COMPUTER SCIENCE EDUCATION 3, 203–221, (1992)

Mathematics of Computing

Hossein Saiedian
University of Nebraska at Omaha

After years of discussion, the consensus among academic computer scientists seems to be that undergraduate computer science programs need more mathematical content. This mathematical content is not to be found in areas such as calculus and differential equation but in areas such as discrete mathematics and mathematical logic. The objective of this article is to re-examine the central role of mathematics in computer science by reviewing the undergraduate computer science curriculum from Curricula '68 to the current Computing Curricula 1991. The trend toward a greater concentration in such courses as discrete mathematics will be explored. Furthermore, we collected and put together comments and concerns of three prominent computer scientists about the importance of mathematics in computing science and lack of mathematical content in computer science programs. We then look at one area of computer science amenable to mathematical treatment. This area, *software specification*, is one of the important phases of the software development. The use of formal methods based on discrete mathematics has been increasingly emphasized recently for software specifications in an effort to produce precise and concise descriptions. We will provide an example to demonstrate the importance of a mathematical approach to arrive at precise specifications and argue against claims that mathematical notations introduce unnecessary complication by pointing out the advantages of and misconceptions about the use of formal notations. The article is concluded by once again emphasizing the integration and applications of mathematics in all areas of computing discipline, especially for software development.

1. INTRODUCTION

Several recommendations for academic undergraduate programs in computer science have been published by several different and indepen-

Correspondence and requests for reprints should be sent to Hossein Saiedian, Dept. of Computer Science, University of Nebraska, Omaha, NE 68182.

The author is grateful to Keith Barker and anonymous referees whose constructive comments and suggestions substantially improved the quality of this article.

COMPUTER SCIENCE EDUCATION 3, 203–221, (1992)

Mathematics of Computing

Hossein Saiedian
University of Nebraska at Omaha

After years of discussion, the consensus among academic computer scientists seems to be that undergraduate computer science programs need more mathematical content. This mathematical content is not to be found in areas such as calculus and differential equation but in areas such as discrete mathematics and mathematical logic. The objective of this article is to re-examine the central role of mathematics in computer science by reviewing the undergraduate computer science curriculum from Curricula '68 to the current Computing Curricula 1991. The trend toward a greater concentration in such courses as discrete mathematics will be explored. Furthermore, we collected and put together comments and concerns of three prominent computer scientists about the importance of mathematics in computing science and lack of mathematical content in computer science programs. We then look at one area of computer science amenable to mathematical treatment. This area, *software specification*, is one of the important phases of the software development. The use of formal methods based on discrete mathematics has been increasingly emphasized recently for software specifications in an effort to produce precise and concise descriptions. We will provide an example to demonstrate the importance of a mathematical approach to arrive at precise specifications and argue against claims that mathematical notations introduce unnecessary complication by pointing out the advantages of and misconceptions about the use of formal notations. The article is concluded by once again emphasizing the integration and applications of mathematics in all areas of computing discipline, especially for software development.

1. INTRODUCTION

Several recommendations for academic undergraduate programs in computer science have been published by several different and indepen-

Correspondence and requests for reprints should be sent to Hossein Saiedian, Dept. of Computer Science, University of Nebraska, Omaha, NE 68182.

The author is grateful to Keith Barker and anonymous referees whose constructive comments and suggestions substantially improved the quality of this article.

mittee's work was the first comprehensive attempt to describe computer science as an independent discipline. The committee broke down the subject areas within computer science into three major divisions: (1) information structures and processes, (2) information processing systems, and (3) methodologies. Information structures and processes was concerned with representations and transformations of information structures and with theoretical models for such representations and transformations. Information processing systems was concerned with systems having the ability to transform information. These systems usually involved the interaction of hardware and software. Methodologies were derived from a broad area of applications of computing that have common structures, processes, and techniques. The committee's view on mathematics in the computer science area were as follows [2]:

> The Committee feels that an academic program in computer science must be well based in mathematics since computer science draws so heavily upon mathematical ideas and methods. The recommendations for required mathematics courses given below should be regarded as minimal; obviously additional course work in mathematics would be essential for students specializing in numerical applications.

According to the committee, the supporting work in mathematics should consist of at least 18 hours including Mathematical Analysis I and II.

2.2. Curriculum '78

In 1978 the Curriculum Committee of the ACM published another report: "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science" [3]. In this report,the committee updated and expanded upon their previous Curriculum '68. The committee also developed a list of objectives for computer science students. These objectives clearly emphasized the practical aspects of computer science. For example, according to the Curriculum 1978,

1. Students should be able to write programs in a reasonable amount of time that work correctly, are well documented, and are readable
2. Students should be able to determine whether or not they have written a reasonably efficient and well-organized program
3. Students should know what general types of problems are amenable to computer solution, and the various tools necessary for solving such problems
4. Students should understand basic computer architectures

1. To provide a coherent and broad-based coverage of the discipline of computing. Graduates should develop a reasonable level of understanding in each of the subject areas and the processes that define the discipline, as well as an appreciation for the interrelationships that exist among them.
2. To function effectively within the wider intellectual framework that exists within the institutions that house the programs.
3. Different undergraduate programs place different levels of emphasis upon the objectives of preparing students for entry into the computing profession, preparing students for graduate study in the discipline of computing, and preparing students for the more general challenges of professional and personal life. Students should be aware of that program's particular emphasis with regard to these three objectives.
4. To provide an environment in which students are exposed to the ethical and social issues that are associated with the computing field.
5. To prepare students to apply their knowledge, singly or as a member of a team environment, to specific, constrained problems and produce solutions.[4]
6. Provide sufficient exposure to the rich body of theory that underlies the field of computing, so that students appreciate the intellectual depth and abstract issues that will continue to challenge researchers in the future.

2.4. Mathematical Maturity According to the Curricula 1991

The Curricula 1991 emphasizes that an understanding of mathematics is essential for students in the computing discipline to master successfully fundamental topics of computing. According to the curricula, all students should take at least one-half of a full academic year of mathematics courses (i.e., at least four or five semester courses) that cover at least the following two subjects:

- Discrete mathematics (covering topics such as sets, functions, propositional and predicate logic, graph theory, proof techniques, combinatorics, and probability)
- Calculus

[4]According to the report, the knowledge implied by this item includes the ability to define a problem clearly; to determine its tractability; to determine when consultation with outside experts is appropriate; to evaluate and choose an appropriate solution strategy; to study, specify, design, implement, test, modify, and document that solution; to evaluate alternatives and perform risk analysis on that design; to integrate alternative technologies into that solution; and to communicate that solution to colleagues, professionals in other fields, and the general public.

cal analysis is dropped in the favor of discrete mathematics and mathematical logic courses. The recommendations very explicitly recommend a course in discrete mathematics with a list of all topics to be covered and further emphasizes that courses in both advanced discrete mathematics and mathematical logic (covering propositional and functional calculi, completeness, proofs, etc.) be considered. As pointed out by a colleague, the Curriculum 1991 view on discrete mathematics and mathematical logic represents an "inverted bell-shaped curve" with respect to Curriculum '68 and Curriculum '78. In 1968, discrete mathematics was considered very important when students going into computer science came mostly from engineering and mathematics was no problem. When computing became an area of study for a wider range of students during the 1970s, the emphasis, as reflected in the Curriculum '78, was decreased. However, as the computing discipline matured, it became evident that its foundations are strongly mathematical as shown in the Curriculum 1991.

4. SHOULD COMPUTER SCIENCE BE MORE MATHEMATICALLY FOCUSED?

Should computer science be more mathematically focused? To answer this question, we examine the views of three prominent personalities in the computing discipline and look at their supporting arguments. Our objective in recollecting these views is to reinforce the emphasis on the role of discrete mathematics from a slightly different angle than those presented in the proposed curricula.

In February 1989, Edsger Dijkstra, who is perhaps the most prominent and well-known computer scientist and who has made great contributions to the field of computer science, gave an invited talk called "On the Cruelty of Really Teaching Computing Science" at the ACM Computer Science Conference. During the course of his presentation, some of the basic assumptions on which the computer science curricula are based were brought under fire by his comments. He attacked software engineering as a doomed discipline and argued that the teaching of computer science should embrace the disciplines of formal mathematics and applied logic.

His attack on software engineering, which he calls "The Doomed Discipline" is backed by what he terms reassuring illusions. In the area of programmer productivity, he cites the illusion that programs are just devices like any others. In the area of quality control, he points to the illusion that what works with other devices works with programs as well. In the area of software production, he examines the illusion that the pains of software production are largely due to a lack of appropriate

Anthony Ralston [10] points out that the general consensus among academic computer scientists concerning the role that mathematics should play in an undergraduate computer science curriculum is that undergraduate computer science programs need more mathematical content, which needs to be more closely integrated with computer science courses than is implied by Curriculum '78. Ralston further elaborates that, when possible, topics in computer science and mathematics should be paired together, and concludes that computer science students could gain valuable knowledge for the first course in computer science through a prerequisite or corequisite mathematics course that provides relevant mathematical material in the context of a full discrete mathematics syllabus. This should be a one-year course in discrete mathematics taught at an intellectual level equivalent to that of an introductory calculus course.

On the importance of discrete mathematics, the Mathematical Association of America [13] published a report authored by Ralston that gives further credence to the importance that discrete mathematics carries in the field of computer science. Among other things, the report states that (1) discrete mathematics should be part of the first two years of the standard mathematics curriculum at all colleges and universities, and that (2) the primary themes of discrete mathematics courses should be the notions of proof, recursion, induction, modeling, and algorithmic thinking and that the topics to be covered should emphasize acquisition of mathematical maturity and of skills in using abstraction and generalization. Furthermore, the report suggests that secondary schools should introduce many ideas of discrete mathematics into the curriculum to help students improve their problem-solving skills and prepare them for college mathematics.

David Gries, also a well-known computer science academician, complains in his excellent article [14] that software engineering, computing, and computing education all suffer from a lack of basic mathematical skills that are needed in dealing with algorithmic concepts. He also states that the formal techniques and their applications in programming that are being taught are reaching the programmers, software engineers, graduate students, and upper-level undergraduates too late in the curriculum. Because of the lack of basic skills needed to apply the techniques successfully, attention must be divided between the teaching of the basic skills and the discussion of their advanced applications. Gries believes that the heart of the problem in software engineering lies in no attempt being made to teach methods for formalizing, for solving by calculation, and for checking calculations. In his opinion, the field relies far too much on intuition and guessing.

Gries suggests that the answer to the current problems in undergraduate computer science programs is to overhaul the beginning of the

The development of any system has to be preceded by a specification of what is required. Without such a specification, the system's developers will have no firm statement of the needs of the would-be users of the system. The need for precise specification is accepted in most engineering disciplines. Software systems are in no less need of precision than other engineering products.

Many aspects of a software system must be specified, including its functionality, performance, and cost. In this section, attention is focused on a system's functionality. There is a general agreement that *precise* specifications are a must to obtain quality software. Informal specifications (e.g., in a natural language) alone are certainly not appropriate because they are normally inconsistent, imprecise, ambiguous, and they rapidly become bulky, making it very difficult to check for their completeness. Semiformal approaches (e.g., data flow diagrams, Structured Analysis and Design Technique, SADT) to specification have been developed since the 1970s to improve the practices used in software development. A particular emphasis in semiformal approaches is on diagrammatical representation of the software system being built. Major problems with semiformal methods include their lack of a precise semantics that can be used to reason about or verify the properties of the software system, and their generally "free" interpretation. To overcome the limitations of informal and semiformal approaches, the use of *formal methods* for software development is often encouraged.

5.1. What Are Formal Methods?

Generally speaking, a formal method consists of a set of symbols and an associated set of well-defined, mathematically based rules for inference and manipulation of symbols. A formal method is used to *model* and *reason* about some characteristics, for example, functionality, of a software. The important aspect of formal methods is the use of formalism for describing a software system and for developing the software based on the techniques of mathematical modeling and reasoning. Mathematical modeling, abstraction, and reasoning will help provide a scientific base for the software system in the same way as they have done for other engineering disciplines. Indeed, if our objective is to make software engineering a true engineering discipline, we must ensure that the methods we use are scientific and, by extension, have a mathematical foundation so that they provide predictable and verifiable results.

Formal methods can be used during various stages of software development and for variety of purposes. They can be used during the early stages for modeling and specification purposes and during later stages for program verification. Whereas formal methods have traditionally

LibSystem

$$members : \mathbb{F}\, Person$$
$$shelved : \mathbb{F}\, Book$$
$$checked : Book \pfun Person$$

$$shelved \cap \operatorname{dom} checked = \emptyset$$
$$\operatorname{ran} checked \subseteq members$$
$$\forall mem : Person \bullet \#(checked \rhd \{mem\}) \leq MaxLoan$$

The state variables include: (1) *members*, a finite set of type Person representing those who can borrow books from the library; (2) *shelved*, a finite set of type Book representing the set of books currently in the library; and (3) *checked*, a partial function (shown by $\pfun$) from type Book to type Person used for representing who borrowed which book. The predicates capture the properties of the system: the first predicate states that no book can be on shelves and checked out at the same time; the second predicate states that only members can check out books; and the third predicate states that each member can borrow as many as *MaxLoan* books. (The symbol # reads as *size of*, while symbol $\rhd$, for range restriction, restricts the set *checked* to those entries whose range is *mem*.) The schema *CheckOut* specifies the properties of the book checkout operation:

CheckOut

$$\Delta LibSystem$$
$$borrower? : Person$$
$$book? : Book$$

$$book? \in shelved$$
$$borrower? \in members$$
$$\#(checked \rhd borrower?) < MaxLoan$$
$$checked' = checked \oplus \{book? \mapsto borrower?\}$$
$$shelved' = shelved \setminus \{book?\}$$
$$members' = members$$

The declaration $\Delta LibSystem$ alerts that the schema is describing a state change. Thus we have to specify clearly which elements of *LibSystem* will be changed. Other items of the signature include *borrower?* and *book?*, both adorned with a question mark symbol to show that they are *input* items. The first three predicates describe the preconditions of checkout: the requested book must be on shelves, borrower must be a member, and must have borrowed less than *MaxLoan*. The last three predicates represent the state changes. Note that all state variables of *LibSystem*

- *Unambiguous interpretation.* Different interpretations are avoided because the elements of the notation have a well-defined meaning.
- *Conciseness.* By using a mathematical notation (such as Z), it is possible to express complicated facts and relations in a short space that would otherwise take a large number of noise words and verbose sentences to describe in a natural language.
- *Ease of reasoning.* Mathematics provide one with the ability to deduce useful results, or to use inference rules and theorems to check results of propositions.
- *Modeling reality.* A mathematical notation has concepts that can be used for abstract modeling of reality. In fact, mathematical notations have been used in other areas such as engineering for modeling and the successful development of reliable systems. The same concepts should be exploited for software development.

In a recent article by Cooke [15], mathematics, theory, and the application of formal methods are explained in greater detail. (In fact, that same issue of *The Computer Journal* is dedicated to formal methods and contains several other inspiring articles.)

5.4. Misconceptions about Mathematical Specification

One misconception about the use of formal methods for software specification is that mathematical methods are not used in industry for real projects. The fact is that they are. There are numerous reports on practical and effective use of formal methods in industry. The range of applications, from small to large systems, possessing attributes such as real time, interactive, robust, and secure, has been sufficiently wide to test the viability of formal methods, and to establish a considerable body of knowledge and experience. A number of authors (e.g., [16, 17]) discuss experience gained in applying formal specification techniques to IBM's CICS transaction processing system. CICS is a large, 20-year-old system and contains over half a million lines of code. The language Z was used by IBM to *respecify* CICS to improve its maintainability.

The industry's concern for mathematical techniques is also evident from a recently completed report by the Computer Science and Technology Board [18]. This report points to the need for strengthened mathematical foundations in software engineering courses:

> As software engineers begin to envision systems that require many thousands of person-years, current pragmatic or heuristics approaches begin to appear less adequate to meet application needs. In this environment, software engineering leaders are beginning to call for more systematic approaches: More mathematics, science, and engineering are needed.

6. CONCLUSIONS

It is evident from the recommendations in Curriculum '68 that those responsible for the recommendations for academic undergraduate programs in computer science knew from the beginning that mathematics is a crucial component in computer science. Also evident from the progressive recommendations toward mathematics from Curriculum '68 to Computing Curricula 1991 is the increased realization of the extent and degree to which discrete mathematics is and should be involved in the computer science areas. In Curriculum '68 it was recommended that at least 18 hours of mathematics be taken by those majoring in computer science. In Curriculum '78, the committee responsible for the report, while recommending that "courses concentrating on discrete mathematics which are appropriate to the needs of computer scientists" should be developed, it overemphasized the practical aspects of computer science. In the current Computing Curricula 1991 there are as many as 21 hours of mathematics and it is recommended that the courses covered must include discrete mathematics and even advanced discrete mathematics.

The views of several computer scientists offer even greater evidence of the importance of mathematics to computer science. Dijkstra points out that: (1) well-chosen formalisms provide a shorthand with which no verbal rendering can compete; (2) by viewing the purpose of mathematical logic as providing a calculational alternative, its potential as a tool for practical proof design can be realized; (3) in proof design, strong heuristic guidance can be extracted from a syntactic analysis of the theorem and from proof theory; and (4) the effective techniques of symbol manipulation are well within the teachable domain. Ralston asserts that there seems to be a general consensus among academic computer scientists for a more mathematical context within undergraduate computer science programs which needs to be integrated more closely with computer science courses and that traditional undergraduate mathematics (i.e., the calculus sequence) is not usually supportive of the computer science curriculum. Gries believes that software engineering, computing, and computing education are all plagued by a deficiency in basic mathematical skills that are needed to utilize algorithmic concepts. He attributes this problem to the fact that generally no attempt is made to teach methods for formalizing, for solving by calculation, and for checking calculations. Gries suggests that, to remedy the problem, undergraduate computer science programs should be revised by merging the contents of programming and mathematics courses. These views show how vital it is that those involved in the computer sciences also be involved in the

[11] A. Berztiss, "A Mathematically Focused Curriculum for Computer Science," *Communications of the ACM*, Vol. 30, No. 5, pp. 356–365, May 1987.

[12] Edsger W. Dijkstra, "A Debate on Teaching Computing Science," *Communications of the ACM*, Vol. 32, No. 12, pp. 1397–1414, 1989.

[13] Anthony Ralston, *Discrete Mathematics in the First Two Years*. The Mathematical Association of America, 1989.

[14] David Gries, "Teaching Calculation and Discrimination: A More Effective Curriculum," *Communications of the ACM*, Vol. 34, No. 3, pp. 45–55, 1991.

[15] J. Cooke, "Formal Methods: Mathematics, Theory, Recipes or What?, *The Computer Journal*, Vol. 35, No. 5, pp. 419–423, 1992.

[16] C. J. Nix and B. P. Collins, "The Use of Software Engineering, Including Z Notation, in the Development of CICS," *Quality Assurance*, Vol. 1, No. 9, pp. 103–110, September 1988.

[17] I. Hayes, "Applying Formal Specification to Software Development in Industry," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 169–178, February 1985.

[18] "Computer Science and Technology Board Report: Scaling Up: A Research Agenda for Software Engineering," *Communications of the ACM*, Vol. 33, No. 3, pp. 281–293, March 1990. Excerpted.

[19] A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, Vol. 7, No. 5, pp. 11–19, September 1990.

[20] J. C. Cherniavsky, "Software Failures Attract Congressional Attention," *Computer Research Review*, Vol. 2, No. 1, pp. 4–5, January 1990.