Teaching mathematically demanding computer science topics to software engineering students: Is there any reason? Is there any hope?

Antti Valmari

Tampere University of Technology, Institute of Software Systems
PO Box 553, FI-33101 Tampere, FINLAND
Antti.Valmari@tut.fi

Background

Many computer science educators, especially theory-oriented, tend to complain about the lack of mathematical skills of new students. For instance, the call for papers of the “Formal Methods in the Teaching Lab” workshop at the Formal Methods 2006 Symposium contains the following words of Dines Björner:

On the ‘third hand’, an increasing number of students de-select formal methods in the curriculum, due to various causes and trends. One cause of the problem is a general mathphobic trend in society and education.

On the other hand, many others think that computer science has been too theory-oriented in the past, and the current situation with mathematics is not really a problem.

I have had to face a perhaps exceptionally radical decrease of students’ mathematical skills in a rather short period of time. In 1995, to be admitted to the curriculum where I teach, an applicant had to get either 66% of the entrance examination points, or 67% of total points obtained from the entrance examination and previous school performance. The curriculum was the third most difficult to get in among the ten curricula of the university. Ten years later the thresholds were 17% and 30%, and the curriculum was the second last of 14. Some positions were left unfilled because of lack of applicants meeting the absolute minimal standards.

This trend has forced me to think a lot about what mathematical topics it is necessary to teach to prospective software engineers, how to teach them, or would it be best to remove mathematical topics altogether from software engineering education. In 2002 a new second-year course on mathematical methods in software engineering was introduced, and I was made responsible of designing and teaching it. I have made some experiments in the context of the course.

Against

Indeed, there are strong arguments in favour of dropping mathematical topics from software courses. One may well claim that new programming and design
techniques, the advance of various libraries, and the increase of performance of hardware have made programming different from what it was ten years ago. For instance, the need to write complicated algorithms has greatly reduced, because good algorithms can nowadays be picked from libraries, and mediocre algorithms give small enough response times with current computers in most cases. A couple of years ago we had to change the way in which we test students’ C++ algorithm project results, because it turned out that reading the input from and writing the output to a file took so much time that it dominated the time consumption of all but the worst solutions. The C++ input/output library is relatively slow because it is prepared for various character encodings, national conventions, etc.

There is, of course, also the pragmatic — or over-pragmatic? — view that what the students do not learn will not benefit them nor their future employers, and what is of no benefit could as well be dropped. More seriously, according to the results published prominently in [6] and available on the net [5], practicing software engineers do consider mathematics and theoretical computer science mostly unimportant for their work.

In the survey, the importance and other aspects of 75 topics were queried. Differential and integral calculus, combinatorics, Laplace and Fourier transforms, and differential equations were among the ten least important, together with such topics as chemistry and robotics. Their marks were at or below 1.6, in the scale from 0 to 5. The most highly appreciated theory topic was programming language theory, with rank 28 and mark 2.7. Computational complexity & algorithm analysis was number 30, and predicate logic number 39 (mark 2.4). The top five topics got marks 3.4 or better, and they were specific programming languages, data structures, software design and patterns, software architecture, and requirements gathering & analysis.

One possible objection to the bad performance of mathematics in the survey is that wrong topics have been taught, and perhaps also in the wrong way. One would expect that, say, predicate logic, set theory and automata theory are much more important in the software profession than calculus. Indeed, their ranks were much higher, namely 39, 48 and 49 (mark 2.1). However, these ranks are still rather low.

In favour

Should we conclude that mathematics is unnecessary, or is it just that software engineers are unaware of the benefit they have got? The general opinion in the public is clearly that programming requires mathematically oriented minds. Also, it is the task of universities to teach fundamental principles, which do not necessarily have immediate applications but affect the way graduates approach problems. Influence on thinking was actually one factor in the above-discussed importance ranking. The other, equally important factor was the usefulness of the details of the specific material to the career. Of the 28 topics that got higher mark from influence than from details, 26 were in the latter half of the total importance ranking. This suggests that a topic cannot have strong influence on
thinking unless its details are useful, or that software engineers are unaware of strong influence when the details are not useful.

My teaching experiences suggest strongly that along with diminishing mathematical skills, students have lost skills whose importance for software engineering is beyond dispute. At some point I started to doubt whether the students taking my software mathematics course really understand program code, and really master the programming language they use. I started to add questions to examinations that test this issue. For instance, in one examination participated mostly by students who had already tried to pass the course without success, I asked what values can \(j\) have at the end of the following program:

\[
j := 1; \text{limit} := A[n] \\
\text{for } k := 1 \text{ to } n \text{ do} \\
\quad \text{if } A[k] \leq \text{limit} \text{ then } tmp := A[j]; A[j] := A[k]; A[k] := tmp; j := j+1 \text{ endif} \\
\text{endfor}
\]

Of 21 students, 6 got both the lower and upper bound right, 4 got only the upper bound right, 1 got only the lower bound right, 8 got both bounds wrong, and 2 failed to realise that the correct answer consists of more than one value. In another examination, participated by the majority of the students who took the course that year, 42 out of 115 students could not say what the following prints:

\[
x = 8; y = 3; \text{cout} << x - y; \text{cout} << x << y;
\]

Further data collected from the examinations and weekly exercises together with discussions with students have made me believe that many of the current students do not try to understand things such as the meaning of a piece of program code as structured entities. Instead, it seems that they try to memorize a collection of patterns, according to which they answer examination questions and write program code. A similar observation was made in a survey conducted by the Institute of Mathematics of Tampere University of Technology [7].

For instance, the use of two successive minus signs in the previous program example does not match any pattern, because it is never needed when writing programs, as the same effect can be obtained by having no minus signs. Furthermore, the lack of space before the binary subtraction operator apparently made many fail to realise that it is the subtraction operator. Thus, rather small things suffice to break the pattern and utterly confuse the student.

So, the lack of mathematical skills seems to lead to a pattern-based programming style, where the person can solve programming problems only as far as the memorized collection of patterns suffices.

Understanding things as structured entities instead of collections of patterns is even more important in requirements analysis and specification. Software engineers spend today much more time with specifications and much less time with concrete program code than they did twenty years ago. Errors, omissions and wrong insights in specifications may become really expensive. On the other hand, correctness of specifications is more dependent on the person’s thinking
skills, because it is not possible to get as concrete and merciless feedback on specifications as are the “syntax error” from the compiler and “core dumped” from the run-time environment.

Therefore, it is essential that software engineers can predict the consequences of their specifications as reliably as possible. I argued in [8] that “the most valuable service that mathematical education can do to the software profession is to teach the students to recognise, formulate and reason with abstractions. The challenge is to get important details correct, even when they are subtle and the end users or customers do not see any problem with them.”

Hope?

So, there is any reason. Or at least I believe so. Is there any hope?

What I can say about this issue is rather weak. Based on the results in [5, 6] and other information, I have become convinced that one major reason for the failure of mathematics in the education of software engineers is that mostly wrong topics have been taught, and the right topics have been taught in the wrong way.

I share the view that calculus, combinatorics and similar topics are not important in software engineering. The problem with logic, set theory and automata theory is that too often they have been taught as fields of mathematics, not as tools for designing software systems. Engineering students do not study calculus from the same books as pure mathematics students. In a similar fashion, we need books in the style of “Set Theory for Software Engineers” and “Advanced Logic for Practical Computer Scientist”. As a matter of fact, there are some such books. The challenge of adapting the material to software engineering students has perhaps been taken most seriously in [3]. The reviews on this book at Amazon.com reveal that while some like it a lot, others find it next to impossible. It seems that the right teaching approach has yet to be found.

As I have already mentioned, a new second-year course “Mathematical Methods in Software Engineering” was introduced in 2002 and given to me to teach. This gave me the chance to implement my ideas about software mathematics education. The design of the course and the reasoning behind the design have been presented in [8].

The clearest observation I have made so far is that it takes a lot to get the average student of the course to do any decent amount of homework. One problem with most curricula in Finland is that although there is a recommended ordering for completing the courses, in practice the students may take them in just any ordering. A student that has failed a course can postpone it until it is the last course that is missing from his/her degree. Therefore, students do not have sufficient incentive to work hard when a course becomes difficult. Furthermore, they do not have time to work hard. Another special problem is that according to [4] and other surveys, Finnish students spend about 20 hours per week for their studies although they are expected to spend 40 hours. (Finnish students do a lot of part-time jobs.)
I have always disliked the idea of making weekly exercises fully obligatory. Unfortunately, I do not any more believe in anything else. I have tried several systems consisting of a combination of obligatory work and the possibility of earning additional points with voluntary work. A well-designed voluntary component attracts the best quarter of the students and introduces a secondary small peak at the high end of the distribution of marks from the examination, while a badly designed component lacks this effect. Neither succeeds in making the worst third or half of students try voluntary exercises, although they are those who need it most.

Another observation is that quite a few students cannot write a readable logical or set-theoretical formula even at late stages of the course. (In theory, they should master this already when entering my course, because it has a course on logic and set theory as a compulsory pre-requisite.) It seems as if they have a very vague idea of the syntax of the formulae, let alone the meaning. It is difficult to see how these students could understand most of the course material. They get occasional homework points from verbal problems and when the teaching assistant accepts their computations as a good try — and, it seems, by plagiarizing solutions.

It is unbelievable that the students cannot learn the syntax of logic and set theory, after having learnt the syntax of C++. In most cases the explanation must be that they just do not bother to study it. During the last two years we have enforced the rule that a solution that is not syntactically almost correct does not yield a point, even if it is well readable. This has helped the situation clearly but not tremendously.

Again, I and my teaching assistants originally felt that syntax is a secondary issue and being harsh with it is not education but teasing. Now we believe that the dull work of learning the syntactical structure well helps a lot in understanding the corresponding semantical structures.

Checking the syntax and simple static semantics of mathematical formulae is nontrivial but clearly within the capabilities of computers, at least if slight deviations from the standard mathematical practice are allowed. My experience suggests that such a syntax-checking program could be of significant help to students at the low end, if its error messages are good enough. Students could be motivated to use the program by sticking to the "syntax must be correct" rule. They could deliver the printout of the tool as a part of their solution, or the tool may send it directly to the teaching assistant.

Such a program would have many advantages. Unlike with a teaching assistant, the student could improve her/his formula and try again an unlimited number of times. Students tend to cause pressure to slowly degrade the standards by arguing with teaching assistants about the importance of such details as misplaced commas, but they cannot argue with a program. Today’s students

---

1 Exercise: compare the precedence and associativity of multiplication and the sin function (interpreted as a prefix operator) to each other, in the light of the well-known formula \( \sin 2x = 2\sin x \cos x \).
also like playing with computers. The usefulness of such a tool is yet another issue where my opinion has changed radically over the last ten years.

These observations are a light-year behind a solution to the problem of students' decreased mathematics skills, but at least they give me a direction to go in. To answer the second question in the title, I do not know, but I still hope that there is some hope. We have not tried everything yet.

Concurrency in particular

What does this all have to do with teaching the specific topic of concurrency?

Concurrent programming is now more important than ever. An obvious reason is the tremendously increased importance of net programming. Another reason is that major microprocessor manufacturers are running out of ways of improving the performance of microprocessors, so they have started to introduce microprocessor chips with multiple cores. As a consequence, it is becoming necessary to teach basic facts about concurrency to almost all prospective software engineers.

Concurrent programming is much less mature than sequential programming. Long ago there was confusion and disagreement about the right control structures, e.g., [2]. Today the main control structures in almost every sequential programming language are while- and for-loops, if-statements and some kind of a case-statement. Similarly, arrays, pointers and objects are now the universally accepted primitives for organizing data. However, there is no universal agreement on concurrent programming primitives, and in many cases the available primitives are not in the programming language but in additional libraries.

Furthermore, concurrent programming is difficult. In [2], Dijkstra wrote that "our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed." It seems strongly that our powers to understand concurrent processes are poorer still. The trend in software engineering seems to be to try to capture the behaviour of a concurrent system as a set of sequential objects such as use cases or message sequences, instead of capturing it as a genuinely concurrent object.

It is commonly agreed that concurrency introduces a particularly troublesome class of programming errors. The behaviour of a concurrent program may vary according to subtle timing differences, so that the same input may lead to different results. It is usually absolutely impossible to test all relevant timings, because there is a huge number of possible timings, and for most of them there is no way of predicting whether it is relevant. If we assume that testing is ten times as efficient in revealing errors as production runs and if the probability of some error occurring in one hour of testing is $10^{-d}$, then the error will most likely not occur during 100 hours of testing ten copies of the system, but will occur when 10,000 copies of the system are used for one year.

Basic topics in concurrency — such as critical sections, mutual exclusion, deadlocks, starvation, and semaphores — have traditionally been taught in courses on operating systems, covering also scheduling, memory management, etc. More recently, people have started to appreciate concurrent programming
as a topic of its own, as is evidenced by the publication of more and more books devoted solely to it, e.g., [1]. In 2005, as a part of a major re-structuring of the curriculum, the operating systems course in Tampere University of Technology was divided into an obligatory course on concurrency and an optional course on operating systems. The experience has been that now that concurrency is the only topic in its course, students take it more seriously. It has also proven difficult enough for the students, justifying devoting a whole course to it.

Because concurrent programming is more difficult and less mature than sequential programming, it would be important that students understand its issues deeply, so that they can adapt to varying concurrency primitives later on in their career, and can get the systems correct despite of the imperfection of the available guidelines and tools. Unfortunately, as long as the students' understanding of sequential programming is rather superficial, I do not believe in the prospect of them mastering concurrency so well that they could compensate for the deficiencies of existing concurrent programming techniques and tools.

My not particularly well-educated guess is that concurrent programming will remain a confusing and difficult topic for several years, until gradually some line of thinking leads to relatively-well working concepts, techniques and tools that make concurrent programming accessible to the masses and come to prevail. I think that we have seen a similar development before. Pointers and dynamic allocation of memory are powerful constructs, without which many programming tasks would be next to impossible to solve. Even so, twenty years ago they were considered so difficult that they were only taught to a subset of programmers. Today, object-oriented thinking offers easy mental tools with which almost every new programmer learns to write programs that, in essence, contain pointers and dynamic allocation of memory, although the programmer does not necessarily use these words.

It seems to me that in concurrent programming, theoretical computer science is further away from practical programming than in sequential programming. Although Turing machines are far from practical programming, each of the steps from programming to practical algorithms to big-\(O\) analysis of time consumption to \(\text{NP}\)-completeness to Turing machines is rather small and natural. I do not recognize an equally strong succession from practical concurrent programming to Petri nets or process algebras. Indeed, in Tampere University of Technology, practical data structures and algorithms are in the same group of courses as theoretical computer science. Also concurrency theory is in the same group, but practical concurrent programming belongs to a different group together with operating systems, embedded systems and the like.

Furthermore, similarly to the situation with practical programming, none of the theoretical models of concurrent programs is nowhere near as generally approved as Turing machines are in sequential programming. Although I claim elsewhere that state machines are an excellent theoretical model of reactive systems and synchronous interaction is a universal theoretical model of interaction and communication [9], I do not believe that all practical concurrent programming could be based on these. (Even so, it would perhaps be worthwhile to teach
something about state machines as part of software mathematics, because state machines are used a lot in software engineering.) When teaching concurrency theory to advanced students, I believe that it is recommendable to teach the fundamental ideas of more than one theory. This would make the students realise that there is no universally adopted theory of concurrency, help them to distinguish between general and theory-specific ideas, and encourage them to seek for ideas from different theories to solve practical problems.

Most theories of concurrency contain the notion of state space in one form or another. Although state spaces can usually be actually constructed only in a minority of cases and only with great effort, they are a useful concept that is easy to understand and helps to illustrate other concepts. The act of constructing a state space of some small system forces one to think carefully about the operational semantics of the concurrent programming notation in use. This brings forward issues that would otherwise easily remain implicit, such as what precisely comprises the total state of the system, and precisely what actions are atomic.

Therefore, teaching the easiest fundamentals of some concurrency theory — such as Petri nets, or Spin and Promela — to a large part of software engineering students would not necessarily be a bad idea. Because the goal would be just to understand some basic concepts, it is not so important which concurrency theory is chosen.

References