11-2006

Alloy, Software Engineering, and Undergraduate Education

Michael J. Lutz
Rochester Institute of Technology

Alloy, Software Engineering, and Undergraduate Education

Michael J. Lutz
Department of Software Engineering
Rochester Institute of Technology

Abstract

RIT's undergraduate software engineering program has a strong emphasis on design, including formal mathematical modeling. However students (and professional software engineers) are skeptical about the use of mathematical models in their day-to-day work. Alloy has proven to be successful in addressing some of this skepticism, but further work is needed to make formal modeling a norm in software development.

Introduction: The RIT Context

In 1996, the Rochester Institute of Technology (RIT) launched the first baccalaureate software engineering program in the United States [1]. Doing so was in the best RIT tradition of offering innovative career-oriented programs in a wide variety of professional disciplines. Our goal was to educate a new type of engineer who could design, develop, and deliver software that was comparable in quality to the products of other engineering disciplines.

The program's foundations are in computer science, mathematics, and natural science. Building on this, students are exposed (via coursework and co-operative education) to key process and product quality issues across the product life cycle. That being said, our program's emphasis is on design, including design synthesis and analysis, modeling, patterns, and software quality attributes. The focus of this paper is on mathematical modeling incorporated in our required formal methods course.

(Software) Engineering and Mathematics

A confession: I am not an engineer by training (though I've played one in industry). My undergraduate degree is in mathematics, and my graduate work was in computer science. Thus my discussion of mathematics and engineering below is based not on formal education, but on observation of practicing engineers in industry and the use of mathematics in engineering curricula (two of my children *are* engineers). For what it's worth, a former engineering dean at RIT told me I think like an engineer; I took that as a compliment.

The first thing to note is that engineers are pragmatic, using any tools that advance their understanding of a problem or help them assess the consequences of proposed designs [2]. Mathematics, of course, while being a very useful tool, is not sufficient – if it were, universities could reclaim a lot of lab space from their engineering schools. Still, much traditional engineering practice involves the use of mathematics, specifically continuous mathematics.

As a general rule, engineers are less interested in proofs of mathematical results than in the application of those results to engineering problems. Stated another way, engineers are intelligent, informed users of mathematics, but they are rarely mathematicians. If formal methods are to have the same effect on software development that continuous mathematics has on traditional engineering, it is imperative that they provide equivalent applicability to practical software problems.

The reluctance of the software industry to adopt more mathematical approaches is due less to math phobia than engineering cost/benefit analysis. Analyzing model properties in languages such as Z and VDM involves manipulating logical formulae, which in turn necessitates some knowledge of axioms, theorems, rules of inference, and proof techniques. The very generality of such systems means tools are either simple example checkers or complex theorem provers requiring significant mathematical maturity on the part of users. Neither of these approaches is appealing to engineers (or, in the RIT environment, to student engineers-in-training).

Fortunately, things are changing. Model-checking has proven its value in analyzing concurrent and distributed systems. At RIT we have successfully incorporated model checking tools [3] in our design courses with little pushback from students – they can see the applicability for themselves. Now, with Alloy [4], we have a promising tool for modeling and analyzing software entities, structures, and their transformations.

Alloy in Undergraduate Education

In the RIT context, Alloy addresses many of the problems we had with student resistance to Z, VDM, and similar formalisms.

The prerequisite discrete mathematics courses provide the necessary background in logic and set theory, but with Alloy students need not resort to proofs from first principles in order to perform useful analysis. Alloy's first-order system means some things cannot be modeled, and its use of SAT algorithms restricts the generality of some results. But used with a modicum of engineering judgment, the notation is sufficient to create and analyze many systems. This is in the best tradition of pragmatic engineering: some information is better than none.

In addition, the concrete syntax, being so similar to C++ and Java, helps overcome students' initial anxiety to the use of mathematics. In my experience, it's a mistake to dismiss the importance of familiar syntax, especially at the undergraduate level.

In the formal methods course, I use several strategies to help students become competent in developing small models. My primary approach is to alternate between lecturing on concepts and exploring their consequences via the Alloy Analyzer. Students access the models on their lab computers, and can mimic my explorations or take side excursions on their own. "What if" and "how would you express this" questions require pairs of students to extend or modify the model on their own. Out of class exercises on related but distinct problems, supported by asynchronous discussions on our course management system, serve to expand student experiences beyond what is possible in class. Finally, the course requires a major team-based modeling project that pushes students to explore issues in scaling Alloy to larger problems.

There are a few areas where further work is required. These have little to do with the Alloy notation or the analyzer tool, but reflect the lack of material on effective use of Alloy:

1. Better documentation is needed on the analyzer tool itself, most particularly on how to use the visualization system. Students spend too much learning how to use color, shapes, and projection in ways that illuminate rather than obscure the analyzer's output.
2. Real case studies – those that show the value of Alloy in industrial software design – would greatly aid in demonstrating the value of Alloy to skeptical students.
3. Students have few problems using Alloy to describe a static model defining legitimate structural states of a system, and in this arena **`fact`**s are an invaluable aid to capturing state invariants. Things get much trickier, however, when moving from such a static model to a dynamic one, where "operation" predicates define state transformations.
    a. First, one must decide how to model time. The two most common approaches are a "primed" notation reminiscent of Z, and an explicit **`Time`** signature whose atoms index time-variant relations. What's missing are heuristics to guide the selection of an appropriate approach.
    b. Second, when **`fact`**s are carried over to the dynamic model (after adjusting as appropriate for time), students develop a false sense of security. Students overlook the need for preconditions to guarantee state closure, and the analyzer is of no help in ferreting out these omissions. The reason is that facts are too strong – because they must hold in the final state of an operation, the analyzer will never produce counterexamples to closure assertions even when required preconditions are missing.
4. These observations illuminate the need for heuristics – patterns or refactoring procedures – to transform static facts into equivalent dynamic predicates and assertions that will help detect missing preconditions. What's required is well-known in the formal modeling community; what's needed is a practitioner friendly approach to performing these model transformations.
5. Finally, guidelines on going from designs to code would help persuade students of Alloy's value. Part of our success in using [3] is due to the connection between formal models and corresponding Java classes and methods.

Summary

Alloy has already proven successful in our formal methods class, with students able to define and analyze interesting systems. The challenge is to build on this success so that students use Alloy to explore design issues in subsequent courses.

References

1. J. Fernando Naveda & Michael Lutz. "The Road Less Traveled: A Baccalaureate Degree in Software Engineering." *1997 Conference on Software Engineering Education and Training.* Virginia Beach, VA. April, 1997.
2. Billy Vaughn Koen. *Definition of the Engineering Method.* ASEE Press, 1985.
3. Jeff Magee & Jeff Kramer. *Concurrency: State Models & Java Programs.* Wiley & Sons, 1999.
4. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.