systems is free of errors is impossible and/or infeasible.

Formal methods for software development have been proposed to overcome these deficiencies. Two important ingredients of any formal method are the use of a formal language for specifications and intermediate designs and the use of mathematical techniques for establishing that systems meet their specification. Further, the testing phase is replaced by formal verification, which involves showing rigorously, using mathematical techniques, that a given system meets its specification.

Formal methods have been in existence for quite some time but remain somewhat inaccessible to an average programmer or to students not having much background in mathematics. The aim of this text book is to introduce formal methods at a level that can be easily assimilated and used by these groups of people; as the name suggests, it adopts a programming approach to illustrate the concepts and techniques of formal methods. The author succeeds, to a greater extent, in this aim.

The book starts with a survey of the concepts and features of programming languages and software engineering, before getting into the basics of formal methods. This survey is rather brief and may serve only as a refresher intended for reviewing concepts relevant for the rest of the book.

The basic concepts of formal methods are introduced and illustrated in the next three chapters. Quite a number of important topics and techniques are touched upon in these chapters. These include symbolic logic as a specification language, correctness proofs of programs, loop invariants, proof by induction, recursion and termination. It is interesting to note that a working knowledge of many of these topics could be gained in the short span of these three chapters. All the concepts are well illustrated with a number of simple examples.

After this general introduction to the concepts of formal methods, the book gets into some details of a specific formal method, namely VDM. (This transition from generality to specifics is rather sudden and could probably have been avoided.) Four important abstract data types of VDM are discussed and their use is illustrated with a number of examples.

The important topic of data type refinement is also discussed and illustrated using a small example. This discussion is primarily in the context of VDM. One chapter is devoted to the use of formal methods in developing designs from specifications, testing and documentation. A complete chapter is devoted to a case study that illustrates the use of formal methods in developing part of a simple information system.

A nice feature about the book is that it contains a
detailed annotated bibliography in the appendix that provides pointers to reference material for a detailed study of many of the basic concepts introduced in the book.

The book is well-written and intentionally pitched at a very low level, making it suitable for undergraduates and practising programmers. It can serve as useful material for a first level course in formal methods and may provide enough confidence to enable the reader to undertake an advanced and more detailed study of the intricacies of formal methods.

S. RAMESH
Indian Institute of Technology, Bombay

DAVID A. SCHMIDT

David Schmidt's new book brings together a large body of recent research material on programming language semantics and type theory. He presents this material in a carefully integrated fashion, and the result is a book that I expect to consult frequently for many years to come.

Chapter 1 introduces the syntax, typing rules, and semantics of a core programming language, with expressions and commands but not (yet) declarations. Chapters 2–4 successively add declarations, parameters and blocks, extending the original core language to a substantial imperative language with relatively advanced features such as parameterized modules. In these three chapters Schmidt studies the abstraction, parameterization, correspondence, and qualification principles in detail. These semantic principles, originally due to Landin and Tennent, suggest ways in which a programming language can be systematically extended. By studying the semantics of languages extended in these ways, Schmidt demonstrates clearly that the principles lead to languages that are exceptionally elegant and powerful.

The next two chapters represent an abrupt change of direction, which I personally found a bit disorienting. Chapter 5 demonstrates that the essential features of a programming language may be understood as a core language extended by record and lambda-abstraction constructions. Chapter 6 presents well-known results about the lambda calculus itself, the simplest conceivable programming language.

Chapter 7 then builds a functional programming language, starting with an applicative core language, and then extending it systematically in a similar manner to Chapters 2–4. This development serves as a second example of the recommended approach to language design, and also gives an opportunity to explore polymorphism and ML-style type inference. The latter entails unification of type terms, which Schmidt rather cutely links into a short study of a Prolog-like logic language, whose semantics entails unification of value terms.

Chapter 8 summarizes recent research into the implications of types-as-values, through the medium of higher-order lambda calculus. Chapters 9–10 conclude the book by exploring intuitionistic type theory, which is based on a striking analogy: the types of a programming language can be viewed as propositions, and the programs themselves can be viewed as proofs.

Schmidt's presentation of denotational semantics is unusual in that he defines the semantics, not of plain phrases, but of phrases attributed by their types. For example, his semantic equation for the core language's while-command is:

\[
\text{while } E \text{ do } C \text{ od} : \text{comm} = \{v(s) = \text{if}(\{E : \text{boolexp}\}(s), \text{w}(\{C : \text{comm}\}(s)), s) \}
\]

Here \(E\) : boolexp stands for a well-typed expression \(E\) of type bool, and \(C\) : comm stands for a well-typed command \(C\). This notation enables us to state explicitly that the meaning of an expression \(E\) : rexpr is a function in domain \(\text{Store} \to [\tau]\) (where \([\tau]\) is the function of values of type \(\tau\)). The conventional notation of denotational semantics enables us to state, less precisely, that the meaning of an expression of any type is a function in domain \(\text{Store} \to \text{Value}\). The extra precision is essential to what follows. For example, an important check on the semantics of a typed programming language is whether evaluation of an expression of type \(rexpr\) is guaranteed to yield a value of type \(\tau\) (the subject reduction property), and with no possibility of run-time type errors (the strong typing property). Schmidt shows how to prove these and other properties.

Notational overload, however, soon becomes a significant problem for the reader. As soon as the core language is extended by declarations, phrases such as expressions must be attributed by 'type assignments' \(\tau\) as well as types. The semantic equation for a while-command becomes rather indigestible:

\[
\text{while } E \text{ do } C \text{ od} : \text{comm} = \{v(s) = \text{if}(\{E : \text{boolexp}\}(s), \text{w}(\{C : \text{comm}\}(s)), s) \}
\]

The reader will also have difficulty in making sense of the diagrams of abstract syntax trees and proof trees. Unaccountably, Schmidt has 'drawn' the tree branches using '(', ')', and '/' characters, which are often hopelessly misaligned. The diagrams are therefore quite unintelligible. The publisher should address this defect as a matter of priority.

My other criticism is a lack of clarity about the book's intended readership. The back cover claims that the book is 'designed for use in a first or second level course on principles of programming languages'. The preface is silent on the intended readership, but claims that instructors who prefer to discuss semantics informally could just skip the sections with 'semantics' in their titles.