# "Mathematical Reasoning in Software Engineering Education"

Peter B. Henderson
Butler University

## Introduction

Engineering is a bridge between science and mathematics, and the technological needs of mankind. Engineering disciplines are fundamentally mathematical and problem solving based. Traditional engineering disciplines, such as chemical, civil, electrical and mechanical, rely heavily upon 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. It relies primarily upon principles of discrete mathematics, especially logic.

What role does mathematics play in software engineering? Consider the following two statements. "Software practitioners do not need or use mathematics." "Software practitioners do need to think logically and precisely." An apparent contradiction since the reasoning underlying software engineering and mathematics are similar [4]. Perhaps when a software practitioner says, "I don't use mathematics" this really means "I don't explicitly use mathematics." Practicing engineers don't explicitly use calculus on a daily basis, but they do implicitly use mathematical reasoning all the time.

Ask traditional engineers if calculus should be eliminated from undergraduate engineering curricula, and the answer would be a resounding "no." There have been arguments by practicing software engineers that mathematics is not that important in software engineering education since "they" don't use it explicitly [5]. But which mathematics, continuous or discrete? The role of discrete mathematics and logic in software engineering is not well defined or understood by either academics or practitioners. This will change as the discipline matures, and academics and practitioners work together to develop its role - similar to the role continuous mathematics plays in traditional engineering disciplines.

## Differences and Similarities Between Traditional and Software Engineering

Articles have been written attempting to describe the similarities and differences between traditional engineering disciplines and software engineering [8]. One of the major differences is that traditional engineers construct real artifacts and software engineers construct non-real (i.e., abstract) artifacts. The foundations of traditional engineering disciplines are mature physical sciences and continuous mathematics, whereas those of software engineering are more immature abstract computer science and discrete mathematics. In physical engineering, two main concerns for a 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 the number of errors per thousand lines of source code. Both disciplines require maintenance, but in different ways.

What are the similarities? Excluding concerns about project cost, management, personnel, etc. perhaps the most relevant is modeling. All engineering disciplines require 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 is mathematical. It is very mature in traditional engineering and very slowly maturing in software engineering.

An example of using mathematical reasoning in both traditional and software engineering will be informative. In electrical engineering consider a resistor-capacitor (RC) circuit. The voltage decay as a function of time ( t ) 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. After taking foundational calculus and differential equations, electrical engineering students take their first electrical circuits course. Here, they develop a mathematical model of the behavior of a RC circuit using differential equations, and solve these to derive the function above. Mathematics based reasoning is used to derive and understand foundational concepts.

Iteration invariants are a foundational concept few computer science or software engineering graduates understand, appreciate or can use effectively. Every iteration has a predicate I(…), the iteration/loop invariant, that captures the underlying meaning of the iteration (e.g., sum values in a list, search a tree structure, compute the tax due by all taxpayers, etc.). Mathematical logic can be used to argue that this predicate satisfies logical constraints as illustrated below for the **while-do** iteration ({ …. } represents logical assertions) :

```
{ pre-condition }
Initialization code
{ I(…) is true }
while  C(…) is true  do
   | { I(…) and C(…) are true }
   |  < code for body of iteration >
   | { I(…) is true }
{ I(…) and not C(…) are true } ⇒  { post-condition }
```

Upon termination (another mathematical issue), { I(…) and not C(…) are true } must logically imply ( ⇒ ) the desired post-condition.

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

## Why is Mathematics Important for Practicing Software Engineers?

Below are some key reasons:

1. Software is abstract.
2. Notations, symbols, abstractions, precision are features common to both mathematics and software engineering.

3. Mathematics is requisite for modeling software systems.
4. Many application domains (engineering, science, economics, etc.) are mathematically based.
5. Mathematical reasoning about software systems is essential.

1. **Software is abstract:** Constructing non-real (abstract) artifacts requires abstract reasoning. What endeavor of mankind has been developed to deal with abstraction? Mathematics!! Indeed, a software system is simply a mathematical model of some process or desired computation. Mathematics provides one tool for reasoning about software systems, and is the only tool available to the practitioner for rigorous reasoning and analysis.

2. **Notations, symbols, abstractions, precision:** **y = ax + b** is familiar from algebra, **count == 0** from programming. Both use notations, symbols and are precise knowing the types of the data and semantics of the operations, both of which are specified mathematically. Learning a formal notation is no harder than learning a programming language. Indeed, it is often easier since the syntax and semantics are cleaner. Programming appeals to our process/imperative oriented minds and programming tools breath life into programs. Mathematics tends to be declarative and static, although tools such as Maple, Mathematica and Axiom help to mitigate this perception.

3. **Modeling software systems:** A model, even a mental one, must be created before beginning construction of any artifact. Much software development is like art - an initial vision slowly takes form. Planned evolution is not considered, and maintenance issues are ignored. This may be acceptable for some projects, but with software projects the more you know and understand up front, the better. Modeling is one vehicle for achieving this, and mathematics is an important tool for building, checking, analyzing and experimenting with models [2].

4. **Application Domains:** Software practitioners need mathematics to communicate effectively with clients and colleagues who are engineers, scientists, mathematicians, statisticians, actuaries, economists, etc. Mathematics is pervasive and a universal language for communication.

5. **Mathematical Reasoning:** One definition of mathematical reasoning 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 to solve problems in any domain [6]." In the most general interpretation, every problem solving activity is an application of mathematical reasoning. For example, consider the benefits of exercises that require students to translate English statements to propositional or predicate logic form. These "modeling" exercises help students to be more precise and inquisitive about the interpretation of English statements. When a client or colleague states "A or B" do they mean "inclusive or" or "exclusive or"? Or when they say "for all …" do they really mean universal quantification? What is the intended meaning when this statement is vacuous?

## What Mathematics is Important for Practicing Software Engineers?

As computer science and software engineering mature, educational foundations are more easily identified. Computing Curriculum 2001 requires discrete mathematics in its core and recommends that it be taken early in the undergraduate curriculum [3]. The current ACM/IEEE undergraduate software engineering curriculum effort has adopted and extended this foundational model [9]. The topics for a two course mathematical foundations core are listed below (here E = Essential, D = Desirable, and O = Optional):

Functions, Relations and Sets (E)
Basic Logic (propositional 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)
Algebraic Structures (O)

Other complementary foundational core material includes:

1. Abstraction
   a) Generalization
   b) Levels of Abstraction and Viewpoints
   c) Data types, class abstractions, generics/templates
   d) Composition

2. Modeling
   a) Principles of Modeling
   b) Pre & Post conditions, invariants
   c) Mathematical models and specification languages (Z, etc.)
   d) Tools for modeling and model validation
   e) Modeling/design languages (UML, OOD, functional, etc.)
   f) Syntax vs Semantics
   g) Explicitness (make no assumptions, or state all assumptions)

Although calculus is not explicitly in the core, it is highly recommended since it enhances mathematical maturity, provides a good contrast with discrete mathematics concepts, and is used by many client and application disciplines. Other recommended mathematical courses include probability and statistics, and linear algebra. Advanced mathematical courses such as graph theory, combinatorics, theory of computing, probability theory, operations research, algebra, etc. might be required or elective courses depending upon the goals of the program and the individual student.

There are currently a number of mathematically oriented software engineering courses including ones with titles like, Formal specifications, Formal methods, Mathematically rigorous software design, Software verification and validation, Software models and model checking, etc. In time, it is expected that the word “formal” will disappear as the discipline matures. The word is rarely used in traditional engineering where formal approaches are the norm.

Specific material will vary between programs. However, an important goal is to ensure that foundational mathematical concepts are used and reinforced in computer science and software engineering courses, in the same way that continuous mathematics is used and reinforced in traditional engineering courses. Achieving this will require time, dedication and rethinking the current software engineering curricula.

## How can Mathematics be Used?

The article “Why Math?” in this issue illustrates several example applications of mathematics in computer science. A more detailed example of mathematical reasoning, based upon iteration invariants is presented below.

Humans tend to apply familiar patterns of thought when solving problems. This works well for most “in-the-box” problems. Consider the following simple linear search problem: “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 – ones we hope our students would ask themselves. How many items are in the list? Can the list be empty? What happens if the specified item appears more than once? What is meant by “first instance?” By location? Knowing the answers to these questions and the given problem information one can formulate representative pre and post conditions - required to ensure the problem is well defined.

Most graduating students would love to see such a question on a comprehensive exit exam[1]. Applying a familiar pattern, or template, leads to a few lines of code – simple.
But do students really understand, or is this an example of rote learning?

Now consider an extension of this problem, binary search. Identify the pre and post conditions, apply a “somewhat fuzzier” familiar pattern, and compose the algorithm. Jon Bentley used this problem with professional programmers - approximately 90% got it wrong [1].

Returning to the linear search problem, let’s consider an alternative solution strategy based on the foundational concept of iteration invariants. First, what is the post condition? It can be formally specified using predicate logic, but for this discussion English will be used. “The desired item = list of items [ location ] **and** the desired item is not found in the list before ‘location’ (i.e., for all locations loc = 1, 2, 3, … , location-1, desired item ≠ list of items [ loc ] )[2].” The latter leads to the iteration invariant, “the desired item has not been

[1] Most colleges and universities don’t have such exams.
[2] Start counting a 1, not 0 as in C, C++, Java.

found yet (i.e., "for all locations loc = 1, 2, 3, … , current_location-1, desired item ≠ list of items [ loc ]" ) as illustrated pictorially in Figure 1. It can be established by iterative comparisons starting from location 1, then 2, etc.

```
 ________________________________________________
| ≠ | ≠ | ≠ | . . . | ≠ |  ?  |  ?  | . . . |  ? |
                            _
  1    2    3  . . .    current_location
```

Figure 1

Start the iteration to ensure the initial invariant is valid, keep the loop iterating to ensure the invariant remains valid, and ensure it is valid at the end since it is necessary to imply the desired post condition. See Figure 2 where the predicate invariant( current_location ) is "for all locations loc = 1, 2, 3, … , current_location-1, desired item ≠ list of items [ loc ]." Also, note that this invariant is vacuously true prior to entering the iteration, as is often the case - a reason why students should understand this fundamental mathematical logic concept. Feel free to complete the algorithm using the 'red stuff' to derive condition C(…) and <body of the iteration>.

```
{ there exists location 'loc' such that  desired item = list of items [ loc ] }
current_location ← 1
{ invariant( current_location ) }
while  C(…)  do
   | { invariant( current_location ) and C( current_location )  }
   |   <body of the iteration>
   | { invariant( current_location ) }

{ invariant( current_location )  and not C( current_location ) } □
{ invariant( current_location ) and desired item = list of items( current_location ) }
```

Figure 2

One may argue that all this 'red stuff' gets in the way. Perhaps, but the 'red stuff' mimics the underlying reasoning our mind uses. It has simply been made explicit here. This is the type of reasoning, from first principles, software engineering students should be able to do. However, this does not mean they must use it all the time. It is a tool available when needed. For example, when a derived software system must be correct, or when understanding or debugging a software system.

What is the iteration invariant for binary search? Intuitively, that the location of the desired item is constrained between two other locations 'low' and 'high' that the algorithm adjusts to converge on the potential location of the desired item (the desired item may not even be in the list). This "intuitive" invariant is only useful if it is used to derive a correct algorithm. What percentage of Bentley's programmers would have gotten it right if they reasoned more mathematically?

## Conclusions

Mathematical reasoning is intrinsic to both traditional engineering and software engineering; however, the foundational mathematics and its use differ. Traditional engineers primarily use continuous mathematics in a calculational mode for modeling, design and analysis (e.g., calculate load on a bridge component, compute wattage of a resistor, determine optimum weight of a car suspension system, etc.). Software engineers mainly use discrete mathematics and logic in a declarative mode for specifying and verifying system behaviors, and for analyzing system features.

A resistor-capacitor circuit and iteration invariants were used to illustrate basic mathematical reasoning. Engineering is much deeper and broader. Engineers are systems architects who understand and can apply the foundational principles of the discipline. Software engineers must learn to use mathematics to construct, analyze and check models of software systems, to compose systems from components, to develop correct, efficient system components, to precisely specify the behavior of systems and components, and to be able to analyze, test and evaluate systems and components. They must 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 tools for mathematical modeling, design, analysis and implementation. These include standard languages for communication (e.g, blue prints, schematic circuit diagrams, etc.) and computer-aided prototyping, design and analysis tools. Comparable tools at the software engineers disposal are slowly 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 or use mathematics [5]. Recall the adage "If {the only tool} you have {is} a hammer, everything looks like a nail." Surveys of current practices [7] simply reflect reality - many software engineers have not been educated to use discrete mathematics and logic as effective tools. Education is one key to ensuring future software engineers can use mathematics and logic as power{ful} tools for reasoning and thinking.

**Acknowledgements** I thank David Hemmendinger, Doug Baldwin and Kim Bruce for suggestions that significantly improved the quality of this paper.

## References

[1] Bentley, J., "Programming Pearls: Writing Correct Programs," Communications of the ACM, Dec. 1983 (26:12) pp. 1040-1045.

[2] Clark, E.M., Grumberg, O., and Peled, D., "Model Checking," MIT Press, 1999

[3] "Computing Curricula 2001 : Computer Science Volume", http://www.acm.org/sigcse/cc2001/, Dec. 15, 2001.

[4] Devlin, K., “The Real Reason Why Software Engineers Need Math,” Communications of the ACM, Oct. 2001 (44:10) pp. 21-22.

[5] Glass, R.L., "A New Answer to 'How Important is Mathematics to the Software Practitioner' ?", IEEE Software, November/December 2000, pp. 135-136.

[6] Henderson, P., et. al., “Striving for Mathematical Thinking,”
SIGCSE Bulletin - Inroads, Vol. 33, No. 4, Dec 2001, pp. 114-124.

[7] Lethbridge, T., "What Knowledge is Important to a Software Professional?," IEEE Computer, **(**33:5**),** May 2000, pp. 44–50.

[8] Parnas, D.L., "Software Engineering Programmes are not Computer Science Programmes", Annals of Software Engineering. Vol. 6, No. 1-4, 1998, pp. 19-37.

[9] Sobel, A. (Editor), “First Draft of the Software Engineering Education Knowledge (SEEK),” http://sites.computer.org/ccse/, Aug. 28, 2002.