Formal Methods: Teaching and Practicing Computer Science at the University Level

Raymond Boute

INTEC — Ghent University, Belgium,
boute@intec.UGent.be http://www.funmath.be

Abstract. At too many universities, CS curricula are not taught at the university level. This causes stagnation in professional practices. The missing element is the pervasive presence of mathematical modeling throughout the curriculum. This is the role of formal methods (FM) in its original sense. Mathematical fundamentals and concepts are crucial, software tools are secondary and even misleading without the former.

Social, professional, educational and local influencing factors are discussed. Recommendations are given for for curriculum structure, for specific key cources and for attitudes to instill in students and educators. As a conclusion, FM should break outside the limitations caused by the conservatism of policy makers but also the self-imposed ones.

1 Observations

Conventions Drawing boundaries between Computer Science and Engineering is more in the interest of departmental power games than epistemologically useful, so we take CS in a wide sense, including (even adopting) the engineering view.

Parnas [21] observes that "Professional engineers can often be distinguished from other designers by the engineers' ability to use mathematical models to describe and analyze their products". The central role of mathematics is a fortiori evident to those viewing CS as a more theroretical activity than engineering.

Hence the use of mathematical modeling is a proper working criterion for what constitutes "university level" in teaching as well as in professional practice.

For completeness, some earlier material [5] will be recalled in passing.

1.1 The level of Computer Science curricula

In classical pure science and engineering disciplines, the pervasive use of mathematics throughout the curriculum has become self-evident since centuries.

Also, quite a large number of universities do teach Computer Science at the university level, and hence give no cause for concern.

Still, too many universities teach CS at the level of pre-Newtonian mechanics. When colleagues who teach classical engineering courses (say, electronics) look at the computing courses in such curricula, they often remark that these courses are merely descriptive, lacking in intellectual content, and altogether little more than inflated programming courses—an opinion reinforced by the kind of assignments: writing programs, more programs and reports, and doing uninstructive projects [20]. Unfortunately, this remark is all too often justified.

Indeed, many CS curricula seem to be designed as a refuge for mathphobic students just to increase the student count. The study by Tucker et al. [28] uses American data, yet reflects European trends equally well (see section 3.3). Instead of devoting the scarce teaching resources to solid fundamentals [20], they are wasted on trendy topics that students could easily pick up on their own. Courses like "Introductory Programming" usually teach some language (which is better learned in passing via the assignments) rather than program design.

Formal Methods, if taught at all, have a small place in the curriculum, looking like an afterthought. Students see this as the best "proof" of their uselesness: if FM were af any value, wouldn't all professors use them? Very few students are mature enough to see that curricula reflect the faculty [12] rather than relevance.

1.2 The level of professional practice

It is eery to see how little software practice has changed over the past 50 years.

Some younger software professionals suggest that things were better in the "old times" [24]. For instance, Spolsky also notes that "programmers seem to have stopped reading books" [25]. However, a 1975 paper [19] already commented on the computer illiteracy of professional programmers!

Google yields ample comments on the phrase "Programmers don't read", indicating that many programmers find the books worthless anyway. However, the titles quoted in the rejection are on programming languages, not program analysis or design. This just reflects the old misconception that the main capability of a programmer is the language.

Lethbridge surveyed "what knowledge is important to a software professional" [17]. The use of the preposition "to" rather than "for", also in the cited paper's abstract, is significant: it indicates that the survey is about subjective views. Circumstantial evidence leaves little doubt that many topics score low with the "average programmer" because they are not sufficiently known or mastered to be applied, rather than because they are truly less relevant or useful.

Hence, whereas Lethbridge advocates that "efforts to develop licensing requirements, curricula, or training programs for software professionals should consider the experience of the practitioners who actually perform the work", it

may be wise to use this information to detect the gaps, and educating professionals in what they need for advancing their professionalism rather than what they want, which is just more of the same in what they already know.

An important question is: how could the extreme conservatism that has been dominating software practices over the past 50 years survive, whereas practice in other engineering disciplines has kept abreast with technology? The answer is manifold, but the common element is complacency.

- Software thrives on the spectacular advances in hardware technology. Increased circuit speed and density often compensate for the stagnation in software design practices.
- Every few years some "method" is promoted that relies on acronyms rather than intellectual content, yet promises panacea with little effort. By contrast: formal methods promise considerably less and require more effort. In other words, adapting Gresham's law somewhat, "continuous injection of bad commodities keeps the good ones out of circulation".
- In the economic situation of the recent decades, computer and information technology have seen continued growth with only rare periods of faltering. Due to this fast pace, there always are plenty of urgent development tasks to be done at all levels. Even people with no computing background who invest a minor effort in self-study can contribute usefully, and many well-paid tasks are easily learned "on the job". With the urgent demand for quantity, and industry still uncertain about the exact qualifications software designers need, level and scientific basis are easily neglected.

The main problem is that many universities cater for the latter category, and actually advertise this attitude as "being responsive to the demands of industry". Demands maybe, needs certainly not. Students do deserve better.

Students are also the key. Indeed, Max Planck is reputed to have said

> Eine neue wissenschaftliche Wahrheit pflegt sich nicht in der Weise durchzusetzen, daß ihre Gegner überzeugt werden und sich als belehrt erklären, sondern vielmehr dadurch, daß ihre Gegner allmählich aussterben und daßß die heranwachsende Generation von vornherein mit der Wahrheit vertraut geworden ist.

Translated: "A new scientific truth usually does not break through in such a manner that its opponents are convinced and declare themselves informed, but rather because its opponents gradually die out, and the emerging generation has become familiar with it beforehand." The task of the universities is clear.

The Formal Methods community should also heed Planck's advice. All too often, the failure of FM is attributed to insufficient arguments for convincing practitioners. This has lead to considerable effort in producing successes in industrial applications that are impressive by themselves, but whose effects es-

sentially remain limited to the industries who benefit and the participanting researchers.

The effective lever for FM is not convincing the practitioners but educating the emerging generations.

2 Realizing the full potential of Formal Methods

2.1 Formal Methods

As observed by Gopalakrishnan [10], the term "formal methods" seems to suggest the sudden discovery by computer scientists of the use of mathematics—so evident in other branches of engineering that a separate appellation is redundant.

Still, the specific term could be justified by the fact that CS/ECE requires a more formal kind of mathematics than the classical mathematical/engineering disciplines, a point also emphasized in Lamport's book on specifying systems [14].

Here "formal" means that expressions are manipulated on the basis of their form (syntax) following precise calculation rules. This contrasts with traditional practice in math/engineering, where expressions are often manipulated on the basis of their interpretation (semantics) in some application domain. The benrefit is in letting the symbols do the work, which is especially useful in areas where intuition for the application domain is still nascent. It also develops a "parallel intuition" for handling symbols that supports domain-oriented intuition [7].

Understood in this wider (actually, more original) sense, Formal Methods point the way not only to teaching CS at the university level, but also provides a refreshing new style to mathematics [9]. The realization has been demonstrated for discrete mathematics [11] and essentially all of engineering mathematics [4, 6].

Hence the non-CS engineering mathematics can equally benefit, and the entire curriculum considerably streamlined by unification.

2.2 The role of tools

To realize the aforementioned potential of FM, the use of software tools has to be led into the proper channels.

For the sake of completeness, we first mention that some tool developers consider that tools are essential to FM or even "the only important thing" [23]. Taken literally, this statement is self-refuting by reductio ad absurdum. Still, it reflects the acceptance by industry of formal methods for solving specific kinds of problems economically. Epistemologically, however, it is far off the mark.

In mathematics, classical engineering and CS/ECE, software tools are meant to alleviate the burden of handling tedious details in calculations and proofs, and to reduce mistakes or avoiding pitfalls. Yet, this works only for users with ample mathematical maturity: for novices it can only lead to sorcerer's apprentice attitudes and even ruin [13]. Indeed, current tools are far from mature:

- No single tool has sufficient scope; even small systems require multiple tools;
- Implementation restrictions cause a very narrow view on mathematics;
- The many irrelevant syntax details and even semantic errors confuse novices.

For instance, tools like Maple and Mathematica seem designed fairly well for calculus and algebra, but in nearly every use for discrete mathematics the author found calculations going astray in unexpected and educationally hazardous ways. Here is an example, arising from number representation. Consider the functions

- `digits` such that, for any natural number n, the base 10 representation is `digits(`n`)`, defined as a function such that `digits(`n`)(`i`)` is the i-th digit; in Maple: `digits := n -> i -> floor (n/10^i) mod 10;`
- `decnum` such that `decnum(`f`)(`k`)` is the number represented by the k lowest digits in representation (function) f;
  in Maple: `decnum := f -> k -> sum(f(i)*10^i, i = 0..k-1);`.

One expects `decnum(digits(`n`))(`k`)` $= n$ if k digits suffice to represent n. Yet `decnum(digits(210))(3);` yields 620 while `decnum(i -> i)(3);` yields 210. The idea that "tools help students discover their errors" thus gets a new meaning!

In mathematics education, the trap of thinking that tool use can obviate solid mathematical reasoning abilities seems to have been largely avoided. Some calculus textbooks [27] even wisely contain examples and assignments involving tool use especially to foster awareness of the pitfalls. Likewise, insofar as Ralston [22] advocates tools over pencil-and-paper simulation of computational algorithms (e.g., "long division") in elementary school, he does not do so at the cost of reduced awareness for numbers and mathematics. To the contrary: he proposes more emphasis on head calculation as an antidote to rote and the errors typical for using calculators without numeric awareness.

Electronics engineers routinely use tools like Maple, Matlab etc. by relying on a good mathematical background which, more than the tools themselves, explains the successful results. Tool vendors do not boast or encourage mathphobia; to the contrary: an announcement for a textbook on Simulink [8] states:

"students should have the appropriate mathematical preparation [such as] calculus and differential equations".

Unfortunately, in the FM area tools are often advertised as "hiding the math" for professionals (thus depriving them of the most powerful intellectual tool) and depicted by some lecturers as an aid to learn math for students (instead of mathematics preparing for tools). Tools are popular because they cater for the affinity of some students for video games, while seemingly "realistic" tool-based projects give them the illusion of becoming "real engineers" quickly and easily: the "sics munce ago i coodnt evun spel enguneer, now i are won" syndrome.

Proper FM courses are essentially mathematical, do not teach tools, but contain assignments where students learn the use of tools and their defects.

3 Curriculum and course design

3.1 The mathematics content of engineering curricula

Learning by analogy from classical engineering disciplines and also considering the huge body of mathematical knowledge generated by CS in recent decades [10], we propose the mathematical content for a computing engineering curriculum.

Table 1. Fundamental engineering mathematics at various levels

(level)	EM. Engineering Math.	CM. Computer Engineering Mathematics
Basic (general)	Analysis, Linear algebra Probability and Statistics Discrete mathematics (combinatorics, graphs)	Formal proposition and predicate calculus Function(al)s, relations, orderings Lambda calculus (basics) Lattice theory, induction principles, ...
Targeted (modeling)	Physics, Circuit theory, Control theory Stochastic processes Information Theory, ...	Formal languages and automata Formal language semantics Concurrency (parallel, mobile calculi etc.) Type theory, ...
"Advanced"	Functional analysis Distribution Theory Hilbert and Banach spaces Measure theory, ...	Category theory Unified algebra Modal logic Co-algebras and co-induction, ...

Table 1 draws an epistemological parallel in terms of topics and levels, not course names. Here "basic" and "targeted" are proper for undergraduates; "ba-

sic" is domain-independent, "targeted" is more domain-oriented. We put "advanced" in quotes because it is debatable: given today's state of the art, some topics are arguably valuable for undergraduates as well.

EM is mathematics for classical engineering, e.g., electrical (EE). CM is the mathematics for Computing Engineering (CE). In view of curriculum design, the columns are meant to complement, not replace each other.

Indeed, considering just the first two rows, the EM topics are generally (and rightly) considered crucial in the formation of every engineer, and there is no reason to start making exceptions for CE's, to the contrary [20]: CE's can learn from the rich variety of examples and styles in mathematical modeling.

Conversely, the new mathematical style that emerged from Computing Science [9] is relevant to all exact sciences, as amply demonstrated for discrete mathematics [11] and for engineering mathematics in general [6]. For instance, [6] shows that it is useful for electrical and computer engineers (ECEs) to be as fluent in calculating with quantifiers ($\forall$, $\exists$) as with derivatives and integrals.

3.2 A Bachelor program in Computer Engineering

A program following the principles set out thus far is outlined in table 2. Note that it is a concept program, intended as an archetype for curriculum design by tailoring it to the local goals and possibilities. It can also be extended by either a 4th Bachelor year or a 2-year Master program.

Whereas most of the topic titles are familiar, the added value is in the unified mathematical modeling throughout all courses. In other words, the use of Formal Methods is ubiquitous, obviating a separate course named "Formal Methods".

The basis is a first-semester course in logic in a form that is useful in the spirit of Gries [12], not only for CS but for all of engineering mathematics, starting with analysis. To reflect this, we called it Application-Oriented Formal Logic. The style is calculational. We briefly outline one actual realization. Apart from the usual proposition calculus (e.g., [11]), the two main elements provided are

a. Generic functionals [4]: using higher order functions supporting systems modeling and mathematical reasoning in a point-free style, and smooth conversion between point-free and the more common pointwise style.
b. Functional predicate calculus [6], supporting the aforementioned fluency in calculating with quantifiers in the context of applications.

This unifying framework forms the bridge between the continuous world of classical engineering and the discrete world of computing. By providing a reference frame, it also facilitates introducing greater diversity in the formalisms

Table 2. A concept BCE Program (3 years, extensible to 4)

	First semester
6	Application-Oriented Formal Logic
6	Mathematical Analysis A
3	Algebra
3	Geometry
6	Physics A (particle & wave mechan.)
6	Introductory Programming

	Second semester
6	Physics B (electricity & magnetism)
6	Mathematical Analysis B
3	Languages and Automata
3	Disctete Math (combinatorics, graphs)
6	Classical Mechanics
6	Algorithms and Data Structures

	Third semester
6	Probability and Statistics
6	Complexity
6	Signals and Systems
3	Elements of Quantum Mechanics
3	Basic Electronics
6	Computer Architecture

	Fourth semester
6	Programming Languages
6	Thermodynamics, heat & mass transfer
6	Database & Information Systems
3	Elements of Quantum Computing
3	Electrical Networks
6	Operating Systems

	Fifth semester
6	Chemistry
3	Digital Systems
3	Information Theory
6	Concurrency
6	Software Engineering A
6	Embedded Systems

	Sixth semester
3	Properties of Materials
3	Elements of Relativity
6	Communications Systems
6	Communication Networks & Protocols
6	Software Engineering B
6	Hybrid Systems

(Legend: the numbers are a measure for the size of the course — see text)

and tools in the various other courses. No valuable time is wasted on "teaching" tool use or language features: these are picked up via the assignments. The courses themselves are fundamental and the appropriate body of knowledge can be structured around mathematical modeling (analysis, specification, design).

This point characterizes the curriculum, rather than a detailed listing of the topics in each course, which would be beyond the scope of a paper anyway. Yet, some additional observations help convey the guiding idea of "unity in diversity".

Many non-CS courses are included, such as physics and engineering, for two reasons: (a) to enhance professionalism, as many students will need some classical engineering in their later career [20], (b) to broaden the intellectual horizon and bring students in contact with a wide variety of modeling techniques.

As in any science curriculum, courses form streams, with dependencies and prerequisites, which restricts ordering somewhat but also provides opportunities.

For instance, arguably the best language choice for Introductory Programming is Scheme, with Abelson and Sussman [1] as a reference textbook. Of course, programming assignments should use Haskell as well. This background in computing meets with Physics A in Classical Mechanics (Sussman and Wisdom [26]). The Lagrange-Hamilton approach used to be covered in classical engineering curricula and has been neglected for some years (perhaps for being less intuitive initially), but the time is ripe for reinstating it. This also paves the way for the later courses on quantum mechanics and quantum computing.

Algorithms and Data Structures emphasizes formal specification and derivation of algorithms (rather than just listing them) and elementary type theory. The main criterion is correctness; other issues are covered in Complexity.

The Signals and Systems course is another crucial link between continuous and discrete modeling. The book by Lee and Varaiya [16] is one of the rare textbooks so far that, in addition to providing a good overview of the field, identify and correct most of the numerous notational defects in classical mathematical notations. Such defects have always hampered clean formal reasoning.

Traditional CS topics like Computer Architecture, Operating Systems and Information and Database Systems may have become less central to curriculum design in the past decades, but they are included as yet another opportunity for exercising and consolidating mathematical modeling techniques. This assumes the traditional content is revamped in this spirit.

Needless to say, Programming Languages is not meant to introduce languages, but to provide an introduction to language modeling, e.g., lattices and fixpoints, denotational semantics and more advanced type theory: language theory is the materials science for software. Application examples are the languages already encountered via the assignments in other courses (say, Scheme, Haskell, Maple, TLA+, Matlab, Simulink, LabVIEW, VHDL, Java, C++, ...).

The sequence starting from Physics A to Elements of Quantum Computing expands the students' horizon with nonstandard computational models.

Software Engineering is left most open to interpretation; the only assumption being that it meets the standards of the remainder of the curriculum. The Formal Methods community has generated a wealth of suitable material. Lecturers may opt for staying within a single series with a wide scope, such as Bjørner's Software Engineering trilogy [2], or devote a full course to a specific approach (such as refinements) using diverse sources, or any other combination or variant.

In a 4-year BCE, table 2 can be extended with Software Engineering C and D, Computer Security, with more advanced topics such as category theory and refinements, and with "integrating" courses combining mathematical modeling techniques. Examples are: Global timekeeping, positioning and navigation

and Embedded Systems in the Automotive Domain. The Automotive -, Train - and Avionics Domain are typical examples of domains where a great variety of modeling techniques must be used in harmony. Hence, although such titles may sound specialistic, the content can be given a wide scope. The educational value of topics with this characteristic is well-known in classical engineering, but the domains idea has been given a new impetus for (E)CE by Dines Bjørner [3].

Course metrics The numbers in table 2 indicate course size in units, assuming a 60 units per year system. The design is adaptable to various local situations.

For instance, regulations at our own School of Engineering stipulate per semester 30 units involving 900 hours of study (classes, guided or lab exercises, self-study), amounting to 30 hours per unit. The smallest useful size of a course is 3 units, with classes over 12 weeks at $1\frac{1}{4}$ h/week. The exam period is 4 weeks. Arithmetic shows that this does not favor activity during the 12 weeks of classes (and that the 900 hours are administrative fiction).

A better option is spreading the courses over 15 weeks, assuming per 3 units weekly 1 hour of classes plus $3\frac{1}{2}$ hours of self-study and lab exercises, if any. Own experience at various universities indicates that homework is considerably more effective than guided exercises in stimulating activity[1], but puts heavier demands on the staff. Anyhow, the total load of 45 hours/week is reasonable, and low in comparison to what is expected at world-class universities.

3.3 Local and international context

Effects of the local context Thus far, too many universities have been unable to integrate mathematical modeling throughout the CS/CE curriculum or approached the university level considered normal in classical engineering fields.

Still, a well-documented design of an actual curriculum integrating mathematics education with software engineering is the BESEME (BEtter Software Engineering through Mathematics Education) project [18].

This report also outlines the impediments to be expected [18, p. 6]:

- Students find it demanding.
- Most instructors must revise notes.

In reality, "revising notes" is a euphemism for acquiring essential background [12]. Whereas students are young and (hence supposedly) flexible, many lecturers do not embrace lifelong learning, especially if it entails novel ways of thinking [9].

Here are two more small "case studies" illustrating other concrete situations.

[1] Regular homework also better prepares students, making 1 week for exams sufficient.

(i) At our School of Engineering, the effects just mentioned have proved quite manifest. As a result, weaving mathematical modeling in the CE curriculum remains a faraway target supported by a minority, but felt as a threat by others.

At one stage, our CE program included a course designed like Application-Oriented Formal Logic and a sequel on formal modeling, both non-optional.

The positive effects were most apparent in how students who had taken these courses applied the acquired abilities in their MSc and PhD dissertations under various supervisors who were themselves not even involved in FM.

Still, courses with a strong mathematical content taught by various lecturers, not only in FM but also in classical engineering, got negative evaluations from a few CE students. This was used as an argument by a certain faction among the faculty to clamp down on the classical engineering course involved, and to relegate FM to just one course, for a few last-year students. The fact that foundations are most efective when laid early in the program was well-known.

(ii) In the design of their CS program, our School of Science made a more constructive use of this fact, and placed Application-Oriented Formal Logic with the contents described earlier in the first semester. The professor teaching the course is one of the author's best former students, although scarcity of qualified teaching assistants may have eroded the application-oriented elements somewhat.

These experiences confirm the observation by many authors regarding the overwhelming importance of general acceptance by the faculty, or at least tolerance from those not wanting to invest effort in new ways of thinking [12, 15, 29]. Other major obstacles against teaching CS/CE at the university level are inbreeding and the idea that CS-related courses can be entrusted to lecturers from a different engineering area (e.g., EE) with just some programming experience.

In principle, students pose fewer problems. Of course, in curricula where formal methods courses are isolated, their mathematical content and lack of applications in other courses demotivates some students. This is why, judging from the literature on teaching FM, so much effort is spent on student motivation, and positive results are strongly highlighted (although difficult to measure).

However, consider for the sake of comparison the student acceptance of Mathematical Analysis by the less mathematically inclined students (the math enthusiasts are not at issue here). Much of the acceptance stems from this topic being an unavoidable part of any first and second year engineering program. Choosing engineering means choosing Mathematical Analysis, even for CE majors. Ubiquitous use in other courses confirms relevance in the classical areas.

If a basic course like Application-Oriented Formal Logic is also made a fixed part of the first-year program (as in case study (ii) above), students tend to

accept it more easily as characteristic for university-level (E)CE education. The basis given can then be assumed without further ado in all other courses, providing further consolidation by various applications, which motivates students by confirming relevance. Whether other courses pick up the thread depends on the lecturers, but at least the basis is there and the other courses can evolve gradually. Moreover, with this background the better students will be more demanding with respect to methodology, and subconsciously or consciously exert gentle pressure on the lecturers if necessary. The attitude of faculty towards this prospect is a decisive factor (and a quality measure), which closes the circle.

In this context, course evaluations must be used wisely. The comments of the students always provide very valuable information for the instructors. However, if students are given the impression (or confirmation) that their feedback is used uncritically for making ad hoc curriculum reforms, their answers will become heavily biased and only serve hidden agendas incompatible with quality.

International context Traditions regarding the mathematical content of engineering curricula vary greatly between countries, and are subject to oscillations and mutual feedback.

For instance, Belgian engineering schools used to have very strong mathematical requirements, enforced by an entrance examination. The high standards were set since the early 19th century by the Grandes Ecoles and in particular the Ecole Polytechnique in France.

Recent politically-inspired European reforms such as the Sorbonne-Bologna declaration have reshuffled the landscape, increasing the differences they pretended to reduce.

For instance, the entrance examination has been cancelled in Flanders, not in Wallonia. The impact has been downplayed, since at that time Flanders scored first among Western countries in the TIMSS (Trends in International Mathematics and Science Study) surveys. However, as a result the mathematically strong programs in secondary school experienced less interest, and declined. As the effects became noticeable in the TIMSS scores, attention shifted to the PISA (Programme for International Student Assessment) criteria. These critera are based on RME (Realistic Mathematics Education), which has strongly influenced math education for young children in some Western European countries. As expected, these countries now tend to get a better place in the PISA rankings than they formerly did under TIMSS, creating the illusion of improvement.

The catch is that RME focuses on concrete "everyday life" problem situations, for which actually just a modicum of common sense suffices. Now common sense is laudable, but genuine mathematics involves considerably more, such as developing abilities for logic and symbolic reasoning and for making

abstraction. These abilities are precisely the most important ones for engineering, classical as well as computer-oriented.

The mathphobia deplored in [28] is certainly not limited to the U.S., where even ample effort is devoted to countering it, but has become a matter of fashion among certain groups of children and adults in Western Europe, even in countries that used to have a strong mathematical tradition. There are indications, needing further study, that some Eastern European countries have escaped this fashion.

Due to these various factors, the mathematical level of students entering engineering schools has decreased in various European countries. Since technology does not comply by lowering its complexity, the first year of the curriculum is important for helping the students to bridge the gap. A framework unifying the mathematics for classical and for computer engineering can be instrumental.

The top scores in the TIMSS are achieved by Asian countries. This indicates considerable potential for maintaining universities at a suitable level to meet the challenges of the future. The names in the literature constitute ample evidence. Judging by the faculty lists of universities in various regions, the U.S. appear to have tapped these intellectual resources more effectively than Europe.

The emphasis on mathematics that was consciously maintained throughout this discussion may appear excessive if one sees mathematics just as a collection of tricks and facts, which is the view held by many laypersons. However, the main educational value of mathematics resides in the development of effective problem solving, reasoning and abstraction abilities and the attitudes it imparts.

4 Conclusion

Referring to the quotation by Max Planck, we have argued that the most effective way for making Formal Methods an evident part of everyday practice is not convincing the current practitioners but investing in the education of future generations. Formal Methods, in the sense of mathematical modeling, can be the lever to lift the entire computing curriculum to the scientific and professional level that would be considered acceptable in classical university-level engineering. It goes without saying that this strategy is not directly needed, yet can still be inspiring, for universities where (E)CE education is already world class.

We have outlined the design principles and to some degree the content of a no-nonsense program that can serve as an archetype for various curriculum designs by tailoring it to the locally available structure and human potential. Graduates from such a program will consider formal methods as evident in their professional practice as classical engineering math in, say, electronics or mechanical engineering.

On the other hand, we have seen that the road to an integrated curriculum is fraught with many impediments, the most important obstacle perhaps being lack of intellectual flexibility and curiosity and, as Gries notes [12], sometimes even mathphobia on the part of the lecturers for the computing-related courses.

The last part of the paper addressed some local and international factors affecting the availability and effective use of intellectual resources in various parts of the world.

One final observation: throughout our argumentations, we have hesitated to invoke one of the primary tasks of a university, because the focus on direct utility in recent years has made even its mentioning suspect. This task is conveying to our students a rich intellectual heritage and stimulating curiosity, and ultimately contributing to their cultural development, not just their professionalism.

Such goals may appear overly ambitious and perhaps lofty today, but the least one can do is avoiding a curriculum ridden with shortcuts.

Acknowledgement The author thanks the anonymous reviewers for their many helpful comments and observations, some of which appear in the text this paper.

References

1. Harold Abelson and Gerald Jay Sussman with Julie Sussman, Structure and Interpretation of Computer Programs. The MIT Press (1996).
2. Dines Bjørner, Software Engineering (3 volumes). Springer (2006).
3. Dines Bjørner, "What do I mean by Domain?". `http://www2.imm.dtu.dk/~db/`
4. Raymond Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons, Johan Jeuring, eds., Generic Programming, pp. 89–119, Kluwer (2003).
5. Raymond Boute, "Can lightweight formal methods carry the weight?", in: David Duce et al., eds., Teaching Formal Methods 2003, Oxford Brookes University (2003). Web: `http://cms.brookes.ac.uk/tfm2003/papers/boute.pdf`
6. Raymond Boute, "Functional declarative language design and predicate calculus: a practical approach", ACM Trans. Prog. Lang. Syst. 27, 5, pp. 988–1047 (2005).
7. Raymond Boute, "Calculational semantics: deriving programming theories from equations by functional predicate calculus", ACM Trans. Prog. Lang. Syst. 28, 4, pp. 747–793 (Jul. 2006).
8. James B. Dabney and Thomas L. Harman, Mastering Simulink 4 (2nd ed.). Prentice Hall (2001).
9. Edsger W. Dijkstra, "How Computing Science created a new mathematical style", EWD 1073. University of Texas at Austin (Mar. 1990).
   Web: `http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1073.PDF`
10. Ganesh Gopalakrishnan, Computation Engineering: Formal Specification and Verification Methods (Aug. 2003).
    Web: `http://www.cs.utah.edu/classes/cs6110/lectures/CH1/ch1.pdf`
11. David Gries and Fred Schneider, A Logical Approach to Discrete Math. Springer (1993).
12. David Gries, "The need for education in useful formal logic", IEEE Computer 29, 4, pp. 29–30 (April 1996).

13. Henri Habrias and Sébastien Faucou, "Linking Paradigms, Semi-formal and Formal Notations", in: C. Neville Dean and Raymond T. Boute, eds., Teaching Formal Methods, pp. 166–184, Springer LNCS 3294 (Nov. 2004).
14. Leslie Lamport, Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002).
Web: http://research.microsoft.com/users/lamport/tla/book.html
15. Edward A. Lee and Pravin Varaiya, "Introducing Signals and Systems — the Berkeley Approach". First Signal Processing Education Workshop, (Oct. 2000).
Web: http://ptolemy.eecs.berkeley.edu/publications/papers/00/spel/
16. Edward A. Lee and Pravin Varaiya, Structure and Interpretation of Signals and Systems. Addison-Wesley (2003).
17. Timothy C. Lethbridge, "What knowledge is important to a software professional?", IEEE Computer 33 5, pp. 44–50 (May 2000).
18. Rex L. Page, "Software is discrete mathematics".
Web: http//www.cs.ou.edu/~beseme/besemePres.pdf
19. A. Parkin, "Professional Programmers – Do They Read?", Computer Bulletin, Series II, 4, p. 23 (1975).
20. David L. Parnas, "Education for computing professionals", IEEE Computer 23, 1, pp. 17–22 (Jan. 1990).
21. David L. Parnas, "Predicate Logic for Software Engineering", IEEE Trans. SWE 19, 9, pp. 856–862 (Sep. 1993).
22. Anthony Ralston, "Let's Abolish Pencil-and-Paper Arithmetic". Journal of Computers in Mathematics and Science Teaching, Vol. 18, No. 2, pp. 173–194 (1999).
Web: http://www.doc.ic.ac.uk/~ar9/abolpub.htm
23. John Rushby and Natarajan Shankar, "Theorem Proving and Model Checking for Software", Tutorial, Fourth Symposium on the Foundations of Software Engineering (Oct. 1996).
Web: http://www.csl.sri.com/users/rushby/slides/fse4tut.ps.gz
24. Joel Spolsky, "The Perils of Java Schools", in: Joel on Software (Dec. 2005).
Web: http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html
25. Joel Spolsky, "Stackoverflow.com", in: Joel on Software (Apr. 2008).
Web: http://www.joelonsoftware.com/items/2008/04/16.html
26. Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer, Structure and Interpretation of Classical Mechanics. The MIT Press (2001).
27. George B. Thomas, Maurice D. Weir, Joel Hass, Frank R. Giordano, Thomas's Calculus, 11th ed. Addison Wesley (2004).
28. Allen B. Tucker, Charles F. Kelemen, Kim B. Bruce, "Our Curriculum Has Become Math-Phobic!", ACM SIGCSEB, SIGCSE Bulletin 33 (2001).
Web: http://citeseer.ist.psu.edu/tucker01our.html
29. Jeannette M. Wing, "Weaving Formal Methods into the Undergraduate Curriculum", Proc. 8th Intl. Conf. on Algebraic Methodology and Software Technology (AMAST) pp. 2–7 (May 2000). Web:
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/calder/www/amast00.html

GRACE TECHNICAL REPORTS

Proceedings of the First International Workshop on Formal Methods Education and Training

Jim Davies, Jeremy Gibbons, Mike Hinchey and Kenji Taguchi (editors)

GRACE-TR 2008–03 2008 October 28

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/