ELSEVIER Science of Computer Programming 23 (1994) 91-101 Science of Computer Programming

Book review [*]

E.W. Dijkstra and C.S. Scholten, *Predicate Calculus and Program Semantics* (Springer, Berlin, 1989)

The book deals with a fascinating subject, indicated in its title: a theory of program semantics based on predicate logic. When I was asked to review this book, I was preparing a course on the subject and accepted with pleasure the opportunity to study what I expected to be an illuminating text from famous authors' hands. Unfortunately my expectation was not met. Moreover I believe the book will not be helpful to those interested in the subject area. What follows is a detailed review that explains my critical judgment and the reasons that I cannot recommend the book. (An abridged version of this review appears in the June 1994 issue of *The Journal of Symbolic Logic.*)

After a summary of the contents of the book, I will discuss separately the presented logical framework and the development of semantics of programs. This will lead to a discussion of the authors' epistemological assumptions on the notion of "proof" and the role of formal methods in mathematics. I will conclude with some critical remarks on expository aspects.

Contents of the book. The book presents a theory of predicate transformer semantics based on first-order logic; predicate transformers are understood as functions from Boolean-valued terms to Boolean-valued terms. The first Part (Chapters 1-6) is devoted to basic definitions and to first-order logic laws and their application to predicate transformers. The second Part (Chapters 7-12) deals with the (mainly, weakest precondition) semantics of a class of structured programs.

Chapters 1-4 are concerned with notation. Chapter 1 (pp. 1-10) discusses at length the notion of Boolean-valued first-order term (called a "structure") in the Frege tradition, which considers first-order formulae as terms which are subject to Boolean evaluation (interpretation). It also introduces a new notation for universal closure. Chapter 2 (pp. 11-16) presents at length the usual conventions for omitting parentheses (binding power of operators and indication of scope of quantifiers and substitution). Chapter 3 (pp. 17-20) introduces a notation for the ι-operator (to be used for the definition of

[*] Occasionally *Science of Computer Programming* publishes multiple reviews of important, seminal books. The present book review follows one by Brian Q. Monahan in Vol. 17 (1-3) (1991), pp. 259-269.

functions via equations) and for function application. Chapter 4 (pp. 21-29) introduces some of the usual notational conventions from logic which allow succinctness in displaying short (i.e., of the order of magnitude of ten lines) derivations. Chapter 5 (pp. 33-80) proves certain propositional (pp. 30-62) and quantificational (pp. 62-80) tautologies from some postulates. This is followed in Chapter 6 (pp. 81-120) by simple laws about duality, monotonicity and how application of predicate transformers distributes over quantification. Chapters 7-9 present weakest precondition semantics for structured nondeterministic programs built up from (essentially) assignment by sequential composition, the alternative construct IF (a kind of nondeterministic multiple case distinction, pp. 121-146) and the repetitive construct DO (a kind of nondeterministic WHILE, pp. 147-189); IF and DO are constructed using lists of guarded commands. Chapter 10 (pp. 190-200) contains heuristic explanations of these constructs in operational terms, and Chapters 11 and 12 (pp. 201-214) explain the strongest postcondition semantics of those constructs.

On Part One. The book fails most seriously in Part One, which deals with first-order logic. The authors state that "this is not a treatise on logic" (pp. vii, 11) and that their "task of designing the theory [viz. of predicate transformer semantics] ... as an exercise in formal mathematics" (p. v) only called for "a few notational adaptations of the predicate calculus ... and the adoption of a carefully designed, strict format for our proofs" (p. v). And yet they do not simply "introduce the reader to the repertoire of general formulae that will be used throughout", but instead they "develop the calculus of boolean structures" and spend 91 out of 215 pages "proving all formulae that have not been postulated" (p. 30). For a book on a theory of predicate transformer semantics, intended to provide "a means for defining programming language semantics in a way that would directly support the systematic development of programs from their formal specifications" (p. v), it would surely have been appropriate to presuppose what is needed from predicate logic, or simply to list and perhaps briefly explain it. It is indeed highly laudable to separate the computer scientist's concern with "systematic development of programs from their formal specifications" (p. v) from the logician's concern with presenting and explaining, in a clean and succinct way, the "repertoire of general formulae that will be used" (p. 30) and proving them "from a restricted repertoire" (pp. vi, 21, 28). Software engineers, like any other engineers, should have no problem in using logico-mathematical laws without knowing all of the details of their justificatory theory. By developing at length their own logical calculus, the authors open themselves to the inevitable comparison with the presentation of first-order logic laws in dozens of treatises on formal logic.

Certainly, a part of the logician's task is to justify logical laws that are to be used. For this purpose, it is indeed often helpful to seek "a modest repertoire" (p. vi) of postulates—or better, a "repertoire" that is "appropriate" or "adequate" for the specific purpose—from which all the laws to be justified (usually relative to a certain semantics) can be proved in a formal way. Therefore, the task consists in providing, within a formal calculus, proofs for all the laws under consideration (so-called completeness) and in justifying the postulates (axioms and rules) used in the proofs (so-called soundness).

An additional epistemological or pragmatic task is to give the reasons that this particular (set of postulates for the) calculus has been chosen. Whether a justification is satisfactory depends, not only on the class of logical laws under study—"which laws will be used"—, but also on where and how one wants to use them (for which subject or purpose, whether for mathematicians or engineers, and whether for pencil and paper or machine manipulations). Note that the request for such justification is a methodological imperative for whatever logical "calculus" is proposed; it cannot simply be dispensed with by a "naive refusal to make some of his [the logician's] cherished distinctions" (p. vii).

The authors—heading for what "seemed at first a rather presumptuous undertaking" (p. vi), namely "a revision of mathematical methodology" (ibid.)—place themselves in the position of trying to redo the modern predicate calculus. Unfortunately, they remain behind what Frege had accomplished in his *Begriffsschrift* in 1879 and his *Grundgesetze der Arithmetik* in 1893 and 1903, not to mention what has been achieved in the subsequent 100 years of research in logic. The authors introduce postulates without any justification of their special status with respect to derived formulae, of their soundness, or of their completeness. They do not even mention that a justification problem exists—except that they say that these postulates suffice to prove from them, in the particular proof style chosen, "the repertoire of general formulae that will be used throughout the remainder of this booklet" (p. 30). One is led to wonder whether new postulates or proof principles will be needed (and if so, which) if one seeks to extend the approach to the semantics of programs with more complicated features than those considered in this book (see below). There is no indication as to whether this repertoire has any other characterization than that of the finite set of the formulae that happen to "be used throughout the remainder of this booklet". Will the process of adding new first-order postulates or laws ever end or reach a limit? For predicate logic, generations of logicians have developed satisfactory and simple answers to that kind of fundamental questions. The authors could and should have used these achievements to their advantage, instead of coming up with a pale imitation of a "theory" of derivations—derivation of some "repertoire" from a few postulates which appear in the book like "mathematical rabbits—pulled out of a hat" (p. ix). Note that everywhere in the book "mathematical rabbits" are very much criticized, and I think correctly so.

This attitude of the authors to the problem of justification is so surprising because it contradicts their own principles, clearly and convincingly stated, in support of a computational approach to program design from specifications: "The 'mindless' manipulation of uninterpreted formulae that is implied by 'letting the symbols do the work' is clearly impossible without a sufficiently clear view of the permissible calculations" (p. 11). Is such a clear view of the calculations within first-order logic not provided by a sound and complete calculus for predicate logic? It is the separation of mathematically precise syntax from semantics and their equivalence (through the soundness and completeness theorems) which allows the "mindless"—but semantically safe—"manipulation of uninterpreted formulae" and, more important, the combined use of syntactical manipulations and semantical reasoning that is so typical and fruitful in applications of predicate logic.

As a reviewer, I still must answer the question whether the particular calculus developed here offers a better proof format than other systems of first-order (formal) logic. The authors themselves do not compare their system with any other system in the literature; indeed, they do not even mention that there are others. But the authors repeat frequently (and mostly when commenting on particular formulae or proofs) their conviction—they call it "evidence"—"that most mathematicians lack the tools needed for the skillful manipulation of formulae" (p. vi). Perhaps, this statement is intended to point out that in the field of formal program specification, most (though not all, see below) of the methods developed during the last 25 years lead to unacceptable combinatorial explosion when applied to real systems. But this phenomenon appears only far beyond the point to which the authors lead us in this book; it certainly does not present itself at the level of deriving mostly well-known and simple laws of first-order logic. The authors looked for a proof format that fits the needs of a human programmer who uses laws of predicate logic for the systematic development of programs from formal specifications. The proposed proof format for predicate logic is geared to truth-value preserving transformations of formulae (Boolean-valued terms) based on equations allowing functional notation. Let me note, en passant, that the latter idea is not new: it has been extensively investigated in algebraic and term logic literature, but the authors seem not to be aware of this. (See, for example, L. Henkin, J.D. Monk and A. Tarski, *Cylindric Algebras* (North-Holland, Amsterdam, 1971), and H. Hermes, *Eine Termlogik mit Auswahloperator*, Lecture Notes in Mathematics, Vol. 6 (Springer, Berlin, 1965), and the references given there). The crucial criticism is this: despite many lengthy explanations of the need for a neat and expressive framework and despite much expression of pride in the (alleged) "beauty" of their system, the authors present a proof format which is considerably clumsier, less transparent, and more difficult to use (in proofs and in teaching "the design of beautiful, formal proofs", p. vi) than various well-known frameworks in the literature.

Let us check this judgment using criteria that the authors themselves regard as establishing the "virtues of presenting formal proofs in a strict format"; in particular, (a) "that it enables a meaningful comparison between alternative proofs of the same theorem, viz. by comparing the text", and (b) the "greater homogeneity it introduces into the task of proof design, a task that becomes a challenge in text manipulation that is more or less independent of what the theorem was about" (p. vi). For comparison let us consider the well-known polynomial representation of Boolean-valued terms due to I.I. Shegalkin (Mat. Sb. 34 (1927) and 35 (1928)) and in dual form to J. Herbrand (Dissertation, 1930), which can be found in various textbooks on first-order logic. Every Boolean-valued term x is (uniquely) representable as a polynomial $\overline{x}$ in its free variables with coefficients $0, 1$; conjunction corresponds to multiplication mod2, exclusive disjunction (not the inclusive "or" to which the authors refer to on page 42) to addition mod2, negation to "+1 modulo 2", true to 1, and false to 0. (By the way, this gives rise to an elegant Boolean ring structure $\mathbb{Z}/2\mathbb{Z}[x_1 \ldots x_n]$, where $\mathbb{Z}$ is the ring of integers.) Hence for arbitrary Boolean-valued terms x, y one has the following equations, where for notational convenience I suppress overlining (identifying z with $\overline{z}$):

$$x \wedge y = xy, \qquad x \dot{\vee} y = x + y, \qquad \neg x = x + 1,$$

and therefore also

$$x \leftrightarrow y = x + y + 1, \qquad x \rightarrow y = xy + x + 1, \qquad x \vee y = x + y + xy.$$

The tautologies are exactly those Boolean-valued terms t whose polynomial representation $\bar{t}$ equals 1 mod 2. In this way, literally all "theorems" proved by the authors in their "calculus of boolean structures" (pp. 30–80) become trivial, and this despite the fact that in polynomial notation they are established in much more general, surely uniform, more homogeneous, more succinct, simpler, and more transparent versions. (These properties are all claimed by the authors for *their* theorems and proofs, in comparison with those in other logical calculi, not specified.)

Here are three "small examples" (p. 43); I do not insist that each of them "shows in its full horror how an unfortunate notation can damage one's manipulative abilities" (ibid.), but they do not lack some small horror of their own. First, half a page (pp. 32/33) is spent in explaining the universal quantification of the formula $x \leftrightarrow y \leftrightarrow y \leftrightarrow x$ as expressing—according to much fussed-over conventions for omitting parentheses—both the symmetry of logical identity (equivalence) and the left- and right-identity property of $(y \leftrightarrow y)$ with respect to equivalence. By the way, it is not clear to me why we should make such a fuss over a mnemonic device—nothing more than a traditional "pons asinorum"—which due to some notational conventions permits in some (how many interesting?) cases the representation of conceptually different ideas by a single logical formula. But never mind, transforming $x \leftrightarrow y \leftrightarrow y \leftrightarrow x$ into polynomial notation (by applying $x \leftrightarrow y = (x+1) + y$ three times, say from left to right) yields $x + 1 + (y + 1 + (y + 1 + x)) = 2x + 2y + 3$. But $2x + 2y + 3 = 1 \bmod 2$, which is an instance of the obvious and general polynomial equation $\sum_i 2n_i x_i + 2m + 1 = 1 \bmod 2$.

The second example involves the definition of conjunction in terms of equivalence and inclusive disjunction—namely $x \wedge y \leftrightarrow x \leftrightarrow y \leftrightarrow x \vee y$ (p. 37)—, introduced as "The Golden Rule" and used with corresponding great pomp. Transforming this definition into polynomial notation yields the (easily generalizable) trivial equation

$$xy = x + 1 + (y + 1 + (x + y + xy)) = xy + 2x + 2y + 2 \bmod 2.$$

The third example comes during a lengthy discussion of propositional logic manipulations of "expressions that make heavy use of the implication". We are told that "in combination with the negation sign, hell breaks loose, as is shown by the following eight expressions..." (p. 61). These eight expressions are nothing more than rewritings, modulo the law of contraposition, of $x \vee y \rightarrow x \wedge y$, and transforming this expression into polynomial notation yields its equivalence to $x \leftrightarrow y$ by:

$$(x + y + xy)xy + (x + y + xy) + 1 = 4xy + x + y + 1 = x + y + 1 \bmod 2$$

—the authors spend half a page for their proof which comes spiced with pompous methodological comments (p. 62). In polynomial notation, no "hell breaks loose" when a busy little demon adds an even number of occurrences of xy or x or y to the above expression for $x + y + 1$: it yields the general elimination rule

$$x + y + 1 = 2kxy + (2i + 1)x + (2j + 1)y + 2l + 1 \bmod 2$$

(This rule contains as special case the above-mentioned eight expressions discussed by the authors.) And so forth for the entire long Chapter 5 and most of Chapter 6.

The book also exhibits some unclear and confusing definitions, paired with clumsy notation, for phenomena which have already a better and well-established definition and notation in the literature. For example, on page 8, an "everywhere operator" is introduced which plays an important role throughout the book. But the "definition" is obscure, elaborated later (see pp. 10, 23, 58) and only at the end the experienced reader is able to guess what was intended. I asked more than ten top logicians and computer scientists to read the relevant early sections and then to explain their meaning. None of them had understood matters, because the notation is "almost as cryptic as Frege's notation", as one from this group said after I had explained to him what the definition and notation meant; and he continued: "only in Frege's days it was a great merit to have invented a full and correct notation at all." The everywhere operator expresses essentially universal closure without explicitly writing down all the variables concerned. The usual logical notation $\tilde{\forall}$ or simply $\forall$, besides being established, is more suggestive; it can also be extended, very naturally and without further words, to existential closure $\tilde{\exists}$, whereas our authors would have to invent yet another new notation. Another use the authors often make of their everywhere operator can also be described much more simply by the well-known tautology rule, which states that the validity of a first-order law is not affected by replacing a propositional variable by a first-order formula (which may contain fresh free individual variables). It comes as no surprise that proof format and notation [ ] for the everywhere operator induce the authors to state and prove "theorems" which express trivial logical facts, for instance, the idempotence of [ ] (p. 32), or the validity of (what *they* call) Leibniz' rule for so called punctual functions which is a case of $\tilde{\forall}(\alpha \rightarrow \beta) \rightarrow (\tilde{\forall}\alpha \rightarrow \tilde{\forall}\beta)$.

Here are two more examples of rather obscure definitions. On page 10, one finds the "definition" of application of [ ] to term pairs, a definition that becomes understandable only much later in the book (p. 58) and essentially expresses the fact that universal quantification distributes over conjunction. (Note that it is rather unfortunate to speak here about "pairing" because conjunction is associative and commutative, but pairing is not.) On page 18, one finds again a rather confusing "definition" of the well-known description (iota) operator in the context of defining functions by equations. The reader has to struggle through misleading phrases such as "the global variables of a scalar" and unfortunate hiding of variables and quantifiers in a context where their presentation would have helped the understanding. Also, it is not clear why the proposed new notation $y : [y = \alpha]$ should replace the usual $\iota y[y = \alpha]$, established by Whitehead and Russell in their "Principia Mathematica" (1910–1913). Logic has taken years of work to develop fairly standard methods and notations. Changing notation without serious reason provides no good service, isolating the reader from the rest of the scientific community.

The authors' decision to use—for an "abstraction" of expressions in (program) variables—the name "structure" instead of the usual name (Boolean-valued) "term" is unfortunate. As a matter of fact, "modern mathematical usage" in logic has *not* been "reluctant to introduce a name for an expression in a number of variables" (p. 4), and logic has provided methods for avoiding the danger that "those names may become quite tricky to manipulate" (p. 4). This is done in full generality, but in particular in term logic, where "formulas"are viewed as Boolean-valued terms. (This comes naturally from a systematic treatment of Whitehead's and Russell's description operator; see, in particular, Chapter 8 in D. Hilbert and P. Bernays, *Grundlagen der Mathematik* (Springer, Berlin, 2nd ed., 1968) and its further development in the book by Hermes cited above.)

I have some reservations also about a number of minor but misleading remarks, which reveal that the authors are not really acquainted with the field of mathematical logic. For example, they seem to believe that they had discovered the usefulness—in proving certain theorems—of not breaking equivalences into separate implications. But from automated theorem proving systems it is well-known that one should, if possible, avoid breaking equivalences into separate implications. However, in the case of formulae such as $\exists x\alpha \leftrightarrow \exists y\beta$, one may have different witnesses in the two directions and therefore need to handle the two implications separately.

Summarizing my judgment of the first part of the book, I recommend to those who wish to learn just the elementary facts which are needed from first-order logic for a formulation of the weakest precondition approach within predicate logic: read one of the excellent texts *Logic and Structure* by D. van Dalen, Springer-Verlag, or *Logic for Computer Scientists* by U. Schöning, Birkhäuser, Boston. In each case, if the proof of the completeness theorem and related material is skipped, there are fewer than one hundred pages to read to reach the point where the book under review switches from logic to semantics of programs.

On Part Two. In the second part of the book, the authors are in their métier; they explain predicate transformer semantics, and write as experts of the field. But there, too, many critical questions remain.

At first sight, it appears surprising that the authors treat only structured (nondeterministic) programs built from assignment (and simple basic programs abort, skip, a nondeterministic stop) by sequential composition, the alternative construct IF , and the repetitive construct DO. Over the past 25 years we have seen many approaches (structural operational, axiomatic, algebraic, denotational, etc.) developed around such simple programs; and some calculi deal with structured programs successfully, in a balanced and transparent way, and not without mathematical elegance. (The reader may look, for example, at the brilliant exposition in *Verification of Sequential and Concurrent Programs* by K.R. Apt and E.-R. Olderog (Springer Texts and Monographs in Computer Science, 1991), where deterministic (pp. 55–105) and nondeterministic (pp. 106–176) sequential programs are treated as a prerequisite for a detailed study of parallel and distributed programs.) But the presentation of the calculus in the book under review, pressed into a corset of an unfortunate notation and proof format, lacks conceptual clarity and surely the beauty that Dijkstra has achieved in his book *A Discipline of Pro-*

gramming (Prentice-Hall, Englewood-Cliffs, NJ, 1976) and advertises as his trademark (see the editors' explanations of the title of the book *Beauty is Our Business. A Birthday Salute to E.W. Dijkstra*, W.H.J. Feijen et al., eds., Springer Texts and Monographs in Computer Science, Vol. 21, 1990).

Secondly I can see no real progress in this calculus over previous presentations of the weakest precondition approach—except for minor technicalities, see, for example, the discussion of the role of or-continuity in the proof of the "Main Repetition Theorem" (pp. 180–185). The treatment of this theorem—a modulo some technical refinement 25 years old theorem which constitutes the culminating point of Chapter 9 on "Semantics of repetition"—reveals a similar kind of unacceptable "incompleteness" (or authors' bias) which we have encountered already in the first part of the book. The "Main Repetition Theorem" establishes the soundness of a specific "rule" for the DO construct. It thereby corresponds to the soundness theorem for loop rules (proof rules for termination) in Hoare-like systems. The natural counterpart is the completeness problem, whether all relevant true formulae can be proved this way. This problem has been extensively studied (see Apt and Olderog, op.cit., Ch. 3.6, Ex. 4.20 and the references given there). The authors of the book under review do not tell the reader what such completeness could mean in their approach; they do not even pose the question or at least mention that such a question can and has been asked.

Finally, it does not become clear whether and how the theory developed by the authors could deal with extensions to, for example, concurrent or reactive programs, programs allowing recursion with parameters, multiprocessing, more sophisticated forms of control and data structures, etc.

Operational semantics seems to be a victim of attack by the authors (pp. 121–125). They argue correctly against an absolute, physical understanding of "machines" and "computation step". But the approach of structural operational semantics à la Plotkin—to mention just one example—is not at all based on such a physical notion of machine. One can also successfully base operational semantics on a notion of (classes of) machines at various levels of abstraction, related hierarchically by the method of stepwise refinement. This permits a transparent, mathematically precise, and simple operational semantics, as well as formal specifications, and mathematical correctness proofs for real languages and systems. The conceptually operational character allows one to couple intuitive insight with mathematical rigor of method and thereby to settle challenging questions by proofs which satisfy appropriate mathematical standards without necessarily being proofs in a "calculus". The framework of "evolving algebras" recently proposed by Y. Gurevich shows that this is indeed possible.

This brings me to a fundamental criticism of the book's approach to design and formal verification of proofs and programs. The authors suggest that we identify the mathematical notion of "proof" with "formal proofs" and the latter with "formal proofs in a strict format" (p. vi), namely the one developed in this book "for a revision of mathematical methodology" (p. vi). The authors praise their proof format rather extensively, and the reader "is invited to share [the authors'] delight" (pp. v–vi). I read the book carefully, slowly (as the authors ask on page vii) and with patience—it took me two months of work to go through it three times. I read it not only "with an open mind"(as recommended by the authors on p. vii) but with an initial very positive bias—but to my disappointment I found nothing which could justify the authors' "hope [that the reader] travelled long and far" (p. vii). It is one thing—reasonable and standard—to adopt, for a specific purpose, an appropriate particular proof format which allows to break down (writing, explaining, and checking of) unavoidable routine calculations into simple sequences of easily manageable steps; see in this respect the use Apt and Olderog make of Dijkstra's proof format for development and verification of programs (op.cit., p. 42, etc.). A completely different and I believe wrong thing is what the authors suggest, namely to press *every* form of construction of and reasoning about programs into such a restricted format. This decision of the authors to limit problems and solutions to those which fit their proof format not only "shows in its full horror how an unfortunate notation can damage one's manipulative abilities" (p. 43) and one's intuition. Worse than that, it shows how the underlying conception of mathematical reasoning as formalistic (calculational) activity limits the horizon of mathematical research to what can be reached by mere symbol pushing. As a matter of fact, the reader who follows the book has to go a long way—the formalist's way—but without being brought far and without reaching a destination one could recommend. From this point of view it appears not any more surprising, but certainly significant, that the authors—after 25 years of intensive work (see p. 184), "in possession of a tool [viz. the theory developed in this book] that surpassed [their] wildest expectations" and that "became an absolute delight to work with" (p. v)", writing a monograph reflecting *the current state of our art*" (p. vii) [my italics]—treat only structured IF/DO programs.

Still more significant in this context seems to me the following fact: at the very point where—after 170 pages of preparation—the authors start with their description of the "semantics of repetition" (Chapter 9), they cannot really give their definitions "without pulling a sizeable rabbit out of the magician's hat" (p. 184)—one of those "mathematical rabbits" the authors frequently and rightly insist on "exorcising" (p. ix). Indeed they declare (p. 171): "We shall now define predicate transformers *wlp.DO* and *wp.DO*. We suggest that in this [NB: 20 pages long] chapter the reader *just accept these definitions as such, without wondering from where they come or what has inspired them* [my italics]. Such *background information* [my italics] will be provided in the next chapter." Chapter 10, entitled "Operational *Considerations*" [my italics], eventually provides the basic intuitions which make the whole thing work. The authors show by their book what is the price for banning simple and rigorous descriptions of the basic operational intuitions from a formal definition of program semantics: the price is a messy apparatus for formal representation, which produces alienation even in the willing reader. The one who learns semantics of programs in this way will have difficulties to appreciate the usefulness of formal (mathematically precise) methods in the area.

Let me state what helps more than the preceding 189 pages of the (220-page) book under review: a clean conceptual framework that allows us to make our basic operational experience mathematically manageable and precise in such a way that the framework turns into an interesting object of study for mathematically inclined theoreticians, and into a really helpful tool for programmers. I mentioned above an example of such a framework. Another, quite different but very illuminating, example for comparison is the book *Compilerbau* by N. Wirth (Teubner, Stuttgart, 1984, 118 pages, expanded version of the last Chapter of Wirth's book "Algorithms + Data Structures = Programs", Prentice-Hall, 1976), a jewel of clarity, precision, and simplicity. In 83 pages, starting from scratch, by stepwise refinement and with good "heuristics from which most of the arguments emerged most smoothly" (p. vi of the book being reviewed), it develops a full compiler for a language that contains (besides assignment, composition, IF-THEN-ELSE and WHILE) procedures and elementary input and output instructions. In my teaching in computer science over the past twenty years, I have found that Wirth's methodology really achieves what the authors claim for their calculus: it "turned the design of beautiful programs into an eminently teachable subject" (p. vi). (Paraphrasing the authors' wording, I have taken the liberty of replacing "formal proofs" by "programs".)

I am afraid "the mathematically inclined computer scientists and the mathematicians with methodological and formal interest"—the authors' declared target audience (p. viii)—may turn their backs on computer *science* should they fall victim to the authors' suggestion that "this was all" (p. vii) one could do to fruitfully use logic, or more generally formal mathematical methods, for systematic development of programs from their specifications. My conception of the role of formal methods in computer science, logic, and mathematics is rather different from what the authors call their "new appreciation of the mathematical activity" (p. vi). My experience shows me that the formal instrumentarium that modern abstract mathematics has created has not diminished but increased the power and the range of "intuitive precise" mathematical reasoning—a reasoning which takes advantage of the help offered by "calculational [forms of] reasoning" (p. vii) but without exhausting itself in pure formalism. I do not believe that purely formalist, merely "calculational reasoning" will solve the difficult and challenging mathematical problems we are facing in computer science.

Summarizing my judgment of the second part of the book under review, I recommend to the interested reader to study Part II (pp. 55–176) of the above-mentioned book by Apt and Olderog. I guess this reading will produce much curiosity to continue reading also of Parts III and IV on parallel and distributed programs, pp. 177–324, 325–416.

On the style of writing. Further criticism concerns expository aspects. The style in which the book is written is rather unusual for a scientific text. It is full of phrases that seem to aim at involving the reader emotionally instead of convincing by rational arguments. The presentation is flat: even the most trivial proposition is called "Theorem"; the reader is given only very scattered hints that would enable him to distinguish between simple and difficult, technical and profound, routine and creative matters or techniques. The authors are so much concerned with (or delighted by) "letting the symbols do the work" (loc. cit.) that they often neglect to provide necessary basic motivation. For example, although *nondeterministic* programs play a basic role for the whole approach in the book, the reader finds only scattered remarks explaining why nondeterministic constructs are useful at all; this is essentially only at the end of Chapter 7 (pp. 145–146). Another example is provided by the definition of determinacy, advertised on the book cover. This definition is purely formal (p. 131), no attempt is made to tell the reader whether for example this "determinacy" of programs means (a) uniqueness of output, or (b) that in all occurring cases the occurring guards are disjoint. But the authors do write that they "think it worthwhile to note that, as a definition, it is very nice and, by being so, gives an encouraging indication that we have introduced appropriate concepts." (p. 132)

Maybe it can be accepted that no references at all are given—although a novice in the field might like to be in a position to relate this text to work in the literature. It is less acceptable that the authors, by alluding to some and hiding or ignoring other historical sources, produce a distorted view of their own achievements. For example, on pp. 6–9, Recorde is celebrated for having introduced, in 1557, the symbol "=" to denote equality, then Boole is celebrated for having assigned (in 1854) to this symbol "in principle the full status of an infix operator that assigned a value to expressions of the form $a = b$", after which the reader is taught that "[the authors'] decision to introduce an explicit symbol for functional application is the natural counterpart of Recorde's decision to do so for equality" (p. 9), namely $f.x$. No mention is made of Frege's and others' fundamental contribution to a correct logical definition of identity (see below). No mention is made of Schoenfinkel, Curry, or the λ-calculus tradition where more than just a notation for functional application can be found. In the same context, the decision to treat logical formulae as Boolean-valued terms is suggested to be an invention of the authors, instead of mentioning Frege or the term logic approach. The authors "honour" Leibniz through baptizing one of their cherished rules "the Rule of Leibniz" (p. 9). This rule expresses preservation of equality under function application: $x = y \rightarrow f(x) = f(y)$. But the authors do not tell us that this implication expresses only a restriction (namely to function application) of the trivial direction of Leibniz' fundamental and profound discovery known in the literature as *principium indiscernibilium*; this principle provided the epistemological ground for the first logico-mathematical definition of identity, given by Frege improving on Boole (the interested reader will find more details in J.M. Bochenski, *Formale Logik* (Alber, Freiburg 1956, pp. 413–416)).

The above criticism deals with the form and content of the *book* under review. There is no doubt whatsoever that Dijkstra has substantially contributed to computer science, and that he will be rightfully remembered for these contributions; in the case of the combination computer science–logic, however, his present contribution will not undergo the same fate. The writing of this review has been a slow and painful process, brought to an end only thanks to encouragement and help from numerous colleagues who obviously remain anonymous. The problem was that I had expected from Dijkstra a truly outstanding book—but found instead a thoroughly disappointing and misleading one.

EGON BÖRGER
Dipartimento di Informatica
University of Pisa
Corso Italia 40
I-56125 Pisa, Italy