2006

Experiences with alloy in undergraduate formal methods

Michael Lutz

Experiences with Alloy in Undergraduate Formal Methods

Introduction

At the core of all engineering endeavors is the modeling of proposed system designs and the use of these models to determine system properties. While some models are physical, the vast majority use mathematics to both describe and analyze the consequences of design decisions. In the case of traditional engineering disciplines, most models are based on continuous mathematics, e.g., calculus and differential equations. The situation is quite different in software engineering, however, where the applicable models are more likely to be drawn from discrete mathematics, logic, and set theory. The term of art for such modeling approaches is *formal methods*.

One complaint about formal methods, voiced by both practitioners and students alike, is the lack of applicability to "real" problems. While some of these objections are undoubtedly based on an unwillingness to learn the relevant mathematics, this does not mean they can be dismissed out-of-hand. To be useful in practice, a modeling method must provide engineers with information that more than compensates for the cost of learning the technique and creating its models. Model checking is one formal method that has proven its value as a tool for describing and analyzing concurrency effects[1,2]. Alloy[3,4,5], a modeling tool created by Daniel Jackson's research group at MIT, provides similar value when modeling and analyzing the structural and behavioral consequences of software design decisions.

This paper reports on the value Alloy has brought to the undergraduate formal methods course within the software engineering program at RIT. The next section introduces Alloy by way of a well-known example problem, the birthday book[6]. This is followed by a section discussing the advantages Alloy for teaching undergraduates, especially as compared to traditional methods such as VDM[7] and Z[8]. The final section discusses some areas where Alloy's support for instruction needs improvement.

Birthday Book Example

Consider a system for maintaining a birthday book (that is, a book that lists birthdays for some set of persons). In Alloy, we would start by defining the necessary signatures: Named sets of indivisible, immutable, atomic objects and the relations that hold among these sets.

```
sig Person{}
sig Date{}
sig BirthdayBook {
    known :  set Person,       // those persons known in this book
    dates :   Person -> Date  // the birth day for each known person
}
```

Here we have defined three signatures, Person, Date, and BirthdayBook, along with two relations, known and dates. The signature declarations implicitly state that the three underlying sets of atoms partition the universe of all atoms (that is, the three sets are pair-wise disjoint and their union is the universe). At the modeling level, however, all that exists are relations – Person,

Date, and BirthdayBook are really unary relations, containing a 1-tuple for each of the elements in the underlying set.

The declaration of a relation within a signature means the relation consists of tuples whose first element is an atom from the signature's underlying set. Thus known is a binary relation mapping each book to those persons recorded in the book, and dates is a ternary relation, whose tuples consist of a book, a person known in that book, and that person's birthday. Or at least that's what we intend: without further constraints there is nothing to ensure the persons known in a book are exactly those whose dates are recorded.

To create the needed constraints we add "facts" – predicates that must hold in any legal state of the system. In our case, we can state our constraint in one fact:

```
fact {
        all b : BirthdayBook | b.known = b.dates.Date
}
```

This fact says "the persons known in book b are exactly those who have a birthday recorded in b" – but how? Consider first the declaration b : BirthdayBook. Since everything in Alloy is a relation, then b must be a relation, and it is – it's a singleton subrelation of BirthdayBook, which is itself a unary relation. This is as close as we can get to a set element in Alloy – a singleton, unary relation. As Alloy uses first-order relational logic, there is no danger of tripping over Russell's paradox, so we can use "element" and "singleton set" interchangeably.

The expression b.known is a *relational join* between the (singleton, unary) relation b and the binary relation known. In Alloy, relational join matches the last column of every tuple from the left relation to the first column of every tuple in the right relation; on a match, the tuples are concatenated and the two matching columns are dropped. In this case, we get the unary relation (set of) Persons who are listed in book b.

The expression b.dates.Date is similar – first we join unary relation b to ternary relation dates, resulting in a binary relation between Persons and Dates. This is then joined (on the right) to the unary relation Date; the effect is to simply "strip off" the Date column from the binary relation, leaving a unary relation (set of) Persons. The equality simply states that the two sets of Persons defined by the joins are identical – just what we want.

So far Alloy seems to be just another formal method: similar to C in syntax, and with its own peculiarities (e.g., everything is a relation), but nothing new. What makes Alloy stand out, however, is its support for exploring the consequences of a design. First of all, we can create predicates describing the properties we wish to see in a solution; the properties become, in effect, temporary constraints in addition to the facts. What is more, we can "run" a predicate and have the tool produce a conforming solution (or tell us that it cannot).

```
pred show() {
    some known
}
run show for 3
```

The show() predicate above has a body that says there must be **some** (one or more) tuples in the known relation. That is, there must be at least one BirthdayBook that knows of at least one Person (and, given our **fact**, this Person has a birthday recorded in the book).

The **run** command instructs the Alloy tool to search for a solution which has at most three elements in each of the declared signatures. Alloy compiles the declarations, facts, and predicate into a Boolean expression that is then sent to a Boolean constraint satisfier (SAT); if the satisfier finds a solution, Alloy displays it in one of several formats. Figure 1 gives the graphical version of one possible solution for our model:

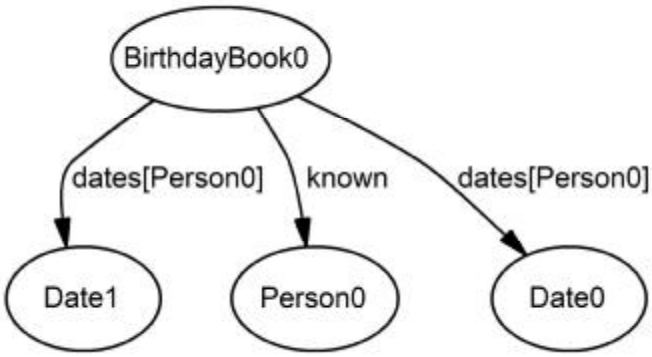

Figure 1 – Solution to **run** show for 3

There's something peculiar about this solution, however – Person0 has two distinct birthdays recorded in the BirthdayBook. Assuming we don't want this, we can add another fact constraining the solution so that no person can have more than one birthday recorded in a given book:

```
fact {
        all b : BirthdayBook, p : Person | p in b.known <=> one p.(b.dates)
}
```

The expression p **in** b.known says relation p is a subrelation of b.known. Given that p is an element (singleton, unary relation) and b.known is a set (unary relation), this is equivalent to the traditional "element of" predicate from set theory. In general, however, both operands of **in** will be relations, in which case we have a subrelation (or subset) test. Indeed, the keyword **in** was chosen for its ambiguity, as it can represent either "element of" or "subset of," depending on the left operand involved.

From the previous discussion, we know that b.dates is a binary relation between Persons and Dates; thus p.(b.dates) is a set (unary relation) consisting of those Dates associated with Person p. The expression **one** p.(b.dates) states this set has exactly one member. Thus the whole predicate, p **in** known <=> **one** p.(b.dates) says a Person is known if and only if the Person has exactly one Date recorded for their birthday. In the context of the universal quantifier, this states that any Person known in any BirthdayBook will have exactly one birthday in the book.

After adding this fact, running the show() predicate produces the solution in Figure 2.

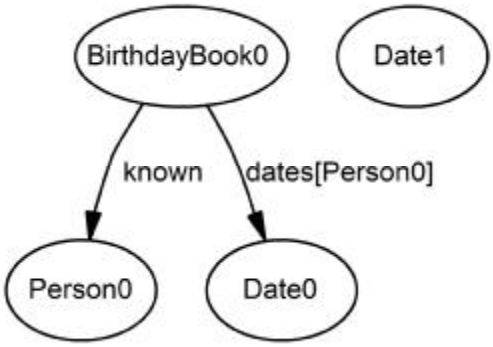

Figure 2 – **run** show() with augmented facts

Alloy has many features and facilities beyond those shown in this simple example, including

- Functions that extract information from the solution state,
- Checkable assertions (i.e., universal claims that follow from the declarations and the facts), and
- State changing operations modeled by predicates relating the pre and post states.

The goal of this section was simply to give a flavor of Alloy; more information can be found in the Alloy documentation[3,4,5].

Pedagogical Advantages

Alloy's primary advantage over traditional methods such as VDM and Z is that it supports analysis and exploration without the need to become a mathematician. Tools for these traditional methods come in two basic forms: simple syntax checkers and complex proof assistants, neither of which is appropriate for undergraduate education. Syntax checkers do little to help students understand the consequences – often quite subtle – of what they design. That is, while the syntax checker can ensure the model is meaningful, it cannot help determine whether that meaning is what is intended.

The only way out of this problem is to do formal proofs of claims made in the model. When done by hand, such proofs are tedious and error prone. When done via proof assistant tools, students soon see the necessity of deep knowledge of both proof theory and the idiosyncrasies of the specific tool they are using. The tradeoff is obvious: Either hope the models says what you want to say, or become expert in mathematics at a level not required of any other engineering discipline[9,10]. In light of this, it is hard to refute student perceptions that formal methods provide no improvement over informal and *ad hoc* methods for designing, validating, and verifying software.

Alloy, on the other hand, requires one to be knowledgeable of discrete mathematics but not an expert mathematician. One need only understand what Alloy's constructs mean and be able to interpret its graphical or textual output in order to use the tool effectively for exploring the consequences of design decisions. The dirty work of finding solutions (or looking for counterexamples to universal claims) is left to the sophisticated SAT systems on which Alloy is

built. One must make compromises, of course – Alloy cannot express higher order constructs, and it is limited to searching finite state spaces – but in practice these compromises are rarely problematical. If a counterexample to a claim cannot be found in a relatively small state space, say 3-5 atoms per signature, then it is highly unlikely (but not impossible) that a counterexample exists in an infinite universe.

There is another advantage that should not be dismissed: Alloy is interactive, allowing users to iterate among design, specification and analysis. This makes Alloy much more attractive to students familiar with interactive, integrated development environments. One can easily explore large state spaces from the keyboard, making design verification much more comprehensive than with unit testing. This interactivity is a boon to instruction as well; I often build a model in class, asking students to fill in key facts, predicates, and assertions, and then I use the tool to see if their solutions are correct. Alloy also makes it easier to take side-tracks that either interest students or reinforce material they find confusing. There is no need to anticipate every possible problem – an impossible task in any event – rather, one can let the nature of student questions and answers direct the creation of a model.

Instructional Needs

Despite its manifest advantages, Alloy is not without problems. Fortunately, none of these involve the tool *per se*, but rather the pedagogical framework needed for effective undergraduate instruction.

First and foremost, a solid undergraduate text based on Alloy is a critical need. It wasn't until Kramer and Magee's text[1] on concurrency in Java that research on safety and liveness in the context of interacting state machines was brought to a level appropriate for undergraduates. Jackson's new book on Alloy[5] is a step in the right direction, but the presentation is a bit too terse for a text. A book that presents Alloy with many examples and periodic review exercises would be a great pedagogical aid.

In addition, a set of real (or at least realistic) case studies is needed, with the studies presented at a level accessible to undergraduates. In part this would serve to provide a rich set of examples that could be emulated; in part it would be useful propaganda to help persuade students that formal methods are worth consideration.

Finally, we need the equivalent of "design patterns" for Alloy. That is, we need prepackaged templates showing proven modeling approaches to common design problems. Such a pattern library would help students become proficient that much sooner, and allow instructors to assign design problems that bring to light the value of formal modeling.

Conclusion

All in all, Alloy is the most satisfying tool I've used in the 15+ years I have been teaching formal methods. My hope is this paper at least sparks some interest in others who teach this material, and that they will consider adopting Alloy or a similar tool. After all, if we are to place software engineering on a firm mathematical foundation, we must do so in a way that makes this useful to practicing engineers. To my mind, Alloy is a step in this direction.

References

1. Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
2. Michael Lutz and James Vallino. “Concurrent System Design: Applied Mathematics & Modeling in Software Engineering Education.” *2005 ASEE Annual Conference and Exposition*, June, 2005.
3. Daniel Jackson. “Alloy: A lightweight object modelling notation.” *ACM Transactions on Software Engineering and Methodology*, November, 2002.
4. *The Alloy Analyzer*. http://alloy.mit.edu/
5. Daniel Jackson. S*oftware Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
6. J. M. Spivey. *The Z Notation*. Prentice-Hall, 1992.
7. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, 1998.
8. Jonathan Jacky. *The Way of Z*. Cambridge University Press, 1997.
9. David Parnas. "Mathematical Methods: What We Need And Don't Need", *IEEE Computer*, April, 1996.
10. Michael Lutz. “Formal Methods and the Engineering Paradigm.” *SEI Conference on Software Engineering Education*, October, 1992.