BY
PETER B. HENDERSON

MATHEMATICAL REASONING IN SOFTWARE ENGINEERING EDUCATION

THE ENGINEERING PROFESSION IS A BRIDGE BETWEEN SCIENCE AND MATHEMATICS AND THE TECHNOLOGICAL NEEDS OF ALL PEOPLE. ALL ENGINEERING DISCIPLINES ARE FUNDAMENTALLY BASED ON MATHEMATICS AND PROBLEM SOLVING. TRADITIONAL ENGINEERING DISCIPLINES, INCLUDING CHEMICAL, CIVIL, ELECTRICAL AND MECHANICAL, RELY ON CONTINUOUS RATHER THAN DISCRETE MATHEMATICAL FOUNDATIONS. SOFTWARE ENGINEERING IS AN EMERGING DISCIPLINE THAT APPLIES MATHEMATICAL AND COMPUTER SCIENCE PRINCIPLES TO THE DEVELOPMENT AND MAINTENANCE OF SOFTWARE SYSTEMS, RELYING PRIMARILY ON THE PRINCIPLES OF DISCRETE MATHEMATICS, ESPECIALLY LOGIC.

Discrete mathematics, especially logic, plays an implicit role in software engineering similar to the role of continuous mathematics in traditional physically based engineering disciplines.

What role does mathematics play in software engineering? Consider the following two statements: Software practitioners do not use mathematics; and Software practitioners need to think logically and precisely. They represent an apparent contradiction in light of the similarity of the reasoning underlying software engineering and mathematics [3]. Perhaps software practitioners who say, I don't use mathematics, really mean, I don't use mathematics explicitly or formally. Many practicing engineers don't explicitly use calculus on a daily basis but do implicitly use mathematical reasoning all the time. Similarly, software engineers

ILLUSTRATION BY JEAN-FRANÇOIS PODEVIN

should learn to use foundational discrete mathematics concepts and logical reasoning at all times.

Ask traditional engineers if calculus should be eliminated from undergraduate engineering curricula; the answer would be no. In contrast, practicing software engineers have argued that mathematics is not that important in software engineering education since practitioners don't use it explicitly [4]. Was it continuous or discrete mathematics (or both) that software engineers considered less important? The answer was unclear. The role of discrete mathematics and logic in software engineering today is not well understood by either academic researchers or industrial practitioners. This lack of understanding will change as the discipline matures and academics and practitioners work together to develop that role, making it similar to the role of continuous mathematics in traditional engineering disciplines.

WHICH HUMAN ENDEAVOR WAS DEVELOPED TO DEAL WITH ABSTRACTION?

MATHEMATICS.

Physical vs. Software Engineering

Recent articles by software engineering educators have described the similarities and differences between traditional engineering disciplines and software engineering; for example, see [9]. One major difference is that traditional engineers construct real, physical artifacts, while software engineers construct non-real, abstract artifacts. The foundations of traditional engineering disciplines are mature physical sciences and continuous mathematics, whereas those of software engineering are less mature abstract computer science and discrete mathematics. In physical engineering, two main concerns for designing any product are cost of production and reliability measured by time to failure. In software engineering, two main concerns are cost of development and reliability measured by number of errors per thousand lines of source code. Both traditional and software engineering disciplines require maintenance but in different ways.

All engineering disciplines involve developing and analyzing models of the desired artifact. However, the methods, tools, and degree of precision differ between traditional and software engineering. Abstract modeling and analysis are mathematical in nature. They are very mature in traditional engineering and maturing slowly in software engineering. An example discussed in the following paragraphs demonstrates how mathematical reasoning is used in both traditional and software engineering.

In electrical engineering, the voltage decay as a function of time t of a resistor-capacitor (RC) circuit is specified by the function $V(t) = Vo\ e^{(-t/RC)}$, where R is the resistance, C the capacitance, and Vo the initial capacitor voltage. This model of the behavior of RC circuits is derived from principles of mathematical circuit design using foundational calculus and differential equations. Electrical engineering students learn this derivation in their electronic circuits course after studying calculus and differential equations. Here, mathematics-based reasoning is used to derive and understand a fundamental concept.

Iteration invariants represent a foundational concept few computer science or software engineering graduates understand, appreciate, or use effectively, even though they are important for deriving, understanding, debugging, and documenting algorithms. Every iteration has a predicate I(S) (S represents the current state of the computation) called the iteration/loop invariant, that captures the underlying meaning of the iteration (such as sum the values in a list, search a tree structure, and compute the tax due by all taxpayers). Mathematical logic can be used to argue that the predicate I(S) satisfies logical constraints, as in Figure 1 for the `while-do` iteration; here, the red stuff in brackets represents logical assertions, and C(S) is a side-effect free Boolean condition. Upon termination, another mathematical issue, { I(S) and not C(S) are true }, must logically imply (=>) the desired post-condition. The use of this approach for linear search is discussed later.

Software engineering students can learn to use mathematical reasoning to model, derive, understand, debug, and document software systems. With enough practice, the underlying mathematical concepts become intrinsic to their thought processes, supporting rather than hindering their thinking.

```
{pre-condition}
Initialization code
{I(S) is true}
while C(S) is true do

  [ {I(S) and C(S) are true}
  [   <code for body of iteration>
  [ {I(S) is true}

{I(S) and not C(S) are true  ==>  post-condition}
```

Figure 1. Logic of a while-do iteration.

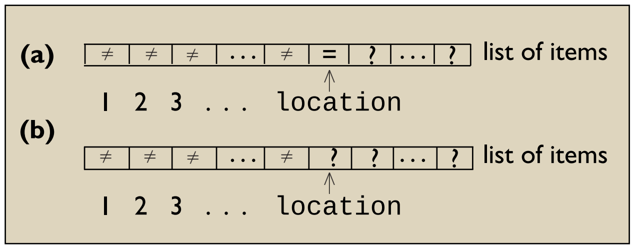

Figure 2. (a) Post condition and (b) potential iteration invariant.

Mathematics and Software Engineering

Key reasons for wanting to learn and use mathematical reasoning include:

Abstract software. Constructing non-real (abstract) artifacts requires abstract reasoning. Which human endeavor was developed to deal with abstraction? Mathematics. Hence one view of a software system is as a mathematically precise model of some desired process or computation. Mathematics is one tool for reasoning about software systems, as well as for practitioners' rigorous reasoning and analysis.

Notations, symbols, abstractions, precision. The expression `y = ax + b` is familiar from algebra, and `count == 0` is familiar from programming. Each uses notations and symbols and is precise, given the types of data and semantics of the operations, specified mathematically. Learning a formal notation is no more difficult than learning a programming language. Indeed, it is often easier, as the syntax and semantics are cleaner. Programming appeals to our innately process/imperative-oriented minds, and programming tools breathe life into programs. Mathematics tends to be declarative and static, though such tools as Axiom, Mathematica, and Maple help mitigate this perception.

Modeling software systems. A model, even a mental one, must be created before construction of any artifact can begin. Most software development is like creating art whereby an initial vision slowly takes form. Planned evolution, along with maintenance issues, is often ignored. This process may be acceptable for some software projects, but with many such projects the more the software engineer can learn and understand early, the better. Modeling is one vehicle for achieving this understanding, and mathematics is an important tool for building, checking, analyzing, and experimenting with models [2]. Moreover, developing precise models using specification languages, including (in order of popularity) Z, Larch, and Alloy, is important for identifying specification errors, which are very costly to correct once a software system is implemented [6].

Application domains. Software practitioners can use mathematics to communicate with their colleagues, including engineers, scientists, mathematicians, statisticians, actuaries, and economists. Mathematics is a rich, comprehensive, universal language for communication between such diverse groups.

Mathematical reasoning. One definition of mathematical reasoning, attributed to an informal working group of computer science and math educators (www.math-in-cs.org) [5] is: "Applying mathematical techniques, concepts, and processes, either explicitly or implicitly, in the solution of problems; in other words, mathematical modes of thought that help us solve problems in any domain." In the most general interpretation, every problem-solving activity is an application of mathematical reasoning. For example, consider the benefits of exercises requiring students to translate English statements into prepositional or predicate logic form; these "modeling" exercises help them be more precise and inquisitive about the interpretation of English statements. When clients or colleagues say, "A or B," do they mean "inclusive or" or "exclusive or"? When they say, "for all ...," do they mean universal quantification? What is the intended meaning of "for all ..." when there are no elements over which to quantify?

Mathematics in Software Engineering

The article "Why Math?" by Bruce et al. in this section describes several example applications of mathematics in computer science. The following simple linear search problem illustrates the use of logical reasoning to derive an algorithm. The problem statement is: Find the location of the first instance of a specified item in a list of items; the specified item is known to be in the list. Develop an algorithm for this problem. Think about the following questions (hopefully the ones students would be asking themselves): How many items are in the list?; Can the list be empty?; What happens if the specified item appears more than once?; and What is meant by "first instance" and by "location"? Addressing these questions and using the given problem information enables software engineers to formulate representative pre and post conditions required to ensure the problem is well defined.

Most undergraduate students would love to see such a problem on a comprehensive exit exam.[1] Applying a familiar pattern, or template, leads to a few lines of code (simple). When developing computer programs, do students simply pattern-match to get an approximate program, then use extensive testing to refine it? Or do they really understand the underlying logic?

√ IN THE MOST GENERAL INTERPRETATION, EVERY PROBLEM-SOLVING ACTIVITY IS AN APPLICATION OF MATHEMATICAL REASONING.

To understand the relevance of this issue, consider a variant of the problem as it was presented to professional programmers in a tutorial session [1]. They had to compose a program for binary search, a search strategy that iteratively divides the list in half. Approximately 90% of them got it wrong, unable to identify proper pre and post conditions, apply a familiar pattern, compose a correct algorithm, or express it in a programming language.

Returning to the linear search problem, consider a solution strategy based on the foundational concepts of logic and iteration invariants. First, the algorithm developer identifies the pre and post conditions to clearly specify the problem. They can be presented formally using predicate logic; for this discussion, I offer a picture of the post condition (see Figure 2a). That is, the desired item is not found ($\neq$) in the list before `location` and the desired item is found (=) at `location` of the list. As the algorithm iterates, the not found ($\neq$), knowledge accumulates by advancing the value of `location` from 1, 2, 3, …; the iteration terminates once the item is found (=). These factors lead to a potential iteration invariant (see Figure 2b) and a potential iteration termination condition, that is, iterate as long as the desired item $\neq$ item in `location` of the list.

Figure 3 outlines a partially complete linear search algorithm with logical assertions and iteration invariants; the color blue denotes the item referenced by the variable `location`. Note that the iteration invari-

[1] Few colleges and universities give such exams.

ant is true prior to entering the iteration when location = 1, that is, it is vacuously true.[2] Indeed, this is often the case for iterations, another reason students must understand the logical concept of vacuous. (You may complete the algorithm using the red stuff to derive the `body of the iteration`.)

One may argue that the red stuff is useless clutter.

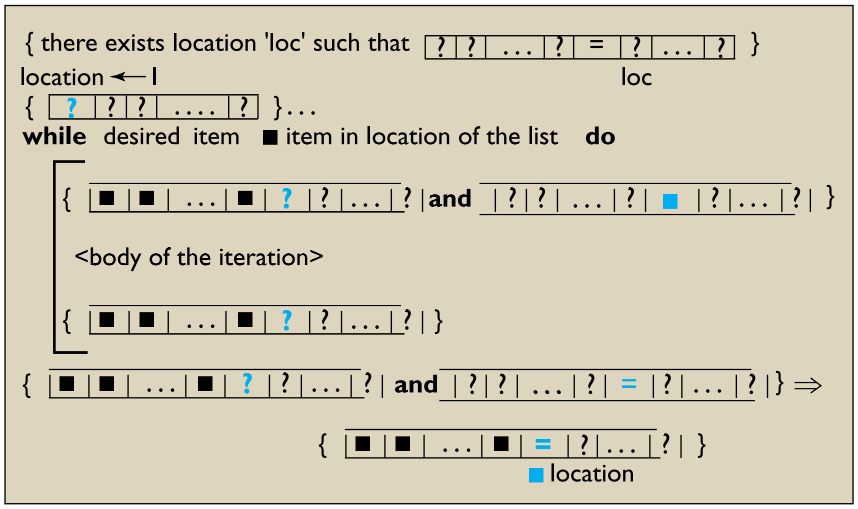
{ there exists location 'loc' such that | ? | ? | . . . | ? | = | ? | . . . | ? | }
location ← l
loc
{ | ? | ? | ? | . . . . | ? | }. . .
while desired item ■ item in location of the list do
{ | ■ | ■ | . . . | ■ | ? | ? | . . . | ? | and | ? | ? | . . . | ? | ■ | ? | . . . | ? | }
<body of the iteration>
{ | ■ | ■ | . . . | ■ | ? | ? | . . . | ? | }
{ | ■ | ■ | . . . | ■ | ? | ? | . . . | ? | and | ? | ? | . . . | ? | = | ? | . . . | ? | } ⇒
{ | ■ | ■ | . . . | ■ | = | ? | . . . | ? | }
■ location

Figure 3. Linear search algorithm.

Perhaps, but it captures the natural reasoning our minds use. It is explicit here. This is the type of reasoning—from first principles—software engineering students should be able to perform. However, this does not mean they must use it all the time but be ready to apply it as needed, when, for example, a derived software system must be correct or when understanding, debugging, or documenting software systems.

What is an iteration invariant for binary search? Understood intuitively, it is that the location of the desired item is constrained between two other locations—low and high—and the algorithm adjusts to converge on the potential location of the desired item; the desired item may not even be on the list. This "intuitive" invariant is useful to software engineers for deriving a correct algorithm. What percentage of the programmers described in [1] would have solved the problem correctly if they had reasoned more mathematically?

Which Mathematics for Software Engineers?

Educational foundations are being identified as computer science and software engineering mature. For example, the ACM/IEEE *Computing Curricula 2001: Computer Science* [10], a guideline for undergraduate computer science programs required discrete mathematics in its core, recommending it be taken early in the undergraduate curriculum. Meanwhile the ACM/IEEE *Computing Curricula 2001: Software Engineering* [7] for undergraduate software engineering programs adopted and extended this foundational model. Included in its two-course mathematical foundations core (E = essential, D = desirable, and O = optional) are the following topics:

Functions, Relations, and Sets (E);
Basic Logic (prepositional and predicate) (E);
Proof Techniques (direct, contradiction, inductive) (E);
Basic Counting (E);
Graphs and Trees (E);
Discrete Probability (E);
Finite State Machines, regular expressions (E);
Grammars (E);
Algorithm Analysis (E);
Number Theory (D); and
Algebraic Structures (O).

Foundational core material included:

Abstraction:
- Generalization;
- Levels of abstraction and viewpoints;
- Data types, class abstractions, generics/templates; and
- Composition;
Modeling:
- Principles of modeling;
- Pre and post conditions, invariants;
- Mathematical models and specification languages;
- Model development tools and model checking/validation;
- Modeling/design languages (such as UML, OOD, and functional);
- Syntax vs. semantics (including understanding model representations); and
- Explicitness (make no assumptions or state all assumptions)

Though not explicitly in the core, a year of calculus was also required to: enhance mathematical maturity and thinking; provide a contrast with discrete mathematics concepts; and ensure sufficient background

[2]There is nothing to make it false, so it must be true "vacuously."

for various client and application disciplines. Statistics and empirical methods were also recommended; personally, I would also add linear algebra. Advanced mathematical courses, including graph theory, combinatorics, theory of computing, probability theory, operations research, and abstract algebra, might also be required, depending on the goals of the program and the needs of individual students.

Mathematically oriented software engineering courses today cover formal specifications, formal methods, mathematically rigorous software design, software verification and validation, and software models and model checking. As software engineering matures, the word "formal" will disappear; it is rarely used in traditional engineering where formal approaches are the norm.

Specific material varies by degree program. However, an important goal is to ensure foundational mathematical concepts are introduced early and used and reinforced in computer science and software engineering courses in the same way continuous mathematics is used and reinforced in traditional engineering courses. It will, however, take time, dedication, and rethinking the current software engineering curricula.

Conclusion

Mathematical reasoning is intrinsic to both traditional engineering and software engineering, though each discipline uses different foundational mathematics. Traditional engineers use continuous mathematics primarily in a calculational mode for modeling, design, and analysis, including to calculate, say, load on a bridge component, compute the wattage of a resistor, or determine the optimum weight of an automobile suspension system. Software engineers usually use discrete mathematics and logic in a declarative mode for specifying and verifying system behaviors and for analyzing system features.

The RC circuit and iteration invariants described earlier illustrate basic mathematical reasoning. However, engineering is significantly deeper and broader than these examples indicate. Engineers are systems architects who understand and apply the foundational principles of the discipline. Software engineers must therefore learn to use mathematics to: construct, analyze, and check models of software systems; compose systems from components; develop correct, efficient system components; specify (precisely) the behavior of systems and components; and analyze, test, and evaluate systems and components. They must therefore understand the theoretical and practical principles of programming and be able to learn and use new languages and tools.

One area where traditional engineering has an advantage is the number, variety, and maturity of tools for mathematical modeling, design, analysis, and implementation, including standard languages for communication in blueprints and schematic circuit diagrams and computer-aided prototyping, design, and analysis tools. Comparable software engineering tools are emerging as the discipline matures.

Evidence supporting the importance of mathematics in software engineering practice is sparse. This naturally leads to claims that software practitioners don't need to learn or use mathematics [4]. Surveys of current practices [8] reflect reality; many software engineers have not been taught to use discrete mathematics and logic as effective tools. Education is the key to ensuring future software engineers are able to use mathematics and logic as power{ful} tools for reasoning and thinking. C

References

1. Bentley, J. Programming pearls: Writing correct programs. *Commun. ACM 26,* 12 (Dec. 1983), 1040–1045.
2. Clark, E., Grumberg, O., and Peled, D. *Model Checking.* MIT Press, Cambridge, MA, 1999.
3. Devlin, K. The real reason why software engineers need math. *Commun. ACM 44,* 10 (Oct. 2001), 21–22.
4. Glass, R. A new answer to 'How important is mathematics to the software practitioner?' *IEEE Software 17,* 6 (Nov./Dec. 2000), 135–136.
5. Henderson, P. et al. Striving for mathematical thinking. *SIGCSE Bulletin (Inroads) 33,* 4 (Dec. 2001), 114–124.
6. Hinchey, M. and Bowen, J., Eds. *Applications of Formal Methods.* Prentice-Hall, London, U.K., 1995.
7. LeBlanc, R. and Sobel, A., Eds. *Computing Curricula 2001: Software Engineering Volume (1st Draft)*, June 25, 2003; see sites.computer.org/ccse/volume/FirstDraft.pdf.
8. Lethbridge, T. What knowledge is important to a software professional? *IEEE Comput. 33,* 5 (May 2000), 44–50.
9. Parnas, D. Software engineering programmes are not computer science programmes. *Annals Software Engin. 6,* 1–4 (1998), 19–37.
10. Roberts, E., Ed. *Computing Curricula 2001: Computer Science Final Report*. IEEE Computer Society, New York, April 2002.

Peter B. Henderson (phenders@butler.edu) is a professor in and head of the Department of Computer Science and Software Engineering at Butler University, Indianapolis, IN.