POLITECNICO
MILANO 1863

Alloy Examples

Traffic Lights
Communication Networks
Luggage Keeper (UML-related)
Recipe Management
Mailboxes (temporal)
Towers and Cubes
Airbus

M Camilli, E Di Nitto, M Rossi

Alloy Examples

Traffic Lights

Traffic Lights

- A semaphore can have one of the following colors: GREEN and RED.
- Two semaphores are used to regulate traffic at an intersection where 2 roads meet. Each road is two-way. The semaphores must guarantee that traffic can proceed on one and only one of the two roads in both directions.
- Provide an Alloy model of the intersection, specifying the necessary invariants.

- **NB:** While building the model, focus on those part of the roads that are close to the intersection. Disregard the fact that a road can participate to more than one intersection.

Traffic Lights - Signatures

```
abstract sig Color{}
one sig GREEN extends Color{}
one sig RED extends Color{}

abstract sig Traffic{}
one sig FLOWING extends Traffic{}
one sig STOPPED extends Traffic{}

sig Semaphore {
 color: one Color
}

sig Road {
 traffic: one Traffic,
}

sig Intersection{
 connection: Semaphore -> Road    //MAPS Semaphore to Road
}{
 #connection = 2
}
```

Traffic Lights - Functions

```
//Retrieves all the Roads of one Intersection
fun getIRoads[i :Intersection]: set Road {
 Semaphore.(i.connection)
}

//Retrieves the Roads connected to one semaphore
fun getSRoads[s :Semaphore]: set Road {
 s.(Intersection.connection)
}

//Retrieves all the Semaphores of one Intersection
fun getSemaphores[i :Intersection]: set Semaphore {
 (i.connection).Road
}
```

Traffic Lights - Properties

```
fact intersectionStructure{
 //Intersection has exactly 2 roads and 2 semaphores
 (all i : Intersection |
  (let s = getSemaphores[i] | #s=2) and
  (let r = getIRoads[i] | #r=2)
 )
 and
 // All semaphores are connected to only one road
 (all sem : Semaphore |
  (let rd = getSRoads[sem] | #rd=1)
 )
}
```

Traffic Lights – Properties (2)

```
//Semaphores of one intersection should display different colors
fact greenIsExclusive{
 all i : Intersection |
  ( let s = getSemaphores[i] | all s1: Semaphore, s2 : Semaphore |
                                  (s1 in s and s2 in s and s1 != s2)
                                  implies s1.color != s2.color )
}

//Traffic flows with GREEN
fact goWithGreen{
 (all s: Semaphore | let r = getSRoads[s] |
                     s.color=RED iff r.traffic=STOPPED)
 and
 (all r: Road | r.traffic=STOPPED implies
                #((Intersection.connection).r)>0)
}
```

Traffic Lights – Commands to run

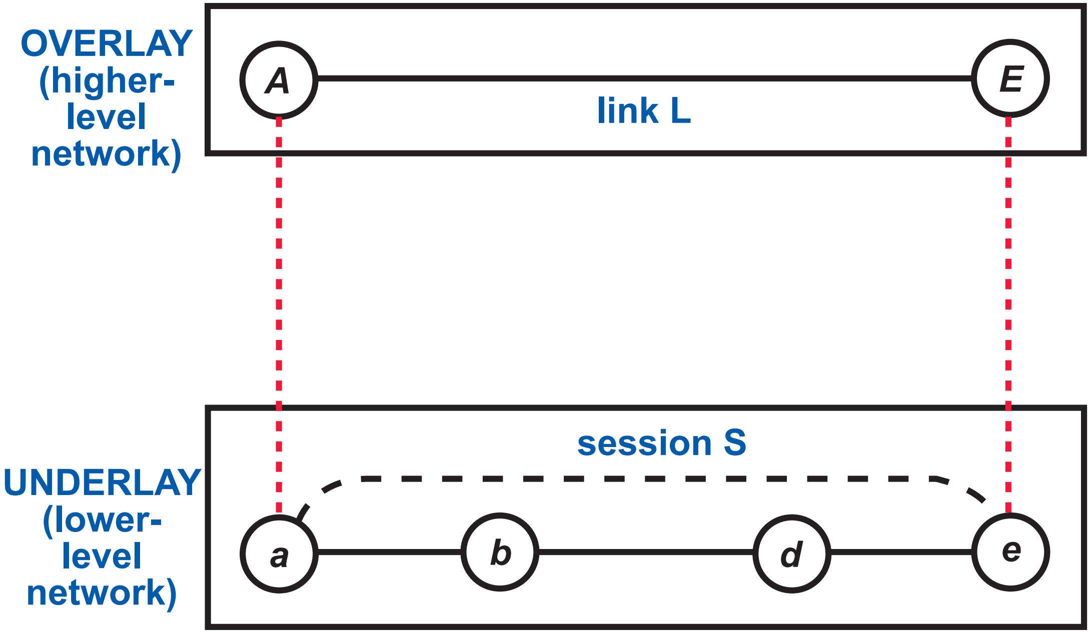
```
pred show{
 #Intersection = 1
}


run show for 8
```

Alloy Examples

Communication Networks (June 28, 2017 exam)

Exercise

- Observe the following figure. It describes a communication system composed of an overlay network linking *nodes* A and E. This overlay is a virtual network built on top of an underlay network. In the figure, the underlay network is composed of four nodes (a, b, d, and e) and link L exploits the links between a, b, d and e in the underlay network to ensure that A and E can communicate.



E is attached to e, or . . .

e is the location of E.

L uses S, or . . .

S implements L.

Exercise (cont.)

- Consider the following Alloy signatures:

```
sig Network {
  uses: lone Network
}{ this not in uses }

sig Node {
      belongsTo: Network,
      isLinkedTo: some Node,
      isAttachedTo: lone Node
}{ this not in isAttachedTo and
   this not in isLinkedTo }
```

Exercise (cont.)

- **A)** Explain the meaning of these signatures with respect to the figure above and indicate which elements in the figure are not explicitly modeled by the two signatures.

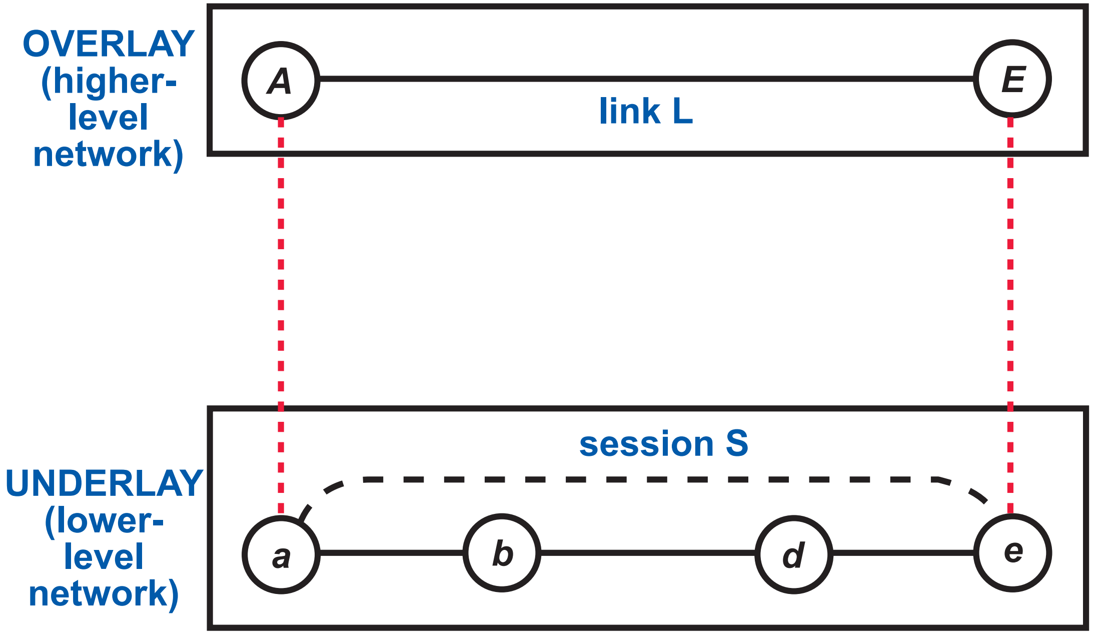
- **B)** Write facts to model the following constraints:
  - Linked nodes have to be in the same network
  - A node belonging to a certain network can only be attached to nodes of the corresponding underlay network
  - If a network is an overlay one, then there should not be nodes in this network that are not attached to other nodes
  - A network should always contain some nodes

- **C)** Write the predicate `isReachable` that, given a pair of `Nodes`, `n1` and `n2`, is true if there exists a path that from `n2` reaches `n1`, possibly passing through any intermediate node.

Solution, Part A

```
sig Network {
  uses: lone Network
}{ this not in uses }
```

```
sig Node {
        belongsTo: Network,
        isLinkedTo: some Node,
        isAttachedTo: lone Node
}{ this not in isAttachedTo and
   this not in isLinkedTo }
```



E is attached to e, or . . .

e is the location of E.

L uses S, or . . .

S implements L.

Solution, Part B

```
// Linked nodes have to be in the same network
fact linkedNodesInTheSameNetwork {
  all disj n1, n2: Node |
           n1 in n2.isLinkedTo implies
                               #(n1.belongsTo & n2.belongsTo) > 0
}

// A node belonging to a certain network can only be
// attached to nodes of the corresponding underlay network
fact isAttachedToInConnectedNetworks {
  all disj n1, n2: Node |
           n1 in n2.isAttachedTo implies
                                #(n1.belongsTo & n2.belongsTo.uses) > 0
}
```

Solution, Part B (cont.)

```
// If a network is an overlay one, then there should not be nodes in
// this network that are not attached to other nodes
fact overlayNodeShouldBeAttached {
  all ntw: Network |
      some ntw2: Network | ntw2 in ntw.uses
        implies
      all n: Node | n.belongsTo = ntw implies n.isAttachedTo != none
}

// A network should always contain some nodes
fact notEmptyNetwork {
  all ntw: Network | some n: Node | n.belongsTo = ntw
}
```

Solution, Part B
(alternative formulation of some facts)

```
// Linked nodes have to be in the same network
fact linkedNodesInTheSameNetwork {
  all disj n1, n2: Node |
           n1 in n2.isLinkedTo implies n1.belongsTo = n2.belongsTo
}

// A node belonging to a certain network can only be
// attached to nodes of the corresponding underlay network
fact isAttachedToInConnectedNetworks {
  all disj n1, n2: Node |
           n1 in n2.isAttachedTo implies
                                    n1.belongsTo in n2.belongsTo.uses
}
```

Solution, Part C

```
//n1 is reachable from n2
pred isReachable[n1: Node, n2: Node] {
  n1 in n2.^isLinkedTo
}
```

Alloy Examples

Luggage Keeper (See also exam of February 16, 2018)

Informal description

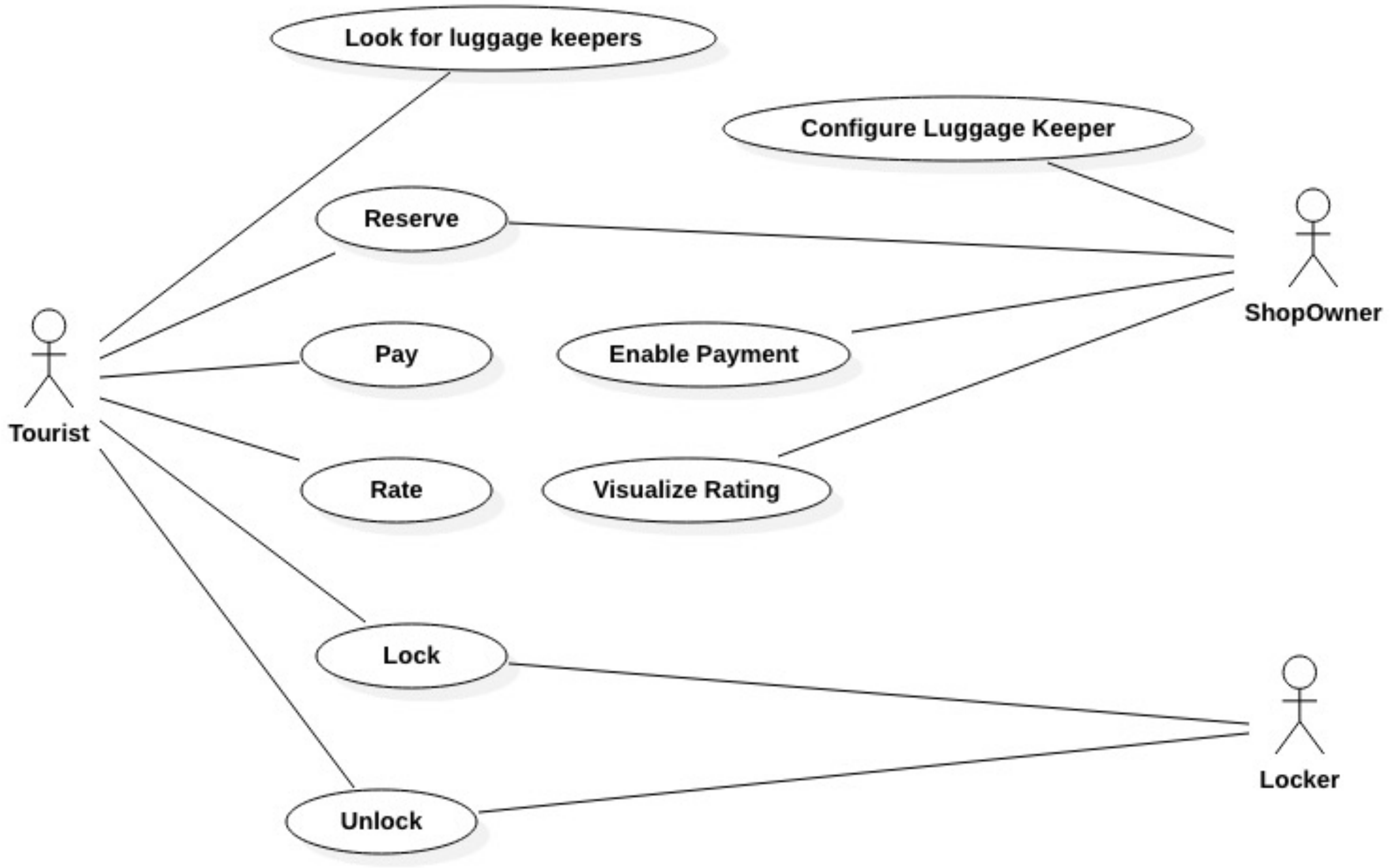
- The company *TravelSpaces* decides to help tourists visiting a city in finding places that can keep their luggage for some time. The company establishes agreements with small shops in various areas of the city and acts as a mediator between these shops and the tourists that need to leave their luggage in a safe place.
- To this end, the company wants to build a system, called *LuggageKeeper*, that offers tourists the possibility to: look for luggage keepers in a certain area; reserve a place for the luggage in the selected place; pay for the service when they are at the luggage keeper; and, optionally, rate the luggage keeper at the end of the service.

(some) possible world and machine phenomena

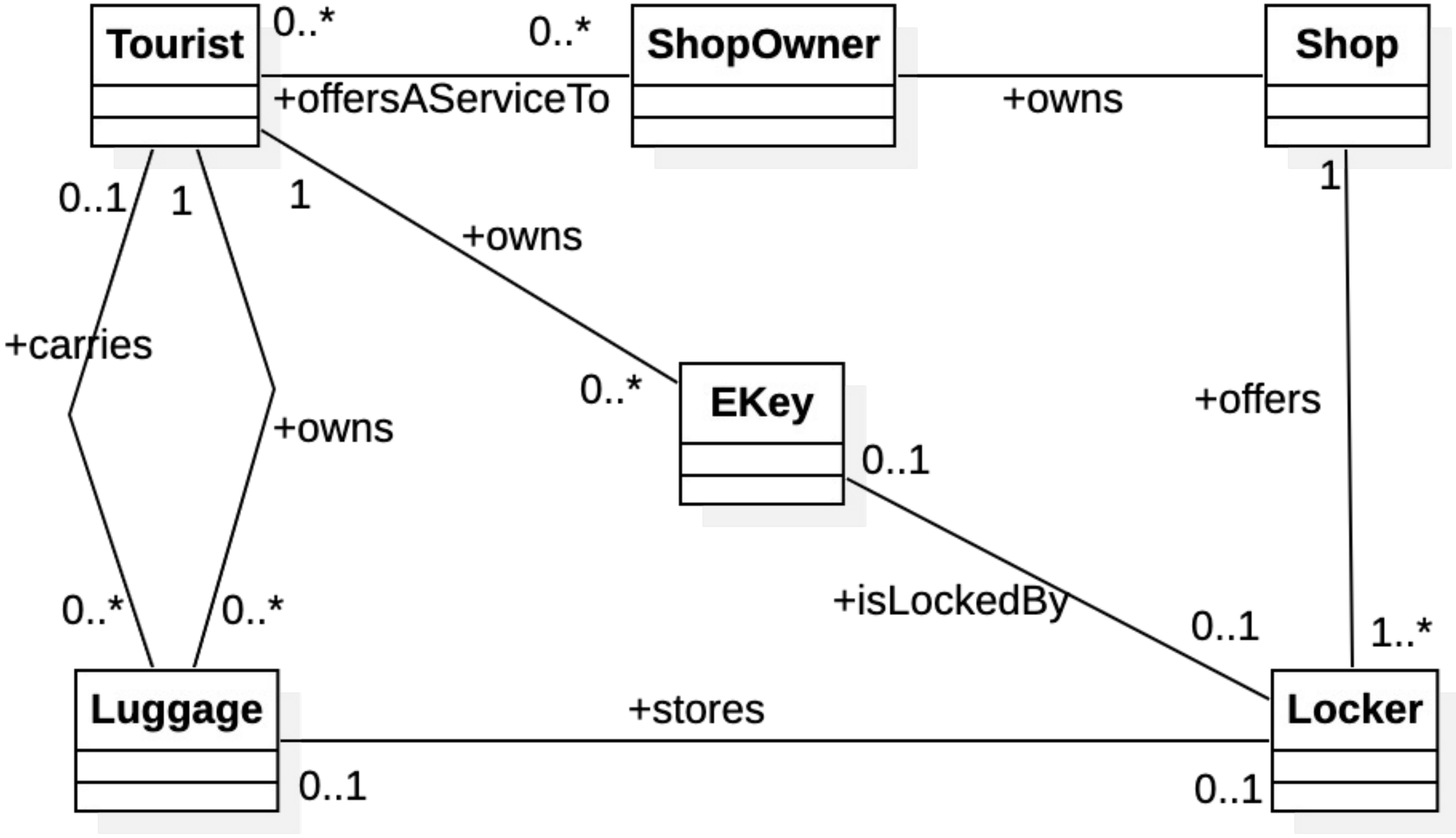
- World phenomena
  - Some users own various pieces of luggage.
  - Some users carry around various pieces of luggage.
  - Some pieces of luggage are safe
  - Some pieces of luggage are unsafe.
  - Small shops store the luggage in lockers.
- Shared phenomena
  - Some lockers are opened with an electronic key.
  - Some users hold various electronic keys.

Use cases

POLITECNICO
MILANO 1863



Problem domain model (class diagram)



Alloy signatures

```
abstract sig Status{}
one sig Safe extends Status{}
one sig Unsafe extends Status{}

sig Luggage{
  luggageStatus : one Status
}

sig EKey{}
```

```
sig User{
  owns : set Luggage,
  carries : set Luggage,
  hasKeys : set EKey
}

sig Locker{
  hasKey : lone EKey,
  storesLuggage : lone Luggage
}

sig Shop{
  lockers : some Locker
}
```

Various constraints could be added
(e.g., the owner of a luggage is unique)

A domain assumption

- any piece of luggage is safe if, and only if, it is with its owner, or it is stored in a locker that has an associated key, and the owner of the piece of luggage holds the key of the locker

```
fact DAsafeLuggages {
  all lg : Luggage |
      lg.luggageStatus in Safe
        iff
      all u : User | lg in u.owns
                     implies
                   ( lg in u.carries
                     or
                     some lk : Locker | lg in lk.storesLuggage and
                                        lk.hasKey != none and
                                        lk.hasKey in u.hasKeys )
}
```

A requirement

- a key opens only one locker

```
fact requirement {
  all ek : EKey | no disj lk1, lk2: Locker | ek in lk1.hasKey
                                             and
                                             ek in lk2.hasKey
}
```

A goal

- for each user all his/her luggage is safe

```
pred goal {
  all u : User, lg : Luggage | lg in u.owns
                               implies
                             lg.luggageStatus in Safe
}
```

Operation `GenKey`

- Given a locker that is free, `GenKey` associates with it a new electronic key

```
sig Locker{
    var hasKey : lone EKey,
    var storesLuggage : lone Luggage
}

pred GenKey[lk : Locker] {
    //precondition
    lk.hasKey = none
    //postcondition
    lk.storesLuggage' = lk.storesLuggage
    one ek : EKey | lk.hasKey' = ek
}
```

We have to make these relations mutable

Alloy Exercises

Recipe Management (June 26, 2023 exam)

Exercise

- Consider an application that manages recipes that are of interest to users and provides suggestions when requested.

- **Question 1**: Define suitable signatures, constraints and facts to describe the following phenomena:
  - Recipes are characterized by a set of ingredients, and by the type of cuisine (Italian, Indian, etc.).
  - Users have ingredients at their disposal.
  - Users have favorite types of cuisine.
  - Users maintain a list of favorite recipes.
  - Users are provided with suggested recipes (which cannot be recipes that the user already favors).
  - Users can be suggested only recipes that include at least 1 ingredient that is already at the user's disposal.

Exercise (cont.)

- **Question 2**: Define a suitable predicate specifying the behavior of a procedure `missingIngredients` that, given a user and a recipe that has been suggested to the user, produces the list of ingredients that the user is missing.

- **Question 3**: Define a suitable predicate specifying the behavior of a procedure `suggestRecipe` that, given a user and a set of possible recipes, produces a subset of the input recipes that can be suggested to the user. The produced subset should be non-empty if in the input set there is at least one recipe that can be suggested to the user.

Solution, Question 1

```
sig Ingredient{}

abstract sig Cuisine {}
one sig Italian extends Cuisine {}
one sig Indian extends Cuisine {}
one sig French extends Cuisine {}
one sig Japanese extends Cuisine {}
// the list of cuisine types is typically finite (it is an enumeration)
// this list should be completed with the various possibilities

sig Recipe {
  ingredients : some Ingredient,
  cuisine: Cuisine
}
```

Solution, Question 1 (cont.)

```
sig User {
  favoriteCuisine : set Cuisine,
  favoriteRecipes : set Recipe,
  availableIngredients : set Ingredient,
  suggestedRecipes : set Recipe
}{
   suggestedRecipes & favoriteRecipes = none
   all sr : suggestedRecipes |
                  sr.ingredients & availableIngredients != none
}
```

Solution, Question 2

```
pred missingIngredients[ u : User, r : Recipe,
                         res : set Ingredient] {
  // pre-condition
  r in u.suggestedRecipes
  // post-condition
  res = r.ingredients - u.availableIngredients
}
```

Solution, Question 3

```
pred suggestRecipe [u : User, possibleRecipes : some Recipe,
                          res : set Recipe] {
  //postcondition
  u.availableIngredients & possibleRecipes.ingredients != none
    implies
  ( some r : possibleRecipes |
             not r in u.favoriteRecipes and
             r.ingredients & u.availableIngredients != none and
             r in res )
  and
  ( all r : res | r in possibleRecipes and
                  not r in u.favoriteRecipes and
                  r.ingredients & u.availableIngredients != none )

  u.availableIngredients & possibleRecipes.ingredients = none
                                                       implies res = none
}
```

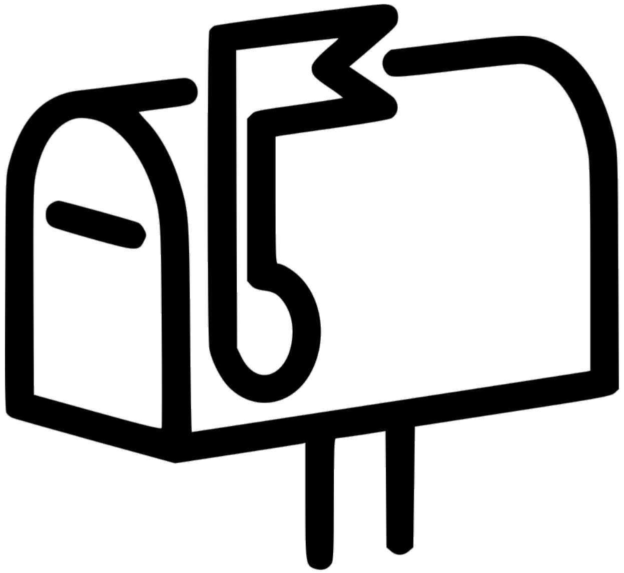
Alloy Exercises

Mailboxes

Example: message deletion from mailbox

POLITECNICO
MILANO 1863

- Model a system handling messages, which can be deleted from a mailbox and later restored
- We introduce the notion of "trash", from which messages can be restored
  - Some messages are in the trash (i.e., they are trashed), others are not



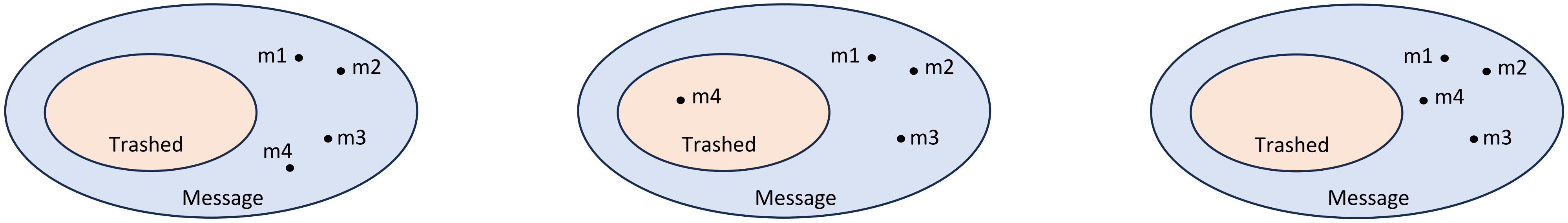
Signatures

This states that `Trashed` is a subset (not necessarily a proper one) of `Message`, i.e., `Trashed` ⊆ `Message` holds

```
var sig Message {}

var sig Trashed in Message {}
```

Signatures (i.e., sets of elements) can also be mutable

Some messages are in the `Trashed` set.
Signatures are mutable, which means that a message can be "regular", then be trashed, then be regular again, etc.



First instant Second instant Third instant

Predicates

- Deletion operation

```
pred delete[ m: Message ]
{    m not in Trashed
     Trashed' = Trashed + m
     Message' = Message    }
```

The subset with the trashed messages changes, while the total set of messages does not

- Predicate capturing the condition that a message can be restored

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

- Restore operation

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

Predicates (2)

- Change in system state: trash is emptied

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

- Predicate representing the fact that the state of the system does not change

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

- Change in system state: new message arrives

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

Behavior of the system

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

Initially the trash is empty

Several things can occur during system execution:

- a message could be deleted or restored
- the trash could be emptied
- new messages could arrive
- nothing happens in the system (no state change)

Assertions
(do you think they hold?)

- If a message is restored, sometimes in the past it had to be deleted

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

- If at a certain point in time all messages are trashed and the trash is emptied, then, from that point on there will be no more messages

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

More assertions
(do these hold?)

- The set of messages never changes

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

- If no messages are ever deleted, then the trash will never be emptied

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

POLITECNICO MILANO 1863

Towers and Cubes

Towers and Cubes (February 13, 2017 exam)

Exercise

POLITECNICO
MILANO 1863

- Consider construction cubes of three different sizes, small, medium, and large. You can build towers by piling up these cubes one on top of the other respecting the following rules:
  - A large cube can be piled only on top of another large cube
  - A medium cube can be piled on top of a large or a medium cube
  - A small cube can be piled on top of any other cube
  - It is not possible to have two cubes, A and B, simultaneously positioned right on top of the same other cube C

Exercise (cont.)

- **Question 1**: Model in Alloy the concept of cube and the piling constraints defined above.
- **Question 2**: Model also the predicate `canPileUp` that, given two cubes, is true if the first can be piled on top of the second and false otherwise.
- **Question 3**: Consider now the possibility of finishing towers with a top component having a shape that prevents further piling, for instance, a pyramidal or semispherical shape. This top component can only be the last one of a tower, in other words, it cannot have any other component piled on it.
  Rework your model to include also this component. You do not need to consider a specific shape for it, but only its property of not allowing further piling on its top. Modify also the `canPileUp` predicate so that it can work both with cubes and top components.

Solution, Question 1

```
abstract sig Size{}
one sig Large extends Size{}
one sig Medium extends Size{}
one sig Small extends Size{}

sig Cube {
  size: Size,
  cubeUp: lone Cube
}{ cubeUp != this }

fact noCircularPiling {
  no c: Cube | c in c.^cubeUp
}
```

Solution, Question 1 (cont.)

```
fact pilingUpRules {
  all c1, c2: Cube |
      c1.cubeUp = c2
        implies
      ( c1.size = Large or
        c1.size = Medium and (c2.size = Medium or c2.size = Small) or
        c1.size = Small and c2.size = Small )
}

// it is still possible for a cube to be on top of two different cubes
// this is not explicitly ruled out by the specification
```

Solution, Question 2

```
pred canPileUp[cUp: Cube, cDown: Cube] {
    cDown != cUp
    and
    ( cDown.size = Large
      or
      cDown.size = Medium and (cUp.size = Medium or cUp.size = Small)
      or
      cDown.size = Small and cUp.size = Small )
}
```

Solution, Question 3

```
// modified signatures
abstract sig Block {}
sig Top extends Block {}
sig Cube extends Block {
  size: Size,
  cubeUp: lone Block
}{ cubeUp != this }

pred canPileUp[bUp: Block, bDown: Block] {
    bDown != bUp and
    bDown in Cube and
    ( bUp in Top
      or
      bDown.size = Large
      or
      bDown.size = Medium and (bUp.size = Medium or bUp.size = Small)
      or
      bDown.size = Small and bUp.size = Small )
}
```

Alloy Exercises

Airbus

Is the UML spec complete?

- We have described
  - All phenomena: Aircraft, wheels, sensor, reverse thrust system
  - A use case EnablingReverseThrust
- Are we missing something?
- Are we representing goals, domain properties and requirements?
  - Goal
    - Reverse_enabled ⇔ Moving_on_runway
  - Domain properties
    - Wheel_pulses_on ⇔ Wheels_turning
    - Wheels_turning ⇔ Moving_on_runway
  - Requirements
    - Reverse_enabled ⇔ Wheels_pulses_on

Is the UML spec complete?

- Pure UML does not help us in expressing assertions
- UML models must be complemented with some formal or informal description of these assertions

Modeling the Airbus braking logic with Alloy

POLITECNICO
MILANO 1863

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

abstract sig AirCraftState {}
one sig Flying extends AirCraftState {}
one sig TakingOff extends AirCraftState {}
one sig Landing extends AirCraftState {}
one sig MovingOnRunaway extends AirCraftState {}
```

- … for landing, we are not considering the movement due to takeoff as it is not relevant to our analysis

Modeling the Airbus braking logic with Alloy (2)

```
sig Wheels {
      retracted: Bool,
      turning: Bool
}{ turning = True implies retracted = False }

sig Aircraft {
      status: one AirCraftState,
      wheels: one Wheels,
      wheelsPulsesOn: one Bool,
      reverseThrustEnabled: one Bool
}{ status = Flying implies wheels.retracted = True }
```

Modeling the Airbus braking logic with Alloy (3)

```
fact domainAssumptions {
    all a: Aircraft | a.wheelsPulsesOn = True
                      iff a.wheels.turning = True
    all a: Aircraft | a.wheels.turning = True
                      iff a.status = MovingOnRunaway}

fact requirement { all a: Aircraft | a.reverseThrustEnabled = True
                                     iff a.wheelsPulsesOn = True}

assert goal { all a: Aircraft | a.reverseThrustEnabled = True
                               iff a.status = MovingOnRunaway}
check goal
```

- No counterexamples are found!
  - But note that, still, this is the wrong model of our world: the spec is internally coherent, but it does not correctly represent the world