6.898 Advanced Topics in Software Design February 6, 2001

Lecture 1: Course Overview and Introduction to Alloy

Lecturer: Daniel Jackson *Scribe: Bill Thies*

1 Course Overview

1.1 Contacting the Lecturer

Please use the email address `dnj+6898@mit.edu` to contact Daniel, to help him organize mail for the course.

1.2 Expectations

The following tasks are expected to be completed by the student. They are described in more detail on the course webpage.

- Scribe one lecture – take notes and write them up in HTML. There might be enough people registered for the course that not everyone will have to scribe.
- Attend class and participate.
- Read assigned papers, and answer a few short questions about the papers before coming to class. These answers should be emailed to Daniel the day before the class meeting.
- Complete three projects:
    1. A modeling exercise. This will be challenging, but not open-ended.
    2. A programming exercise.
    3. A term project. The project can be of three types:
        (a) Do an open-ended research project. This can be done in a team or individually, and will result in a paper with an oral presentation.
        (b) Build something as a design project. This must be done individually, and requires only a written presentation.
        (c) Re-design an existing piece of software, as a design "clinic". This must be done individually, and requires only a written presentation.

In all, the course should require an average of about 6 hours/week of outside work, but the work will be sporadic since there are only a few assignments.

1.3 Collaboration Policy

It is fine to discuss solutions in a group, but all work that is handed in must be *written* individually.

1.4 Course Outline

This course will focus on advanced topics of software design. We will concentrate on how to express, analyze, and realize high-level design ideas, and will examine current techniques, languages, tools, and methods in the field. We will try to bring together perspectives from different communities, including:

1. The programming language community, especially those interested in functional programming.
2. The formal methods community.
3. The "extreme programming" community, which embraces open-source models with communal incremental refactoring of code.
4. The theoretical design community, *e.g.* those that are interested in decoupling theory. This area the lecturer is less familiar with, but is interested in exploring as part of the course.

Speaking of communities, Alan Donovan is organizing a programming languages reading group that meets from 4-5 p.m. on Tuesdays. Everyone is welcome to join.

1.4.1 The Lecturer's Biases

Daniel gave a disclaimer as to his own biases in software design. They include:

1. **He is a reductionist**. How can you capture the essence of software? What makes a design different from other designs? How can designs be most simple?
2. **He likes tools**. It's kind of strange to note that software designers are one of the only "computer scientists" who don't necessarily use computers in their work – currently, they just think about the design without much help of automated tools. This is especially true for the early stages of design. Some people are concerned that software design is a craft, and shouldn't be reduced to a science–a tool would be too rigid and restrictive. However, most people don't take this view about structured programming, even though it met with some resistance at first. Is it possible to factor out repeated human tasks and capture common structures in the design process, too?
3. Regarding formal methods, **he likes lightweight formalism**. Some people argue for reducing all of software design into a mathematical formula, but this has the unappealing aspects of 1) not really capturing everything in the design, and 2) not being practical. Instead, the use of formal methods should be risk-driven, motivated by "bang for the buck". For example, if a product needs to be highly reliable, then lightweight methods can be valuable for providing a precise design and a mechanical analysis of the program's properties.

1.4.2 Topics to be Covered

There are four main topics planned for the course, although they are flexible:

1. **Modeling Languages**, including the idioms and analyses that are characteristic of each. We will start with Alloy, the language being developed in Daniel's group. This will provide a clean notation to use for discussing design ideas in the remainder of the class. Also, we'll consider JML.
2. **Design Patterns**. There are two types of patterns that we will consider:

- *Patterns of Problems*. These include Martin Fowler's book on refactoring, and Michael Jackson's work on problem frames. The latter work takes the approach of classifying design problems into known categories before attempting solutions.
- *Patterns of Solutions*. This includes the classic Design Patterns book by the Gang of Four.

3. **Programming Language support** for software design. Examples include:

   (a) *Functors*, which are constructs in ML for building module generators and sharing constraints.

   (b) *Type classes*, which are provided in Haskell as a decoupling mechanism for something related to overloading.

   (c) *Units*, a theoretical idea that is being developed by Matthias Felleisen (who is teaching a related type-theory course at Northeastern this spring). Units are being incorporated into the Java pre-processor.

   (d) *Open Classes*, which allows one to dynamically add methods to a class.

   (e) *Aspect-Oriented Programming*, which is a technology for separation of concerns in software development.

4. **Theoretical underpinnings**, including Nam Suh's work on software design, as well as Design Rules by Thomas Baldwin and Kim Clark, which views design from the perspective of economic theory.

2 Introduction to Alloy

We'll discuss the Alloy language at the beginning of this class, both as a foundation for discussing design concepts later, and as a means for understanding a modeling language. Alloy has some operators that are generalized versions of normal operators, which makes it very succinct and expressive.

2.1 Overview

The high-level ideas and properties of Alloy are as follows:

1. **Declarative Modeling**. Alloy is for recognizing that a program has a given property. It answers the modeler's question "how can I recognize that property P has happened?" rather than the engineer's question of "how can I make property P happen?" Also, Alloy code can be executed, *e.g.* to demonstrate that a sorting algorithm works.

2. **Expresses Structure**. Everything in Alloy is done with relations – there are no sets or other mathematical entities in the language. This keeps things simple.

3. **First-Order**. All the logical constraints in Alloy are first-order, which allows them to be analyzed automatically with a tool. This is in contrast to some higher-order logics that are harder to analyze.

4. **Slick Syntax**. The syntax is designed to be compact and completely textual, so that you don't need LaTeX to get it onto paper.

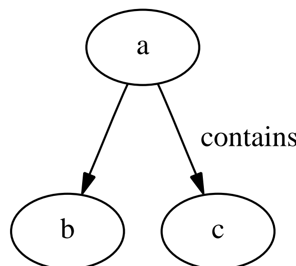
The following sections describe the main concepts of Alloy in more detail.

2.2 Atoms

An *atom* is the basic modeling object in Alloy. An atom is:

1. **Indivisible**. It has no parts.
2. **Immutable**. It doesn't change over time.
3. **Uninterpreted**. That is, there is no "theory" coming with it–for example, numbers don't count as atoms.

We pretend that things are atoms even when they're not, since not many things in real life strictly satisfy the above properties. For example, we can model a compound object a that contains two components b and c by treating all three objects as atoms and defining a *contains* relation:



2.3 Basic Types

Every atom in Alloy belongs to a single *basic type*. That is, there is no subtyping in Alloy, and all basic types are disjoint. We write basic types in capital letters:

The types of atoms are not declared, but inferred by Alloy from the types of relations (see below) that are used with the atoms.

2.4 Relations

Every expression in Alloy is a relation. A relation is a set of tuples, where each element of the tuple is of a basic type. One can visualize a relation as a fixed-width table with one or more columns and zero or more rows; each row is a tuple of the relation. All of the entries in a given column must have the same basic type. As in other applications, the number of columns is called the *arity* of the relation, and the number of rows is called the *cardinality*.

Alloy allows arbitrary k-relations (the arity can be any natural number). We'll refer to 1-relations as *unary*, 2-relations as *binary*, and 3-relations as *ternary*. Sometimes people use the word "relation" to describe only binary relations. We'll consider sets to be unary relations, as they are just an unordered list of atoms.

Example

We are going to define some relations. First we give their types:

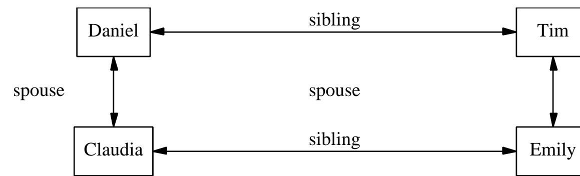


Figure 1: Example of some relations. Here, a bi-directional arrow is used for shorthand to indicate a relation in both directions.

```
spouse:  <PERSON, PERSON>
sibling: <PERSON, PERSON>
Daniel:  <PERSON>
Tim:     <PERSON>
Claudia: <PERSON>
Emily:   <PERSON>
```

Each of the relations above is a *variable*. `Daniel` is a unary relation (which we also refer to as a *set*), and in this case we presume that it contains only one element (*i.e.,* it is is a *singleton*)–we use the word *scalar* to refer to singleton sets. We could say that `Daniel` is an instance of the `DANIEL` atom. Atoms are never declared explicitly; Alloy is concerned only with variables, and it "generates" atoms.

Now let us specify some values for the relations above:

```
spouse:  { (Daniel, Claudia),
           (Claudia, Daniel),
           (Tim, Emily),
           (Emily, Tim) }

sibling: { (Daniel, Tim),
           (Tim, Daniel),
           (Claudia, Emily),
           (Emily, Claudia) }
```

As depicted in Figure 1, these relations illustrate the lecturer's remark: "If you didn't know me, you wouldn't know that I'm a living example of a commuting diagram!"

To get a taste of Alloy, we can express this remark in the language:

```
Some ( Daniel.spouse.sibling &
       Daniel.sibling.spouse )
```

This states that the intersection between Daniel's spouse's siblings and his sibling's spouses is non-empty.

Notes

A few notes about relations:

- Note the tuple-notation `(a, b)` denotes an *ordered* list of `a` and `b`. As usual, the notation `{a, b}` denotes an unordered set.
- In Alloy (unlike in some set theories), there is no distinction between a variable a, a 1-tuple of a, a singleton set a, and a singleton set of the 1-tuple of a. That is, in Alloy syntax, `a = (a) = {a} = {(a)}`.

Type	Symbol	Meaning
Set	+	union
	-	difference
	&	intersection
Comparison	in	subset
	=	equality (have same set of tuples)
Relational	$\sim$	transpose
	$\rightarrow$	product
	.	join

Table 1: Binary operators in Alloy.

- The relations defined above are all *homogeneous*, meaning that they relate variables of the same type. Alloy also supports *heterogeneous* relations between variables of different types. For example, one could define a relation for a routing table as `table: <ROUTER, IP, LINK>`

2.5 Operators

2.5.1 Binary Operators

Some of the binary operators that Alloy supports are shown in Table 1. A few of the operators deserve further explanation:

Transpose

The transpose operation is only defined for binary relations. So, for example, `~parents = child` .

Product

The product operation is defined as follows:

$$
\begin{aligned}
[[p \rightarrow q]] = \{\ & (s_1, \ldots, s_n, t_1, \ldots, t_m) \\
& \mid\ (s_1, \ldots, s_n) \in [[p]] \ \wedge \\
& \quad (t_1, \ldots, t_m) \in [[q]]\ \}
\end{aligned}
$$

Roughly speaking, the product operation takes all combinations of elements in p followed by elements in q. For example, if we have the following relations:

```
Knows:        <PERSON, PROGRAMMING-LANGUAGE>
MITGrad:      <PERSON>
LISP-Dialect: <PROGRAMMING-LANGUAGE>
```

then the assertion `MITGrad→LISP-Dialect` would mean that all MIT graduates know all dialects of LISP.

Join

The join operation is defined as follows:

$$
\begin{aligned}
[[p.q]] = \{\ & (s_1, \ldots, s_{n-1}, t_2, \ldots, t_m) \\
& \mid\ (s_1, \ldots, s_n) \in [[p]] \ \wedge \\
& \quad (t_1, \ldots, t_m) \in [[q]] \ \wedge \\
& \quad s_n = t_1\ \}
\end{aligned}
$$

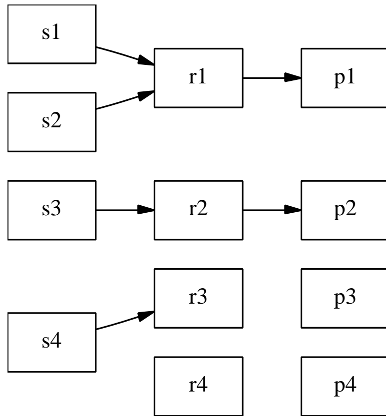
That is, the join operation "links together" tuples from p and q that have matching ends, and the element that matches is omitted from the linked set. For example, if we have the following relations:

```
room:  <STUDENT, ROOMS>
phone: <ROOM, PHONE>
```

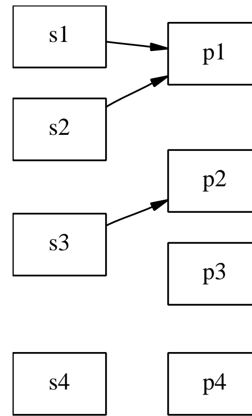
with the following values:

```
room:  { (s1, r1), (s2, r1), (s3, r2), (s4, r3) }
phone: { (r1, p1), (r2, p2) }
```

Then the expression `room.phone` gives you `{ (s1, p1), (s2, p1), (s3, p2) }`, as illustrated by the following diagram:



The `room` and `phone` relations.



The `room.phone` relation.

As an example of the join operator, we can note that `Daniel.parents = ~parents.Daniel` .

2.5.2 Constants

Before we can define the closure operator, we need some expressions for constructing standard expressions in Alloy. These expressions are as follows:

Expression	Type Signature	Meaning
none[r]	$\langle t_1, t_2\rangle \to \langle t_1, t_2\rangle$	Given a relation r, returns the empty relation $\{\}$ within the same domain as r
iden[e]	$t \to \langle t, t\rangle$	Given an expression e of basic type t, returns the relation $\{(a, a) \mid a \in t\}$ – that is, the relation that maps each element to itself.
univ[r]	$\langle t_1, t_2\rangle \to \langle t_1, t_2\rangle$	Given a relation $r : \langle t_1, t_2\rangle$, returns the relation $\{\langle a, b\rangle \mid a \in t_1, b \in t_2\}$ – that is, the universal relation that maps every element of type t_1 to every element of type t_2.

In the use of these constants in the following sections, we are slightly sloppy about the arguments that are passed to the constructors. This issue will become clear in the following lectures once we talk about declaring *signatures*.

2.5.3 Closure

We can now define the closure operators:

Operator	Meaning
$\hat{}r$	$r + r.r + r.r.r + \ldots$
$*r$	iden$[r]$ + $\hat{}r$

For example, `ancestor = ^parent` .

2.5.4 Navigation Expressions

Let's compare the use of the join operator as a navigator for relations with the navigation of Java fields and methods via the same syntax. In both Alloy and Java, you could write something like `x.f.g` . The difference is that in Alloy, `x` could map to either none, one, or many variables, whereas in Java it can map to either one or none (if the object is `null`).

In addition, Alloy allows *branching* and *iterative* constructors as part of join operations, which further tightens the syntax. An example of branching is `Daniel.(spouse + sibling)`, which gives both the spouse and siblings of Daniel. An example of iteration is `Daniel.^friend`, which gives the entire network of Daniel's friends. As another example, if `next` is a list, then `l.^next` gives all the elements of the list. Finally, the following assertion states that there are no cycles in a list:

```
^next & iden[list] = none[list]
```

2.6 Properties of Operators

We could consider the following properties of binary relations (we give them with their Alloy syntax):

- **Homogeneous.** The types in the relation are the same.
- **Heterogeneous.** The types in the relation are different.
- **Functional.** Each atom is mapped to at most one other: `no a:t | #a.r > 1`
- **Injective.** No two disjoint elements map to the same element. We have to express this in set terms in case the relation is not functional (an element maps to more than one element):

  `no a:t | #a.~r > 1`
- **Reflexive.** `all a:t | a->a in r`
- **Symmetric.** `all a:t | a->b in r => b->a in r`
- **Transitive.** `all a,b,c:t | (a->b in r && b->c in r) => (b->c in r)`
- **Irreflexive.** `no a:t | a->a in r`
- **Anti-symmetric.** `all a,b:t | (a->b in r && b->a in r) => a=b`

2.7 Examples

Finally, we consider some examples of binary relations and the properties that these relations might have. In designing software, it is useful to think about these properties before doing an implementation.

Containment

```
folderContents: <FOLDER, MSG>
```

This relation is heterogeneous, which implies that folders can't contain other folders. Also, it is usually injective in most implementations, where a message is only in one folder. However, it wouldn't have to be this way (you could have overlapping folders).

```
groupContains: <OBJ, OBJ>
```

This is a homogeneous relation. We would expect it to be irreflexive, since groups shouldn't contain each other.

Labeling

```
ip: <MACHINE, IP>
```

This relation maps machines to IP addresses. We would expect it to be injective at any given time, since no two machines have the same IP address at once. However, since a time is not specified as part of the tuple, the relation as written could cover many times and thus might not be injective.

Security

```
canReadDoesBy: <PERSON, PERSON>
```

This relation is expressing if one person can read what another person does. We would expect it to be transitive, but not symmetric.

Graphics

```
occludes: <WINDOW, WINDOW>
```

This relation expresses whether or not a window on the screen occludes another. We would expect it to be asymmetric, since windows can't occlude each other. However, it is not transitive (even in one dimension!)

Grouping

```
conflicts: <MACHINE, MACHINE>
```

We didn't have time to talk about this one.

Linking

```
links: <URL, URL>
```

We didn't have time to talk about this one.

3 Next Time

Next week, we'll look at the Alloy language itself on Monday and use it to consider software idioms on Wednesday.