Towards More Formalism in Software Engineering Education

Hossein Saiedian
Dept. of Computer Science
University of Nebraska
Omaha, NE 68182

Abstract

Improving students' understanding of software engineering concepts and increasing the productivity of software engineers requires new ways of thinking and reasoning about software and better ways of producing it. To gain intellectual control over the software development process and become more productive and efficient, we encourage the use of formal methods. The use of formal methods requires that students be educated and software engineers be trained. We make a distinction between education and training. Education is the long term goal designed to build a foundation of knowledge for the students while training is a short term activity designed for software engineers. The challenge to educators is to provide the appropriate foundation for software engineering students. This issue is discussed in the paper.

1 Background

It has been more than 20 years since universities established academic programs in computer and information sciences. Graduates of these programs are hired by industry and government to develop and maintain computer programs which are used to keep banking and hospital records, assist in controlling air traffic, maintain inventories, etc. Sometimes these computer programs help engineers design buildings and bridges and build cars and airplanes. Thus the *non-engineering* graduates of computer science programs develop products that may be used for building engineering artifacts such as cars and airplanes. It is time to examine and ensure that this back door to engineering possesses appropriate *engineering* knowledge. Engineering is viewed as the application of scientific knowledge and formal methods for designing useful products and for other practical purposes. Software engineering must be viewed in the same way.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ACM-24thCSE-2/93-IN,USA
© 1993 ACM 0-89791-566-6/93/0002/0193...$1.50

The computer science programs thus must embody enough material to ensure that future software engineers are able to use scientific knowledge and formal methods in the construction of computer programs. Computer science students must be convinced that software engineering is in fact an engineering discipline.

According to David Gries [3], a pioneer in program development, software engineering lacks a sense of professionalism and responsibility because:

- few software engineers know how to write a good specification,
- few software engineers view a specification as a legal contract, and
- few software engineers stand 100% behind their software product.

An explanation of these conclusions is that the software engineering students have not been exposed to means of acting professionally, and without the means, no amount of exhortation to act professionally can help [3]. The students are weak on the fundamental concepts; their computer science knowledge is primarily focused on the narrow areas of programming languages and classical operating systems problems. Most importantly, they are never exposed to the discipline associated with engineering. They accept the bizarre inconsistency and unpredictable behavior of current systems as normal.

In this paper, we *emphasize* curriculum materials which are in the direction of increased formalism in software development. This includes discussion of necessary course work and tools that would help students appreciate the need for formal methods. Our goal is not to propose a new curriculum; nor is it to single out any particular department's curriculum[1] or any particular report as a paradigm.

[1] We do believe that many current undergraduate curricula are focused on the very narrow areas of programming, programming languages, and operating systems and that treatment of mathematics and logic in these curricula is often quite shallow.

2 Formal Courses

In this section we discuss the courses that we believe should be emphasized more strongly in the computer science curriculums for exposing the students to the concepts related to formal methods.

2.1 Discrete Mathematics

Discrete mathematics is a study of calculations involving a finite number of steps and is the foundation for much of computer science. It focuses on the understanding of concepts and provides invaluable tools for thinking and problem solving. Discrete mathematics is especially important when a computer science student is not required to study much of ancillary mathematics.

Computer science students should take at least a one semester course in discrete mathematics covering fundamentals topics such as set theory, functions, relations, graphs, and combinatorics. There have been few attempts to teach these topics to freshmen/sophomores in a thorough fashion that would relate discrete mathematical concepts to computer science and software development.

2.2 Mathematical Logic

Logic is fundamental to many of the notations and concepts in computing science. Mathematical logic allows students to mathematically formulate and solve a wide class of problems, is fundamental in understanding the meaning of algorithms, and represents the foundation of logic programming. Students thus must have deep understanding of concepts such as decision procedures and higher order logic, and the relationship between set theory and lambda calculus. A one semester course which focuses on these concepts is essential.

In his influential textbook, *The Science of Programming*, Prof. David Gries has the following to say about the importance of mathematical logic:

> "The research was fraught with lack of understanding and frustration. One reason was that computer scientists in the field, as a whole, did not know enough formal logic. Some papers were written simply because the authors didn't understand earlier work; others contained errors that wouldn't have happened had the authors been educated in logic ... We spent a good deal of time thrashing, just treading water, instead of swimming, because of our ignorance. With hindsight, I can say that the best thing for me to have done ten years ago would have been to take a course in logic. I persuaded many students to do so, but I never did so myself."

The above statements imply that professors of computer science should also consider taking a course in mathematical logic if they do not possess an adequate background.

The mathematical logic course and the discrete mathematics course should enhance a student's ability for abstract specification and the mathematical skills for specifying, manipulating, and analyzing programs.

2.3 Formal Specification

In addition to the above courses, students should take a formal specification project course. Such a course ties together the abstract concepts learned in the discrete mathematics and logic courses and provides an opportunity to make practical use of these concepts.

Students must have sufficient experience to be able to appreciate the need for proper specification. This experience may be developed in such a class and could come from the ad hoc development of a program of some complexity or better still, from attempts to modify a poorly documented and poorly modularized system.

The above course should not simply survey several different approaches but rather give in-depth practice with one or two approaches that have proven useful (e.g., Z).

One misconception about formal methods is that is that they are too mathematical and too complicated requiring a Ph.D. to understand them. Formal methods *are* based on mathematics. However, the mathematics of formal methods are not difficult to learn. Using them requires some training, but experience has shown that such training is not difficult and that people with only high school math can develop the skills to write good formal specifications. Most popular formal specification languages (e.g., Z and VDM) employ only a limited branch of mathematics consisting of set theory and logic. The elements of both set theory and logic are easily understood and are taught early in high school these days. Certainly, anyone who can learn a programming language can learn a specification language like Z. In fact learning a specification language such as Z should be easier than a programming language like COBOL. Z is smaller; it is abstract and is implementation-independent.[2] The specification of a problem in Z is shorter and much easier to understand than its expression in a programming language like COBOL.

[2]For example, Z uses data formalized data types (e.g, sets) instead of types which conform to those generally in use in programming languages (e.g., arrays).

The negative perception of the role of mathematical techniques in software specification is very unfortunate. When problems become very large and complex in other engineering disciplines, they turn to mathematics for help. Unfortunately, some software engineers feel that formal methods and mathematical tools are of academic interest only and that real software systems problems are too large and complex to be handled by formal methods and mathematical tools.

The need for a broader use of mathematical techniques and concerns for lack of rigor and accountability in software engineering is not felt just by the computer scientists. Consider, for example, a recent report released by the Subcommittee on Oversight of the House of Representatives Committee on Science, Space and Technology [2]. This report addresses the problem of software reliability and quality and criticizes universities for not providing adequate education for software engineers. In an article summarizing this congressional report, Cherniavksy [1] writes:

> "[... there is] a fundamental difference between software engineers and other engineers. Engineers are well trained in the mathematics necessary for good engineering. Software engineers are not trained in the disciplines necessary to assure high-quality software ..."

Another negative perception and perhaps fear about formal specification languages is the presence of mathematical symbols. But such symbols are introduced to make mathematics easier and less wordy. The difficulty in learning is not symbols any more than difficulty in learning Farsi is learning Farsi alphabets.

The advantages of mathematics and its positive role in producing good specifications are summarized by Ince [5] as:

- Conciseness. While a natural language is verbose and wordy, a mathematically-based notation is concise and contains concepts that can be used to represent complicated relations that would otherwise be expressed in a large number of words.

- Ease of Reasoning. Mathematics provides one with the ability to deduce useful results, or to use reasoning rules and theorems to check results of propositions.

- Unambiguous and Well-defined Meanings. Compare the following example of informal specification:

  > *The lamp voltage will always be a whole number of volts between 3 and 6.*

  with the following mathematical expression:

  $$voltage \geq 3 \wedge voltage \leq 6$$

  While the former can be read ambiguously, the latter expression has only one interpretation. In addition to the above, even if two notations, one mathematical and one informal, can be "proved" to be equivalent, they may differ in the ease with which they can be evaluated or manipulated. It is fair to say that mathematical expressions are easier to manipulate than informal expressions.

- Modelling Reality. Engineers very often use mathematics to model reality and have been very successful in developing reliable systems.

- Suppression of Unnecessary Detail. This is because mathematicians use abstract values instead of actual values and because they use symbols instead of large structures.

- Mathematics is constant. For example, the concept of set theory and the methods of representing it, including its operations, properties, etc., has been the same almost from its very genesis. Compare this with all different methods invented for software specification and design during the last 20 years.

3 Training for Professionals

Since their jobs are product oriented, software engineers require a different kind of education than that typically offered by research institutions and computer science departments. The ideal approach for educating the practitioners is to develop a curriculum for a graduate professional degree (analogous to an MBA degree but perhaps with less course work). Such a curriculum would cover the necessary background for using formal methods (e.g., discrete mathematics courses covering sets and logic) and would present a variety of principles, tools, and skills in applying formal methods during software development. Such a professional curriculum is, unfortunately, not very practical now but it should be considered for the near future.

The professional degree is not the only approach. A good deal of knowledge concerning formal methods in software engineering resides in professional workshops in industry and can be attained through apprenticeship. Typical workshops on formal methods present

concepts and comparisons of various types of specifications for different software components (e.g., data structures, files, single procedures, composite objects, programs, etc.). Examples are developed and the relationships between formal specifications and other topics such as logic programming, program verification and "clean-room development" are illustrated.

We suggest the following hints for the software engineering professional:

- Training in discrete mathematics covering elementary set theory and logic should be the first step. For those who have a mathematical background but are unfamiliar with the basic concepts of set theory and propositional logic one or at most two days suffices to introduce the ideas. For others one week of training is required.
- Training in a particular formal method such as Z or VDM should be the next step. Such training typically takes one to three weeks, once the participant has the necessary mathematical background.
- Tutoring and consultation in a real project is helpful, so is participation in workshops where one can tackle a problem with the help of a tutor.

Most practitioners perceive formal methods as academic tools which are difficult to use. They are reluctant to use them despite their considerable advantage over traditional methods. A study needs to be done to discover what it will take to move formal methods from this unfair perception into a wider acceptability within software projects. Case studies must be developed to demonstrate the applicability of formal methods with the intention of convincing the practitioners that the benefits outweigh the difficulties of transition and result disseminated.

To demonstrate that formal methods pay off, more realistic, large scale examples performed in conjunction with industry (e.g., IBM's CICS) are necessary. These industrial case studies not only are necessary for advancing the technology and demonstrating the potential benefits, they also help identify the needs of companies that adopt formal methods and enhance the integration of formal methods with current software engineering practices. In general, formal methods tend to address semantic issues rather than pragmatics of a software. Managers and practitioners are however most concerned with pragmatics issues. An understanding would help researchers delineate more precisely where formal methods are most useful.

Case studies also assist in identifying the limits of formal methods. Formal methods have proven very useful for specification of functional properties of a system. Non-functional properties of a software system such as reliability, cost, performance, portability, man-machine interface, and resource consumption of running programs are difficult or impossible specify by means of formal methods. Research needs to be done to find out if formal methods can in fact be used for such purposes. It is only in the practical applications that the limits or constraints of formal methods are revealed.

4 Students Need Tools

Students learn more by active participation than just by observing. Theoretical concepts (such as discrete mathematics and graph theory) should be reinforced with hands on experience in labs. Since such courses should be taught early in the college career (to provide necessary background for high-level courses), educators must ensure that the students learn the concepts well. As is often the case, the students have difficulty with theoretical concepts that are described in the books using definitions, theorems, and proofs. A tool which visualizes theoretical concepts and allows a student to experiment with these concepts creates a creative environment. Such a tool is helpful in solving various discrete math problems which would be tedious to work by hand. Freed from mechanical aspects of these calculation, the students can focus their attention on the concepts which form the basis of the material being studied.

One factor limiting the use of formal methods is the lack of investment in automated tools and support structures to reduce the efforts of applying these methods. In fact, lack of support tools is often seen as a major barrier the to use formal methods. A key factor in the acceptance of high-level languages has been the presence of a comprehensive set of tools to support the user. If formal methods languages are to achieve the same level of acceptance, they too will require extensive automated support. Support tools may reduce the learning time, thereby aiding their wide spread use. Automated tools may include:

- Special editing environments
- Syntax checkers
- Animation tools
- Refinement and Proof Tools

For example, a special editing environment for the specification language Z would provide a specifier with a number of pop-up menus from which the specifier could view global schemas, local schemas, state

schemas, operation schemas, defined sets, etc. The editor would also make schema creation, modification, deletion, etc., more flexible.

In addition to the above, good interfaces to specification languages, transformation tools for taking popular methods and converting them into formal methods, and tools for inferring from specifications to assist software validation are needed.

5 Conclusions

Formal methods enable a software engineer to specify a system via a rigorous mathematical notation. Such an approach to specification can eliminate many of the problems associated with software development such as ambiguity, impreciseness, incompleteness and inconsistency. The errors are discovered and corrected more easily; moreover error detection is accomplished not through an ad hoc review, but through application of mathematical analysis. When used during the early stages of software development, formal methods enable the software developer to discover and correct errors that otherwise might go undetected, therefore increasing the quality of the software and its maintenance and decreasing its failure rate as well as its maintenance cost. Although such properties are the objective of all specification methods, the use of formal methods results in a much higher likelihood of achieving these ideas. While it may be easier to educate software engineering students in formal methods within an academic setting, it is less easy to convince the industry to accept such methods. More research and case studies are required to minimize industry's misconception of formal methods.

There is a rather large volume of literature on formal methods, their applications, and their advantages. In particular, the the September 1990 issue of *IEEE Software*, 7 (5), contains a series of readable tutorial articles discussing different aspects of formal methods. These are excellent articles to be studied in a senior-level software engineering or software specification course. The article by Anthony Hall [4] about the myth of formal methods is particularly recommended. More technical papers appear in the complementary issues of the *IEEE Computer* and *IEEE Transactions on Software Engineering* of the same month.

Acknowledgment

The author is grateful to the anonymous referees for their comments and corrections.

References

[1] J.C. Cherniavsky. Software failures attract congressional attention. *Computer Research Review*, 2(1):4–5, January 1990.

[2] GPO 052-070-06604-1. Bugs in the program — problems in federal government computer software development and regulation. Superintendent of Documents; Government Printing Office; Washington, D.C., 20402, 1989.

[3] D. Gries. Positional statement on the foundation of software engineering. In G.X. Ritter, editor, *Information Processing 89*. Elsevier Science Publishers, North-Holland, 1989.

[4] A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, September 1990.

[5] D. Ince. *An Introduction to Discrete Mathematics and Formal System Specification.* Oxford University Press, 1990.