A Perspective on Formal Methods in Education

Jayadev Misra
Menlo Park, 6/09/2008

1 Introduction

It is tempting for me to give a speech about my latest research. Fortunately, for you, I will resist the temptation.

Let me describe my experience with formal methods in computer science education. For an incoming grad level class, I would have a quiz in the very first lecture. Around the time Shankar, Dave and Pat were students, I would have put questions like, (1) what is a monotonic function, (2) what is a well-founded order, (3) prove a given program correct with respect to a specification. Typically, 50% of the domestic students and 75% of the Indian ones would do reasonably well in a test like that. Over the years the questions became far more lenient —what is a loop invariant, what is a partial order, what is strong induction—, and the grading became equally lenient. Yet, the scores still fell precipitously. Last I checked, just a handful, less than 5 in a class of 40, knew the term "invariant". These results have become so predictable that I stopped giving the quiz. Instead, I ask the teaching assistant to cover some preliminary material starting with predicate calculus.

Things are not better elsewhere. A young colleague, an assistant professor from a top-4 school, told me cheerfully that he does not know predicate calculus, but he can pick it up any time he wants to, because the material is so simple.

Some years back, I was asked to develop a course, called "Theory in Programming Practice". It is required of all our undergraduates. They take the course after taking 3, or so, programming courses, and 3 formal courses. I am constantly hampered in teaching this course by the students' lack of formal/abstract training: inability to generalize appropriately, inability to see the core of a problem, inability to trust symbols and manipulate them. Every concept has to be explained in relation to reality, in words or pictures, with an elaborate example. I call this "low bandwidth communication".

I had caricatured this phenomenon in a satirical piece some years ago [3]. A Roman mathematician of antiquity is explaining why the Roman numeral system is superior to the decimal system. When you add 9 sheep to 17 sheep, he explains, the Roman system offers a clear link to reality. At every stage during the computation, you can divide the sheep into two classes, those who have been counted and those who remain to be. In the decimal system, the notion of a carry corresponds to nothing in the real world; an unnecessary complication.

Computer is a wonderful device for experimentation. The students are often tempted to test out an idea rather than study its foundation. And, they do it even when it takes much longer to experiment. I contend that availability of experimentation tools, even when imperfect, makes it far more difficult to interest students in formal methods. The areas in which experimentation is infeasible, such as formal semantics, has the highest acceptance of formal training. Conversely, testing an algorithm is far simpler than proving it; so there is very little emphasis on program verification in introductory algorithm books. A paper I read recently, "Our curriculum has become math-phobic" [4], cites some interesting statistics. Among the 5 most popular books on data structures, each in the neighborhood of 800 pages, only one book devoted 8 pages to program correctness; others none at all. This is particularly infuriating for algorithms that cry out for a formal treatment, because the informal treatment is usually labored, verbose, incomplete, or just plain wrong. Once I took a page from a leading book on algorithms to a faculty meeting, and asked if any one can explain to me what the authors may be hinting at. I had spent 5 fruitless hours on that page, on Knuth-Morris-Pratt string searching algorithm, a topic that I knew very well. An invariant would have been welcome! The message to the students is crystal clear: it is best to muddle through with words, pictures and plausible explanations than make an honest effort at a proof.

Let me lighten up by quoting that mythical Roman mathematician again [3]. The supporter of the decimal system asks incredulously: how can you multiply in your system,? Say, you have the following arithmetical problem. It takes two Christians to keep a lion happy for one hour. How many Christians must be supplied to keep 3 lions happy for 4 hours? And, the Roman responds, much like our contemporary software designers, that he would run a live simulation.

2 The Trend

I am afraid that the trend is negative, very negative. You may expect that my department at the University of Texas would be doing much better, given that its faculty includes, among others, Bob Boyer, J Moore, Warren Hunt, Allen Emerson and Vladimir Lifschitz. The reality is sobering. We have not hired a junior faculty member in formal methods since Allen Emerson in 1983. We can not justify hiring faculty members because so few students plan to work in these areas, the only exception being Security-related research. Thus, there is a vicious circle: few needed, few supplied, few created.

In the paper that I had mentioned earlier [4], the authors cite an interesting exchange. A committee was developing recommendations for undergraduate CS curriculum in 2001. The proposed curriculum was reviewed by a group of computer scientists, who were probably sympathetic to theory. The reviewers

“expressed concern about the diminishing amount of theory and the lack of integration between theory and practice in the core curriculum. This letter made some concrete recommendations that would restore this balance. After careful consideration, the committee decided not to accept these recommendations, for the following stated reasons:

- The centrality of theoretical areas (algorithms and programming language theory) is declining relative to that of many other areas in computer science (net-centric computing, graphics, human-computer interaction, information management, software engineering, and social and professional issues).
- The curriculum model must serve the widest variety of institutions.
- The curriculum model must serve the interests of the present and the future, rather than retreating to a “computer science of the past”.”

In my own department, we are undergoing a curriculum revision. The committee charged with revision has recommended reducing theoretical courses from 3 to 2 for core requirements. And, program correctness seems to be an optional topic. I am convinced that the students will not be hampered in the other courses by their lack of formal training. A typical operating system course does not ask for a formal definition of deadlock, nor does an architecture course attempt to justify a cache-coherence protocol. As long as the world is the way it is, lack of formal knowledge is not an inconvenience.

3 Recommendation

I am against teaching formal methods for their own sake. There are wonderful theories, like Recursive Function Theory, that we don’t teach, because people believe that it has little relevance to practical computer science. Unfortunately, a large number of computer scientists believe that we are getting by just fine without formal methods. There is no way to refute that argument without appealing to an alternative world, that things would be much better, productivity much higher, if practitioners knew theory. We have not supplied any such empirical proof.

What the industry seems to want is change that is unobtrusive. A leading computer scientist, more of a practitioner, told me recently that the greatest contribution of formal methods have been the theory of types. The type checker lurks in the background, helpfully slapping your wrist at the right moments. You have to be minimally trained to enjoy its benefits. The view espoused by many is that the best formal method is what you don't see. It is a bunch of tools, much like the tools of linear algebra or graphics, whose foundations are known only to a few, but the benefits are enjoyed by many.

This brings me to the most outstanding success story in formal methods in the last 20 years, Model Checking. It meets several criteria of success: push-button technology (though with many limitations), automatic counterexample generation, and applicability to actual industrial-strength problems, which is a rarity among formal methods. Twenty years ago, electrical engineering students in my class knew logic to the extent of truth-table creation. Today, any student aspiring to do circuit design has to be familiar with the interface to a model checker; she would know how to express certain basic safety and progress properties in CTL*, or some equivalent logic.

Let me tell you about my pet project on tool development. Five years ago, I teamed up with Tony Hoare to push his vision of a verifying compiler [2, 1]. Over the years, with the help of many scientists, many of them in this room, the vision has become more concrete and realistic. We would like to see developed a suite of compatible tools capable of routinely proving, with minimal human assistance, industrial strength programs of hundreds of thousands lines of code. We would like these tools to be usable before, during and after the design. Tony would like to see complete functional verifications stretching from buffer overflow and null pointer dereferencing to properties of concurrent, real-time and embedded systems, and analysis of numerical round-off errors. The goal is to provide complete assurance about the software. Tony realizes that this may be a hopelessly unachievable goal, yet as true scientists we should pursue perfection.

I see opportunity and opportunity in a project of this dimension. The project will surely fail if the formal method educators do not produce sufficient number of well-trained graduates. By contrapositive arguments, this project, if properly funded, will allocate significant resources toward formal method education and research. Secondly, the aspect that truly interests me is that it will energize several core areas of Computer Science: programming languages and type theory, semantics, specification logics, decision procedures and their data structures, concurrency, real-time and embedded system design, and large-scale software engineering for integrating tools from different parties. Lastly, it will have an impact on computer science education. We can all communicate at higher bandwidth using symbols, rather than words and pictures.

References

[1] C.A.R. Hoare and Jayadev Misra. Verified software: Frequently asked questions. *EASST Newsletters*, (ISSN 1861-0668):18–30, December 2005.

[2] C.A.R. Hoare and Jayadev Misra. Verified software: Theories, tools, experiments. *EASST Newsletters*, (ISSN 1861-0668):5–17, December 2005.

[3] Jayadev Misra. A visionary decision. In Manfred Broy, editor, *Proc. 9th International Summer School on Constructive Methods in Computer Science*, volume F 55 of *NATO ASI Series*, pages 1–3, Marktoberdorf, Germany, July, 1988, 1989. Springer-Verlag.

[4] Allen B. Tucker, Charles F. Kelemen, and Kim B. Bruce. Our curriculum has become math-phobic! In H. Walker, R. McCauley, J. Gersting, and I. Russell, editors, *SIGCSE Technical Symposium on Computer Science Education*, pages 243–247, Charlotte, North Carolina, USA, 2001. ACM Press.