Why, How and What Should be Taught about Formal Methods?

Maximiliano Cristiá

CIFASIS and UNR, Rosario, Argentina
cristia@cifasis-conicet.gov.ar

I started to teach formal methods (FM) in an undergraduate, mandatory course of a Computer Science degree at Universidad Nacional de Rosario (Argentina) in 1998. After all these years I still have more questions than answers. In particular, I still can't find satisfactory answers to questions I asked to myself back in 1998. However, at the same time, I still *believe* that I am doing the right thing. I would like to share and discuss with all of you my doubts and what I have learned during all these years regarding what should or should not be taught about FM to undergraduate students.

This talk will be more a collection of questions and doubts rather than (firm) statements. So let's begin with some of them. Should we teach FM to undergraduate students? I dislike absolute answers to sociological questions (teaching is, after all, a sociological activity). For instance, should we teach FM regardless of the length of the plan of study? The answer would be yes provided FM are considered so fundamental to the profession that none of them can graduate without passing a course on FM. So, are FM that fundamental to the profession of programmers? Here, the discussion can be forked again: a) Fundamental to do what? What is the profession under the umbrella of informatics? Does the professional working on the avionic system of an aircraft need the same education as a system administrator?; and b) Do we really mean *programmers*? Why don't we say *software engineers*? Is the same a programmer as a software engineer? Should both of them know FM? But, let's go back to the previous question, the one referring to 'fundamentals'. Are FM fundamental to Computer Science or to Software Engineering? I think that if they are fundamental to something it's (just) to Software Engineering. David Parnas has already stated the difference between Computer Science and Software Engineering [2]. He has been quite critical about how FM were designed and used. He says that they don't reflect the way mathematics is used in other engineering disciplines. And this makes software engineers to seldom use FM. So, should we teach FM to undergraduate students of Software Engineering degrees?

Let's assume for a moment that we should teach FM to future software engineers. What FM should we teach? As you know, there are dozens of mathematical approaches to software development that fall within the realm of FM. It simply makes no sense to teach all of them as it make no sense to teach all the programming languages. Choosing what programming languages to teach might seem easier: just pick the top 3 used in industry. Why 3? Why not just 1? Why the top used in industry? Are the top programming languages used in industry fundamental to the software professional? I think that most of us will agree

that 'used in industry' is not the best criterion to choose what programming languages we should teach. This is a fortunate conclusion because it would be very hard to use it to choose what FM to teach. Instead, we would agree that we should teach programming languages belonging to different programming paradigms. Why the paradigm-based criterion seems better? In my opinion, because it allows students to use the best tool for each problem. Hence, I use the same guidelines to select a menu of FM to be taught. (Yes, I teach more than one formal method!) Why? Because from my point of view they are tools that fit best for different problems. For example, a Statecharts model is good to describe reactive systems, while a Z model is better to describe a payroll system. As you wouldn't admit that a Software Engineering plan of study including only one programming language is complete, why would you deem it as complete if it includes only one formal method? Aren't there different formalization paradigms? Probably yes, but, wasn't there the point that FM are fundamental to software engineers? If we should teach them because they are fundamental, then seeing them as tools could seem a contradiction to some of us.

Once we agree on what formal methods we should teach, we need to decide how are we going to teach them. Let's say we decided to teach Z. Should we start by teaching set theory? Probably not because we can assume students have already learned set theory. Should we start by teaching its syntax and semantics? I guess that we all have already learned that this is not the best way. *If FM are tools*, then the best way to learn them is by using them. Then, we present some (functional) requirements and show to students how they can be formalized in Z. At this point many technical questions can be raised. Should we continue by teaching how to verify the model? Should we continue by teaching how to implement the model? Should we teach refinement calculus? In my opinion the answers to these questions depend on many factors (such as the length of the plan of study), but in general I tend to think that 'no' is the right answer in most of the cases. For example, I have found more useful to show that: a) formal models could be automatically turned into prototypes; b) to link the formal model with the design of the system (i.e. the document describing the software components, their functionality and the relationships between them [1, chapter 4]); and c) to show how test cases can be generated from the formal models. All these activities would allow students to put FM in the bigger picture of software production.

References

1. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of software engineering (2nd ed.). Prentice Hall (2003)
2. Parnas, D.L.: Software engineering programs are not computer science programs. IEEE Software 16(6), 19–30 (1999), http://doi.ieeecomputersociety.org/10.1109/52.805469