Dahl, Ole-Johan

Verifiable programming. (English) Zbl 0790.68005
New York, NY etc.: Prentice Hall. IX, 269 p. (1992).

"This book", to quote the author, "is primarily intended for readers who know, from personal experience, that programming is in fact very difficult". The author is one of the founding fathers of Simula and of ideas now known as "object-orientation". The book is well-written and rather terse.

In quite a few cases, the author considers program development as a document-based process, both due to the programmer's need to read other people's documents and also due to his need to create his own documents. Obviously, understandability of these large and complicated documents – specifications, user manuals, and code – is of utmost importance. However, this does not always happen, and therefore "countless hours are spent in guesswork, trial, and retrial". Abstraction is essential to reduce the volume of reasoning for large systems, i.e., to master their complexity. Usually, documents of less than desirable quality are not abstract enough or not precise enough [(*) *H. Kilov* and *J. Ross*, Information modeling, an object-oriented approach, Prentice-Hall (1994)]. The book shows how to describe programs in an abstract and precise manner.

To promote abstraction, the author notes the importance of modularization: the external properties of a module are simpler than the sum of internal detail, leading to substantial simplification. To promote precision, the book provides a way to formally define important programming concepts and focuses on program verifiability by means of reasoning about the program text. To do that, a specification – with respect to which a program will be verified – is essential; moreover, reasoning hints are necessary in the program text. These reasoning hints, in the form of preconditions, postconditions, and invariants, should be provided before and during (rather than after!) the development of a program. A specification can also be formulated using these concepts. This "additional information" substantially enhances understanding of the program text (assertion boxes in flow diagrams have been proposed for this purpose, as the author notes, in 1947 (!) by *J. von Neumann* and *H. H. Goldstine* [Planning and coding for an electronic computing instrument, Part 1, Vol. 1 in: The mathematical and logical aspects of an electronic computing instrument, Institute for Advanced Study, Princeton (1947)]). Obviously, a correct program should better be understandable. Even if formal verification is not intended, a verifiable program means a program that is easy to verify, and the book tells the reader how to write such programs.

The book often deals with rather traditional material. It starts with first order predicate logic. A strongly typed language is used. Regrettably, the author recommends using truth tables and trying out all combinations of truth values to verify the very first rules of propositional calculus, although this approach certainly does not scale up. Almost immediately at the beginning, the author describes his general approach to formal reasoning. The challenge in program developments is in "problem and system specifications, as well as other kinds of program documentation", whereas the proofs required for good program verification are uninteresting mathematically – as now and then shown in the book – and very tedious (up to being impractical for human use) because of the volume of reasoning. Therefore these proofs should be mechanized to the extent possible. It is interesting to contrast this with E. W. Dijkstra's approach to verification: in Dijkstra's opinion, these proofs are very important and are to be done by humans, so reasoning has to be reduced – by humans – to a doable amount. Both Dahl and Dijkstra notice difficulties in proof construction and understanding due to the use, in Dijkstra's terms, of rabbits taken out of the magic hat (Dahl uses "mathematical intuition" and, later, "programming intuition and ingenuity"...). Proofs produced by Dijkstra's school (e.g., in [*A. J. M. van Gasteren*, On the shape of mathematical arguments, Lect. Notes Comput. Sci. 445 (1990; Zbl 0705.68004), (**) *E. W. Dijkstra*, A discipline of programming, Prentice-Hall (1976; Zbl 0368.68005)]) at times seem to be more elegant than certain proofs in the book under review. Obviously, a library of reusable lemmas ("specifier's tool kit") helps a lot, both in human and mechanical theorem proving.

Hoare logic is presented next in a rather detailed manner. Restricted forms of goto's are justified and dealt with, although the proof rules for these constructs look quite complicated. Documentation includes mythical program sections – auxiliary constants and variables that do not "contribute to the actual results of the program", but are very useful for understanding (compare [*C. Morgan*, Programming from

specifications, Prentice-Hall (1990; Zbl 0697.68018)]). In particular, input/output is appropriately treated by using mythical variables for representing input and output histories. Axioms and proof rules of Hoare logic are formulated on the basis of informal reasoning, although the author stresses that appropriate foundations may be built formally. He mentions serious problems in use of constructs that are easy to implement, but difficult to reason about, such as pointers, dynamic binding of variables, aliasing, and so on (compare [*F. Bauer* and *M. Woessner*, Algorithmic language and program development, Springer Verlag (1982; Zbl 0486.68007)]). Therefore only static variable binding is used, and functions do not have side effects. Throughout the book, the author pays special attention to partial logics (and definedness) as partial functions are often encountered in typical computer programs. All ill-defined function application in the author's constructive approach aborts (preferably) or does not terminate. This AID (abort when ill- defined) principle leads to important consequences with respect to abstraction: only if it is satisfied reasoning at an appropriate abstraction level becomes possible. Regrettably, this principle is often violated in existing systems.

Abstract data types are treated next. An abstract data type definition is based on the set of functions syntactically associated with the type, so that every such function has an object of this type as its distinguished parameter. (A type is a set of values generated by the functions contained in its generator basis. Generators are widely used in reasoning, including reasoning about equality, and also in proofs.) Some design considerations are provided for certain of these functions. This approach corresponds to the classical object model; the generalized object model describing joint behavior of a collection of objects [(∗)] for which all parameters of a function are equally important (so that there is no notion of a distinguished parameter) is not considered. An invariant of an abstract data type is also not considered. Subtyping and parameterized types (including the need for assumptions about formal type parameters) are described in some detail, but at times the descriptions, especially for parameterized types, are rather sketchy.

A library of standard, sometimes parameterized, types (including enumerations, integers, disjoint unions (like free types in Z), sequences, maps (somewhat like finite partial functions in Z), and so on) is surveyed in a rather detailed manner, with many useful observations.

The final rather lengthy section describes type simulation needed to transform abstract types "good for reasoning", and therefore as simple as possible, into appropriate computationally efficient types.

The last chapter is rather short and treats classes and objects. Pointers are used to refer to high-volume data objects, and the danger of aliasing (and side effects) is considered again. The author notes that for efficiency reasons high-volume (composite) data objects have to be updated in small increments. Traditional object-orientation is treated using Algol 60 blocks as a starting point, whereby the author notes that there may be "a reason to prohibit all direct accesses to components from outside the class, in order to prevent the confusion of different levels of abstraction". The chapter (and the book) concludes with defining a class simulating the type set T and a discussion of pointer aliasing introduced for efficiency purposes. In both cases, the reasoning in proofs is quite tedious.

The author carefully shows that abstraction is essential to reduce the volume of reasoning for large systems, i.e., to master their complexity.

Sound programming advice is provided in the book by stressing the need to design, e.g., the loop invariant and the loop hand in hand. However, it does not always show program design considerations: for example, some ("maintenance"?) exercises require verification of existing programs the design considerations for which are not provided (compare, e.g., Exercise 4.49 with the treatment of updating a sequential file in [(∗∗)], Chapter 15).

No particular prerequisites are required, except some kind of mathematical and programming maturity. Regrettably, the presentation is somewhat uneven: for example, section 5.5 explains bags and equivalence classes while at the same time providing a rather terse treatment of less well-known issues. The author often includes pragmatic remarks very useful even for those readers who are not going to use full formal verification.

The book includes quite a few interesting examples and exercises. Answers to exercises are not provided.

Reviewer: H.Kilov (Millington)

MSC:

68-02	Research exposition (monographs, survey articles) pertaining to computer science	Cited in **8** Documents
68N01	General topics in the theory of software	
68Q60	Specification and verification (program logics, model checking, etc.)	
68Q65	Abstract data types; algebraic specification	

Keywords:

abstract data types; program development; program verifiability; reasoning; formal verification; program documentation; Hoare logic; abstraction; type simulation; object-orientation; pointer aliasing