Making Formal Methods Popular: The Crux is Math Education!

Franz Lichtenberger

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University
Linz, Austria
franz.lichtenberger@risc.jku.at

Abstract. Although on many occasions, especially at FM conferences, highlights of the use of Formal Methods in software development are presented, FM plays just a minor role in both the everyday work of software engineers as well as Computer Science and Software Engineering curricula. To me, one of the main reasons for the status quo is that mathematics education, as it is usually done today, does not enable students to understand and to use Formal Methods.
Software engineering is a young engineering discipline that is different in many respects from the classical engineering fields. To me the most distinguishing point is the kind of mathematics that serves the respective fields well. Classical, calculus-based engineering mathematics is of no use. Just putting more emphasis on Discrete Mathematics, as recommended and usually done, is – by far – not enough.
I will report on an alternative approach to teaching introductory mathematics to students of CS and SE which started already in the 1980s. The whole first year is dedicated to teach "The Language and Methods of Mathematics". It is also an introduction into FM, i.e. in the second semester students learn to do Hoare-style correctness proofs, for example. Implementing such radical changes in the curriculum is, of course, also a "political" problem. Some of these aspects will be discussed, for example that this will be possible only if SE is regarded a discipline of its own, not just a part of CS. In many cases it will also be necessary to take math education away from math departments. Real ($\mathbb{R}$-) mathematicians have a hard time understanding the mathematical needs of software engineers. In the last section I will present some personal thoughts about which advanced mathematical topics could play a major role in future SE education, as for example (partial) differential equations do in classical engineering mathematics. For me, these topics come from algebra, aiming at "conceptual mathematics", which essentially is category theory.

1 Introduction

Although on many occasions, especially at FM conferences, highlights of the use of Formal Methods in software development are presented, FM plays just a minor role in both the everyday work of software engineers as well as Computer Science and Software Engineering curricula.

There are reasons to believe that this cannot be changed dramatically, or even that it should not be changed. Many practitioners regard FM as not applicable and thus unimportant for the main stream, relevant just in some niches. I assume that most participants of a workshop on FM in SE education will disagree. So, what can we do to bring FM closer to the main stream in software development?

To me, one of the main reasons for the status quo is that mathematics education, as it is usually done today, does not enable students to understand and to use Formal Methods. This seems a bit remote, at first glance. In this paper, which is no research paper proper, but more an extended position statement, I will try to explain my point of view and give recommendations for how to make FM more popular.

2 Mathematics in CS and Engineering

It is generally agreed 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. The mathematics taught there is well-established, almost exclusively calculus-based, since the advent of computers numerical mathematics plays an important role as well.

Software Engineering is a young engineering discipline that is different in many respects from the classical engineering fields. To me the most distinguishing point is the kind of mathematics that serves the respective fields well. Classical, calculus-based engineering mathematics is of no use. Just putting more emphasis on Discrete Mathematics, as recommended and usually done, is – by far – not enough.

There is a lot of material, mostly developed by high-ranking committees of IEEE Computer Society and ACM, giving guidelines for CS and SE curricula both at the undergraduate and the graduate level or describing the "body of knowledge" in the respective fields. All of that is valuable and great, I just criticize one point: the attitude towards mathematics. Let's start with SE2004 ([1]). Just saying that "Discrete mathematics is the mathematics underlying all computing, including software engineering [and so on]" and choosing two introductory courses on discrete structures from the CS volume, shows me that SE is not regarded as a discipline of its own. The third course, called Statistics and Empirical Methods, is relevant to SE, as it is for any engineering discipline. By the way, in GSwE2009 ([2]) software engineering is treated as a subfield of systems engineering (be careful: in this document SE stands for systems engineering, whereas SwE is software engineering). In SWEBOK ([3]) it is basically the same: mathematics and FM play just a minor role.

It seems that everybody is – more or less – satisfied with this kind of math teaching. In our community as well I rarely hear complaints that students are not prepared for Formals Methods in introductory math courses although FM is nothing else than software development strictly based on mathematics. Many practitioners (and academics) regard FM, as already said in the introduction, as not applicable and thus unimportant. This is different from the classical engineering disciplines. As Peter Henderson put it in [4]: "... there mathematics is intrinsically used for "everything" - modeling, analysis, reasoning, design, verification, validation, statistical analysis, etc. In SE it is more like an afterthought called Formal Methods." I am convinced that bringing FM to the mainstream of software development needs a different kind of math teaching.

3 Introductory Math for Software Engineering

It is pretty clear which basic mathematical topics are important for Formal Methods, an early book covering this is [5]. Many FM courses start with introducing these topics, some curricula go far beyond. I personally like [6] very much which was developed at TU Berlin. To my knowledge all these courses come *in addition* to a traditional math education, and mostly late in the curriculum, often as graduate courses only. My plea is to teach these topics right away from the beginning, *instead of* the usual things like calculus, linear algebra, or discrete math.

An Alternative Approach

My view on a proper, realistic mathematics education for software engineers is perhaps characterized best by part of the title of a talk I gave in 2002 at a conference on teaching mathematics: "It Should Be Radically Different!" That is different from math for the classical engineering disciplines, and thus different from how it is taught at most places today. In [7] I give the reference with a link to the paper.

The paper is more than ten years old and I probably would use different examples and other wording today. There I use the example of defining the notion of "abstract data type" mathematically, i.e. via algebraic specification, to show that a different type of math is needed than classical, calculus-based engineering math.

For more than 30 years we have been teaching introductory mathematics to students of Computer Science (called "Informatik", at Johannes Kepler University, Linz, Austria) and of Software Engineering (at the University of Applied Sciences Upper Austria) in a different way. Unfortunately the textbook we used ([8]) is in German and out of print (see link to pdf copy). The concept was developed by Bruno Buchberger already in the late 1970s and I had the privilege to contribute to the project from the beginning. The whole first semester is dedicated to teaching "the language and methods of mathematics". By showing several case studies we try to analyze 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), proving the correctness and analyzing the complexity of the algorithms, and finally giving a structured documentation and presentation of the problem and its solution.

Teach specifying and proving!

Although there is a lot more to be said about our approach, in this writing I simply want to underscore two topics that are dear to me: specifying and proving.

What is the most important thing that one has to teach to SE students from the beginning? Teach them to specify, then to specify, and once again – to specify! In our introductory math course students have to write their first formal problem specification after four weeks. You just need the basic language of mathematics for that, i.e., propositional and predicate logic. But you need much more time to explain the details than is usually allocated for these topics in introductory math courses for CS. Mathematicians regard this use of logic as trivial, and in textbooks on discrete math it's usually worth just a few pages. Specification means translating natural language into the (formal) language of mathematics, but also the other direction is important: learning to read mathematics, for example formal definitions, and to explain it in one's own words. Our observation is that students just out of high school are totally unprepared for this kind of mathematics and have a hard time learning it.

Based on the aforementioned, one could think that proofs are of no importance to software engineers. This might be true for "deep" mathematical proofs, but they definitely should learn how to prove properties of or relations between the artifacts they produce or work with, be it specifications, designs, implementations, protocols, or others. Such proofs are "shallow" and they have to be formal, not just rigorous, because they are mainly done by computers. No human can check millions or billions of states of a system, nor can she or he have the patience (and time!) to deductively prove some intermittent assertion produced by a proof assistant that fills several pages.

In our approach students have to do (Hoare style) correctness proofs of programs at the end of their freshmen year. Before that we teach how to do proofs in general, i.e. natural deduction style proofs in predicate logic. I personally think that one has to start that early with these topics, otherwise it is too late. I mean too late to fulfill the main purpose of teaching mathematics to software engineers: to enable them to use formal methods, i.e., methods strictly based on mathematics.

Most important are the basics: logic and sets

To repeat it: I think that it is absolutely necessary to teach these basic skills to enable students to use Formal Methods. But, is this also sufficient? At FM conferences, especially in the industrial track, highlights of FM applications are regularly presented. Often one has the feeling that (at least) a PhD in math or CS is needed to do that work.

On the contrary, Leslie Lamport once gave a talk at a Formal Methods Education conference with the title "All I Really Need to Know I Learned in High School" ([9]). I agree with Lamport. All you need is the very basics, logic and sets - of course to an extent that this becomes your working language. Without doubt, there are also topics from advanced mathematics that are useful and probably even necessary in software engineering. For me, these topics come from algebra, as described later.

My experience shows that it takes (at least) a whole semester to teach just logic and sets, i.e., to bring students to the point where they, for example, really understand the meaning of implication, that they know that disjunction usually is used differently in math and everyday language, that they not only are able to check the validity of De Morgan's laws by truth tables but also to use them – especially in the version with quantifiers – to express the meaning of a natural language statement in different but equivalent ways, and that they can work with the basic set operations (plus functions and relations, especially equivalences and orderings as required in all curriculum proposals).

Where to put Discrete Math and Calculus?

Some words about where to put discrete math and calculus in a SE curriculum: I personally prefer not to have a DM course but to put most of the topics into other courses like Algorithms and Data Structures. This way you get the topics when and where you need them. If a topic is not covered there, for example modular arithmetic, then it should be taught when it is needed, say in a course on cryptography. So one can also better scale the number of proofs one does in detail. For SE students I think it is, for example, not necessary to give all the proofs concerning why RSA encryption works, computational scientists should know these proofs in detail. And if a topic does not show up in any course then it is not needed.

This does not apply to the very first topics of DM, i.e. sets and logic. (I personally prefer to teach logic before sets, I could explain in length why.) As argued in the previous section, these basics should be put into an introductory math course for SE, and treated there extensively.

It should be clear that *every* engineer should have courses towards quality assurance and control that are based on probability and statistics. Here, of course, calculus comes into play. Our SE students do not have a calculus course. Calculus is reviewed during the probability and statistics course. I think this is a very good place because the relationship between summation in the discrete case and integration in the continuous case, as well as differentiation, can be demonstrated well. Here I have to remark that this is possible in an educational system (like in my home country, Austria) where math in high school (”Gymnasium”) focuses very much on calculus. If freshmen can remember anything from high school math, it is how to differentiate and how to integrate. This may be different in other school systems.

4 Changing the Curriculum: Political Considerations

Before I come to some ”political” statements on how to enable such radical changes in the curriculum, especially towards math teaching for software engineering, I want to state that I am an educator and I belong to the people who have problems with the term ”computer science” per se, and I think that the fuzziness of that term is the main reason that it is difficult to impossible to find – and teach – the ”right” mathematics that could serve the whole field in a way that satisfies everyone.

To me, the big question is the one that has been discussed for decades now, namely, the ”science vs. engineering” problem. It is important to know whom we are educating: scientists or engineers? In my opinion we should not try to do both at the same time.

There is a wide spectrum of professional activities in our field: at the one end we have computing (computational) scientists and at the other software engineers. It is very clear to me that the former should become full-fledged mathematicians, with an emphasis on all aspects of computing, whereby ”Concrete Mathematics” a la Knuth ([10]) has to play an important role. The mathematical education of the latter group, however, must be different.

For all that is in between – call it CS, IT, ICT, IS, IM, System Science or Engineering, Data Science or Engineering, Web Science or Engineering, Big Data, Pervasive Computing, or any other name for a (sub) field – it is not clear to me what the right math is, and therefore I refrain from comment.

I suggest developing a software engineering curriculum incorporating mathematics as sketched above, and implementing it – more or less – independently of CS departments. Yes, and it should really be an engineering curriculum. I think that it would be good to have a ”Department for Software Science and Engineering” at every institution that grants a degree in SE. This collaboration of people who are working scientifically on foundational issues, together with others transferring their insights to engineering practices is common in classical engineering disciplines. It could help SE to become a mature engineering discipline.

Three Recommendations

My first advice for Software Engineering as a discipline is to divorce from Computer Science. You will be the underdog in this relationship forever. Don't go in anger. Help each other, there are a lot of problems you could and should solve in common.

This is important especially for curriculum development. I see no chance that the radical changes in introductory math education I propose can be done in CS curricula in general. For new SE curricula I definitely see the chance.

My second recommendation is to take math education for software engineers away from mathematics departments. Real ($\mathbb{R}$ -) mathematicians have a hard time understanding the mathematical needs of software engineers. Many mathematicians who have an affinity towards SE and FM work in CS or SE departments anyway. Those software scientists can do the job excellently. To make it clear: math should be taught by mathematicians, but there is no need that they come from math departments.

My third suggestion is that SE should try to get integrated into the community of the old engineering disciplines as a new member. This could help SE in many, more organizational matters like accreditation, for example, to become a mature and well recognized engineering discipline.

5 Advanced Math for SE: Algebra and Category Theory

In this last section I want present some personal thoughts about which advanced mathematical topics could play a major role in future SE education, as for example (partial) differential equations do in classical engineering mathematics. The first year of classical engineering math is usually devoted to calculus and linear algebra. Based on what I said about math education for SE, my favorite name for the course would be "Language and Methods of Mathematics".

The second year I would just call "Algebra", starting with classical topics (groups, rings, fields ...), enhanced by topics like, for example, process algebra, relational algebra, computational algebra. Soon I would come to a topic that I find important for SE: algebraic specification of Abstract Data Types (ADT), which means students learn about heterogeneous algebras, basic notions like signatures, and also some category theory for dealing with semantics of ADT specifications.

As computing scientists need a lot of "concrete math" a la Knuth (see [10]), I think software engineers should learn "conceptual mathematics" (see [11]), which essentially is category theory (yes, this abstract nonsense, you know). In many situations to be modelled one has a class of objects (like specifications, designs, implementations, programs, languages, queries, XML-documents, ...), and some transformation between such objects which often is associative. In most cases one also has some identity transform (often just "do nothing") which already means that the objects together with the transformation form a category.

Algebraic specification became popular in the early 1980s. In recent years also coalgebraic modelling gained momentum. Finite and, more important, infinite behavior of systems of all kinds can be modelled conveniently using a coalgebraic approach. For a nice introduction see [12]. I am convinced that one could rewrite all textbooks on automata, transition systems, reactive systems, parallel and distributed processes, and the like, in coalgebraic language – it just does not make sense, i.e. it does not pay off today. Perhaps this could change if students would learn about coalgebras in their algebra course.

Students should learn about duality, which is one of the most important, yet basic, notions in category theory. Just "reversing the arrows" in a categorical definition often defines another notion that makes sense. Examples are monomorphism and epimorphism, i.e. injective and surjective functions in the category SET, product and coproduct (direct sum), initial and final objects, pullback and pushout, and – very important – induction and coinduction for doing definitions and proofs.

"Data processing" was an ancient name for our field. To see that data and processes are just two sides of the same coin, i.e. dual in categorical language, the first modelled algebraically, the second co-algebraically, is a beautiful insight that all students of SE should be able to enjoy.

6 Conclusions

In this personal statement on how to make Formal Methods more popular, i.e. bring them closer to the main stream of SE education and practice, I come to the conclusion that mathematics education has to be changed in a way that students early in the curriculum learn to understand and to use Formal Methods.

In accordance with the overall objectives of the FMSEET'15 workshop, as stated in the Call for Papers,

– I find that FM education in a typical undergraduate curricula is in a sad state, almost irrelevant,
– I give some recommendations for improving the situation which are mainly political and far reaching, and
– I am, of course, willing to exchange knowledge and experience in any forum aiming to support FM education.

References

[1] IEEE Computer Society and ACM Joint Task Force on Computing Curricula, *Software Engineerig 2004 - Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*, `http://sites.computer.org/ccse/`
[2] Integrated Software and Systems Engineering Curriculum (iSSEc) Project, *Graduate Software Engineering 2009 (GSWe2009)*, `http://www.gswe2009.org/`
[3] P. Bourke, R.E. Fairley (eds.), *SWEBOK V3.0 - Guide to the Software Engineering Body of Knowledge*, `http://www.computer.org/portal/web/swebok`
[4] P. Henderson, personal e-mail to F. Lichtenberger, May 4, 2012.
[5] J. Woodcock, M. Loomes, *Software Engineering Mathematics: Formal Methods Demystified*, Pitman Publ., London, 1988.
[6] H. Ehrig, B. Mahr, F. Cornelius, M. Große-Rhode, P. Zeitz, *Mathematisch-strukturelle Grundlagen der Informatk*, 2. Aufl., Springer, Berlin, 2001.
[7] F. Lichtenberger, *Mathematics Education for Software Engineers: It Should be Radically Different!*, Proc. ICTM2, 1-6 July 2002, University of Crete, Greece. (`http://www.math.uoc.gr/~ictm2/Proceedings/pap305.pdf`)
[8] Buchberger B., Lichtenberger F.: *Mathematik für Informatiker I: Die Methode der Mathematik*, Springer-Verlag, Heidelberg, 2. Auflage, 1981. `http://www.risc.jku.at/publications/download/risc_2230/mathematik_informatiker_bookmarks.pdf`
[9] L. Lamport, *All I Really Need to Know I Learned in High School*, invited talk at CoLogNET/ Formal Methods Europe Symposium on Teaching Formal Methods, Gent, Belgium, Nov. 18-19, 2004.
[10] R. L. Graham, D. E. Knuth, O. Patashnik, *Concrete Mathematics - A Foundation for Computer Science*, Addison-Wesley, Reading, Mass., 2nd ed., 1994
[11] Lawvere F.W., Schanuel St.H.: *Conceptual Mathematics: A first introduction to categories*, Cambridge University Press,2nd Ed., 2009
[12] Jacobs B., Rutten J.: *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1977), pp. 222-259.