Discrete Mathematics for Computing Students: A Programming Oriented Approach with Alloy

Leo C. Ureel II
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
Email: ureel@mtu.edu

Charles Wallace
Department of Computer Science
Michigan Technological University
Houghton, Michigan 49931-1295
Email: wallace@mtu.edu

***Abstract*—Students in computing disciplines need a strong basis in the fundamentals of discrete mathematics. Traditional "offline" approaches to teaching this material provide limited opportunities for the kind of interactive learning that students experience in their programming assignments. We have been using the Alloy language and analysis tool to teach concepts in discrete structures and logic in an exploratory, programming-intensive way. We report on our efforts to build scaffolded Alloy exercises for newcomers to discrete mathematics, and we report on some initial findings based on our experiences with students.**

I. Discrete Mathematics in Computing Education

The field of discrete mathematics provides a vital analytical toolset for computing disciplines. Students in computer science, software engineering, and related programs need a foundation in topics like logic, proof, graph theory, and combinatorics. In many undergraduate computer science programs, these tools are presented in a specialized departmental course. This arrangement is largely one of convenience, putting the choices of course content directly into the hands of computing faculty. A course tailored to computing can effectively distill the content of multiple mathematics courses into a single course. Apart from convenience, is there any meaningful way in which a discrete mathematics course taught in a computer science department differs, or should differ, from one taught in a mathematics department?

We argue that bringing an interactive programming spirit to the computer science discrete mathematics course is valuable for two reasons. First, problem solving through the medium of the machine is the essence of computer science. Computer science majors enjoy engaging with computers, and any course in our curriculum should take advantage of this intrinsic motivation. Second, a programming approach to discrete mathematics affords active learning. Students in our course are building a conceptual model of the elements of discrete mathematics; with an appropriate tool that supplies feedback on the soundness of their model, they gain ownership of their own learning.

II. The Alloy Language and Analyzer

In our discrete mathematics course for computer science students, we use the modeling language Alloy and the Alloy Analyzer [1] to provide a hands-on experience with automated feedback. In the spirit of the name "Alloy", the logic at the heart of this tool combines the quantifiers of predicate logic with the operators of the relational calculus to provide a highly expressive specification language [9]. An Alloy program asserts a set of constraints on mathematical structures over a program-specified signature. The Alloy Analyzer searches within programmer-specified bounds for models satisfying an Alloy program, through a SAT-based constraint solver. If a satisfying model is found, the analyzer displays it graphically; otherwise, it identifies a minimal unsatisfiable core of the program.

Alloy has been used in a wide array of application areas, including security, computer architecture, program verification, and model driven development. It has also been used extensively as a teaching tool, but reports of its use focus solely on more advanced topics in formal methods or software modeling [2], [5], [15]. Our use of Alloy in an introductory discrete mathematics course appears to be novel. It is worth noting there is previous successful work implementing functional programming in discrete math courses [4], [16], [7]. Additionally, Barwise and Etchemendy's long-standing learning tool *Tarski's World*, with similar aims to ours but without the benefits (and possible pitfalls) of Alloy's greater expressivity.

An obvious and understandable concern is that a high-powered tool like Alloy in the hands of students at an early stage of development can be dangerous. The declarative nature of Alloy constitutes a conceptual departure from their still nascent conception of what programming is. Alloy does not look like the imperative Algol-style languages they have been exposed to, and it can be easy for them to despair at learning a new language and a new style of computing. It is crucial to provide scaffolding for early concept formation and then gradually remove this scaffolding.

III. Scaffolding Alloy for Beginners

We find that the feedback provided by the Alloy Analyzer eliminates common misconceptions among students. Compared to a traditional approach where students simply submit written answers to homework problems, students working on Alloy problems get immediate critique of the well-formedness and satisfiability of their responses. A common student response as they wrestle with programming exercises is that they

"don't understand Alloy". The root of the problem, however, is rarely the Alloy language, since its structure is so close to that of standard quantified formulas and relational expressions. Working through such student problems usually reveals that they have a misunderstanding of the structure of standard mathematical notation. With a traditional pencil-and-paper exercise, students can remain in a false state of confidence, and deficiencies in understanding are not exposed until after the exercise is submitted for grading.

Alloy's expressive flexibility allows us to use it throughout the course. Constraints may be expressed in Alloy as first-order formulas, as relational expressions, or a combination of both. This flexibility gives us an opportunity as educators to illustrate a conceptual continuum in the course material. Starting with sets and relations, we state Alloy constraints as relational expressions. Later, when we cover first-order logic, we introduce this other style of expression in Alloy and show how it compares to the relational style. Finally, we address problems in graph theory using both styles, and taking advantage of the Alloy Analyzer's visualization functionality.

Students can engage with Alloy at various levels of sophistication. For our early discrete mathematics course, we do not expect students to build sophisticated programs from scratch. Rather, we design exercises carefully to take students from *observers* of program behavior to *tweakers* of search parameters to *builders* of more substantial constraints.

A. Observing

In the observer model, students run a complete Alloy program and use the Analyzer's search powers to find satisfying models that they must then interpret. We use this model when introducing the concept of equivalence relation. In a simple Alloy program, we define a binary `friend` relation over the signature `Kitteh`:

```
sig Kitteh { friend: set Kitteh }
```

Using predefined library predicates `reflexive`, `symmetric`, and `transitive`, we define a predicate that tests whether the `friend` relation has the three essential properties of an equivalence relation:

```
pred kittehWorld {
  reflexive[friend, Kitteh]
  symmetric[friend]
  transitive[friend]
}
```

Finally, we specify a search for satisfying models — those containing exactly 8 `Kittehs`:

```
run kittehWorld for exactly 8 Kitteh
```

The Alloy Analyzer finds all satisfying models (modulo isomorphism) and presents them one by one in a simple visual manner. The concept of equivalence class emerges naturally in a visual way (Figure 1). By looking at successive models, including non-obvious corner cases (*e.g.,* the identity relation), students are able to discover the concept of equivalence relation on their own.

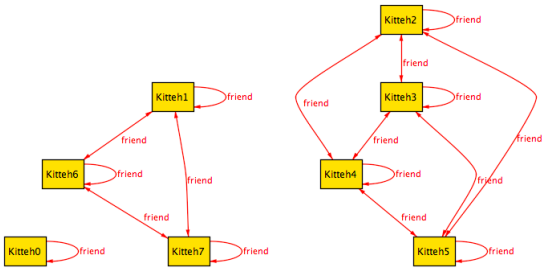

Fig. 1. Alloy visualization of equivalence relation.

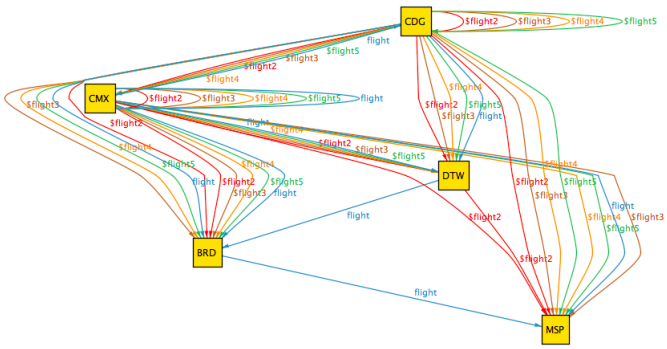

Fig. 2. Alloy visualization of complex airport model.

B. Tweaking

In the tweaker model, students play with highly constrained parameters of a complete Alloy program. In one example, a binary "flight" relation `flt` over `Airports` is defined, and the model is constrained to a set of five named airports: `CMX`, `DTW`, `MSP`, `BRD`, `CDG` (the keyword `one` specifying that exactly one of each airport appears in a satisfying model):

```
abstract sig Airport {flt: set Airport}
one sig CMX, DTW, MSP, BRD, CDG
  extends Airport {}
```

The tweaker exercise asks whether a model exists with this counterintuitive property: "anywhere you can get to in five flights, you can get to in two flights; but there are places you can get to in two flights that you cant get to in five flights". Once they understand the composition operator (`.`), students are prepared to write a simple predicate to search for this model:

```
pred show {
  flt.flt.flt.flt.flt in flt.flt
  flt.flt not in flt.flt.flt.flt.flt
}
```

The initial models that Alloy finds can be quite complex, with many pairs in the relation `flt` (Fig. 2). As a further challenge, students are asked to find a minimal satisfying model (in terms of the size of `flt`). This is done simply by

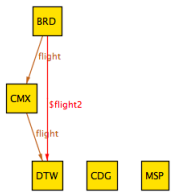

Fig. 3. Alloy visualization of minimal airport model.

adding a constraint `#flight = 2` (after some trial and error to find the correct integer value). Students are asked to show and interpret the visualization of a satisfying model (Fig. 3).

C. Building

In the builder model, students are charged with completing portions of an Alloy program. One such exercise asks students to create a model for execution traces of a parallel program (with two processors A and B). A "happens-before" relation `hb` is defined over operations, which are classified as A-operations, B-operations, or S(ync)-operations:

```
abstract sig Op { hb: set Op }
sig AOp, BOp, SOp extends Operation { }
```

The constraints on execution traces are declared in English, and the students must translate these formally into Alloy code. A solution is given below:

```
pred parallel {
-- hb is irreflexive and antitransitive.
  irreflexive [hb]
  antitransitive [hb]
-- AOps and BOps are unordered.
  no hb & AOp->BOp
  no hb & BOp->AOp
-- Reflexive, transitive closure of hb,
-- restricted to AOps and SOps, is total.
  totalOrder [*hb, AOp+SOp]
-- Reflexive, transitive closure of hb,
-- restricted to Bops and Sops, is total.
  totalOrder [*hb, BOp+SOp]
}
```

In building this solution, students use standard intersection (`&`) and union (`+`) operators, as well as more sophisticated operators like Cartesian product (`->`) and reflexive, transitive closure (`*`). They can continually check their solutions by executing the code and examining the satisfying models (Fig. 4.)

Later in the course, when predicate logic is covered, students may use quantification to express constraints, as in these exercises defining common graph properties (here as predicates defined over an edge relation `e` and a set of vertices `s`):

```
pred clique[e: V->V, s: set V] {
  all u,v: s | u->v in e
}
pred vertexCover[e: V->V, s: set V] {
  all u,v: V | u->v in e implies
    (u in s or v in s)
}
```

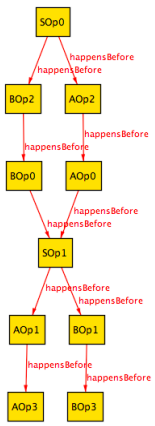

Fig. 4. Alloy visualization of parallel execution.

Students can test their preliminary solutions and determine whether the generated graphs have the desired properties.

IV. Preliminary Results

As part of a larger effort to build a cognitive apprenticeship model into our introductory curriculum [10], we seek not only improvements in student performance but also insights into what students do and how they utilize automated feedback as they work through problems in Alloy. In this paper, our findings focus on preliminary qualitative studies of student learning, based on one-on-one problem solving sessions.

During the Spring 2015 term, Alloy was introduced to our second year discrete mathematics course. The students in the course come primarily from computing majors (Computer Science, Software Engineering, and Computer Engineering). For many students, this is their first time they are studying the mathematical underpinnings of computation. Students participated in whole class instruction and discussions concerning use of Alloy in the context of logic-based, graph theoretic assignments. Two sets of semi-structured clinical interviews were then conducted. Eight students were given an interview task to build a predicate that tests for Hamiltonian paths in a directed graph

Students were given the outline of the Alloy program and were asked to complete it. The task required them to apply their knowledge of graph theory and predicate logic from the course to complete the formal Alloy description. Students could execute their descriptions at any time to generate graphs satisfying their description. A portion of the scaffolded code appears below:

```
pred hamPath(p: Node->Node,
  start: Node, end: Node) {
-- three things have to be true
-- for p to be a Hamiltonian path
-- from start to end:
```

```
-- (1) every node (except one)
-- has a successor in the path
  all n: Node - end | one n': Node |
    n -> n' in p
-- (2) every node (except one)
-- has a predecessor in the path
  -- (your code here)
-- (3) no node appears more than once
-- in the path
  -- (your code here)
```

Students were allowed to ask questions and encouraged to "think out loud" while filling in the missing parts of the Hamiltonian Path description. We transcribed and coded their conversations. A selection of emergent codes appears below:

- Explanation: Students often explained what they were doing and used those explanations as feedback to help problem solve.
- Exploration: Students tried changes to the code and executed it just to see what would happen.
- Code Interference: At times, previous experience with imperative programming interfered with encoding formal mathematics in Alloy.
- Translation: Students struggled with reading Alloy code in English, or would be able to articulate a constraint in English but couldn't put it into Alloy.
- Self Doubt: Students expressed doubt about their own code.
- Repair: Students finding themselves at a dead end after execution would turn back to the code and iterate between writing and executing code.
- Interpreting exeuctions: Students used the executable features of Alloy to generate instances and solve problems.
- Breakthroughs: Students struggling with a building challenge would experience "a-ha" moments.

We noted an interesting relationship between Repair and Explanation: in several instances, students entered a "virtuous cycle" where they would write code, execute it and encounter a syntax error or unexpected result, then backtrack and talk through their intended result to get to a more refined Alloy solution. An example is given below.

> Student: For all n ...the you would want to use one p in ...probably have to use the divider `...for all n node one n in n.^p`. [Instructor: Run it and see what happens.] [Student runs program resulting in error.] Its a set...do that no...[Instructor: Tell me in English what you want to say.] For every node in this set or for all sets...you cant have more than one n in this set.
>
> Instructor: Right. For every n I can look at the nodes reachable from n...What has to be true of all those nodes that are reachable from n?
>
> Student: They cant be duplicate.
>
> Instructor: Exactly. So if I start with n, what has to be true about every thing after n?
>
> Student: They have to be unique values.
>
> Instructor: [Draws graph.] If I have a graph like this and I look at n and take the transitive closure, what s going to be in the transitive closure?
>
> Student: n will be in transitive closure. [Instructor: And we dont want that to happen right?] Yeah. So n not n ...[types] `one n in n.^p`. I dont know, I want to have in. [Instructor: Read it out to me.] For all n node ...n cannot be in a list of what can be reached from n.
>
> Instructor: It sounds good to me. Lets run it. [Student runs program.] Looks like Hamiltonian path. But maybe we got lucky, lets try another. Hamiltonian Path?
>
> Student: Yes. [Keeps looking through the generated graphs. Finds a graph with just one node.] Is that a path?
>
> Instructor: Is a path of length zero a Hamiltonian Path?
>
> Student: No...Yes! So were done.

This cycle of coding and explaining, followed by execution and interpretation, seems highly promising for building competence in these discrete mathematics concepts among our students.

A post interview revealed an interesting instance of Code Interference, with the concept of iteration interfering with the more abstract notion of transitive closure:

> *Instructor:* It was interesting. I noticed you started going down a path that was really close, but not quite right. But eventually you caught on. How about where were there places where you had n a-ha moment.
>
> *Student:* I think in that third one, when you were drawing the graph and showed me the transitive closure, that really helped me figure out what I was supposed to do with the third one whereas before I was *thinking in terms of it looping through itself* [emphasis ours] while finding everything else but I think I already kind of had that.

V. Future Directions

Our experiences with Alloy in an introductory discrete mathematics setting show promise. Due to the substantial shift in programming paradigm that it represents, and the relative lack of supporting materials for early students, we are structuring our Alloy materials as lab exercises, where the instructor can provide the just-in-time feedback that we saw was part of the "virtuous cycle" of coding and explaining. Our exercises follow the structure of POGIL (Process Oriented Guided Inquiry Learning) [6], [11], [8], in which small groups of students run through a learning cycle of *exploration*, *concept invention* and *application*, discovering concepts as they go (in a way similar to our "Observer" example).

References

[1] Alloy Community. `http://alloy.mit.edu/alloy/index.html`

[2] Ball, T. and Zorn, B. (2015). Teach Foundational Language Principles. *Communications of the ACM* 58 (5), 30–31.
[3] Barwise, J. and Etchemendy, J. (2002). *Language, Proof and Logic.* CSLI Publications.
[4] Bolick, B., Gopalakrishnan, G., Jacobsen, C., & Tuttle, T. (2013) Toward Revamping Discrete Structures: Concurrency through Functional/Constraint Programming Integration. University of Utah, Formal Methods Research.
[5] Boyatt, R.C. and Sinclair, J.E. (2008). Experiences of Teaching a Lightweight Formal Method. Proceedings of Formal Methods in Computer Science Education.
[6] T. Eberlein *et al.* (2008) Pedagogies of engagement in science: a comparison of PBL, POGIL, and PLTL. *Biochemistry and Molecular Biology Education* 36: 262–273.
[7] Farahani, A. (2009). Use of Python in Teaching Discrete Mathematics. In American Society for Engineering Education. American Society for Engineering Education.
[8] H.H. Hu and T.D. Shepherd (2013). Using POGIL to help students learn to program. *ACM Transactions on Computing Education* 13(3).
[9] Jackson, D. (2012). *Software Abstractions: Logic, Language and Analysis (Revised Edition).* MIT Press.
[10] Kumar, S., Ureel, L., and Wallace, C. (2015). Agile Communicators: Cognitive Apprenticeship to Prepare Students for Communication-Intensive Software Development. Proceedings of AGILE'15.
[11] C. Kussmaul (2012). Process oriented guided inquiry learning (POGIL) for computer science. Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh NC, 373–378.
[12] Lin, C. C., & Zhang, M. (2003). The use of computer animation in teaching discrete structures course. Midwest Instruction and Computing Symposim (MICS).
[13] Marion, B., & Baldwin, D. (2007). On the implementation of a discrete mathematics course. Committee report, SIGCSE.
[14] Page, R. L. (2003). Software is discrete mathematics. ACM SIGPLAN Notices, 38(9), 79-86.
[15] Tarkan, S. and Sazawal, V. Chief Chefs of Z to Alloy: Using a Kitchen Example to Teach Alloy with Z. In *Teaching Formal Methods*, Lecture Notes in Computer Science 5846, 72–91.
[16] VanDrunen, Thomas (2011). The Case for Teaching Functional Programming in Discrete Math. Educators' and Trainers' Symposium at SPLASH (formerly OOPSLA) 2011, https://cs.wheaton.edu/ tvandrun/dmfp/case4dmfp.pdf