Introducing Model Checking to Undergraduates

Abhik Roychoudhury

Department of Computer Science, National University of Singapore
abhik@comp.nus.edu.sg

Abstract. Introducing temporal logics and model checking to undergraduate students is usually an involved activity. The difficulty stems from the students' lack of exposure to logics, unfamiliarity with reactive systems and lack of conviction that model checking search can lead to anything practical. Here, I narrate some experiences in attempting to overcome these stereotypes over a period of five years at the National University of Singapore.

1 Introduction

Teaching of formal techniques has always been a topic of discussion and debate in Computer Science (CS) education. CS academics have underlined the importance of encouraging formal system development practices by trying to incorporate them into the CS curriculum (*e.g.* see [2]).

However, in reality the task of convincing students of the value of formal methods could be a formidable one. This is typically because of the paucity of formal methods courses in the CS curriculum which results in students' inherent lack of exposure to formal techniques. Often times, we face the following arguments from our students (or even our colleagues).

- CS students (particularly undergraduates) are not strong enough to learn formal methods, or
- It is difficult to get CS students used to the rigor of formal methods (even if they are capable), or
- Formal methods is mostly about mathematics which is of limited value for building (computer) systems.

As a formal methods educator, I feel that all of these arguments are false. The question is how to do we get past such stereotype arguments which may reside in the minds of our students (who might think that formal methods courses are to be avoided) or even our colleagues (who might perceive formal methods to be of little value).

In this year's International Conference on Software Engineering (ICSE), there was a panel on Formal Methods where the panelists were asked — *if you have $10 million for promoting formal methods how would you invest it ?* Several panelists underscored the value of education for such investment. More importantly, an interesting analogy [1] was drawn with engineering undergraduates who reguarly learn complex mathematical concepts (such as differential equations) thereby strongly alluding to the incorrectness of the usual argument that Computer science undergraduates are not strong enough to learn formal methods.

In this article, I describe some experiences in teaching formal methods to undergraduate students over five years. Primarily, I deal with issues arising from exposing undergraduate students in Computer Engineering to a course on model checking. *All course materials* have been made freely available from

`http://www.comp.nus.edu.sg/∼abhik/CS4271`

Before proceeding to elaborate on the teaching methods, I give some background information about the course which may be helpful in judging the applicability of my teaching methods.

2 Background Information about the Course

The course in question was offered as an *elective module* once a year for five years at the National University of Singapore. It was part of the Computer Engineering curriculum, that is, it was offered to our Computer Science students specializing in Computer Engineering. These students take a wide variety of Computer Science courses with a concentration of courses on embedded system design. Consequently, they are required to take four electives on system design from various courses such as:

– *Critical Systems and their Verification*
– *Hardware Software Codesign*
– *Mobile Computing*
– *Performance Analysis of Embedded Systems*
– *Embedded Software Design*
– *An advanced course on Computer Networks*
– *An advanced course on Programming Language Design and Implementation*
– ...

The course we are discussing here is the first one in the above list – *Critical Systems and their Verification*. Note that the other courses in the list are more focused on (embedded) system performance rather than formal techniques. So, our course (which was offered as an elective) is inherently somewhat different from the other elective modules.

Our course on formal verification is taken by third or fourth year Computer Engineering undergraduates with most of the students coming from the fourth year. The total number of students in the Computer Engineering programme is approximately 75 out of which approximately 45 students have opted for the formal verification module in the last three offerings of the module. The exact enrolment numbers over the five years are as follows.

2001-02	2002-03	2003-04	2004-05	2005-06
3	32	34	45	54

Since our verification course was offered as part of the Computer Engineering curriculum, we could not require a course in Logic to be a pre-requisite (since a Logic course in not mandatory in our Computer Engineering curriculum). This in fact made the offering of the course more challenging. The only pre-requisites of our course on formal verification were:

– a first year undergraduate course on Discrete Mathematics (which gives the students brief exposure to propositional and predicate logic), and
– a first year undergraduate course on Computer Organization (which gives the students some exposure to combinational/sequential circuits, buses etc).

Note that most of our students were in their fourth year and they only had a brief introduction to logic in their first year of undergraduate study. Hence it was necessary to communicate to them that the course goes beyond logic/discrete maths. On the other hand, it was also important to refresh their background on logics while introducing temporal logics. In the next section, I proceed to outline the main strategies that were adopted as an attempt to enhance the students' learning experience. Some of these strategies are standard ones, while some were learnt gradually by offering the course multiple times.

3 Strategies to Enhance Students' Learning Experience

For enhancing the students' learning experience, we need to elicit more student interest and participation by relating the techniques (in this case model checking) to real-life. However, this is often done in a rather extreme way by mentioning dramatic historical disasters which happened due to lack of formal verification. Too often we motivate a formal verification technique by mentioning the Arianne space shuttle disaster, or the Therac-25 accidents. If we (the formal methods educators) decide to be a bit more down-to-earth while motivating our techniques, at most we refer to the Intel Pentium floating-point error from 1994 (which resulted in substantial financial loss for Intel). Clearly, mentioning these historical incidents to the students serve an important purpose— they get the students' initial attention/interest. However, from my experience, this interest is often *difficult to retain* — possibly because many of these historical disasters seem to be far removed to the students. Emphasizing these historical incidents also serves to emphasize the students' perception that formal methods is something "exotic" — a perception we as educators should fight against.

Students need to understand, they do not need to be surprised As a first step, I have avoided mentioning historical disasters in my lectures for the purpose of motivating formal verification. Instead, in the first lecture, I try to refer to existing industry practice in "verification and validation" — why these practices do not amount to formal verification, and what needs to be done to achieve formal verification. Since my course is part of a programme with Embedded System focus, I refer to some existing Electronic Design Automation industry practices in this regard (methods like in-circuit emulation). I try to explain how the existing methods are intrusive to the design process and how a model-based technique can help the design cycle. This results in a rather different pedagogical style, where the aim is to discuss the system design cycle with the students rather than impressing/surprising the students with the power of formal verification.

Presenting formal verification as a tool to improve the system design cycle helps. It removes the misconception that formal methods are required only for very very safety-critical systems which normal engineers need not be bothered with. However, until and unless the students can get some amount of gratification from using formal methods, they are quite likely to forget about it once the semester ends. Often, we (as formal methods educators) take a view that the theory should be taught prior to the tool. In a course focusing on model checking this would mean that the students need to learn about Kripke Structures, Temporal Logics, Explicit-state checking, Binary Decision Diagrams (BDD) and Symbolic Checking — even before they can write a single line of code in a model checker! Clearly, such an approach is unlikely to evoke student interest. We could try to improve the state of affairs by teaching only Kripke Structures, Temporal Logics and explicit-state checking prior to discussing model checkers. However, from my experience, a significant fraction of the students still feel lost by the time the model checkers are introduced. To effectively teach model checking, it is important to discuss system modeling (from requirements) as early as possible.

Discuss System Modeling as early as possible To address this issue, I try to familiarize the students with (at least) the input language of a model checker even before they learn temporal logics and model checking. So, the rough flow of my course (which focuses on model checking) is

– Transition Systems and Kripke Structures
– SMV model checker and case studies
– Temporal Logics
– Explicit-state Model Checking
– Binary Decision Diagrams (BDDs)
– Symbolic Model Checking using BDDs

A few points need to be emphasized at this stage. When we discuss SMV and its case studies, I try to pick moderate sized but real-life case studies. These examples serve an important purpose — they are not toy examples, but they are not so large that their modeling cannot be discussed in details. I believe this is more effective than mentioning some very large case studies, where the students may be more surprised/impressed but they may not understand the intricacies of modeling a real-life protocol. Also, note that when we discuss the SMV model checker and case studies, the students have not yet been introduced to temporal logics. Hence, the properties being verified in the case studies are mentioned informally at this stage and they are formalized in subsequent lectures. The detailed flow of the course is available (in the form of a **Lesson Plan**) from

`http://www.comp.nus.edu.sg/~abhik/CS4271/lesson-plan.html`

Unfamiliarity (with temporal logics & reactive systems) breeds contempt From my experience, students are often uncomfortable with one of the following.

- connection between program behaviors and transition systems,
- understanding reactive systems which have execution traces of infinite length
- interpreting temporal logic formulae over infinite execution traces.

The first hurdle is relatively easy to overcome. A refresher revision hour on operational semantics might help in this regard. However, since the students are typically familiar with transformational systems it takes them substantial time to make the conceptual switch to systems with execution traces of infinite length. This can be aided by presenting (successively more complex) example transition systems in class and telling the students to list out the infinite execution traces of the given transition system. Getting familiar with reactive systems (and infinite length execution traces) is often the primary hurdle in the minds of students. Once this barrier is overcome, they can (relatively) easily adapt to the concept of Linear-time temporal logic (LTL) and its operators. Branching-time temporal logics are then covered by building on Linear-time logics.

Finally, keep it project-based The final point that I want to discuss here is a lesson that was learnt the hard way. In retrospect, it is probably an obvious lesson but it was not obvious (to me) when I started teaching the course. To give the students hands-on experience with the model checking tools, I had the option of designing a series of assignments or allowing them to choose term-projects. From a pragmatic point of view, managing an assignment-based course is easier (for grading and other purposes). I ran the course in two successive years in two different modes (project based and assignment-based). The student response was overwhelmingly in favor of the project-based version. In retrospect, this was so for more reasons than one.

- A term project allows the student some choice and encourages some independent exploration for fixing the project as well as during modeling/validation.
- A term project gradually builds on itself during the entire semester and is more substantial. This way the students can see the benefit of using model checking on some substantial-sized examples.

I feel that engaging the students in a medium-sized term project might be the best way to convince them of the applicability of formal techniques. However, if the entire class does the same term-project it becomes a bit like an extended assignment depriving the students of a sense of independent exploration. For this reason, it might be important to allow students (individually or in groups of 2-3) choose different term projects even if the module administration becomes difficult.

An initial list of possible project ideas that I gave out to students in my course is available from

`http://www.comp.nus.edu.sg/~abhik/CS4271/proj-ideas.html`

Needless to say, students did projects outside this list as well. We should note that for a course based on independent term projects there are several adminstration issues involved such as counseling the students on deciding their project (particularly this needs to be done at the beginning of the semester when the students are not yet familiar with formal tools/techniques). If the course is offered multiple times, there is the additional issue of modifying/upgrading the list of project ideas in subsequent years.

One should emphasize here that *prior to actually doing their term projects, the students go through substantial experience in modeling and analysis of several case studies*, particularly when I introduce the SMV tool. These include

- medium-sized examples which are fleshed out in full details for the students to grasp intricacies of system modeling (such as the examples in [4]), and
- larger scale real-life protocol verification examples (such as lessons learnt from model checking the AMBA system-on-chip bus protocol running on ARM processors [3]) which lets the students appreciate the value of modeling and model checking.

4 Discussion

The lessons mentioned in this paper should be (at least partially) applicable to various formal methods courses — even those not covering model checking. The generic versions of these lessons are as follows.

- Do *not* rely on historical incidents to motivate formal methods.
- Emphasize system modeling (from requirements) rather than focusing only on verification techniques.
- Introduce verification tools prior to techniques as far as practicable.
- Allow students freedom in doing term projects (rather than assignments or fixed projects) even if module administration becomes difficult.

I sincerely hope that these general issues (which I learnt gradually over a period of five years) and the course materials (which I have made available through the Internet) will be useful to fellow formal methods educators in other universities and institutes.

References

1. John C. Knight. Position statement in Panel “Formal Methods: Too little or too much?”. In *ACM Intl. Conf. on Software Engineering (ICSE)*, 2006.
2. Peter J. Denning and others. A debate on teaching Computing Science. Essays in response to Edsger W. Dijkstra’s lecture “On the cruelty of really teaching Computer Science”. *Communications of the ACM*, 32(12), pages 1397-1414, 1989.
3. Abhik Roychoudhury, Tulika Mitra and S.R. Karri. Using formal techniques to debug the AMBA system-on-chip bus protocol. *Design Automation and Test in Europe (DATE) 2003.*
4. J.M. Wing and M. Vaziri-Farhani. A case study in model checking software systems. *Science of Computer Programming, 28, 1997.*