Alloy Documentation Documentation

Alloy Board

Apr 22, 2020

Contents:

CHAPTER 1

Introduction

1.1 About Alloy

Alloy is a tool for analyzing systems and seeing if they are designed correctly. You can read more here and download it here.

1.2 About This Guide

This is my tentative proposed documentation for the Alloy language.

This is not the official Alloy documentation. It's a staging ground for material that's intended to *become* official documentation, but for now I'm writing this independently, not as part of the alloy board.

This is a work in progress. This documentation is incomplete and changes may be made. There are also a lot of things that still need to be added.

This is meant as a reference, not a tutorial.

This is not comprehensive. Certain things are intentionally excluded, like bitshift operators, specific internal modules, anonymous predicates, and Alloy 3 legacy syntax. Generally these are things that I personally believe are only useful in very specific niche cases, and otherwise serve to confuse things.

1.3 How to Read

I've added commentary in the markup of the pages that is invisible in the final output. If you see something

```
.. like this
```

Then it will not be rendered. If I used a special `run` predicate to generate a particular visualization, I put it in a comment.

Anything with a `[]` is an advanced topic and unecessary to learn if you're just a beginner.

Issues with the documentation can be raised here.

1.4 About the Author

I'm Hillel. I also wrote a guide to TLA+ and have a blog and a twitter.

CHAPTER 2

Language

2.1 Signatures

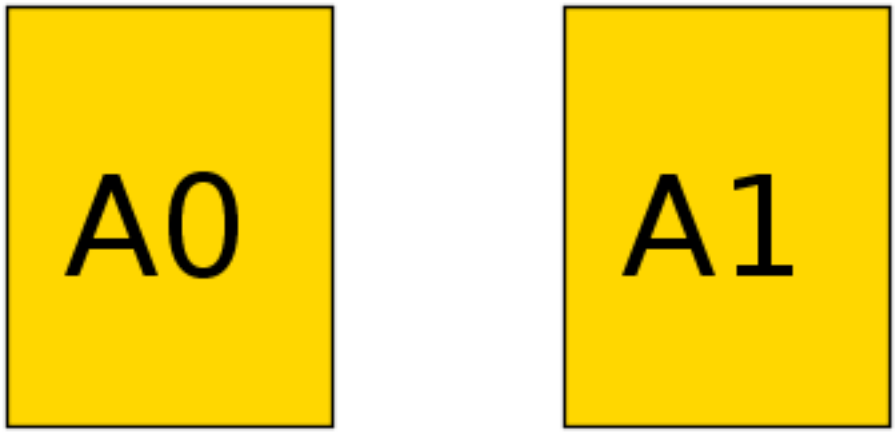
A **signature** expresses a new type in your spec. It can be anything you want. Here are some example signatures:

- Time
- State
- File
- Person
- Msg
- Joke
- Pair

Each model will have some number of each signature, called **atoms**. Take the following spec:

```
sig A {}
```

The following would be a valid model:



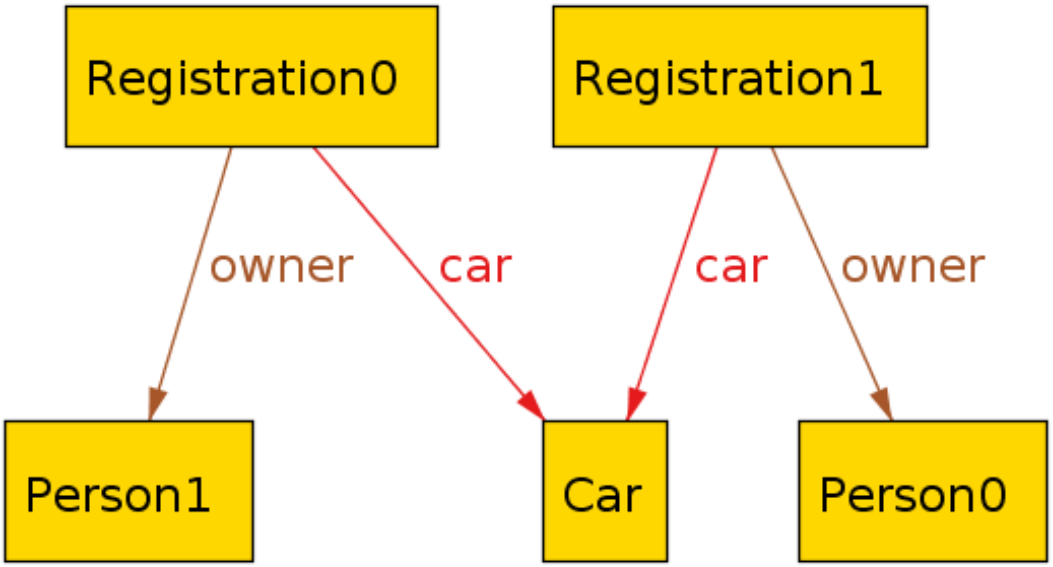
Here we have two atoms `A$1` and `A$0`. Both count as instances of the `A` signature. See *visualizer* for more on how to read the visualizations.

Usually we care about the relationships between the parts of our systems. We don't just care that there are people and accounts, we care which people have which accounts. We do this by adding **relations** inside of the signature body.

```
sig Person {}
sig Car {}

sig Registration {
  owner: Person,
  car: Car
}
```

This defines a new `Registration` type, where each Registration has an `owner`, which is a Person, and a `car`, which is a Car. The comma is required.



Tip: Extra commas are ignored. So you can write `Registration` instead like this:

```
sig Registration {
   , owner: Person
   , car: Car
}
```

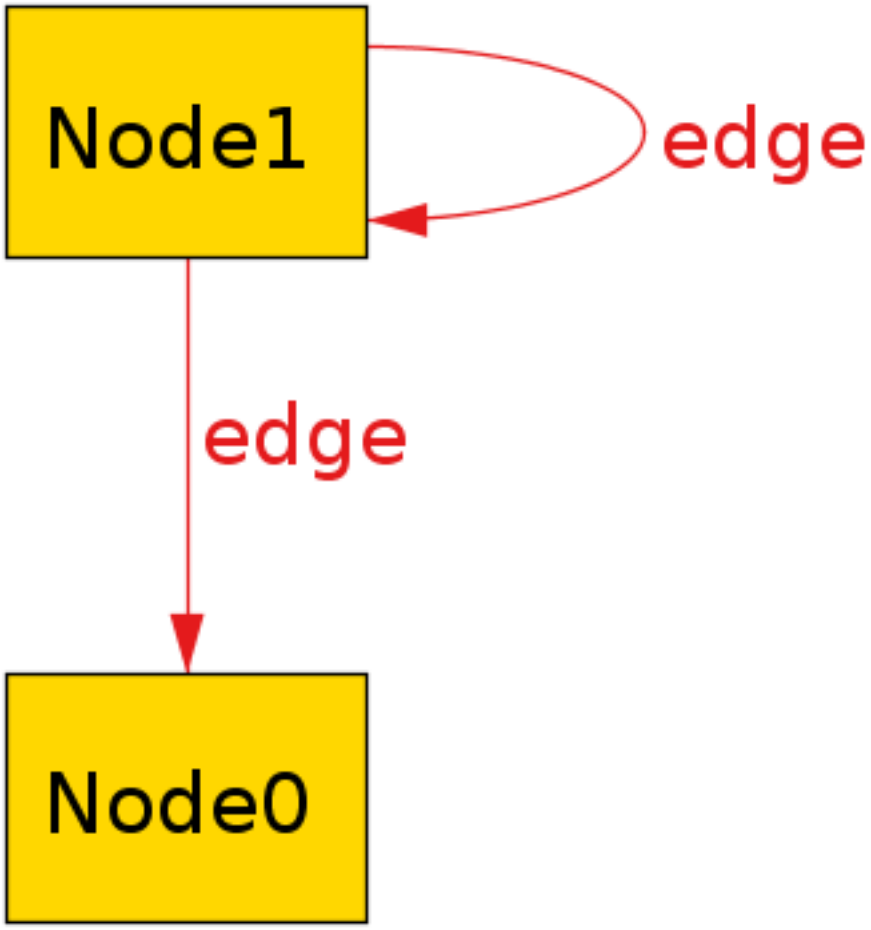
2.1.1 Relations

The body of a signature is a list of **relations**, which show how the signatures are connected to each other.

Relations are separated by a comma. The list can also start and end with a comma. Standard convention is to prefix every relation with a comma.

Relations *can* refer to the same signature. This is valid:

```
sig Node {
  , edge: set Node
}
```



Alloy can generate models where a relation points to the same atom. For this reason we often want to add contraints to our model, such as *Facts* or *Predicates*.

Note: Each relation in the body of a signature actually represents a *relation* type. If we have:

```
sig A {r: one B}
```

Then `r` is set of relations in `A -> B`. See *Sets and Relations* for more information.

Different signatures *may* have relationships with the same name as long as the relationship is not *ambiguous*.

Multiplicity

Each relation has a **multiplicity**, which represents how many atoms it can include. If you do not include a multiplicity, it's assumed to be `one`.

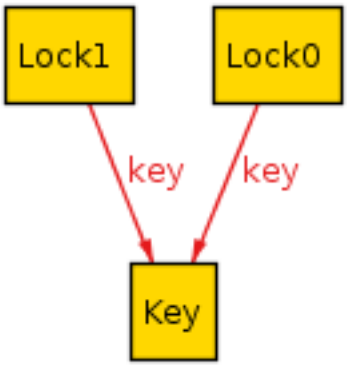
one

The default. `r: one A` states that there is *exactly one* A in the set.

```
sig Key {}

sig Lock {
  , key: one Key
}
```

This says that every lock has exactly one Key. This does *not* guarantee a 1-1 correspondence! Two locks can share the same key.



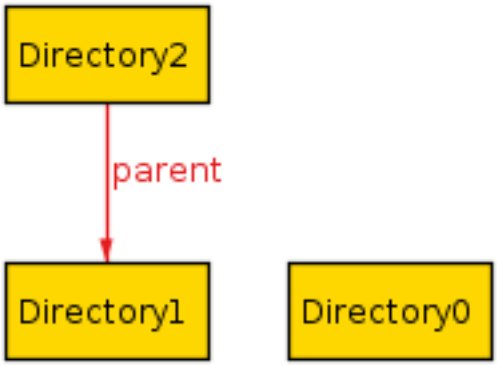
If no multiplicity is listed, Alloy assumes to be `one`. So the above relation can also be written as `key: Key`.

lone

`r: lone A` states that *either* there is one A in the set, *or* that the set is empty. You can also think of it as "optional".

```
sig Directory {
  , parent: lone Directory
}
```

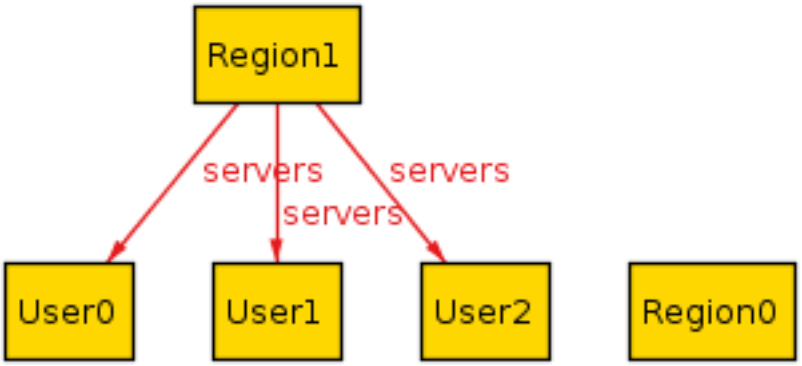
This says that every directory either has one parent, *or* it does not have a parent (it's a root directory).



set

`r: set A` states that there can be any number of A in the relation.

```
sig User {}
sig Region {
  servers: set User
}
```

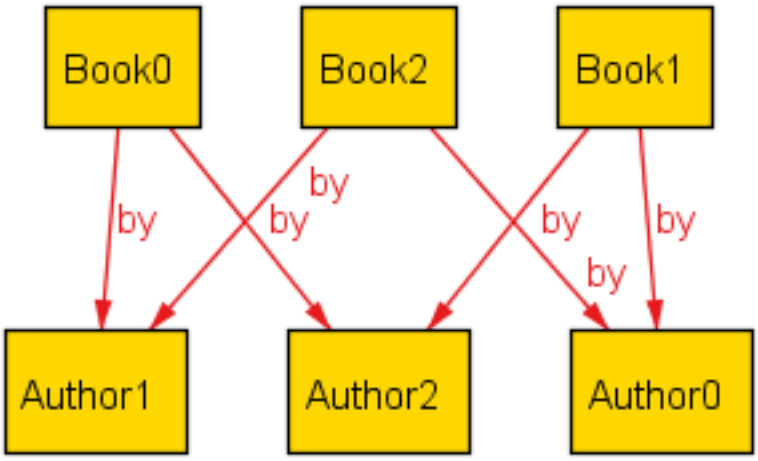


some

`r: some A` states that there is *at least one* A in the relation.

```
sig Author {}

sig Book {
  by: some Author
}
```



disj

`disj` can be prepended to any multiplicity to gaurantee that it will be disjoint among all atoms. If we write

```
sig Lock {}
sig Key {
  lock: disj one Lock
}
```

Then every key will correspond to a *different* lock. If we instead write

```
sig Lock {}
sig Key {
  locks: disj some Lock
}
```

Then every key will correspond to one or more locks, but no two keys will share a lock.

seq

See here for more info.

Field Expressions

A field can be a simple *expression* over other signatures.

```
sig Resource {
  permissions: set (User + Group)
}
```

In addition to full signatures, the expression may contain `this`, which refers to the specific atom itself.

```
sig Node {
  -- no self loops
  , edge: set Node - this
}
```

A *dependent field* is one where the expression depends on the values of other fields in the atom. The dependencies must be fields defined either in the signature or its *supertype*.

```
sig Item {}

sig Person {
  , favorite: Item
  , second: Item - favorite
}
```

Multirelations

Signatures can have multirelations as fields:

```
sig Door {}
sig Card {}

sig Person {
  access: Card -> Door
}
```

In this case `access` is a ternary relationship, where each element of `access` is a relation of form `Person -> Card -> Door`. Using the *dot operator*, if `access = P -> C -> D`, then `P.access = C -> D` and `access.D = P -> C`.

Multirelations have a special kind of multiplicity:

```
r: A m -> n B
```

This says that each member of `A` is mapped to `n` elements of B, and `m` elements of `A` map to each element of B. If not specified, the multiplicities are assumed to be `set`.

As an aide, use the following table:

m	n	Meaning
set	set	No restrictions
set	some	Each A used at least once
set	one	Each A is mapped to exactly one B (total function)
set	lone	Each A is mapped to at most one B (partial function)
some	set	Each B mapped to at least once
some	some	Every A mapped from and every B mapped to
some	one	Each A used exactly once, each B used at least once
some	lone	Each A used at most once, each B used at least once
one	set	Each B used exactly once, no other restrictions (one A can map to two B atoms)
one	some	Each B used exactly once, each A used at least once
one	one	Only satisfiable if #A = #B, bijection
one	lone	At most #A arrows, exactly #B arrows, each A used at most once
lone	set	Each B used at most once
lone	some	Each A used at least once and each B used at most once
lone	one	Each A used exactly once, each B used at most once
lone	lone	Each A used at most once, each B used at most once

Not all multiplicities will have valid models. For example,

```
sig A {}
sig B {}
one sig C {
  r: A one -> one B
}

run {} for exactly 3 A, exactly 2 B
```

Since `r` must be 1-1, and there's different numbers of A and B sigs, nothing satisfies this model.

Multirelations can go higher than ternary using the same syntax, but this is generally not recommended.

2.1.2 Signature Multiplicity

In addition to having multiplicity relationships, we can put multiplicities on the signatures themselves.

```
one sig Foo {}
some sig Bar {}
//etc
```

By default, signatures have multiplicity `set`, and there may be zero or more in the model. By making the signature `one`, every model will have exactly one atom of that signature. By writing `some`, there will be at least one. By writing `lone`, there will be zero or one.

2.1.3 Subtypes

We can make some signatures subtypes of other signatures.

in

Writing `sig Child in Parent` creates an *inclusive* subtype: any Parent atoms may or may not also be a Child. This is also called a "subset subtype".

```
sig Machine {}

sig Broken in Machine {}
sig Online in Machine {}
```

In this case, any Machine can also be Broken, Online, both, or neither.

+

A single inclusive subtype can be defined for many parent signatures. We can do this by using the set union operator on the parent signatures.

```
sig Bill, Client {}

sig Closed in Bill + Client {}
```

extends

Writing `sig Child extends Parent` creates a subtype, as with `in`. Unlike `in`, though, any Parent atom can only match up to *one* extension.

```
sig Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
```

In this case, any Machine can also be a Server, a Client, or neither, but not both.

Something can belong to both `extend` and `in` subtypes.

```
sig Machine {}
sig Broken in Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
```

A Machine can be both a Server and Broken, or a Client and Broken, or just one of the three, or none at all.

abstract

If you make a signature `abstract`, then all instances of the signature will be extensions, and there will be no signatures that are still the base.

```
abstract sig Machine {}
sig Broken in Machine {}

sig Server extends Machine {}
sig Client extends Machine {}
```

Here any machine **must** be either a Server or a Client. They still may or may not be Broken.

> **Warning:** If there is nothing extending an abstract signature, the abstract is ignored.

Tip: You can place multiple signatures on the same line.

```
sig Server, Client extends Machine {}
```

Subtypes and Relationships

All subtypes are also their parent type. So if we have

```
sig B {}
sig C in B {}

sig A {
 , b: B
 , c: C
}
```

Then the `b` relation can include `A -> C`, but `c` **cannot** include `A -> B`.

Child Relations

Children automatically inherit all of their Parent fields, *and also* can define their own fields. We can have:

```
sig Person {}
sig Account {
  , person: Person
}

sig PremiumAccount in Account {
  , billing: Person
}
```

Then all `Account` atoms will have the `person` field, while all `PremiumAccount` atoms will have both a `person` field and a `billing` field.

Note: This also applies to *Implicit Facts*. If Account has an implicit fact, it automatically applies to PremiumAccount.

It is not possible redefine a relationship, only to add additional ones.

2.1.4 Enums

Enums are a special signature.

```
enum Time {Morning, Noon, Night}
```

The enum will always have the defined atoms in it. Additionally, the atom will have an *ordering*. In this case, Morning will be the first element, Noon the second, and Night will be the third. You can use enums in facts and predicates, but you cannot add additional properties to them.

Tip: If you want to use an enumeration with properties, you can emulate this by using `one` and signature extensions.

```
abstract sig Time {}

one sig Morning, Noon, Night extends Time {
    time: Time
}
```

You can also use this to make enumerations without a fixed number of elements, by using `lone` instead.

Warning: Each enum implicitly imports *ordering*. The following is invalid:

```
enum A {a}

enum B {b}

run {some first}
```

As it is ambiguous whether `first` should return `a` or `b`. If you need to use both an enum inside of a dynamic model, be sure to use a *namespace* when importing *ordering*.

2.2 Sets and Relations

2.2.1 Sets

A set is a collection of unique, unordered elements. All Alloy expressions use sets of atoms and *Relations*. All elements of a set must have the same arity, but can otherwise freely mix types of elements.

Adding a signature automatically defines the set of all atoms in that signature. Given

```
sig Teacher {}
sig Student {
  teacher: Teacher
}
```

Then the spec recognizes `Student` as the set of all atoms of type `Student`, and likewise with the `Teacher` signature and the `teacher` relationship.

Everything in Alloy is a set. If `S1` is a `Student` atom, then `S1` is the set containing just `S1` as an element.

There are also two special sets:

- `none` is just the empty set. Saying `no Set` is the same as saying `Set = none`. See *Expressions*.
- `univ` is the set of all atoms in the model. In this example, `univ = Student + Teacher`.

Note: By default, the analyzer also generates a set of integers for each model, which will appear in `univ`. This can almost always be ignored in specifications (but see # below).

Set Operators

Set operators can be used to construct new sets from existing ones, for use in expressions and predicates.

- `S1 + S2` is the set of all elements in either `S1` or `S2` (set union).
- `S1 - S2` is the set of all elements in `S1` but not `S2` (set difference).
- `S1 & S2` is the set of all elements in both `S1` and `S2` (set intersection).

```
S1 = {A, B}
S2 = {B, C}

S1 + S2 = {A, B, C}
S1 - S2 = {A}
S1 & S2 = {B}
```

-> used as an operator

Given two sets, `Set1 -> Set2` is the *Cartesian product* of the two: the set of all relations that map any element of `Set1` to any element of `Set2`.

```
Set1 = {A, B}
Set2 = {X, Y, Z}

Set1 -> Set2 = {
  A -> X, A -> Y, A -> Z,
  B -> X, B -> Y, B -> Z
}
```

As with other operators, a standalone atom is the set containing that atom. So we can write `A -> (X + Y)` to get `(A -> X + A -> Y)`.

Tip: `univ -> univ` is the set of all possible relations in your model.

Integers

Alloy has limited support for integers. To enforce bounded models, the numerical range is finite. By default, Alloy uses models with 4-bit signed integers: all integers between `-8` and `7`. If an arithmetic operation would cause this to

overflow, then the predicate is automatically declared false. In the *Evaluator*, however, it will wrap the overflowed number.

Tip: The numerical range can be changed by placing a *scope* on `Int`. The number of the scope is the number of bits in the signed integers. For example, if the scope is `5 Int`, the model will have all integers between `-16` and `15`.

All arithmetic operators are over the given model's numeric range. To avoid conflict with set and relation operators, the arithmetic operators are written as *Functions*:

```
add[1, 2]
sub[1, 2]
mul[1, 2]
div[3, 2] -- integer divison, drop remainder
rem[1, 2] -- remainder
```

You can use receiver syntax for this, and write `add[1, 2]` as `1.add[2]`. There are also the following comparison predicates:

```
1 =< 2
1 < 2
1 > 2
1 >= 2
1 != 2
1 = 2
```

As there are no corresponding symbols for elements to overload, these operators are written as infixes.

> **Warning:** Sets of integers have non-intuitive properties and should be used with care.

#

`#S` is the number of elements in `S`.

Sets of numbers

For set operations, a set of numbers are treated as a set. For arithmetic operations, however, a set of numbers is first summed before applying the operator. This is equivalent to using the `sum[]` function.

```
(1 + 2) >= 3 -- true
(1 + 2) <= 3 -- true
(1 + 2) = 3  -- false
(1 + 2).plus[0] = 3 -- true
(1 + 1).plus[0] = 2 -- false
```

2.2.2 Relations

Given the following spec

```
sig Group {}
sig User {
  belongs_to: set Group
}
```

`belongs_to` describes a **relation** between `User` and `Group`. Each individual relation consists of a pair of atoms, the first being `User`, the second being `Group`. We write an individual relation with `->`. One possible model might have

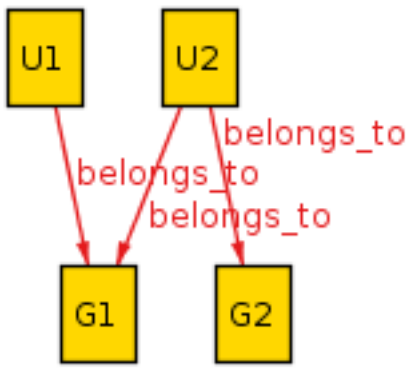
```
belongs_to = {
  U1 -> G1 +
  U2 -> G1 +
  U2 -> G2
}
```

Relations do *not* need to be 1-1: here two users map to `G1` and one user maps to both `G1` and `G2`.

```
abstract sig Group {}
abstract sig User {
   belongs_to: set Group
}

one sig U1, U2 extends User {}
one sig G1, G2 extends Group {}

fact {
  belongs_to = {
  U1 -> G1 +
  U2 -> G1 +
  U2 -> G2
}
}
```



Relations in Alloy are first class objects, and can be manipulated and used in expressions. [This assumes you already know the set operations]. For example, we can reverse a relation by adding ~ before it:

```
~belongs_to = {
  G1 -> U1 +
  G1 -> U2 +
  G2 -> U2
}
```

The . Operator

The dot (`.`) operator is the most common relationship operator, and has several different uses. The dot operator is left-binding: `a.b.c` is parsed as `(a.b).c`, not `a.(b.c)`.

`Set.rel`

Return every element that elements in `Set` map to, via `rel`. This also works for individual atoms.

```
U1.belongs_to = G1
(U1 + U2).belongs_to = {G1, G2}
```

Tip: In this case, we can find all groups in the relation with `User.belongs_to`. However, some relations may mix different types of atoms. In that case `univ.~rel` is the domain of `rel` and `univ.rel` is the range of `rel`.

For *Multirelations*, this will return the "tail" of the relation. Eg if `rel = A -> B -> C`, then `A.rel = B -> C`.

`rel.Set`

Writing `rel.Set` is equivalent to writing `Set.~rel`. See *~rel*.

```
belongs_to.G1 = {U1, U2}
G1.~belongs_to = {U1, U2}
```

`rel1.rel2`

We can use the dot operator with two relations. It returns the inner product of the two relations. For example, given

```
rel1 = {A -> B,          B -> A}
rel2 = {B -> C,
        B -> D,          A -> E}

rel1.rel2 = {
        A -> C,
        A -> D,          B -> E}
```

In our case with Users and Groups, `belongs_to.~belongs_to` maps every User to every other user that shares a group.

[]

`rel[elem]` is equivalent to writing `elem.(rel)`. It has a lower precedence than the `.` operator, which makes it useful for *Multirelations*. If we have

```
sig Light {
   state: Color -> Time
}
```

Then `L.state[C]` would be all of the times `T` where the light `L` was color `C`. The equivalent without `[]` would be `C.(L.state)`.

iden

`iden` is the relationship mapping every element to itself. If we have an element `a` in our model, then `(a -> a) in iden`.

An example of iden's usefulness: if we want to say that `rel` doesn't have any cycles, we can say `no iden & ^rel`.

Additional Operators

Note: You cannot use `~`, `^`, or `*` with *higher-arity relations*.

~rel

As mentioned, `~rel` is the reverse of `rel`.

^ and *

These are the **transitive closure** relationships. Take the following example:

```
sig Node {
  edge: set Node
}
```

`N.edge` is the set of all nodes that `N` connects to. `N.edge.edge` is the set of all nodes that an edge of `N` connects to. `N.edge.edge.edge` is the set of all nodes that are an edge of an edge of N, ad infinitum. If we want every node that is connected to `N`, this is called the transitive closure and is written as `N.^edge`.

`^` does *not* include the original atom unless it's transitively reachable! In the above example, `N in N.^edge` iff the graph has a cycle containing `N`. If we want to also include `N`, use `N.*edge` instead.

`^` operates on the relationship, so `^edge` is also itself a relationship and can be manipulated like any other. We can write both `~^edge` and `^~edge`. It also works on arbitrary relationships. `U1.^(belongs_to.~belongs_to)` is the set of people that share a group with `U1`, or share a group with people who share a group with `U1`, ad infinitum.

Warning: By itself `*edge` will include `iden`! `*edge = ^edge + iden`. For best results only use `*` immediately before joining the closure with another set.

Advanced Operators

<: and :>

`<:` is *domain restriction*. `Set <: rel` is all of the elements in `rel` that **start** with an element in `Set`. `:>` is the *range restriction*, and works similarly: `rel :> Set` is all the elements of `rel` that **end** with an element in Set.

This is mostly useful for directly manipulating relations. For example, given a set S, we can map every element to itself by doing `S <: iden`.

++

`rel1 ++ rel2` is the union of the two relations, with one exception: if any relations in `rel1` that share a "key" with a relation in `rel2` are dropped. Think of it like merging two dictionaries.

Formally speaking, we have

```
rel1 ++ rel2 = rel1 - (rel2.univ <: rel1) + rel2
```

Some examples of ++:

```
(A -> B + A -> C) ++ (A -> A) = (A -> A)
(A -> B + A -> C) ++ (A -> A + A -> C) = (A -> A + A -> C)
(A -> B + A -> C) ++ (C -> A) = (A -> B + A -> C + C -> A)
(A -> B + B -> C) ++ (A -> A) = (A -> A + B -> C)
```

It's mostly useful for modeling *Time*.

Note: When using multirelations the two relations need the same arity, and it overrides based on only the first element in the relations.

Set Comprehensions

Set comprehensions are written as

```
{x: Set1 | expr[x]}
```

The expression evaluates to the set of all elements of `Set1` where `expr[x]` is true. `expr` can be any expression and may be inline. Set comprehensions can be used anywhere a set or set expression is valid.

Set comprehensions can use multiple inputs.

```
{x: Set1, y: Set2, ... | expr[x,y]}
```

In this case this comprehension will return relations in `Set1 -> Set2`.

2.3 Expressions and Constraints

2.3.1 Expressions

Expressions are anything that returns a number, boolean, or set. Boolean expressions are also called *Constraints*.

Information about relational expressions are found in the *Sets* and *Relations* chapters. There are two additional constructs that can be used with both boolean and relational expressions.

See also:

Integer expressions

Let

`let` defines a local value for the purposes of the subexpression.

```
let x = A + B, y = C + D |
   x + y
```

In the context of the `let` expression `x + y = (A + B) + (C + D)`. `let` is mostly used to simplify complex expressions and give meaningful names to intermediate computations.

If writing a boolean expression, you may use a `{}` instead of `|`.

`let` bindings are not recursive. A `let` binding may not refer to itself or a future `let` binding.

Warning: As with *predicate* parameters, `let` can *shadow* a global value. You can use the `@` operator to retrieve the global value.

implies - else

When used in conjunction with `else`, `implies` acts as a conditional. `p implies A else B` returns A if p is true and B if p is false. `p` must be a boolean expression.

If A and B are boolean expressions, then this acts as a constraint. The `else` can be left out if using `implies` as a constraint. See *below* for details.

2.3.2 Constraints

Bar expressions

A bar expression is one of the form:

```
some x: Set |
  expr
```

In this context, the expression is true iff `expr` is true. The newline is optional.

Paragraph expressions

If multiple constraints are surrounded with braces, they are all `and`-ed together. The following two are equivalent:

```
expr1 or {
  expr2
  expr3
    ...
}

expr1 or (expr 2 and expr3 and ...)
```

Constraint Types

All constraints can be inverted with `not` or `!`. To say that A is not a subset of B, you can write `A !in B`, `A not in B`, `!(A in B)`, etc.

Relation Constraints

=

`A = B` means that both sets of atoms or relations have the exact same elements. `=` *cannot* be used to compare two booleans. Use `iff` instead.

in

- `A in B` means that every element of A is also an element of B. This is also known as a "subset" relation.
- `x in A` means that x is an element of the set A.

Note: The above two definitions are equivalent as all atoms are singleton sets: x is the set containing x, so `x in A` is "the set containing just x is a subset of A".

---

size constraints

There are four constraints on the number of elements in a set:

- `no A` means A is empty.
- `some A` means A has *at least one* element.
- `one A` means A has *exactly one* element.
- `lone A` means A is either empty or has exactly one element.

In practice, `no` and `some` are considerably more useful than `one` and `lone`.

Note: Relations are each exactly one element, no matter the order of the relation. If `a`, `b`, and `c` are individual atoms, `(a -> b -> c)` is exactly one element, while `(a -> b) + (a -> c)` is two.

---

disj[A, B]

`disj[A, B]` is the predicate "A and B share no elements in common". Any number of arguments can be used, in which case `disj` is *pairwise-disjoint*. This means that `disj[A, B, C]` is equivalent to `disj[A, B] and disj[B, C] and disj[A, C]`.

Boolean Constraints

Boolean constraints operate on booleans or predicates. They can be used to create more complex constraints.

All boolean constraints have two different forms, a symbolic form and an English form. For example, `A && B` can also be written `A and B`.

word	symbol
`and`	`&&`
`or`	`||`
`not`	`!`
`implies`	`=>`
`iff`	`<=>`

The first three are self-explanatory. The other two are covered below:

`implies` (`=>`)

`P implies Q` is true if Q is true whenever P is true. If P is true and Q is false, then `P implies Q` is false. If P is false, then `P implies Q` is automatically true. `P implies Q else T` is true if P and Q are true or if P is false and T is true.

(Consider the statement `x > 5 implies x > 3`. If we pick `x = 4`, then we have `false implies true`).

`iff` (`<=>`)

`P iff Q` is true if P and Q are both true or both false. Use this for booleans instead of =.

Tip: `xor[A, B]` can be written as `A <=> !B`.

Quantifiers

A **quantifier** is an expression about the elements of a set. All of them have the form

```
some x: A |
  expr
```

This expression is true if `expr` is true for any element of the set of atoms `A`. As with `let`, `x` becomes a valid identifier in the body of the constraint.

Instead of using a pipe, you can also write it as

```
some x: Set {
  expr1
  ...
}
```

In which case it is treated as a standard paragraph expression.

The following quantifiers are available:

- `some x: A | expr` is true for *at least one* element in `A`.
- `all x: A | expr` is true for *every* element in `A`.
- `no x: A | expr` is **false** for every element of `A`.
- [A] `one x: A | expr` is true for exactly one element of `A`.
- [A] `lone x: A` is equivalent to `(one x: A | expr) or (no x: A | expr)`.

As *discussed below*, `one` and `lone` can have some unintuitive consequences.

Tip: As with all constraints, `A` can be any set expression. So you can write `some x: (A + B - C).rel`, etc.

Multiple Quantifiers

There are two syntaxes to quantify over multiple elements:

```
-- 1
some x, y, ...: A | expr

-- 2
some x: A, y: B, ... | expr
```

For case (1) all elements will be drawn from `A`. For case (2) the quantifier will be over all possible combinations of elements from A and B. The two forms can be combined, as in `all x, y: A, z: B, ... | expr`.

Elements drawn do **not** need to be distinct. This means, for example, that the following is automatically false if A has any elements:

```
all x, y: A |
  x.rel != y.rel
```

As we can pick the same element for `x` and `y`. If this is not your intention, there are two ways to fix this:

```
-- 1
all x, y: A |
  x != y => x.rel != y.rel

-- 2
all disj x, y: A |
  x.rel != y.rel
```

For case (1) we can still select the same element for `x` and `y`; however, the `x != y` clause will be false, making the whole clause true. For case (2), using `disj` in a quantifier means we cannot select the same element for two variables.

`one` and `lone` behave unintuitively when used in multiple quantifiers. The following two statements are different:

```
one f, g: S | P[f, g] -- 1
one f: S | one g: S | P[f, g] -- 2
```

Constraint (1) is only true if there is *exactly one* pair f, g that satisfies predicate P. Constraint (2) says that there's exactly one f such that there's exactly one g. The following truth table will satisfy clause (2) *but not* (1):

f	g	P[f, g]
A	B	T
A	C	T
B	A	T
B	C	T
C	B	T
C	A	F

As C is the only one where there is *exactly one* g that satisfies P[C, g]. As a rule of thumb, use only `some` and `all` when writing multiple clauses.

Relational Quantifiers

When using a *run* command, you can define a `some` quantifier over a relation:

```
sig Node {
    edge: set Node
}

pred has_self_loop {
    some e: edge | e = ~e

}

run {
    has_self_loop
}
```

When using a *check* command, you can define `all` and `no` quantifiers over relations:

```
assert no_self_loops {
        no e: edge | e = ~e
      }

check no_self_loops
```

You **cannot** use `all` or `no` in a run command or use `some` in a check command. You **cannot** use higher-order quantifiers in the *Evaluator* regardless of the command.

2.4 Predicates and Functions

2.4.1 Predicates

A predicate is like a programming function that returns a *boolean*. While they are a special case of Alloy functions, they are more fundamental to modeling and addressed first.

Predicates take the form

```
pred name {
       constraint
}
```

Once defined, predicates can be used as part of boolean expressions. The following is a valid spec:

```
sig A {}

pred at_least_one_a {
  some A
}

pred more_than_one_a {
  at_least_one_a and not one A
}

run more_than_one_a
```

Warning: Predicates and functions **cannot**, in the general case, be recursive. Limited recursion is possible, see *here* for more info.

Parameters

Predicates can also take arguments.

```
pred foo[a: Set1, b: Set2...] {
  expr
}
```

The predicate is called with `foo[x, y]`, **using brackets, not parens**. In the body of the predicate, `a` and `b` would have the corresponding values.

Receiver Syntax

The initial argument to a predicate can be passed in via a `.` join. The following two are equivalent:

```
pred[x, y, z]
x.pred[y, z]
```

2.4.2 Functions

Alloy functions are equivalent to programming functions. They have the same structure as predicates, but also return a value:

```
fun name[a: Set1, b: Set2]: output_type {
  expression
}
```

Tip: if a function is constant (does not take any parameters), the analyzer casts it to a constant set. This means if we have a function of parameter

```
fun foo: A -> B {
    expression
}
```

Then `^foo` is a valid expression.

Overloading

Predicates and functions may be overloaded, as long as it's unambiguous which function applies. The following is valid:

```
sig A {}

sig B {}
```

(continues on next page)

(continued from previous page)

```
pred foo[a: A] { --1
  a in A
}

pred foo[b: B] { --2
  b in B
}

run {some a: A | foo[a]}
```

As when `foo` is called, it's unambiguous whether it means (1) or (2). If we instead replaced `sig B` with `sig B extends A`, then it's ambiguous and the call is invalid.

Overloading can happen if you import the same parameterized module twice.For example, given the following:

```
open util/ordering[A]
open util/ordering[B]

sig A, B {}
run {some first}
```

It is unclear whether `first` applies to `A` or `B`. To fix this, use *Namespaced imports*:

```
open util/ordering[A] as u1
open util/ordering[B] as u2

sig A, B {}
run {some u2/first}
```

Parameter Overrides

The parameters of a function (or predicate) can shadow a global value. In this case, you can retrieve the original global value by using `@val`.

```
sig A {}

pred f[A: univ, b: univ] {
  b in A  -- function param
  b in @A -- global signature
}
```

2.4.3 Facts

A fact has the same form as a global predicate:

```
fact name {
  constraint
}
```

A fact is *always* considered true by the Analyzer. Any models that would violate the fact are discarded instead of checked. This means that if a potential model both violates an assertion and a fact, it is not considered a counterexample.

```
sig A {}

-- This has a counterexample
check {no A}

-- Unless we add this fact
fact {no A}
```

Tip: For facts, the name is optional. In addition, the name can be a string. So this is a valid fact:

```
fact "no cycles" {
  all n: Node | n not in n.^edge
}
```

Implicit Facts

You can write a fact as part of a signature. The implicit fact goes after the signature definition and relations. Inside of an implicit fact, you can get the current atom with `this`. Fields are automatically expanded in the implicit fact to `this.field`.

```
sig Node {
  edge: set Node
} {
  this not in edge
}
```

This means you cannot apply the relation to another atom of the same signature inside the implicit fact. You can access the original relation by using the `@` operator:

```
-- undirected graphs only
sig Node {
  , edge: set Node
}
{
  all link: edge | this in link.edge -- invalid
  all link: edge | this in link.@edge -- valid
}
```

2.4.4 Macros

A macro is a similar to a predicate or function, except it is expanded before runtime. For this reason, macros can be used as part of signature fields. Parameters to macros also don't need to be given types, so can accept arbitrary signatures and even boolean constraints. Macros are defined with `let` in the top scope.

```
let selfrel[Sig] = { Sig -> Sig }
let many[Sig] = { some Sig and not one Sig }

sig A {
  rel: selfrel[A]
}

run {many[A]}
```

See here for more information.

2.5 Commands

A *command* is what actually runs the analyzer. It can either find models that satisfy your specification, or counterexamples to given properties.

By default, the analyzer will run the top command in the file. A specific command can be run under the `Execute` menu option.

2.5.1 `run`

run tells the analyzer to find a matching example of the spec.

`run pred`

Find examples where `pred` is true. If no examples match, the analyzer will suggest the predicate is inconsistent (see *unsat core*). The predicate may be consistent if the *scope* is off.

```
sig Node {
    edge: set Node
}

pred self_loop[n: Node] {
  n in n.edge
}

pred all_self_loop {
  all n: Node | self_loop[n]
}

run all_self_loop
```

The analyzer will title the command as the predicate.

```
Executing "Run all_self_loop"
   Sig this/Node scope <= 3
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 SkolemDepth=1 Symmetry=20
   79 vars. 12 primary vars. 101 clauses. 4ms.
   Instance found. Predicate is consistent. 5ms.
```

`run {constraint}`

Finds an example satisfying the ad-hoc constraint in the braces.

```
// some node with a self loop
run {some n: Node | self_loop[n]}
```

Tip: The analyzer will title the command `run${num}`. You can give the command a name by prepending the `run` with `name:`:

```
some_self_loop:run {some n: Node | self_loop[n]}
```

2.5.2 check

check tells the Analyzer to find a counterexample to a given constraint. You can use it to check that your specification behaves as you expect it to.

Unlike with `run` commands, `check` uses **assertions**:

```
assert no_self_loops {
    no n: Node | self_loop[n]
}

check no_self_loops
```

Asserts may be used in `check` commands but not `run` commands. Assertions may not be called by other predicates or assertions.

You can also call `check` with an ad-hoc constraint:

```
check {no n: Node | self_loop[n]}
```

`check` can also be given a named command.

2.5.3 Scopes

All alloy models are **bounded**: they must have a maximum possible size. If not specified, the analyzer will assume that there may be up to three of each top-level signature and any number of relations. This is called the **scope**, and can be changed for each command.

Given the following spec:

```
sig A {}
sig B {}
```

We can write the following scopes:

- `run {} for 5`: Analyzer will look for models with up to five instances of each A and B.
- `run {} for 5 but 2 A`: Analyzer will look for models with up to two instances of A.
- `run {} for 5 but exactly 2 A`: Analyzer will only look for models with *exactly two* A. The exact scope *may* be higher than the general scope.
- `run {} for 5 but 2 A, 3 B`: Places scopes on A and B.

If you are placing scopes on all of the signatures, the `for N except` is unnecessary: the last command can be written as `run {} for 2 A, 3 B`.

Tip: When using *Arithmetic Operators*, you can specify `Int` like any other signature:

```
run foo for 3 Int
```

Note: You cannot place scopes on relations. Instead, use a predicate.

```
sig A {
  rel: A
}

run {#rel = 2}
```

Scopes on Subtypes

Special scopes *may* be placed on *extensional subtypes*. The following is valid:

```
sig Plant {}

sig Tree extends Plant {}
sig Grass extends Plant {}

run {} for 4 Plant, exactly 2 Tree
```

`Grass` does not need to be scoped, as it is considered part of `Plant`. The maximum number of atoms for a subtype is either it or its parent's scope, whichever is lower. The parent scope is shared across all children. In this command, there are a maximum of four `Plant`s, exactly two of which will be `Tree` atoms. Therefore there may be at most two `Grass` atoms.

In contrast, special scopes *may not* be placed on *subset types*. The following is invalid:

```
sig Plant {}

sig Seedling in Plant {}

run {} for 4 Plant, exactly 2 Seedling
```

Since `Seedling` is a subset type, it may not have a scope. If you need to scope on a subtype, use a constraint:

```
run {#Seedling = 2} for 4 Plant
```

2.6 Modules

Alloy **modules** are similar to programming languages and act as the namespaces. Alloy comes with a standard library of *utility modules*.

2.6.1 Simple Modules

```
open util/relation as r
```

Imports must be at the top of the file. Modules may import new signatures into the spec.

Modules can be imported multiple times under different namespaces.

Namespaces

A module can be namespaced by importing `as` a name. Namespaces are accessed with `/`. This is also called a **qualified** import.

```
open util/relation as r

-- later

r/dom
```

2.6.2 Parameterized Modules

A parameterized module is "generic": its functions and predicates are defined for some arbitrary signature. When you import a parameterized module, you must pass in a signature. Its functions and predicates are then specialized to be defined for that signature.

```
open util/ordering[A]

sig A {}

run {some first} -- returns an A atom
```

Normally *ord/first* returns an abstract `elem`. By parameterizing the module with `A`, the function now returns an `A` atom.

The input must be a full signature and not a subset of one.

A parameterized module can be imported multiple times using *Namespaces*.

Note: The following built-in modules are parameterized: *ordering*, *time*, *graph*, and `sequence`.

2.6.3 Creating Modules

The syntax for a module is

```
module name
```

At the beginning of the file.

Private

Any module predicate, function, and signature can be preceded by `private`, which means it will not be imported into other modules.

```
module name

private sig A {}
```

Creating Parameterized Modules

```
module name[sig]

-- predicates and functions should use sig
```

CHAPTER 3

Tooling

3.1 Analyzer

The **Alloy Analyzer** is the tool that actually checks your spec.

3.1.1 Configuring the Analyzer

The analyzer converts the model into a SAT expression to solve. Some of the options are configurable. By default you should not need to change any of these- most performance issues are better solved by improving the spec itself. The following all affect the runtime of the analyzer. All of these are under the "Options" toolbar of the IDE:

- **Allow Warnings:** When "no", the analyzer will halt if the model has any warnings. Warnings usually, but not always, correspond to errors in the spec.
- **Maximum Memory:** How much RAM the analyzer is allowed to use when solving.
- **Solver:** The SAT solver to use for finding the model. Different solvers may have different performance on different specs. The SAT model and Kodkod model can also be output to a temporary file here. There are additional special options for MiniSat with *Unsat Core*, below.
- **Skolem Depth:** see below
- **Recursion Depth:** see below
- **Record the Kodkod input/output:** when `true`, the Kodkod output for the run can be seen by clicking the *Predicate* link in the output.

```
Executing "Run run$1"
   Sig this/Univ scope <= 3
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 Skol
   0 vars. 0 primary vars. 0 clauses. 5ms.
   Instance found. Predicate is consistent. 9ms.
```

```
Option Record the Kodkod input/output changed to fals
Executing "Run run$1"
   Sig this/Univ scope <= 3
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=4 Skol
   0 vars. 0 primary vars. 0 clauses. 4ms.
   Instance found. Predicate is consistent. 2ms.
```

- **Prevent Overflows:** If an *arithmetic operation* would overflow the model, the predicate is treated as false.

Recursion Depth

Predicates and functions are normally not recursive- they may not call themselves. The following is an invalid predicate:

```
sig Node {
    edge: set Node
}

fact {no iden & ^edge}

pred binary_tree[n: Node] {
    #n.edge <= 2
    all child: n.edge |
    // recursive call
        binary_tree[child]
}

run {some n: Node | binary_tree[n]}
```

This is invalid because all Alloy models must be bound, and recursive calls can lead to an unbound model. Normally you should restructure your model to not need a recursive call. If you *must* have a recursive predicate or function, you can set the `recursion depth` to a maximum of 3.

The recursion depth is treated as a "fact": the analyzer will not look for models with a greater recursion depth, even if it would lead to a valid example or counterexample.

Warning: Increasing the recursion depth will slow down your spec.

Unsat Core

By default Alloy is packaged with Minisat, which also has an *Unsat Core*. When "MiniSat with Unsat Core" is selected as the solver, the analyzer can isolate which constraints prevent the analyzer from finding a counter/example. See here for more information.

Note: By default, the Windows version of Alloy does not come with MiniSAT.

Warning: The "Core Granularity" option is not strictly increasing in terms of information: a slower setting might, in some circumstances, lead to the core providing *less* information. Given the following model:

```
sig Node {
    edge: some Node
}

fact {some Node}

run {no edge}
```

All granularity settings will highlight three formulas *except* for "expand quantifiers", which will only highlight two. However, all three constraints are required to make the predicate inconsistent.

3.2 Visualizer

Given the following model

```
abstract sig Object {}

sig File extends Object {}

sig Dir extends Object {contents: set Object}
one sig Root extends Dir { }


fact {
 Object in Root.*contents
}

assert RootTop {
 no o: Object | Root in o.contents
}
check RootTop // This assertion should produce a counterexample
```

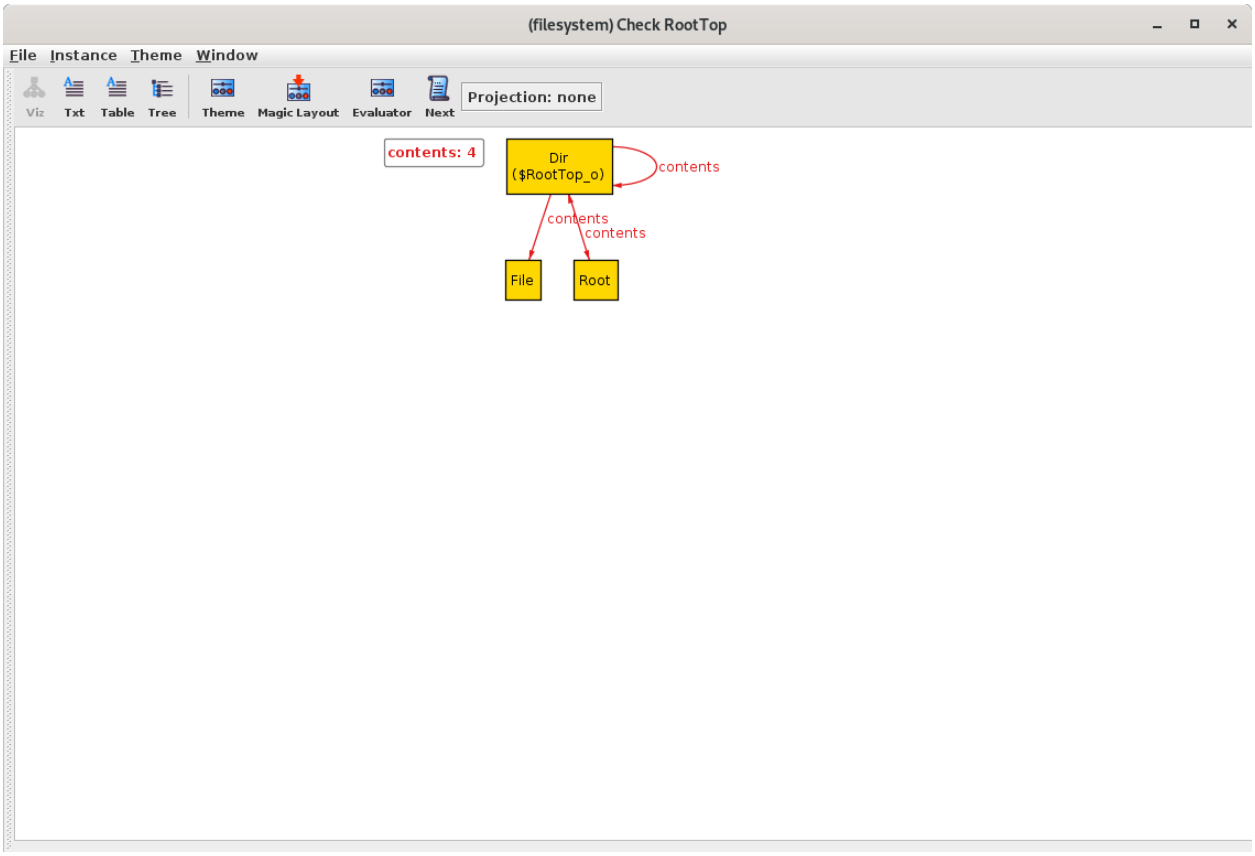
::

fact "demo" {

let unroot = Dir - Root | contents = Root -> unroot + unroot -> Dir + ((Dir - Root) -> File)

}

One counterexample to `RootTop` is a directory that contains the root. In the Visualizer, this looks like



This page will cover all functionality of the visualizer.

Attention: The XML for this example can be downloaded `here`.

Tip: The model visualizer names atoms like `Atom$0`, `Atom$1`, etc. This can sometimes be hard to follow in the visualizer. To give things qualified names, instead write:

```
abstract sig Base {
  -- relations here
}

lone A, B, C, D extends Base {}

run {} for 2 Base
```

This will guarantee the values have better names.

3.2.1 Menu Bar Functions

File

- **Open/Export To**: Visualizations can either be exported to a graphviz diagram or to XML. Saved XML can be reloaded into the visualizer with the open option without needing to first reevaluate the spec.

Instance

- **Show Next Solution/Next:** Returns another solution that satisfies the model (or is a valid counterexample). The evaluator can only move forward in solutions, not backwards. If you would like to see a previous solution, you will need to rerun the model.

Themes

See *Reusing Themes*.

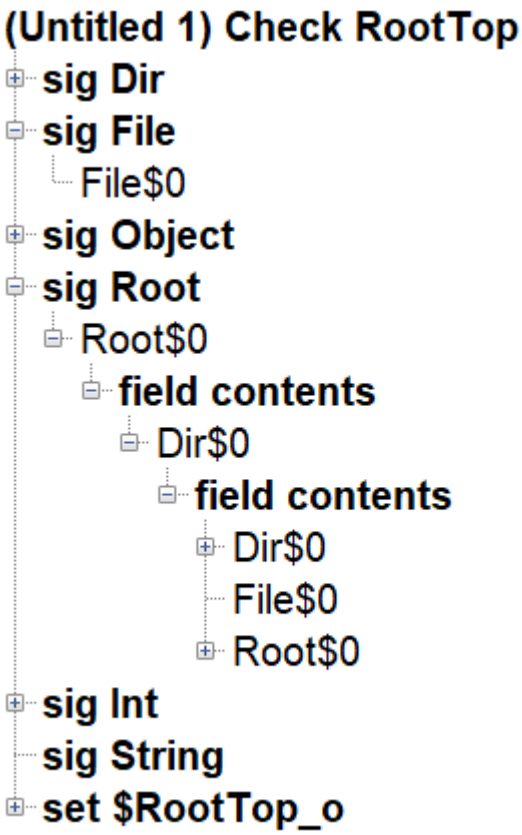
3.2.2 Output options

The standard view is the **visualizer**. The other three options are

- Text

```
seq/Int={0, 1, 2, 3}
String={}
none={}
this/Object={Dir$0, File$0, Root$0}
this/Dir={Dir$0, Root$0}
this/Dir<:contents={Dir$0->Dir$0, Dir$0->File$0, Dir$0->Root$0, Root$0->Dir
↪$0}
this/Root={Root$0}
this/File={File$0}
skolem $RootTop_o={Dir$0}
```

- Tree



- Table

this/Dir	contents
Dir^0	Dir^0
	File^0
	Root^0
Root^0	Dir^0

^{1}

this/Object	Dir^0	File^0	Root^0

Warning: The table view shows the atoms in a human readable form, for example, writing `Root`^{0} instead of `Root$0`. In the *evaluator*, however, you need to write `Root$0`.

Themes

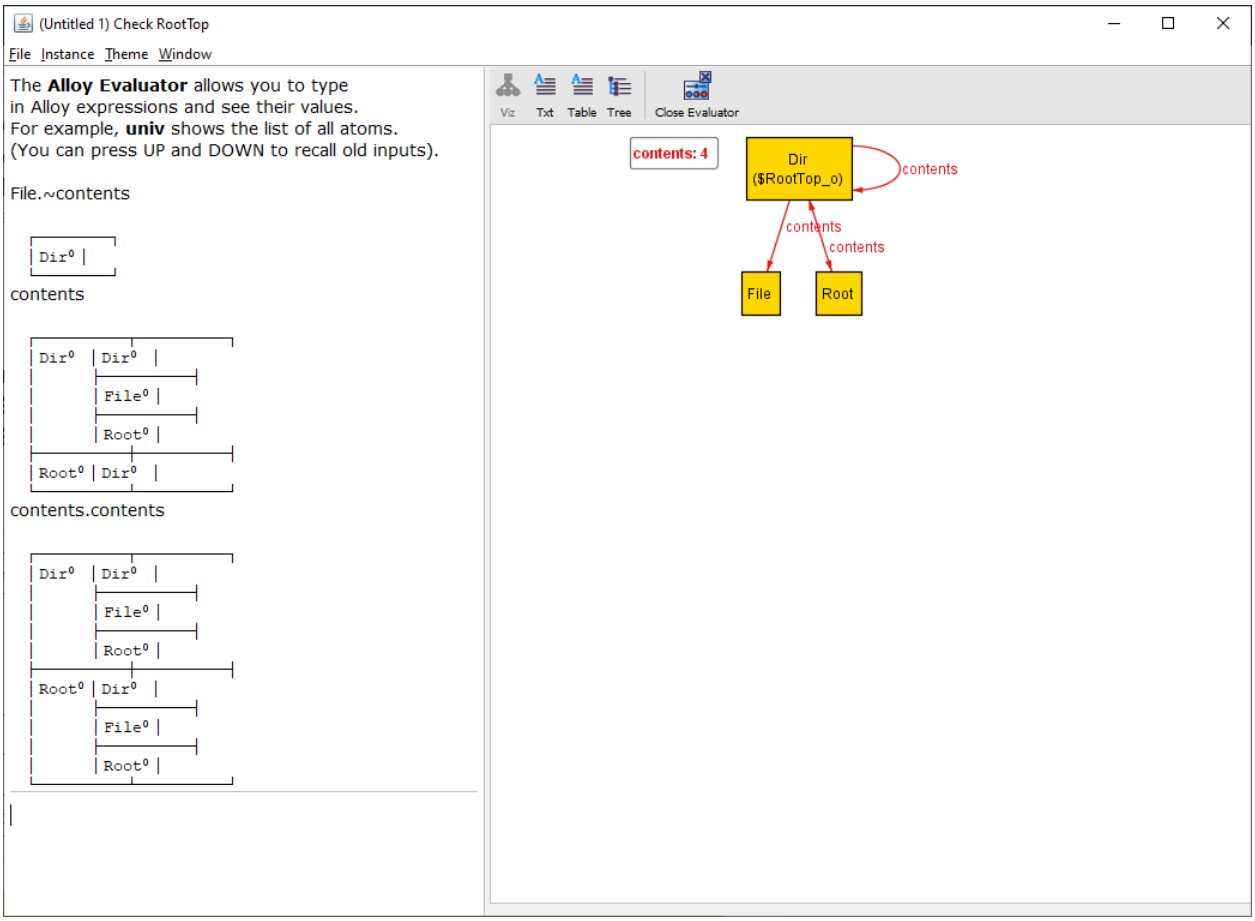
See *Themes*.

Magic Layout

Automatically generates an appropriate theme for the visualization.

3.2.3 Evaluator

The evaluator can be used to run arbitrary commands against the existing model. It cannot create new signatures, only investigate the current signatures and relations you currently have. You cannot define a new functions or predicates in the evaluator, only evaluate *Expressions*.



Note: While the evaluator can evaluate most expressions, it does not have the full capacity of the alloy analyzer. For example, polymorphic domain restriction will not work. Additionally, integer overflow will wrap instead of raise an error.

3.2.4 Projection

Projections break a complex model into multiple subviews. Instead of showing relations from the projected signature, each relation will be represented as a text label on the corresponding targets of the relation.

3.3 Themes

Diagrams produced in the *Visualizer* can be **themed**. Options are:

Note: All options will not be changed until you click the "Apply" button.

3.3.1 General Graph Settings

- **Use Original Atom Names**: Instead of showing atoms as `Atom`[1], the visualizer will show them as `Atom$1`. Default False.
- **Hide Private Sigs/Relations**: Hides signatures and relations that are internal to an imported module. Default True.
- **Hide Meta Sigs/Relations**: ???

3.3.2 Types and Sets

In this case, "set" refers to any nodes that are important to the visualizer, such as the results of *Commands*. All settings are hierarchical: any setting labelled `inherit` will use the same setting as the parent type/set. Settings for sets override settings for signatures.

The following settings are universal to all nodes:

- **Color**, **Shape**, and **Border**.
- **Name** (to the left of color): What to call the nodes. This does not effect the evaluator or tree, text, and table views.
- **Show**: Whether or not the node is visible in the visualization at all. Default True.
- **Hide Unconnected Nodes**: If a node doesn't have any relation to any other node, hide the node. Default True.

The following are specific to signatures:

- **Number Nodes**: Nodes with the same name and same type are suffixed with a number.
- **Project over this Sig**: See *Projection*.

The following are specific to sets:

- **Show as Labels:** Set membership is shown as a text label on the matching atoms.
- **Show in Relation Attributes**: If the element of the set appears in another node's textual attributes, also list that it belongs to the set.

3.3.3 Relations

For all of these options, we assume the relation is `rel: A -> B`

- **Show as Arcs**: Represent the relationship as an arrow from A to B. Default True.
- **Show as Attributes**: Represents the relationship as the label `rel: B` on node A. A relationship can be shown as both arcs and attributes at the same time. Default False.
- **Influence Layout**: If True, the visualizer will try to account for the arrows when laying out the node graph. If False, the visualizer will first lay out the graph, and then draw the relation. Default True.

- **Weight**: The visualizer will try to minimize the weights of the larger arrows first.
- **Layout Backwards**: Layout the graph as if the relation was `B -> A`. The final layout will still show the arrow as `A -> B`. Default False.
- **Merge Arrows**: If True, the relationship `A -> B + B -> A` with be represented as a single double-sided arrow. If false, the relationship is represented as two single-sided arrows. Default True.

Reusing Themes

Themes can be saved and loaded under the `theme` top menu bar. Themes can also be reset here.

3.4 Markdown

Thanks to Peter Kriens, Alloy supports Markdown syntax. Files written in Alloy Markdown have an `.md` extension, just like regular Markdown files. The Alloy Analyzer can read in Markdown files the same way it reads in `.als` files.

For an example of an Alloy Markdown file, see Peter Kriens's Dining Philosophers.

3.4.1 Header

A Markdown Alloy file must start with a YAML header: three dashes on the first line, followed by fields and values in YAML format, followed by three more dashes. For example:

```
---
title: Dining Philosophers
---
```

3.4.2 Alloy sections

After the YAML header, the Alloy parser interprets all text as Markdown. To start an Alloy section, use fenced code blocks with `alloy` as the language identifier. For example:

````
# Literate Programming

Let us change our traditional attitude to the construction of programs:␣
↪Instead
of imagining that our main task is to instruct a computer what to do, let us
concentrate rather on explaining to human beings **what** we want a computer␣
↪to do.

```alloy
sig Foo {} // this is Alloy syntax
```

If you find that you're spending almost _all_ your time on theory, start␣
↪turning
_some_ attention to practical things; it will improve your theories. If you␣
↪find
that you're spending almost all your time on practice, start turning some␣
↪attention
to theoretical things; it will improve your practice.
````

3.4.3 GitHub Pages support

The YAML header can be used in conjunction with GitHub Pages. GitHub Pages allow you to maintain a website via Github, the YAML header is then used to encode metadata information such as title and layout. This example website shows the dining philosophers example rendered using Jekyll and served from GitHub Pages.

CHAPTER 4

Modules

4.1 boolean

Emulates boolean variables by creating `True` and `False` atoms.

```
-- module definition
module util/boolean

abstract sig Bool {}
one sig True, False extends Bool {}

pred isTrue[b: Bool] { b in True }

pred isFalse[b: Bool] { b in False }
```

In our code:

```
-- our code
open util/boolean

sig Account {
  premium: Bool
}
```

Booleans created in this matter are not "true" booleans and cannot be used as part of regular *Constraints*, IE you cannot do `bool1 && bool2`. Instead you must use the dedicated boolean predicates, below. As such, `boolean` should be considered a proof-of-concept and is generally **not recommended** for use in production specs. You should instead represent booleans using *subtyping*.

4.1.1 Functions

All of the following have expected logic, but return `Bool` atoms:

- `Not`

- `And`
- `Or`
- `Xor`
- `Nand`
- `Nor`

So to emulate `bool1 && bool2`, write `bool1.And[bool2].isTrue`.

4.2 graph

Graph provides predicates on relations over a parameterized signature.

```
open util/graph[Node]

sig Node {
  edge: set Node
}

run {
  dag[edge]
}
```

Notice that graph is parameterized on the **signature**, but the predicate takes in a **relation**. This is so that you can apply multiple predicates to multiple different relations, or different subsets of the same relation. The graph module uses some specific terminology:

This means that in a completely unconnected graph, every node is both a root and a leaf.

4.2.1 Functions

`fun roots[r: node-> node]`

Return type `set Node`

Returns the set of nodes that *are not connected to* by any other node.

> **Warning:** this is *not* the same meaning of *root* as in the *`rootedAt`* predicate! For the predicate, a *root* is a node that transitively covers the whole graph. Interally, `util/graph` uses `rootedAt` and not `roots`.

`fun leaves[r: node-> node]`

Return type `set Node`

Returns the set of nodes that *do not connect to* any other node.

Note: If `r` is empty, `roots[r] = leaves[r] = Node`. If `r` is undirected or contains enough self loops, `roots[r] = leaves[r] =none`.

`fun innerNodes[r: node-> node]`

Returns All nodes that aren't leaves

Return type `set Node`

4.2.2 Predicates

`pred undirected [r: node->node]`
`r` is *symmetric*.

`pred noSelfLoops[r: node->node]`
`r` is *irreflexive*.

`pred weaklyConnected[r: node->node]`
For any two nodes A and B, there is a path from A to B or a path from B to A. The path may not necessarily be bidirectional.

`pred stronglyConnected[r: node->node]`
For any two nodes A and B, there is a path from A to B *and* a path from B to A.

`pred rootedAt[r: node->node, root: node]`
All nodes are reachable from `root`.

> **Warning:** this is *not* the same meaning of *root* as in the *roots* function! For the function, a *root* is a node no node connects to. Interally, `util/graph` uses `rootedAt` and not `roots`.

`pred ring [r: node->node]`
`r` forms a single cycle.

`pred dag [r: node->node]`
`r` is a DAG (directed acyclic graph): there are no self-loops in the transitive closure.

`pred forest [r: node->node]`
`r` is a dag and every node has at most one parent.

`pred tree [r: node->node]`
`r` is a forest with a single root node.

`pred treeRootedAt[r: node->node, root: node]`
`r` is a tree with node `root`.

4.3 ordering

Ordering places an ordering on the parameterized signature.

```
open util/ordering[A]

sig A {}

run {
  some first -- first in ordering
  some last  -- last in ordering
  first.lt[last]
}
```

`ordering` can only be instantiated once per signature. You can, however, call it for two different signatures:

```
open util/module[Thing1] as u1
open util/module[Thing2] as u2

sig Thing1 {}
sig Thing2 {}
```

Warning: `ordering` forces the signature to be *exact*. This means that the following model has no instances:

```
open util/ordering[S]

sig S {}

run {#S = 2} for 3
```

In particular, be careful when using `ordering` as part of an assertion: the assertion may pass because of the implicit constraint!

See also:

Module *time*

Adds additional convenience macros for the most common use case of ordering.

`Sequences`

For writing ordered relations vs placing top-level ordering on signatures.

4.3.1 Functions

`fun first`

Returns The first element of the ordering

Return type `elem`

See Also `last`

`fun prev`

Return type `elem -> elem`

See Also `next`

Returns the relation mapping each element to its previous element. This means it can be used as any other kind of relation:

```
fun is_first[e: elem] {
  no e.prev
}
```

`fun prevs[e]`

Returns All elements before `e`, excluding `e`.

Return type `elem`

See Also `nexts`

`fun smaller[e1, e2: elem]`

Returns the element that comes first in the ordering

See Also larger

fun min[es: set elem]

Returns The smallest element in `es`, or the empty set if `es` is empty

Return type `lone elem`

See Also max

4.3.2 Predicates

pred lt[e1, e2: elem]

See Also `gt`, `lte`, `gte`

True iff `e1 in prevs[e2]`.

4.4 relation

All functions and predicates in this module apply to any binary relation. *univ* is the set of all atoms in the model.

4.4.1 Functions

fun dom[r: univ->univ]

Return type `set univ`

Returns the domain of `r`. Equivalent to `univ.~r`.

fun ran[r: univ->univ]

Return type `set univ`

Returns the range of `r`. Equivalent to `univ.r`.

4.4.2 Predicates

pred total[r: univ->univ, s: set]
True iff every element of `s` appears in `dom[r]`.

pred functional[r: univ->univ, s: set univ]
True iff every element of `s` appears *at most once* in the left-relations of `r`.

pred function[r: univ->univ, s: set univ]
True iff every element of `s` appears *exactly once* in the left-relations of `r`.

pred surjective[r: univ->univ, s: set univ]
True iff `s in ran[r]`.

pred injective[r: univ->univ, s: set univ]
True iff no two elements of `dom[r]` map to the same element in `s`.

pred bijective[r: univ->univ, s: set univ]
True iff every element of `s` is mapped to by *exactly one* relation in `r`. This is equivalent to being both injective and surjective. There may be relations that map to elements outside of `s`.

pred bijection[r: univ->univ, d, c: set univ]
True iff exactly `r` bijects `d` to `c`.

pred reflexive[r: univ -> univ, s: set univ]
`r` maps every element of `s` to itself.

pred irreflexive[r: univ -> univ]
`r` does not map any element to itself.

pred symmetric[r: univ -> univ]
`A -> B in r implies B -> A in r`

pred antisymmetric[r: univ -> univ]
`A -> B in r implies B -> A notin r`. This is stronger than `not symmetric`: *no* subset of `r` can be symmetric either.

pred transitive[r: univ -> univ]
`A -> B in r and B - > C in r implies A -> C in r`

pred acyclic[r: univ->univ, s: set univ]
`r` has no cycles that have elements of `s`.

pred complete[r: univ->univ, s: univ]
`all x,y:s | (x!=y => x->y in (r + ~r))`

pred preorder[r: univ -> univ, s: set univ]
`reflexive[r, s] and transitive[r]`

pred equivalence[r: univ->univ, s: set univ]
`r` is reflexive, transitive, and symmetric over s.

pred partialOrder[r: univ -> univ, s: set univ]
`r` is a partial order over the set `s`.

pred totalOrder[r: univ -> univ, s: set univ]
`r` is a total order over the set `s`.

4.5 ternary

`util/ternary` provides utility functions for working with 3-arity *Multirelations*. All functions return either an element in the relation or a new transformed relation.

Table 1: `util/ternary`

f	f[a -> b -> c]
dom	a
mid	b
ran	c
select12	a -> b
select13	a -> c
select23	b -> c
flip12	b -> a -> c
flip13	c -> b -> a
flip23	a -> c -> b

4.6 time

Automatically imports an ordered `Time` signature to your spec.

> **Warning:** `Time` internally uses the *`ordering`* module. This means that the signature is forced to be exact.

See also:

Module *`ordering`*

4.6.1 Macros

`let dynamic[x]`

Arguments

- **`x`** (*`sig`*) – any signature.

Expands to `x one -> Time`

`dynamic` can be used as part of a signature definition:

```
open util/time

abstract sig Color {}
            one sig Red, Green, Yellow extends Color {}

sig Light {
  , state: dynamic[Color]
}
```

At every `Time`, every `Light` will have exactly one color.

`let dynamicSet[x]`

Arguments

- **`x`** (*`sig`*) – any signature.

Expands to `x -> Time`

Equivalent to `dynamic`, except that any number of elements can belong to any given time:

```
open util/time

sig Keys {}

one sig Keyboard {
  pressed: dynamicSet[Keys]
}
```

CHAPTER 5

Techniques

5.1 Boolean Fields

Often software properties are expressed as booleans, either true or false. But Alloy doesn't have a native boolean type. There are two ways to emulate this.

For this example we want something akin to

```
sig Account {
  premium: bool -- invalid, not a real type
}
```

5.1.1 Subtyping

Using *in* subtypes is the standard way to model booleans:

```
sig Account {}
sig PremiumAccount in Account {}
```

Then the boolean can be tested with `a in PremiumAccount`. This method has a number of advantages:

- `PremiumAccount` and `Account - PremiumAccount` can be used in the signatures of other fields.
- Premium accounts can have additional fields.
- You can create a custom *theme* for premium accounts.
- The analyzer will generally be faster.

lone fields

Booleans can also be represented using `lone`:

```
one sig Premium {}
sig Account {
  , premium: lone Premium
}
```

Then the boolean can be tested with `some a.premium`, and the set of all premium accounts is `premium.Premium`. Using lone is somewhat simpler than using a subtype, but it's less flexible overall, and the boolean cannot be used in the fields of other signatures.

The `Boolean` module

If subtyping is insufficient, you can also use the *`boolean`* module. This is generlaly not recommended.

5.2 Dynamic Models

A **dynamic model** represents something changing over time. Alloy does not have a first-class notion of time, and dynamics must be encoded as part of the relationships. This is normally done by placing an *`ordering`* on some signature, which then used to emulate time. There are several common ways of doing this.

See also:

`time`

A module with some utility macros to better model time.

5.2.1 Changing Object Signature

When only one entity is changing and there is only one instance of the signature to consider, we can place the ordering on the signature itself. Then each atom of that signature represents the state of the entity at a different point in time.

Listing 1: Full spec downloadable `here`

```
open util/ordering[Light] as ord

abstract sig Color {}
one sig Red, Yellow, Green extends Color {}

sig Light {
  color: Color
}
```

While there are multiple `Light` atoms, they all represent the same physical "traffic light", just at different points in time.

The dynamic model is modeled by a state change predicate, which relates each light to the next light in the sequence.

Listing 2: Full spec downloadable `here`

```
pred change_light[l: Light] {

  let l' = l.(ord/next) {
    l.color = Red => l'.color = Green
    l.color = Green => l'.color = Yellow
    l.color = Yellow => l'.color = Red
```

(continues on next page)

(continued from previous page)

```
  }

}
```

Conventionally *ordering* is *not* namespaced. We do it here to make the imported functions clearer.

Traces

A *trace* is a *fact* that describes how the system will evolve, by constraining "valid models" to ones where the system evolves properly.

Listing 3: Full spec downloadable here

```
fact Trace {
  ord/first.color = Red

  all l: Light - ord/last |
    change_light[l]
}
```

first and last are from the *ordering* module. This sets the first light to red and every subsequent light to the next color in the chain.

Note: We write Light - last because no last.next, so the predicate would be false.

5.2.2 Time Signatures

For more complex specifications we use a Time signature. This is useful if we have multiple things that are changing or multiple properties that can change. The ordering is placed on Time and the signature fields are all related to that time.

Listing 4: Full spec downloadable here

```
open util/ordering[Time]

sig Time {}
```

Representing Boolean Properties

If the changing value is a boolean, the best way to represent that is to have a field that is set Time, which represents the times where that field is true:

Listing 5: Full spec downloadable here

```
sig Key {
  pressed: set Time
}

pred press[k: Key, t: Time] {
  t not in k.pressed
```

(continues on next page)

(continued from previous page)

```
  t.next in k.pressed
}

pred release[k: Key, t: Time] {
  t in k.pressed
  t.next not in k.pressed
}

pred changed[k: Key, t: Time] {
  press[k, t] or release[k, t]
}
```

We will assume that only one key can be pressed or released at a given time. This means that the trace must specify that some key changes *and also* that no other key changes.

Listing 6: Full spec downloadable `here`

```
fact Trace {
  no first.~pressed
  all t: Time - last |
    some k: Key {
      changed[k, t]
      all k': Key - k |
        not changed[k, t]
    }
}
```

Note: We could have instead written it this way, using `one` instad of `some`:

```
all t: Time - last |
  one k: Key |
    changed[k, t]
```

Which would have enforced that no other keys changed by default. Using `one` in these contexts can be error-prone, so it's avoided for the purposes of this page.

Conventionally, state-change predicates are written to take two time parameters, where the trace then passes in both `t` and `t.next`.

```
pred release[k: Key, t, t': Time] {
  t in k.pressed
  t' not in k.pressed
}
```

Representing Arbitrary Properties

Information beyond booleans can be encoded with *Multirelations*.

Listing 7: Full spec downloadable `here`

```
open util/ordering[Time]

sig Time {}
```

(continues on next page)

(continued from previous page)

```
sig User {
  -- current place in time
  , at: Page one -> Time

  -- all visited pages
  , history: Page -> Time
}
```

Writing `Page one -> Time` indicates that for every Person and Time, there is exactly one page. Writing `Page -> Time` also any number of pages per person/time.

Note: There's no innate reason why we use `Page -> Time` instead of `Time -> Page`. However, making Time the end of the multirelation is conventional.

Listing 8: Full spec downloadable `here`

```
sig Page {
  -- pages reachable from this page
  link: set Page
}

pred goto[u: User, p: Page, t, t': Time] {
  -- valid page to change to
  p in u.at.t.link

  -- change pages
  p = u.at.t'

  -- save new page in history
  u.history.t' = u.history.t + p
}
```

When using multirelations of form `rel = A -> B -> Time`, we get the value of `b` at time `t` with `a.rel.t`

Listing 9: Full spec downloadable `here`

```
pred stay[u: User, t, t': Time] {
  u.at.t' = u.at.t
  u.history.t' = u.history.t
}
```

`stay` is a "stuttering" predicate which makes it valid for a user to not change at the next time step. Without it `goto` would have to be true for every single user at every time in the trace.

Listing 10: Full spec downloadable `here`

```
fact trace {
  -- everybody starts with their initial page in history
  at.first = history.first
  all t: Time - last |
    let t' = t.next {
      all u: User |
        u.stay[t, t'] or
```

(continues on next page)

(continued from previous page)

```
        some p: Page | u.goto[p, t, t']
    }
}
```

5.2.3 Common Issues

Incomplete Trace

The trace must fully cover what happens to all dynamic signatures. If not, weird things happen. Consider the following alternate trace for the browsing model:

```
fact Trace {
  at.first = history.first -- everybody starts with their initial page in history
  all t: Time - last |
    let t' = t.next {
      one u: User |
        u.stay[t, t'] or
        some p: Page | u.goto[p, t, t']
    }
}
```

This is true iff *exactly one* User either stays or goes to a valid page. This allows them to do anything that's not covered by the two predicates. A user may go to an unlinked page, or stay on the same page but change their history to include a page they never visited.

"No Models Found"

`ordering` implicity makes the ordered signature *exact* and this cannot be overridden. If a dynamic spec does not exist, it's likely due to this.

```
open util/ordering[State]

sig State {}

run {#State = 2} -- no models
```

5.2.4 Limitations

There are some limitations to what we can model in a dynamic system.

- Alloy cannot tell if a system has **deadlocked**. A deadlock is when there is no valid next state as part of the trace. If the trace is encoded as a fact, then the entire model is discarded. If the trace is encoded as a predicate, Alloy will provide any model that doesn't match the trace as a counterexample.
- Alloy cannot test that some property is guaranteed to happen in infinite time, aka **liveness**.
- Alloy cannot emulate **fair** dynamic systems.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

U

W