# MATHEMATICS EDUCATION FOR SOFTWARE ENGINEERS: IT SHOULD BE RADICALLY DIFFERENT!

**Franz LICHTENBERGER**
Research Institute for Sybolic Computation (Risc-Linz)
Johannes Kepler University
and
Department of Software Engineering
Polytechnic University of Upper Austria
and
Software Competence Center Hagenberg (SCCH)
A-4232 Hagenberg, Austria
e-mail: Franz.Lichtenberger@scch.at

## ABSTRACT

Software engineering is a young engineering discipline that is different in many aspects from the classical engineering fields. For me the most distinguishing point is the kind of mathematics that serves the respective fields well. By giving examples I will try to show that classical, calculus based mathematics is of no help for defining central notions in software engineering, like "abstract data type". Thus, mathematics education for software engineering students should be radically different from the traditional curricula for science and engineering students. In particular, the changes to be made go far beyond putting more emphasis on discrete mathematics as done in many math curricula for computer science students.

I will report on our introductory mathematics course that that we have now taught at the Polytechnic University of Upper Austria for several years. The whole first year is dedicated to teach "The Language and Methods of Mathematics". I will also report on experiments with using the THEOREMA language and system in the lab exercises for this course, both about highlights and problems. THEOREMA is being developed by Bruno Buchberger and his team at Risc-Linz and aims at combining general predicate logic proof methods and special proof methods in one coherent system.

An important observation is that students are in no way prepared for this kind of mathematics after high school. Since computers and information technologies continue to gain more and more importance in our lives, the ability to developed software with mathematical rigour will be a crucial asset for the competitiveness of the software industry of any country in the future. This implies that changes in the high school mathematics curricula towards usability for software engineering should also be considered.

# 1 Introduction

In the early years of computer science and engineering there was a lot of discussion where the field should be positioned in the landscape of university education. It was also not clear how to name the various programs, several of them differing only very little: Computer science, computing science, computational science, information science, systems science, systems engineering, computer engineering, software engineering, information science, informatics, etc.

Since most of the early programs grew out of mathematics and electrical engineering curricula, one main point in the discussion was if computer science should be considered a science or an engineering discipline. The advent of software engineering as its own field has clarified things partially: it is generally agreed that software engineering is an engineering discipline, whereas computer science is (sic!) a science. This fact should be reflected in the respective educational programs. David L. Parnas ([9]) gives an excellent and exhaustive account on this theme in his paper titled "Software Engineering Programs are not Computer Science Programs".

It is generally agreed as well that mathematics plays an important role in the curriculum of the classical engineering disciplines like Civil, Mechanical or Electrical Engineering. Often up to 30% of an engineers education is devoted to mathematics.

If one looks at journals and proceedings in the field of software engineering education, the topic of teaching mathematics does not seem to have much importance. Several well-known software engineering educators such as, again, David L. Parnas ([8]), say that mathematics in a software engineering program should be essentially the same as for the classical engineering disciplines. I strongly oppose this opinion, and on the contrary will try to show the opposite by giving an example that classical engineering mathematics does not really help a software engineer in working in his or her profession. I agree that software engineering clearly is an engineering discipline. But it is different in various aspects, and the most distinguishing point for me is the kind of mathematics that well serves the respective fields well.

I have expressed some of these thoughts already at a conference on Engineering Education ([7]). Here I try to elaborate in more detail on the points that distinguish math for software engineers from math for classical engineers. Additionally, our experience with using the THEOREMA system in lab exercises is more recent.

# 2 Mathematics for Software Engineers

Civil, mechanical, or electrical engineers usually model aspects of our physical reality where space and time are continuous quantities. Thus it is understandable that classical engineering mathematics is primarily based on calculus. With the advent of the computer discrete mathematics gained more and more importance and thus in a typical computer science curriculum discrete mathematics plays an important role. Often discrete math is even taught before calculus - and then praised as a radical reform in teaching mathematics.

Classical engineering mathematics also differs strongly from pure mathematics. Mathematicians are primarily interested in deep theorems and general properties of classes of functions, expressions, algebras, etc., whereas engineers primarily use well-

known mathematical entities to model aspects of reality and to do calculations in these models. Nowadays engineers should know how to use modern tools like program libraries and computer algebra systems, but the mathematical topics they need are settled and hardly change over the years. I therefore believe that it is a mistake to treat math education for computer science and software engineering as basically the same, as done recently in a working group at an established conference on computer science education ([5]).

When we think about a software engineer designing and implementing a software system for controlling robots, he will need a lot of classical engineering mathematics like geometry and differential equations. The point is that he does not need this math because he is a software engineer but because he is working in the area of robotics. If he worked, for example, in the banking area on modelling workflow, he would need a rather different kind of math (if any). Since software engineers can work in any area of the economy it is virtually impossible to teach all the mathematics they possibly could need. We thus propose to place emphasis on teaching the methods of mathematics and thus enable students to learn arbitrary topics by themselves when needed. This approach is described below.

Nevertheless, one should also try to identify the mathematics that every software engineer should learn. Analysing this question, I came to the conclusion that this kind of mathematics is very different from classical engineering mathematics. Let me try to explain this with the following example.

# 3 An Example: Abstract Data Types

The concept of an abstract data type (ADT) is fundamental in programming. Students learn to base their programs on that concept in the first year. Every introductory textbook on computer science, programming, or algorithms and data structures gives a "definition" of that notion. In [1] it reads like this:

> "We can think of an abstract data type (ADT) as a mathematical model together with a collection of operations defined on that model."

This informal definition usually suffices for daily programming work, but if one wants to *prove* properties of an ADT or relations between ADTs (i.e. does a proposed ADT implement another ADT correctly?) one has to base such proofs on *mathematical* definitions. A math curriculum for software engineers clearly should provide such formal definitions and teach the students how to use them. The best definition of ADT I know is an algebraic one and taken from [3]. Since this book is written in German, we also cite [4] where the definition is similar.

> **Definition:** Let $\Sigma$ be a Signatur. Then $\Sigma$–$ALG$ denotes the category of all $\Sigma$–algebras with all $\Sigma$-algebra-morphisms between them. An *abstract data type* of $\Sigma$ is a full subcategory of $\Sigma$–$ALG$.

I would like to use that definition early in my math courses for software engineers, but this is virtually impossible. The students are in no way prepared for this kind of mathematics when coming from high school. Traditional, calculus based engineering mathematics does not help here at all.

This should make the dilemma clear: something that even mathematicians sometimes call "abstract nonsense" - which is taught in the late undergraduate or graduate curriculum only - i.e. some concepts of category theory, is necessary to define such basic notions as "Abstract Data Type" in a mathematically precise way. One should see that a mathematics curriculum for software engineers has to be very different from traditional engineering curricula.

# 4 Teaching "The Methods of Mathematics"

Already in the early eighties Bruno Buchberger (the founder of the RISC Institute) started to develop a new mathematics curriculum for computer science students at Johannes Kepler University in Linz (Austria). I had the privilege to contribute to this project from the beginning and to teach the course for almost two decades. The whole first semester is dedicated to teaching "The Language and Methods of Mathematics". By showing several case studies we try to analyse and teach all those aspects of mathematics that are necessary to treat the whole problem solving process, i.e. starting from the formal specification of a problem, then developing recursive and iterative algorithms for solving the given problem (which often means to conjecture and to prove some mathematical facts), prove the correctness and analyse the complexity of the algorithms, and finally give a structured documentation and presentation of the problem and its solution. The lecture notes of the first semester were published in [2], the whole curriculum is described in [6].

Since the Polytechnic University of Upper Austria at Hagenberg started a four-year software engineering program in 1993 this course is taught as the only mathematics course in the first year. At the beginning Mathematica was used to demonstrate the basic concepts, now we use THEOREMA in the lab exercises. Although we teach basically no mathematical contents that are new to the students, I am convinced that this course serves the students better in developing the skills needed for their professional career than a calculus or discrete math course would do. The more traditional math courses are taught in the second and third year.

One could argue that this kind of methodological training comes too early in the curriculum since there is nothing "to abstract from" at the beginning of a university program. We believe on the contrary that students have been exposed to enough mathematical topics in high school to be able to demonstrate and teach the technique of problem solving with mathematical methods explicitly. We also believe that this methodological training should make it easier for the students to learn and understand arbitrary mathematical contents in the following years.

# 5 Experiments with THEOREMA

THEOREMA is a language and system which aims at combining general predicate logic proof methods and special proof methods. It is written in Mathematica and thus the user interface and the computational power of Mathematica are available. THEOREMA is being developed by Bruno Buchberger and his team at Risc-Linz. With THEOREMA it is possible to define new notions in "natural" mathematical notation, do computations using them, and prove theorems about these notions in one single environment.

Unfortunately it is not possible to show the nice features of an interactive system like THEOREMA in a paper like this. This will be the main part of the oral presentation at the conference. I suggest to visit the homepage of the project ([10]).

A typical homework example which is given at about mid term of the first semester is the following:

> Define the notion of "a finite sequence is in descending order" according to the following three explanations. A sequence is in descending order if
>
> 1. the left of two neighbouring elements is at least as big as the right one,
> 2. all elements to the right of an arbitrary element are at most as big as this one, and all elements to its left are not smaller than it,
> 3. comparing two arbitrary elements, the left one is at least as big as the right one.

In the second semester, when proving is the main topic, they have to prove that the three definitions they gave are equivalent. THEOREMA can produce these proofs automatically. The presentation of the proofs is essentially the same as if done carefully by hand.

The results of our experiments are promising: students appreciated that they were "forced to rigor". Also, prove construction – a topic that is generally regarded as very difficult – could be "demystified" by showing the computer made but human readable proofs. One drawback still is that the start-up effort is very high. We hope that this problem will vanish with future versions of THEOREMA.

# 6 Implications for the High School Curriculum

High-school math is based on calculus and thus serves well as a preparation for classical engineering mathematics. Clearly much energy is put into defining the basic notions, like *continuous function* precisely. Comparing this with the ADT example, it is definitely not sufficient to give an informal definition like

> A real function is called *continuous* if it can be drawn with a pencil in one stroke (without gaps etc.)

The formal definition is much more complicated:

$$f \text{ is } \textit{continuous} \text{ at } x :\Longleftrightarrow \bigwedge_{\varepsilon>0} \bigvee_{\delta>0} \bigwedge_{|y-x|<\delta} |f(y) - f(x)| < \varepsilon$$

Nevertheless, every engineer has to learn – and should understand – this (or an equivalent) definition of this important notion, even if he will never in his professional life use it in this way. It is important for him to know that the mathematics he uses is based on such solid grounds.

When heading towards a formal definition of *abstract data type* one first has to introduce the notion of *signature*:

A *signature* $< S, \Sigma >$ consists of a set $S$, whose elements are called *sorts*, and a family of sets $\Sigma = (\Sigma_{w,s})_{w \in S^*, s \in S}$.

For each $w \in S^*$, $s \in S$, $\Sigma_{w,s}$ is a set whose elements are called operation symbols.

For each $\sigma \in \Sigma_{w,s}$, $w$ gives the *domain* and $s$ the *codomain* of $\sigma$. $|w|$ is the *arity* of $\sigma$; $\sigma$ is called *nullary* or a *constant*, if $w = \lambda$.

In this definition $S^*$ denotes the set of finite words over (the alphabet) $S$ and $\lambda$ the empty word. Instead of $< S, \Sigma >$ we often denote the signature just $\Sigma$.

I believe that this definition is easier to understand with the right preparation than the definition of *continuous function* given above. Nevertheless, students at university have big difficulties understanding it since they are not prepared for this different kind of basic mathematics. Today high school math usually culminates in teaching differential and integral calculus and it's Fundamental Theorem, sloppily written:

$$\int_a^b f'(x)dx = f(b) - f(a)$$

In order to prepare students for the mathematics I would like to start with at university I would recommend that, for example, Birkhoff's theorem of 1935 stating the completeness and consistency of the equational calculus be taught already in high school:

$$E \models e \iff E \vdash e$$

(Here $e$ is an equation over a signature $\Sigma$ and $E$ a set of equations over $\Sigma$)

I know that this is just wishful thinking and far from having a chance to become reality. It's even worse: all the nice new tools like computer algebra systems, TI 92, etc. make teaching classical, calculus oriented math easier and sometimes even fun. The more formal kind of mathematics I need plays almost no role any more.

# 7 Conclusions

I have argued that although Software Engineering clearly is an engineering discipline the mathematics that could serve the field well is quite different from classical engineering mathematics. I would like to explicitly state the following three points to characterize this type of mathematics:

- The main emphasis is on **modelling of non-physical realities**. The mathematical tools needed go beyond traditional discrete mathematics and come primarily from (universal) algebra and logics. Even classical propositional and predicate logics often are not sufficient, so that one has to use others like modal, temporal, or non-monotonous logics.

- **Define the new instead of use the known**: Research on new types of, say, differential equations is not a task of a classical engineer; mathematicians do that job. But if one considers abstract data types to be essentially classes of algebras, as proposed above, defining or specifying new classes of algebras is everyday work for a software engineer. He will not try to detect deep mathematical theorems valid in these algebras but definitely will have to prove some basic properties of the objects and operations he modelled.

- **Prove instead of compute**: In the classical engineering disciplines mathematics is primarily used to compute numerical values. Symbolic computations are almost exclusively done as a preparation for numerical computations, often the results are visualized graphically. A software engineer should use mathematical reasoning to prove properties of the programs or software systems he designed. Training in doing formal proofs should thus be mandatory.

Since high school math in no way prepares for that kind of mathematics, it is very difficult to introduce these topics at the university level. Nevertheless, I believe that it would pay off to take this challenge since it could improve the quality of software in general. Since computers and information technologies continue to gain more and more importance in our lives, the ability to developed software with mathematical rigour will be a crucial asset for the competitiveness of the software industry of any country in the future.

# References


[1] Aho A.V., Hopcroft J.E., Ullman J.D.: *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass., 1983.

[2] Buchberger B., Lichtenberger F.: *Mathematik für Informatiker I: Die Methode der Mathematik*, Springer-Verlag, Heidelberg, 2. Auflage, 1981.

[3] Ehrich H.-D., Gogolla M., Lipeck U.W.: *Algebraische Spezifikation abstrakter Datentypen*, Teubner, Stuttgart, 1989.

[4] Ehrig H., Mahr B.: *Fundamentals of Algebraic Specification*, Springer, Berlin, Vol. I, 1985.

[5] Henderson P.B. (Chair): ITICSE-2001 Working Group Report: *Striving for Mathematical Thinking*, ACM-SIGCSE Bulletin, Vol. 33. Nr.4., Dec. 2001, pp 114-124.

[6] Lichtenberger F., Buchberger B.: *Mathematik für Informatiker: Ein algorithmenorientierter Ansatz an der Universität Linz*, Zeitschrift fr Hochschuldidaktik, Jahrgang 9, 1985, Sonderheft 10, pp 103-110.

[7] Lichtenberger F.: *Mathematics Education for Software Engineers: An Underestimated Challenge*, Proc. 27th International Symp. on Engineering Paedagogics, Moscow, Sept. 1998.

[8] Parnas D.L.: *Teaching for Change*, Proc. 10th IEEE Confonference on Software Engineering Education and Training, Virginia Beach, April 13-16, 1997, p. 174.

[9] Parnas D.L.: *Software Engineering Programs Are Not Computer Science Programs*, IEEE Software, Nov./Dec. 1999.

[10] THEOREMA Homepage: http://www.theorema.org