Type systems are defined in a similar style—a typed program is one whose type annotations satisfy certain invariants. Type inference is viewed as the minimum solution of constraints derivable from the program. Types are defined as sets of classes which a value might belong to; operations accepting any class derived from an original class are seen as accepting the cone consisting of the set of all such derived classes. The syntactic explanations of inheritance and genericity above can be thus shown to be type correct in a pleasing manner. Palsberg and Schwartzbach successfully (to my mind) argue the need to explain object-oriented semantics as discussed rather than to force-fit the ideas to a $\lambda$-calculus meta-language (note also that Abadi and Cardelli have since proposed a calculus of objects which shares a similar aim but focuses on more theoretical issues concerning typing). This action encourages them to propose that genericity be explained by substitution of classes in others rather than by the notion of application (concerning typing). The result acts as a contract between customer and system developer whose properties may be either derived by proof or observed by prototyping. Applied to the latest stages of the design cycle, formal methods provide techniques for producing correct code: for the correct termination of loops, invocation of procedures, and so on. In between those extremes of abstraction they offer criteria for ensuring the functional correctness of one description with respect to a more abstract one. The lower-level description whilst chosen to incorporate bottom-up constraints and efficiency concerns must meet those criteria to be valid.

Thus crucial to formal methods is a formal description, or specification, at a given level of abstraction. How it is constructed depends on its location in the development cycle; techniques for requirements capture are quite different from those for downcoding into a programming language.

This book is about the construction of specifications. However it considers description only at the level of mutual recursion and polymorphism differs from object-polymorphism especially given the problems of the former with mutability which is at the heart of object-orientation—but this would probably have resulted in a very different book. In between those extremes of abstraction it is difficult to see what to make of secrets explained.

By way of (slight) criticism, I would have welcomed more explanation of C++, particularly the rôle of virtual and non-virtual member functions which could have justified late binding rather better than its proclamation in Section 2.2.5. Moreover, the authors' claim of the leniency of C++ assignment between pointers to class and derived-class is not supported by the C++ annotated reference manual (Ellis and Stroustrup). If another chapter were to be added I would strongly suggest one on multiple inheritance and sharing—I believe the authors' methods would cast light here just as well as they have succeeded for types, inheritance and genericity. I personally would also be interested in discussion of the way in which ML-style polymorphism differs from object-polymorphism especially given the problems of the former with mutability which is at the heart of object-orientation—but this would probably have resulted in a very different book.

In addition to the book itself, the main translations and algorithms have all been implemented and made available to the public at large via ftp. I found these easy to obtain and use (daimi.aau.dk warns one to use the new file server at ftp.daimi.aau.dk) but one needs to have access to Scheme.

This is a book which I would commend to those from outside the object-oriented community (and many inside) who wish to understand the range of object-oriented language design possibilities and understand the semantic choices in detail. I welcome its emphasis on type security which is achieved without the excess formalism sometimes found. I concur with the authors' assessment of its utility for final year undergraduate teaching and I would further recommend it for anyone interested in semantics and types of object-oriented languages. In addition to the technical content, the English is indistinguishable from that of many a native speaker (with the notable exception of the use of 'overwrite' for 'override'), the authors have taste in graphic design as befits camera-ready copy and there are very few, unimportant, typographical errors. I enjoyed reading it.

**J.G. Turner and T.L. McCluskey**


Formal methods seem to be assured a place in the Computing Science curriculum. Some universities base their degrees entirely on formal methods; others offer a single course on formal methods, in final year. That reflects growing commitment in industry to the application of formal methods at various stages in the design cycle.

Applied to the early stages of the design cycle formal methods provide a notation for requirements capture. Formalising requirements forces the customer to decide at the very start of the development process what he wants, when a change of mind is far less costly; the result acts as a contract between customer and system developer whose properties may be either derived by proof or observed by prototyping. Applied to the latest stages of the design cycle, formal methods provide techniques for producing correct code: for the correct termination of loops, invocation of procedures, and so on. In between those extremes of abstraction they offer criteria for ensuring the functional correctness of one description with respect to a more abstract one. The lower-level description whilst chosen to incorporate bottom-up constraints and efficiency concerns must meet those criteria to be valid.

This book is about the construction of specifications. However it considers description only at the level of requirements capture, and discusses neither the development cycle nor criteria for a specification to meet a more abstract one (though the VDM terminology for that, reification, is mentioned once by name). Without some treatment of such issues it is difficult to see what to make of a specification once it has been constructed! In what way does it constrain its refinements? Readers of this book must presumably rely on intuition for that, which seems to undermine the whole benefit of formality.
Anyway, from now on we must interpret 'specification' at the requirements level only.

The book is very p. c. (pedagogically correct). It has been laid out clearly. Each chapter has a helpful introduction, accurate summary, some exercises (mostly rather routine) and a brief list of references with further reading. Two contrasting styles of specification are treated, with two chapters devoted to comparative examples. There is an introductory chapter and a final 'drawing-together' chapter; and there is a glossary. Clearly the authors have gone to some trouble to make the book enjoyable to read and easy to understand.

Here is a description of the contents. The first chapter discusses what constitutes a specification (that is, a formal statement of functional properties). It is followed by an extremely informal introduction to those notions from set theory required later. Then follow several chapters introducing the model-based notation, VDM. A system is specified as an abstract data type, with states and operations on them. Both are described using set theory; VDM provides notations for structuring those descriptions. It is because the states of the data type are constructed in set theory that the method is said to be model based. The types of sequences and maps are described in detail and a chapter is devoted to a 'blocks world' case study.

An operation is specified in VDM by giving: its variables with their types; the precondition required for the operation; and its postcondition, or predicate relating the state before the operation to the state after. The final chapter on VDM describes a method for prototyping VDM specifications in Prolog. The treatment is discursive as it must be in the absence of criteria for a prolog program to meet a specification. But it indicates how to replace some non-executable structures like set, sequence and map with executable structures like lists; and how to execute the postcondition of an operation by regarding it as a prolog program. The restriction required of prolog that its predicates appear in horn-clause form is not discussed; nor is the treatment of negation in the specification (perhaps to be interpreted using prolog's 'closed world' assumption?). However the gross inefficiency of such prototypes is discussed, and the blocks-world example animated using the techniques described. The specification used for animation is the second of two given in the case-study chapter; as already mentioned the relationship between them is not analysed and so the reader must take on trust that the more concrete meets its more abstract specification.

Next the algebraic approach to specification is introduced. Algebraic specification concentrates on the operations of an abstract data type and describes their type and the relationship between them whilst abstracting the underlying state. The date types of queue and binary tree are used to exemplify the method. It is because of the way that an algebraic specification is interpreted that the approach is described as algebraic. The relevant theory is summarised fairly informally in a separate chapter. A simple neural net is specified as a case study in algebraic specification, and the result helpfully contrasted with a VDM specification. Prototyping algebraic specifications in OBJ3 is supported by brief discussions of prototyping using functional languages (too brief) or prolog.

The last chapter comprises 'background, comparison and summary' in which the choice of specification language is discussed and some history is given of the two styles of specification. Because of its style the book seems more likely to appeal, as a text, to those studying an isolated course on formal specification than to those studying a more mathematically-based course. The authors have no doubt made the correct choice in adopting a discursive style; but with little extra effort the mathematical ambiguities and oversights could have been eliminated.

Strengths of this book are its educational style, its examples, its choice of two styles of formal notation and its comparison of them. Weaknesses are the very loose mathematical style, the scanty treatment of logic and dirth of laws for reasoning about formulae of predicate calculus, and the lack of challenging exercises. The reader should be aware of the authors' choice to consider specifications only at the level of requirements capture and the consequent decision to discuss prototyping of those specifications rather than refinement between levels of abstraction. (That choice is taken in the case of algebraic specification, in spite of their having surveyed the semantic framework required to discuss those matters.) In the case study of the neural net, the authors give an operational, relatively low-level specification (transitive closure, a way of giving a more abstract and less operational description in this and many similar circumstances, is not mentioned in the book). In the blocks-world case study, a specification and more operational design are presented but their relationship not discussed. Those seem to provide missed opportunities for discussing the range of levels of abstraction at which a specification might be pitched, with the chance to describe to the reader how systems are developed hierarchically using formal methods.

J.W. SANDERS
University of Oxford

GIL LEMAN and LARRY RUDOLPH

Lerman and Rudolph have written a readable survey of parallel machines covering the thirty years to 1992. However, this is not a book for the novice. It is assumed that the reader has a reasonable knowledge of the field of parallel processing and its concepts. That said, you don't have to be an expert to read this book.

This survey is itself both substantial and yet, inevitably, incomplete, something for which the authors