ALLOY

↗ Notebook	📒 SOFTWARE ENGINEERING 2
Useful Documents	CourseSlides.pdf EducationalModuleGit TextBook Exam-Example.pdf Class Diagram.html Alloy Examples.pdf Alloy6_Exercise.pdf Alloy6_Dynamic_Exercise.pdf

INFO/LINKS

https://alloytools.org/k

Alloy4Fun

Syntax

▼ Sig statements

Syntax for writing sig statements

In its simplest form, a

signature

looks like this:

`sig name { }`

Following each signature declaration, we find a signature body whose within a pair of braces. In the signature body, we can define a series of relations for which the set defined by the signature is the domain. For instance, in the **FSObject** signature body, we define a relation called "**parent**" which relates **FSObject**s to **Dir**s.

A sig may also have fields, which look like this (brackets indicate optional text):

```
sig name {
  //most fields look like this:
  field-name-1: field-type-1,
  //A signature may refer to signatures not
  // defined until later in the model.

  //multiple fields with the same type look like this:
  field-name-2a, field-name-2b: field-type-2
}
```

A signature extending another signature is a partition of that other signature:

```
sig A extends C {
  //fields
}

sig B extends C {
  //fields
}
```

means that **A** and **B** are disjoint subsets of C. If **A** and **B** have identical bodies, then we can get the same effect by writing

```
sig A, B extends C {
  //fields
}
```

We can use **in** instead of **extends** to get a slightly different effect -- **in** declares that one sig is a subsets of another, but does not enforce that it is disjoint from other sigs with the same parent. That is,

```
sig A in C {
  //fields
}

sig B in C {
  //fields
}
```

means that **A** and **B** are both subsets of **C**, but they are not necessarily disjoint. Once again, if **A** and **B** have identical bodies, we can use the shorthand

```
sig A, B in C {
  //fields
}
```

In general, two (or more) sigs which have the same definition may be declared together, like this:

```
sig name1, name2 {
  //fields
}
```

the **abstract** keyword is analogous to the **abstract** keyword of OO languages like Java, although it is not a perfect analogy. Consider some signature defined as follows:

```
abstract sig Foo {
  //fields
}

sig Bar extends Foo {
  //fields
}

sig Cuz extends Foo {
  //fields
}
```

By using the **abstract** keyword, we make guarantee that there will be no **Foo** that is not also either a **Bar** or a **Cuz**. However, had there been no signatures declared as subsets of **Foo** or extending it, then this restriction is relaxed; if a signature has no subsets then writing **abstract** has no effect.

You can force there to always be exactly one instance of (element in) a signature by using the **one** keyword. This also makes sure that every reference to this signature refers to the same atom.

```
one sig name {
    // fields
}
```

Examples:

```
sig FSObject { }

sig Dir {
    contents: set FSObject
}

sig FileSystem {
    root: one Dir,
    objects: set FSObject,
    contents: Dir lone->set FSObject,
    parent: FSObject set->lone Dir
}
```

▼ Quantifiers

The Alloy quantifiers are

- **all x:X | formula** every **x** of type **X** satisfies **formula**. If there are no **X**, then this statement is trivially true.
- **some x:X | formula** one or more **x** of type **X** satisfy **formula**. If there are no **X**, then this statement is trivially false.
- **no x:X | formula** exactly zero **x** of type **X** satisfy **formula**
- **one x:X | formula** exactly one **x** of type **X** satisfies **formula**
- **lone x:X | formula** either zero or one **x** of type **X** satisfies **formula**

Note that a 'formula' is something that evaluates to a boolean value, as opposed to an 'expression' which evaluates to a (relational) value. With "some", "no", one", and "lone", you can also apply them to a set or a relation like this:

- **some X** there is at least one **X**
- **no X** there are no **X**'s
- **one X** there is exactly one **X**
- **lone X** there are either zero or one **X**'s

The basic format for a quantifier is

```
  quantifier variable:type | formula
```

where the formula may includes references to the quantifier variable. An alternative notation is:

```
  quantifier variable:type { formula }
```

Examples:

```
//Does any directory contain itself?
  some d: Dir | d.parent = d

//every file is some directory
  all f:File | some d:Dir | f.parent = d

//no two directories have exactly the same contents
  no dir1, dir2: Dir | dir1!=dir2 && dir1.contents=dir2.contents
```

The above features can, of course, be combined to produce arbitrarily complex formulas.

However, be warned that nested quantifiers may cause your model to become intractable. In generaly, try not to stack more than two or three of them together.

▼ Relational Join

Relational Join

a.b is the composition of the relations **a** and **b**. For each tuple in **a** of the form (a0, a1, ..., an, X), and each tuple in **b** of the form (X, b1, ..., bn), the new relation **a.b** contains a tuple (a0, a1, ..., an, b1, ... bn). Only the last atom in tuples of **a** are matched with the first atom in tuples of **b**. Note that the matched atom is removed from the resulting tuple.

Due to some clever definitions of Semantics, it can also be used to access fields of a signature. See levels of understanding for examples of different ways to interpret the dot.

▼ How to think about an Alloy model: 2 levels

There are three basic levels of abstraction at which you can read an Alloy model.

- The highest level of abstraction is the OO paradigm. This level is helpful for getting started and for understanding a model at a glance; the syntax is probably familiar to you and usually does what you'd expect it to do.

- The middle level of abstraction is set theory. You will probably find yourself using this level of understanding the most (although the OO level will still be useful).
- The lowest level of abstraction is atoms and relations, and corresponds to the true semantics of the language. However, even advanced users rarely think about things this way.

Consider the following excerpt. A signature **sig S**, extending **E**, has a field **F** of type **T**.

```
sig S extends E {
    F: one T
}

fact {
    all s:S | s.F in T
}
```

The fragment can be roughly interpreted in an Object Oriented sense.

- **S** is a class
- **S** extends its superclass **E**
- **F** is a field of **S** pointing to a **T**
- **s** is an instance of **S**
- **.** accesses a field
- **s.F** returns something of type **T**

The fragment can be safely read as being about sets, elements, and relations among those sets and elements. Most users find this way of thinking intuitive after a little practice and does not lead to errors as the OO approach occasionally does.

- **S** is a set (and so is **E**)
- **S** is a subset of **E**
- **F** is a relation which maps each **S** to exactly one **T**
- **s** is an element of **S**
- **.** composes relations
- **s.F** composes the unary relation **s** with the binary relations **F**, returning a unary relation of type **T**

In some sense, we are modeling a field of a class as a relation which maps instances of that class to instances of that field.

▼ Set Operations

Alloy supports the standard set operations:

- union (+): **t** is in **p+q** if and only if **t** is in **p** *or* **t** is in **q**.
- intersection (**&**): **t** is in **p&q** if and only if **t** is in **p** *and* **t** is in **q**.
- set subtraction (): **t** is in **p-q** if and only if **t** is in **p** *but* **t** is *not* in **q**.
- set membership/subset (**in**): Set membership and subsets are both denoted **in**. The same symbol is used, since Alloy does not distinguish between atoms and singleton sets. To understand why atoms and singleton sets are treated this way, you should read more about how everything is a relation in Alloy.

▼ Syntax for *appended facts*

An appended fact is written by placing constraints in curly braces immediately after a **sig** statement. They can make a model clearer by logically grouping related constraints together.

```
sig name {
  // fields of the signature
}{
  // appended fact constraints
}
```

Any field mentioned in the appended fact has an implied "**all this:name |**" at its beginning, and an implied "**this.**" in front of it. Thus:

```
sig Person {
    closeFriends: set Person,
    friends: set Person
} {
    closeFriends in friends
}
```

is really saying

```
sig Person {
    closeFriends: set Person,
    friends: set Person
}

fact {
    all x:Person | x.closeFriends in x.friends
}
```

Consequently, appended fact can be more concise (and thus clearer) than the equivalent non-appended fact. However, there is also a danger in forgetting that *every field* mentioned in an appended fact is prepended with "**this.**".

▼ Assert

While a **fact** is used to *force* something to be true of the model, an **assert** is a claim that something *must already* be true due to the rest of the model. Assertions are verified (or falsified) using a **check** statement.

```
assert name-of-assertion {
  // list of constraints
}
```

example:

```
assert acyclic {
  no d: Dir | d in d.^contents
}
```

Each assertion *must* have a name, since we will need to refer to them by name in **check** statements.

▼ Syntax for *check* statements

With Alloy, you can automatically verify (up to a specified scope) or falsify (with a counterexample) an **assert** statement, by using a **check** command.

```
check name-of-assert for integer
```

The integer defines the scope -- the maximum size of each set in the model that will be considered. Specifically, it is the maximum size of the top-level signatures (sets). If there is any solution in that scope or less, Alloy is guaranteed to find it. Thus, we say Alloy is 'complete up to scope'.

For example, the statement

```
check acyclic for 5
```

checks the **acyclic** assertion, and tells Alloy to only examine cases where there are at most 5 top level sigs/sets.

Larger scopes take longer to solve, and often result in more complicated examples. Thus it is good practice to reduce the scope as low as it can go but still get a solution, before trying to visualize solutions.

You can refine you definition of scope with the **but** keyword. For example:

```
check acyclic for 5 but 1 fileSystem, 7 FSObject
```

The **but** command lets you specify exceptions to the main scope. In this case, Alloy will only examine 1 file system, up to 7 file system objects, and up to 5 of any other set.

▼ Multiplicity Markings

When defining a field of a signature to be a relation, you write something like this:

```
sig University {
  groupMember: Student -> Professor
}
```

Here, each University knows which student is a member of which professor's research group. That is, there is a relation from Universities to Students to Professors. By writing multiplicity markings to the declaration of the relation, we can add additional constraints in a concise way. For example,

```
sig University {
  groupMember: Student -> one Professor
}
```

means that each student is in exactly one professor's research group. If we instead wrote this,

```
sig University {
  groupMember: Student -> lone Professor
}
```

Then we are saying that some students may be in no research group, but no student is in more than one. Lastly, we can write

```
sig University {
  groupMember: Student -> some Professor
}
```

to say that each student is in one or more research groups. When no multiplicity markings are present, there are not constraints.

You can also write multiplicity markings before the arrow. For example,

```
sig University {
  groupMember: Student some -> Professor
}
```

This says that every professor has at least one student in his or her research group. We can put markings on both sides of the arrow as such:

```
sig University {
  groupMember: Student some -> lone Professor
}
```

Here, every research group has at least one student, and each student is in up to one research group.

You can also place the **lone** and **one** multiplicity markings on signatures to control how many copies of that signature can exist (or, really, how many elements can ever be in that set defined by that signature). For example:

```
one sig Root {...}
lone sig University {...}
some sig InitialState {...}
```

▼ **pred** (predicate) statements

```
pred name [parameter1:domain1, parameter2:domain2] {
    constraint1
    constraint2
    constraint3
}
```

If the inputs satisfy all of the constraints listed in the body, then the predicate evaluates to true. Otherwise it evaluates to false.

In an imperative model, you might ask yourself "*how can I make X happen?*". In a declarative model (like Alloy), you ask yourself "*how would I recognize that X has been accomplished?*".

One reflection of this is that predicates in Alloy only ever evaluate to *true* or *false*, while function may evaluate to relations.

A predicate returns true or false. A similar construct is a function which returns a value.

Remind me what a function is.

▼ **fun** (function) statements

```
fun name [parameter1: domain1, paramter2: domain2, ..., parameterN: domainN]: domain {
   [body must evaluate to a value in Domain]
}
```

For example:

```
fun between [lower:Int, upper:Int]: Int {
   {answer: Int | lower<answer && answer<upper }
}
```

Notice that in this case we have written a set comprehension as the body of the function. If there is more than one solution to the comprehension, then the function will return the set of all solutions.

You can put constraints in the parameter list as reminders to yourself, although they are not actually enforced. For example:

```
fun name [param: some Int]: lone Int {
    ...
}
```

A function evaluates to a value. A similar construct is a predicate which returns either true or false, but is otherwise written the same.

Remind me what a function is.

▼ Relations as Ordered Tuples

Alloy represents relations as sets of ordered tuples, and it can some times be useful to think of them this way. Intuitively, this is just an exhaustive list of every way to legally fill in the relation.

A (binary) relation mapping file system objects to their parent directories is really a set of ordered pairs. The first entry in each pair is a file system object and the second is its parent directory. In one instance, it's entries might look like this:

```
{< fileA, DirA >,
 < fileB, DirB >,
 < fileC, DirA >}
```

Now consider a relation that maps pairs of nodes in a network to their cost. This relation would be said to have *arity 3*, and its tuples might look like this:

```
{< nodeA, nodeB, 100 >,
 < nodeA, nodeC, 25 >,
 < nodeB, nodeD, 37 >,
 < nodeC, nodeD, 14 >}
```

Of course, if we didn't write sufficient constraints on the model, the relaion might have more than one cost per pair:

```
{< nodeA, nodeB, 100 >,
 < nodeA, nodeB, 50 >,
 < nodeA, nodeC, 14 >,
 < nodeC, nodeA, 37 >}
```

Since relations are just sets, you can take intersections, unions, set differences, and so on on them, just like any other sets. For instance, you might have a relation **roads**, which is an arity-2 relation on cities, stating which one have roads connecting them. It's tuples might look like this:

```
{< cityA, cityB >
 < cityA, cityC >
 < cityB, cityD >
 < cityC, cityD >}
```

There might be another relation **dirtRoads** which is also a relation on pairs of cities. We might write the constraint

```
  fact physicalWorld {
//dirt roads are roads
    dirtRoads in roads

//roads are two-way
    roads = ~roads
    dirtRoads = ~dirtRoads
  }
```

The ~ operator reverses a relation by reversing the tuples in the relation. Thus our earlier sample falue for **roads** did not satisfy the **physicalWorld** fact. It should have been

```
{< cityA, cityB >
 < cityB, cityA >
 < cityA, cityC >
 < cityC, cityA >
 < cityB, cityD >
 < cityD, cityB >
 < cityC, cityD >
 < cityD, cityC >}
```

We also might want to talk about non-dirt roads, by writing something like this:

```
all r:(roads - dirtRoads) | r.foo ...
```

If **dirtRoads** were

```
{< cityC, cityD >
 < cityD, cityC >}
```

then **(roads - dirt**

▼ Temporal Connectors

FUTURE	PAST
ALWAYS	HISTORICALLY
EVENTUALLY	ONCE
AFTER	BEFORE
UNTIL	SINCE
RELEASES	TRIGGERERD
;	(NO DUAL)

Historically F

1	2	3	4	i	6	7	8

Always F

1	2	3	4	5	i	7	8

Eventually F

1	2	3	4	5	i	7	8

For some $k \geq i$ (becomes true in at least one value in the future)

Once F

1	2	3	4	i	6	7	8

Has happened at least once in the past (included i)

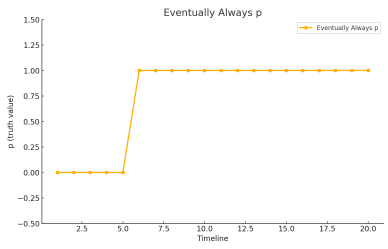
F Until G

1	2	3	4	5	i	7	8

F is true until G happens (G can happen also in “i”, and if G is true in k > i then F is true from i)

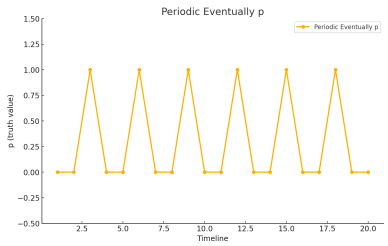
Eventually Always F

describes a stabilization property: after a certain point, F becomes permanently true.



Always Eventually F

describes a recurrence property: F will continue to become true repeatedly throughout the system's behavior.



After F

1	2	3	4	5	i	7	8

Same as saying F'

Before F

1	2	3	4	5	i	7	8

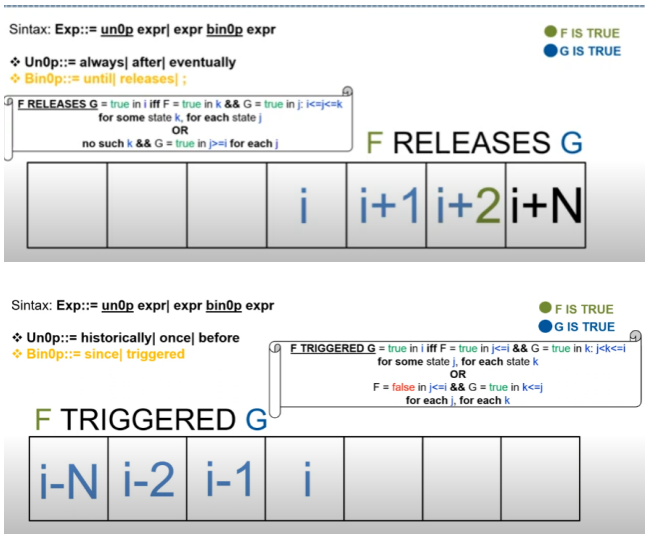
False in state 0

F Since G

1	2	3	4	5	i	7	8

G can be true in *i*

F ; G ⇒ D and after G



▼ Examples

```
sig Person {
  var liveness: Liveness
}
```

```
enum Liveness { Alive, Dead}

pred Die [p: Person] {
 p.liveness = Alive
 p.liveness' = Dead    // next state
}

fact NoResurrection{
  always ( all p:Person | p.liveness = Dead
        implies
                   always p.liveness = Dead
                 )
}

// We need both "always" -> the first one is used to say that
// the fact stands even for future states.
// The second states that the future states of Person are always Dead

assert NoResurrection{
  always ( all p: Person | p.liveness = Dead
                    implies after
                  p.liveness = Dead
                 )
}

// check NoResurrection for 3

fact noImmortality {
  always ( all p:Person | p.liveness = Alive
                   implies after ( eventually p.liveness = Dead )
              )
}

fact NoDeadThenAlive{
   always (
        all p:Person | p.liveness = Alive implies historically p.liveness = Alive
                  )
}

fact DeadSinceDeath {
  always (
                    all p:Pearson |  p.liveness = Dead implies once Die[p]
                )
}

fact AliveUntilDeath {
  always ( all p:Person | p.liveness = Alive implies (p.liveness=Alive until p.liveness =

}


assert AliveUntilDeath2 {
  always ( all p:Person | p.liveness = Alive  implies ( Die[p] releases p.liveness = Alive
```

```
}

assert DeadSinceDeath2 {
 always (all p:Person | p.liveness = Dead implies ( Die[p] triggered p.liveness = Dead ) )
}

assert DeadSinceDeath2 {
  always ( all p: Person | p.liveness = Dead implies ( p.liveness= Dead since Die[p] )  )
}

run{#Person = 4 and some p: Person | (p.liveness = Alive; p.liveness = Alive; p.liveness =
```

▼ disj

`disj` can be prepended to any multiplicity to guarantee 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.

▼ iff

`iff` (`<=>`)

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

▼ impliesed

Given arbitrary formulas `P`, `Q`, `T`, `P implies Q else T` is the same as `(P and Q) or (not P and T)`.
Without the
`else` , if P is false, we can't say anything about Q

▼ **Set comprehension**

- The binary relation between directories and the contained objects

```
{ d : Dir, o : Object | some d.contents.o }
= {(Root,File),(Root,Dir0),(Dir0,Dir1)}
```

~

Examples

▼ File System Model

```
// A file system object in the file system
sig FSObject { parent: lone Dir }
```

```
// A directory in the file system
sig Dir extends FSObject { contents: set FSObject }

// A file in the file system
sig File extends FSObject { }

// A directory is the parent of its contents
fact { all d: Dir, o: d.contents | o.parent = d }

Given any directory d and any (FSObject) o in d's contents, d must be the parent of o.
// alternative :
fact { all d: Dir, o: d.contents | o.parent = d }

fact { all d: Dir | all o: d.contents | o.parent = d }

fact { all d: Dir, o: FSObject | o in d.contents => o.parent = d }

fact { all d: Dir | all o: FSObject | o in d.contents => o.parent = d }

// All file system objects are either files or directories
fact { File + Dir = FSObject }

or
abstract sig FSObject {...}

// There exists a root
one sig Root extends Dir { } { no parent }

fact { FSObject in Root.*contents }

// The contents path is acyclic
assert acyclic { no d: Dir | d in d.^contents }

// Now check it for a scope of 5
check acyclic for 5

// File system has one root
assert oneRoot { one d: Dir | no d.parent }

// Now check it for a scope of 5
check oneRoot for 5

// Every fs object is in at most one directory
assert oneLocation { all o: FSObject | lone d: Dir | o in d.contents }

// Now check it for a scope of 5
check oneLocation for 5
```

```
// A file system object in the file system
abstract sig FSObject { }

// File system objects must be either directories or files.
```

```
sig File, Dir extends FSObject { }

// A File System
sig FileSystem {
  root: Dir,
  live: set FSObject,
  contents: Dir lone-> FSObject,
  parent: FSObject ->lone Dir
}{
  // root has no parent
  no root.parent
  // live objects are reachable from the root
  live in root.*contents
  // parent is the inverse of contents
  parent = ~contents
}

pred example { }

run example for exactly 1 FileSystem, 4 FSObject
```

▼ Family

```
abstract sig Person {
 father: lone Man,
 mother: lone Woman
}

sig Man extends Person {
wife: lone Woman
}

sig Woman extends Person {
husband: lone Man
}

fun grandpas [p: Person] : set Person {
p.(mother+father).father
}

pred ownGrandpa [p: Person] {
p in p.grandpas
}

run ownGrandpa for 4 Person

fact {
no p: Person |
p in p.(mother+father) or
p in (p.(mother+father)).(mother+father) or
p in ((p.(mother+father)).(mother+father)).
(mother+father) …
}
```

```
fact {
no p: Person | p in p.^(mother+father)
}

fact {
all m: Man, w: Woman |
m.wife = w iff w.husband = m
}

fact {
wife = ~husband
}

fact {
no p: Person | p in p.^(mother+father)
wife = ~husband
}

assert NoSelfFather {
no m: Man | m = m.father
}
check NoSelfFather

fact SocialConvention {
no ((wife+husband) & ^(mother+father))
}

fact SocialConvention2 {
all m: Man, w: Woman |
( m.wife = w and w.husband = m )
implies
not( m in w.^(father+mother) or
w in m.^(father+mother) )
}

fun ancestors [p: Person]: set Person {
p.^(mother+father)
}

fact noCommonAncestors {
all p1: Man,
p2: Woman | p1->p2 in wife
implies
ancestors[p1] & ancestors[p2] = none
}

fact SocialConvention3 {
no ( (wife+husband)
&
(mother+father).~(mother+father) )
}

assert Stronger {
noCommonAncestors implies SocialConvention3
```

```
}
check Stronger for 8
assert NotStronger {
SocialConvention3 implies noCommonAncestors
}
check NotStronger for 8

fact SocialConvention4 {
all p1: Person, p2: Person |
p1 in p2.(mother+father)
implies
p1.(mother+father) & p2.(mother+father) = none
}
```

▼ Address Book

```
sig Name, Addr {}


  sig Book {
  var addr: Name -> lone Addr
}

pred show {}
run show for 3 but 1 Book

pred show [b: Book] {
#b.addr > 1 }

pred show [b: Book] {
#b.addr > 1
#Name.(b.addr) > 1 }

pred show [b: Book] {
#b.addr > 1
some n: Name | #n.(b.addr) > 1 }

// DYNAMIC:

pred show [b: Book] {
  #b.addr = 0
  after ( #b.addr = 1
  and
  after #b.addr = 2 )
}

pred show [b: Book] {
#b.addr = 0 ; #b.addr = 1 ; #b.addr = 2 )
}

run show for 3 but 1 Book, 3 steps

pred add [b: Book, n: Name, a: Addr] {
```

```
b.addr' = b.addr + n -> a
}

pred del [b: Book, n: Name] {
b.addr' = b.addr - n->Addr
}
```

▼ Mailbox

```
var sig Message {}

var sig Trashed in Message {}

pred delete[ m: Message ]
{ m not in Trashed

 Trashed' = Trashed + m
 Message' = Message }


 pred restoreEnabled[ m:Message ]
{ m in Trashed }

pred restore[ m: Message ]
{ restoreEnabled[m]
 Trashed' = Trashed - m
 Message' = Message }

 pred deleteTrashed
{ #Trashed > 0
 after #Trashed = 0
 Message' = Message-Trashed }

 pred doNothing
{ Message' = Message
 Trashed' = Trashed }

 pred receiveMessages
{ #Message' > #Message
 Trashed' = Trashed }

 fact systemBehavior {
no Trashed
always (
( some m: Message | delete[m] or restore[m] )
or
deleteTrashed
or
receiveMessages
or
doNothing
)
}
```

```
assert restoreAfterDelete {
 all m: Message |
 always restore[m] implies once delete[m]
}

assert deleteAll
{ always ( ( Message in Trashed and deleteTrashed )
implies
after always no Message) }

assert messagesNeverChange {
 always (Message' in Message and Message in Message')
}

assert ifMessagesNotDeletedTrashNotEmptied
{ ( always all m : Message | not delete[m] )
implies
always not deleteTrashed }
```

▼ **InterRail**

```
//  INTER-RAIL

sig Traveler{
   var status: one Status,
   var position: one City,
   pass: some City,
   startingPoint: one City,
   var visitedCities: set City
} {
// IF THERE ARE SOME VAR INVOLVES IN A FACT, DONT PUT IT HERE!!!
   startingPoint in pass
   #pass > 3
}

abstract sig City{
   connect: some City
}{
   // doesnt connect itself
   this not in connect
}

one sig Milan, Paris, London, Amsterdam, Berlin, Copenaghen, Stockholm extends City{}

abstract sig Status {}

one sig Travelling, Arrived extends Status{}


// CONSTRAINTS:

// If a City c1 connects c2, then c2  conncects c1
fact fact1{
```

```
   all disj c1,c2: City | c2 in c1.connect implies c1 in c2.connect
}

fact fact2{
   // starting from one city, I can reach all the others
   all c: City | c.^connect = City - c
}

fact initialState{
   // every Traveler is travelling at the beginning
   all t: Traveler | t.status = Travelling and t.visitedCities = none
   startingPoint = position
}

pred example{
   #Traveler = 2
   always( some t: Traveler, c: City |  move[t,c] or stutter )
   eventually all t:Traveler | t.status = Arrived
}

pred stutter{
  status' = status
  position' = position
  visitedCities' = visitedCities
}

run example for 6

pred move[t:Traveler, c: City]{
   t.status = Travelling
   c != t.position and c not in t.visitedCities and c in t.pass

   t.position' = c
   c = t.startingPoint implies t.status' = Arrived else t.status' = t.status
   t.visitedCities' = t.visitedCities + c

   all t2: Traveler- t |  t2.status' = t2.status and t2.startingPoint' = t2.startingPoint and
     t2.position' = t2.position
}


fact fact3{
   always(
   all t: Traveler | t.status = Arrived implies always(t.status = Arrived)
)
}

// Traveler visit all cities in their pass.

fact fact4{
   eventually all t:Traveler | t.visitedCities = t.pass
}
```

▼ TreasureHunt

```
// The Treasure Hunt

abstract sig HuntElement{}

sig Island extends HuntElement {
   connect: some Island
}{
   this not in connect
}

one sig StartingIsland in Island{}

sig Treasure{
  var position: one HuntElement  // treasure can be in the island or then taken by an Adventu
}{
   // startingIsland doesnt have Treasure
   position not in StartingIsland
}

sig Adventurer extends HuntElement{
  var isAt: one Island
}

fact fact1{
   // Islands are connected by one-way or two-way bridges
   some disj i1,i2: Island | i2 in i1.connect and i1 in i2.connect
   some disj i1,i2: Island | i2 in i1.connect and i1 not in i2.connect
}

fact initialCondition{
   all a: Adventurer |  a.isAt = StartingIsland
   all t: Treasure | t.position in Island
}

run{
   #Adventurer > 2
   #Treasure > 2
} for 10

fact fact2{
   // from StartingIsland, I have to be able to reach all the others and come back
   StartingIsland.^connect = Island
}

fact fact3{
   // each Island has at most one treasure
   all t: Treasure | t.position not in (Treasure - t).position
}

// An adventurer can move to an island only if it is connected by a bridge from their current
```

```
pred move[a: Adventurer, i: Island]{
   i not in a.isAt and i in a.isAt.connect

   a.isAt' = i

   position' = position

}

pred stutter {
  isAt' = isAt
  position' = position
}

pred takeTreasure[a:  Adventurer] {
   a.isAt in Treasure.position

   let tr = position.(a.isAt) |  tr.position' = a

   a.isAt' = a.isAt

}

fact events {
   //Either a move happens or nothing changes ( needed for when no move happens )
  always ( (some a: Adventurer | some i: Island | move[a, i] or  takeTreasure[a]  ) or stutte
  // eventually some a:Adventurer | takeTreasure[a]
  some a: Adventurer | (eventually takeTreasure[a])
  some a: Adventurer | not (eventually takeTreasure[a])

}

fact movingWithTreasure{
  always(  all a:Adventurer | all t: Treasure | t.position = a implies eventually a.isAt = Sta
}

fact carryOneTreasure{
  always(  all t: Treasure,  a: Adventurer | t. position = a implies a not in (Treasure-t).pos
}
```

▼ **RoadNetwork**

```
enum Status { Accident, Normal}

abstract sig RoadNetwork{}

sig Crossroad extends RoadNetwork{
   var status: one Status
}

sig Road extends RoadNetwork{
    connect:  Crossroad -> Crossroad
}{
   #connect = 1
}

// a Road always connects two different Crossroads
fact roadFact{
    all r: Road | all c1,c2 : Crossroad | c1 -> c2 in r.connect implies c1 != c2
}

fun connected : Crossroad -> Crossroad {
  Road.connect
}


sig Car{
   var located: one RoadNetwork,
   var nextMove: lone Crossroad
}

fact crossRoadConnection{
       // every Crossroad is connected to at least one road ( origin or destination )
    all c: Crossroad | some r: Road | c in  r.connect.Crossroad or  c in r.connect[Crossroad]
}
```

```
fact directions{

   some r1,r2: Road,  c1,c2 : Crossroad | r1.connect = c1->c2 and r2.connect = c2->c1
   some r1,r2: Road,  c1,c2 : Crossroad | r1.connect = c1->c2 and not r2.connect = c2->c1

}


fact initialCondition{
    some Car
    all c: Car | c.located in Crossroad and c.nextMove = none
}


pred example{
   #Car = 3
   all c: Car | c.located not in (Car-c).located
   // some c: Car | after carMoves[c] and (after after carMoves[c]) and (after after after ca
   // all c: Car | after after after after after c.located = (Car-c).located and c.located in

}

run example for 6


// for better visualizaiton

fun origin: Crossroad -> Road{
   { c1: Crossroad, r2 : Road | c1 in Road.connect.Crossroad and r2 in connect.Crossroad.c1 }
}

fun destination: Road -> Crossroad{
   { r1:Road , c2 : Crossroad | c2 in   Road.connect[Crossroad]  and r1 in connect.c2.Crossroa
}


pred carMoves[c: Car]{
    c.located in Crossroad implies (
        c.located' =  origin[c.located]
    )

    c.located in Road implies (
        c.located' = destination[c.located]
    )
}

pred stutter[c:Car]{
    c.located' = c.located
        status' = status
}
```

```
fact{
    always(
        all c: Car | carMoves[c] or stutter[c]
    )
}

pred example2{
   example
   some c: Car | after carMoves[c] and (after after carMoves[c]) and (after after after carMo
   all c: Car | after after after after after c.located = (Car-c).located and c.located in Cr
   all c: Crossroad | c.status = Normal

}

run example2 for 10




fact noCarsLeaveAccident {
  always all c : Crossroad, car: Car | c.status = Accident and car.located in destination.c
}

fact someAccident{
  after always some c : Crossroad | c.status = Accident
}



assert trafficStuckInAccident {
  always all c : Crossroad, car : Car |  c.status = Accident and car.located in Road and dest
                                                                  implies
                                                                  car.located =car.located'
}

check trafficStuckInAccident for 10
```