# Alloy

Pietro Braione

University of Milano-Bicocca

pietro.braione@unimib.it

## What is Alloy?

- A formal notation for specifying models of systems and of software
- With a tool to simulate and verify the documents written in it
- Available at http://alloytools.org

## The Alloy language

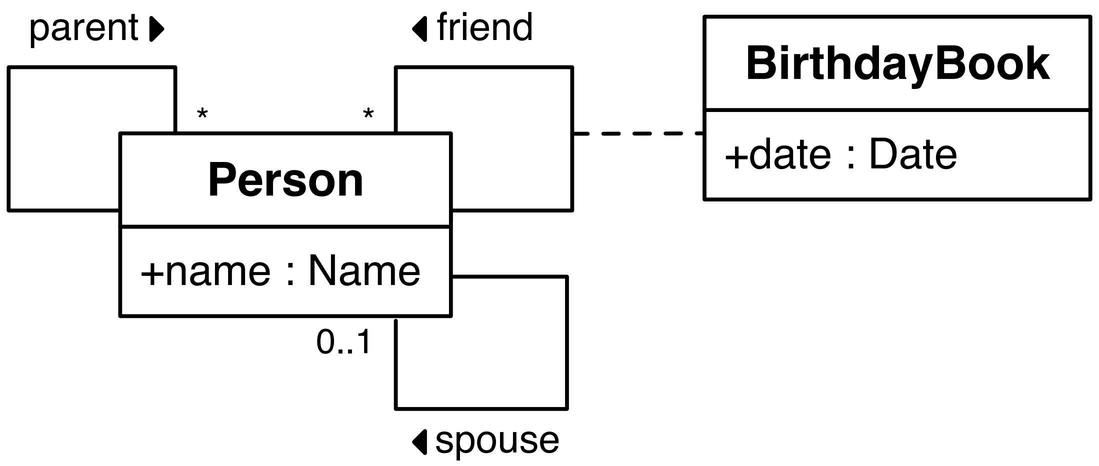
## Features of the language

- Declarative language based on relational algebra and first-order logic
- Allows to define complex structures as you would do in an OO language or in UML
- Allows to define complex relations between states as, e.g., state transitions
- What can be described with this language? E.g.
  - The heap memory of an OO program
  - Databases in E-R format
  - Configurations in a web application
  - Topologies of a network
  - …

## Running example

- There are persons and birthday books
- Each person has:
  - 1 name
  - 0..1 spouse
  - 0..n parents
  - 1 birthday book
- Each birthday book:
  - Contains a list of people
  - For each, it reports his/her birthday

# In UML


parent ▶
◀ friend
BirthdayBook
+date : Date
*
*
Person
+name : Name
0..1
◀ spouse

## An Alloy document

```
module birthday_book
```

$\leftarrow$ **Modules** : organize Alloy specifications

## An Alloy document

```
module birthday_book

sig Date { }

sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    birthdayBook: Person -> Date
}
```

**Modules** : organize Alloy specifications

**Signature** definitions:
Define entities and relations

## Meaning of Alloy documents

- It’s quite similar to the meaning of E-R or UML diagrams:
  - There are entities (**atoms** in Alloy parlance)…
  - …and **relations** over them
- Atoms are featureless, indivisible and immutable: all their features come from relations
- Relations are, as usual, sets of ordered tuples of atoms, and they are always finite
- An Alloy **world** is composed by atoms and relations
- A world that satisfies an Alloy specification is a **model** of it

## Signatures (1)

- Signatures define the “types” (“classes”) of an Alloy specification
- For instance, `sig Date {}` means that there must exist a set of atoms, where each atom stands for a date
- It does not say anything else on these atoms: Which are their attributes, how many of them exist…
- Also, the signature defines a unary relation (a set), with name `Date`, that contains all the atoms with that type
- Similarly, `sig Person { … }` states that there are some other atoms, the persons, distinct from the dates, and implicitly defines the set `Person` of all these atoms

## Signatures (2)

- Signatures are also used to define relations
- Relations definitions are enclosed in curly braces after the `sig` declaration

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    birthdayBook: Person -> Date
}
```

Each person has a name, and the name is a `String`

$\Downarrow$

There is a binary relation between the atoms of type Person and the atoms of type `String`

## Multiplicity constraints

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    …
}
```

Each person has…

## Multiplicity constraints

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    …
}
```

Each person has…

Exactly one name

# Multiplicity constraints

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    …
}
```

Each person has…

Exactly one name

One or no spouse

## Multiplicity constraints

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    …
}
```

Each person has…

Exactly one name

One or no spouse

Zero, one or more than one parent

## Multiplicity constraints

```
sig Person {
    name: String,
    spouse: lone Person,
    parents: set Person,
    …
}
```

Each person has…

Exactly one name

One or no spouse

Zero, one or more than one parent

More kinds of constraints:

- `one` : exactly one (can be omitted)
- `some` : one or more
- `no` : exactly zero

## Non-binary relations

```
sig Person {
    …
    birthdayBook: Person -> Date
}
```

There is a ternary relation between `Person`, `Person` and `Date`

# A possible model

Date

| |
|---|
| D1 |
| D2 |

String

| |
|---|
| ANN |
| BILL |
| JOHN |
| LISA |
| PAM |

Person

| |
|---|
| P1 |
| P2 |
| P3 |
| P4 |
| P5 |

Person <: name

| | |
|---|---|
| P1 | ANN |
| … | |

Person <: parents

| | |
|---|---|
| P4 | P2 |
| P4 | P3 |

Person <: spouse

| | |
|---|---|
| P1 | P4 |
| P2 | P3 |

Person <: birthdayBook

| | | |
|---|---|---|
| P1 | P3 | D1 |
| P1 | P4 | D2 |
| P3 | P2 | D1 |

## Example of facts

- No person is married with a sibling

## Example of facts

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

## Example of facts

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

- Each person is in the birthday book of one of the people in his/her birthday book

## Example of facts

- No person is married with a sibling

- Each person is in the birthday book of one of the people in his/her birthday book

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)

  all p: Person |
    some q: p.birthdayBook.Date |
      p in q.birthdayBook.Date
}
```

## Let’s consider the first fact…

- No person is married with a sibling

$\Downarrow$

For no atom p of type Person

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

## Let’s consider the first fact…

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

$\Downarrow$

For no atom p of type Person exists an atom

# Let’s consider the first fact…

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

$\Downarrow$

For no atom p of type Person exists an atom in the intersection

## Let’s consider the first fact…

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

$\Downarrow$

For no atom p of type Person exists an atom in the intersection of p’s parents

# Let's consider the first fact…

- No person is married with a sibling

```
fact {
  no p: Person |
    some (p.spouse.parents & p.parents)
```

$\Downarrow$

For no atom p of type Person exists an atom in the intersection of p's parents and the parents of p's spouse

## Expressions

- Logical
- Relational
- Numeric (integer)

## Logical operators

- and (&&), or (`||`), not (`!`)
- implication (=>), if-then-else (`F => G else H`), inverse implication (<=), logical equivalence (<=>)
- Quantifiers (`all`, `some`, `no`, `lone`, `one`)
- Subset: in and =
  - `r in s` is true iff `r` is a subset of `s` (relation inclusion)
  - `r = s` equivalent to `(r in s && s in r)`

## Relational operators

- Set-theoretic operators
  - Union (+), intersection (&), difference (-)
  - Remember! Sets are actually unary relations
- Relational operators, binary
  - Join ( . , [ ] ), cartesian product (->), domain restriction (<: ), range restriction ( :>)
- Relational operators, unary
  - transpose (~), transitive closure (^), transitive and reflexive closure (*)

# The join operator

- The join operator ( . ) is used to “navigate” through relations
- Its definition is quite similar to that of natural join used in database theory’s relational algebra, with the following differences:
  - The common columns are the last one of the left operand and the first one of the right operand
  - The common columns are projected away
- Note that in Alloy all variables are considered sets with just one element
- This allows to do the join of a variable with another

## Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?

p

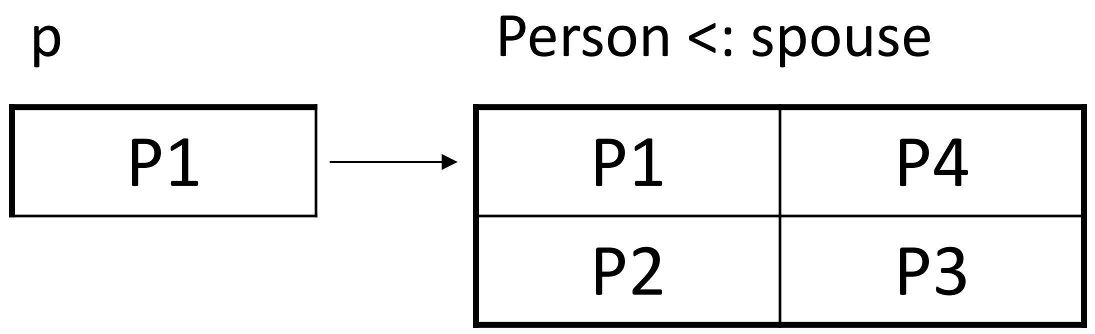
| |
|---|
| P1 |

Person <: spouse

| | |
|---|---|
| P1 | P4 |
| P2 | P3 |

# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?



p

| |
|---|
| P1 |

Person <: spouse

| | |
|---|---|
| P1 | P4 |
| P2 | P3 |

# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?

p

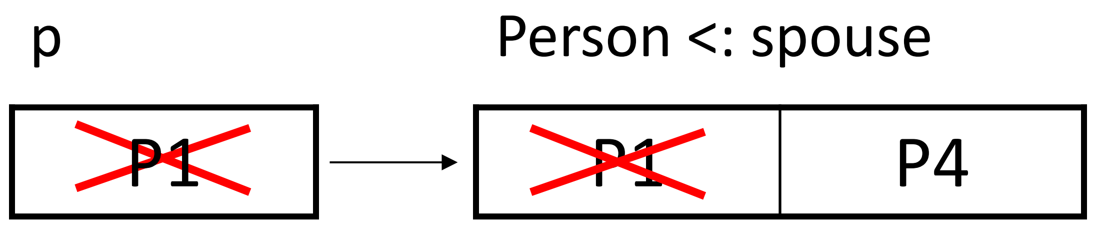
| |
|---|
| P1 |

Person <: spouse

| | |
|---|---|
| P1 | P4 |
| ~~P2~~ | ~~P3~~ |

# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?



# Example of join operator

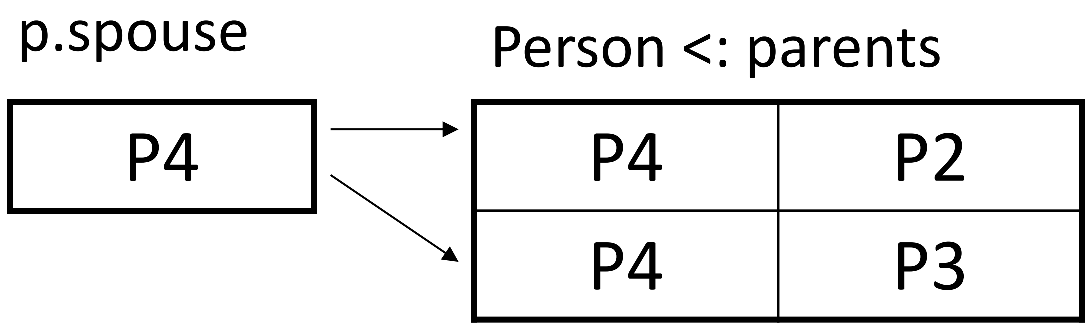
- Let us consider p = {P1}
- What is p.spouse.parents?

p.spouse

$\boxed{P4}$

# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?



p.spouse

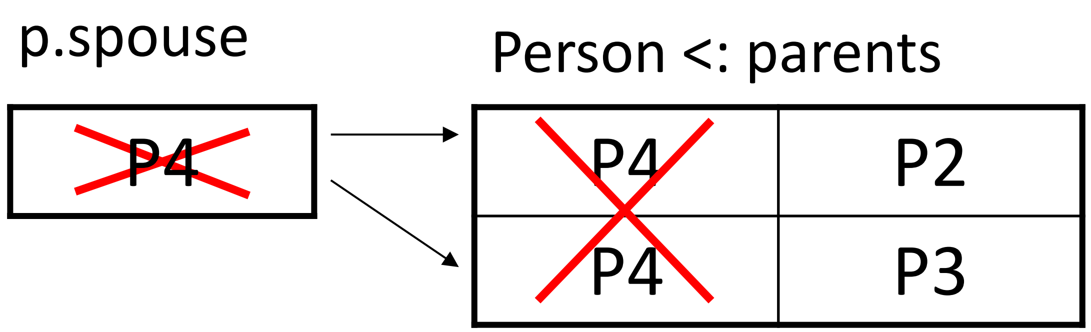
| |
|---|
| P4 |

Person <: parents

| | |
|---|---|
| P4 | P2 |
| P4 | P3 |

# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?



# Example of join operator

- Let us consider p = {P1}
- What is p.spouse.parents?

p.spouse.parents

| |
|---|
| P2 |
| P3 |

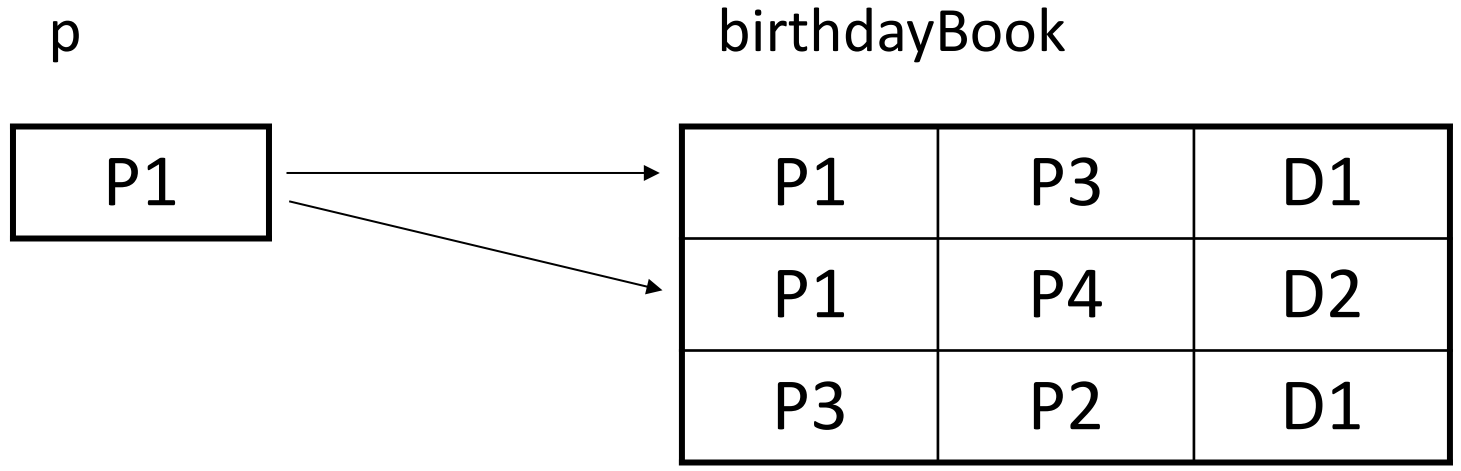
## Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}

```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

# Another example

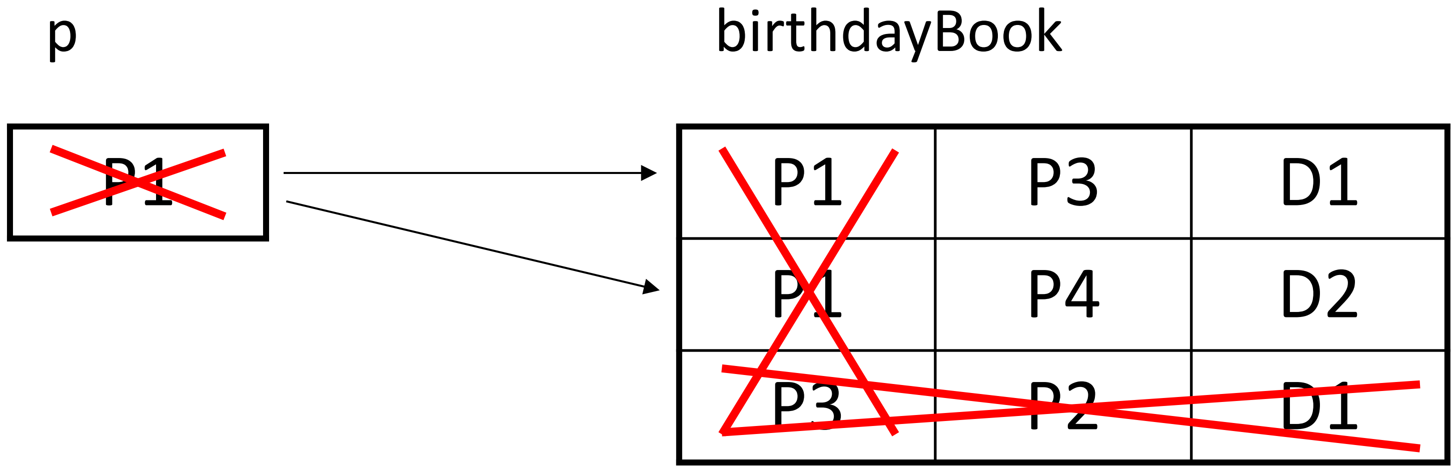
- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

# Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

## Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}

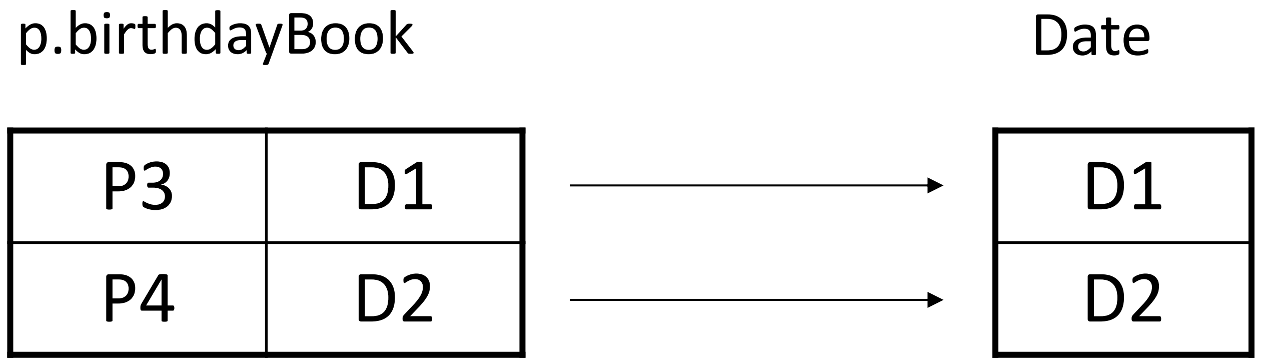
p.birthdayBook

| | |
|---|---|
| P3 | D1 |
| P4 | D2 |

```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

# Another example

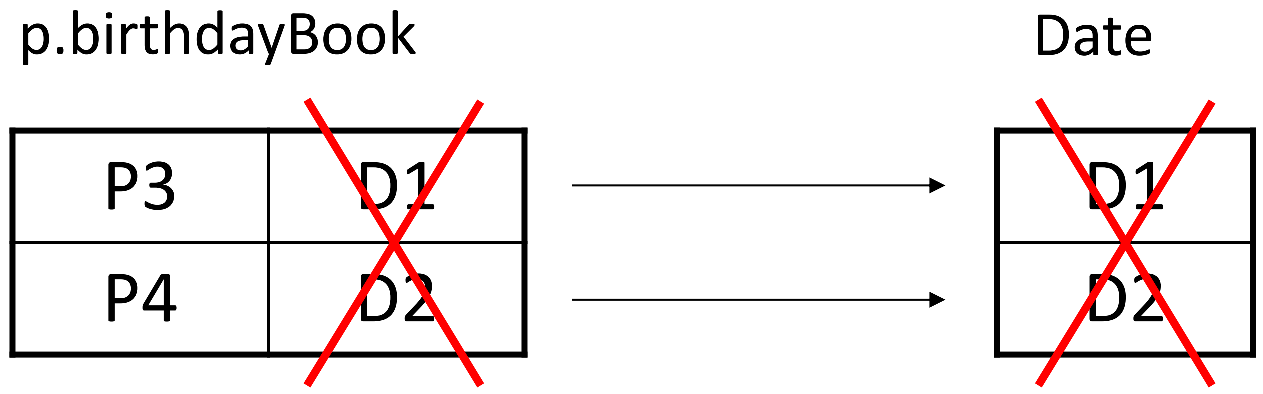
- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

## Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

# Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}

p.birthdayBook.Date

| |
|---|
| P3 |
| P4 |

```
all p: Person | some q: p.birthdayBook.Date |
  p in q.birthdayBook.Date
```

# Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}

p.birthdayBook.Date

| |
|---|
| P3 |
| P4 |

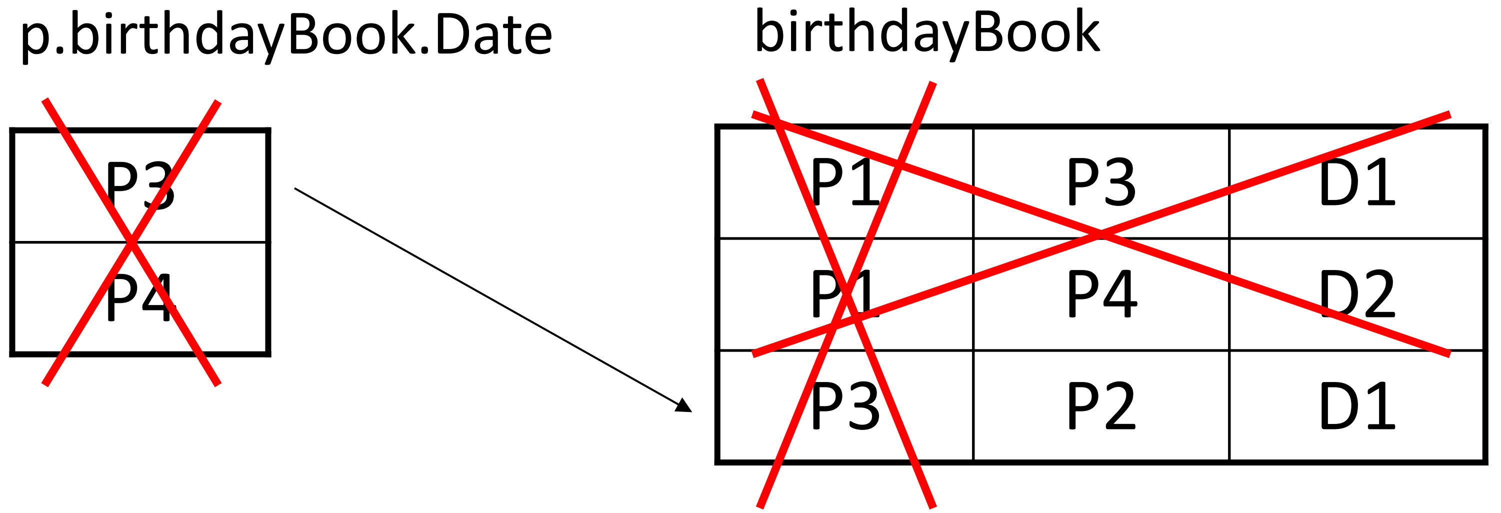
birthdayBook

| | | |
|---|---|---|
| P1 | P3 | D1 |
| P1 | P4 | D2 |
| P3 | P2 | D1 |

```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

# Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}



p.birthdayBook.Date

| |
|---|
| ~~P3~~ |
| ~~P4~~ |

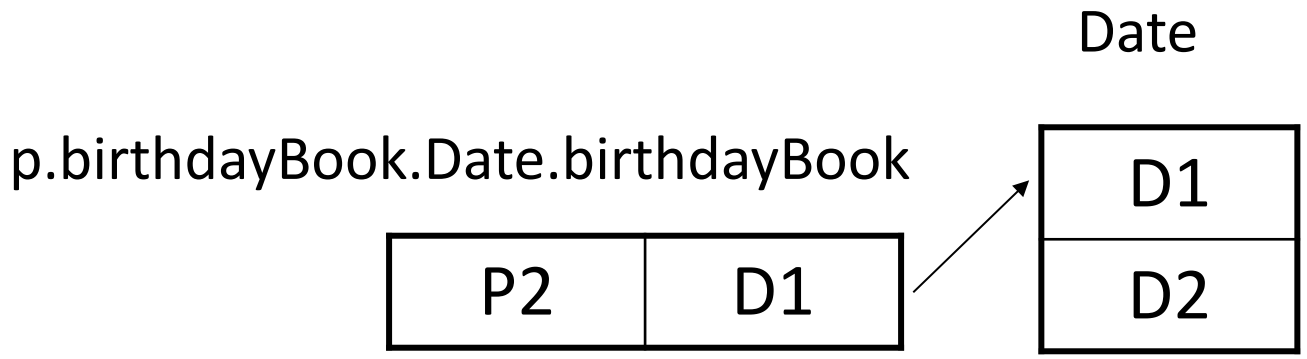
birthdayBook

| | | |
|---|---|---|
| ~~P1~~ | ~~P3~~ | ~~D1~~ |
| ~~P1~~ | ~~P4~~ | ~~D2~~ |
| ~~P3~~ | P2 | D1 |

```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

## Another example

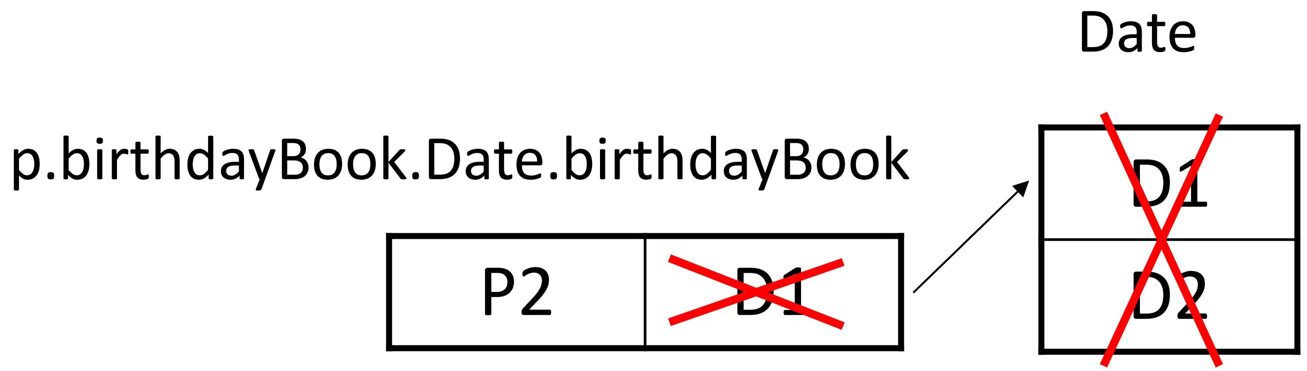
- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

## Another example

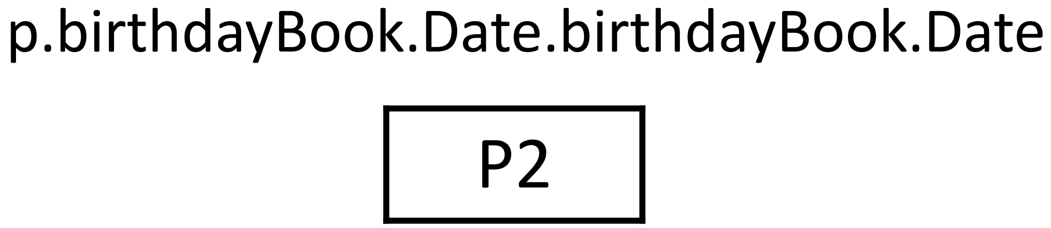
- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

## Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}



```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

## Another example

- Does our example world comply with the second fact?
- We will consider only p = {P1}

NO!

p.birthdayBook.Date.birthdayBook.Date

$\boxed{\text{P2}}$

```
all p: Person |
  p in p.birthdayBook.Date.birthdayBook.Date
```

## Functions and predicates

- Parametric expressions (predicates are logical, functions are not)
- Can be used to simplify expressions / give name to expressions

```
fun commonParents[x: Person, y: Person]: set Person {
  x.parents & y.parents
}

pred areSiblings[x: Person, y: Person] {
  some commonParents[x, y]
}

fact {
  no p: Person | areSiblings[p, p.spouse]
  …
}
```

## The Alloy Analyzer

# The Alloy Analyzer

- An environment for editing and analyzing Alloy documents
- Two possible analyses:
  - **Checking**: given an assertion (a predicate), does a model exist that falsifies the assertion?
  - **Simulation**: given a predicate, does a model exist that satisfies the predicate?
  - (note that they actually are equivalent)
- The analysis is **bounded**

## Structure of an Alloy document

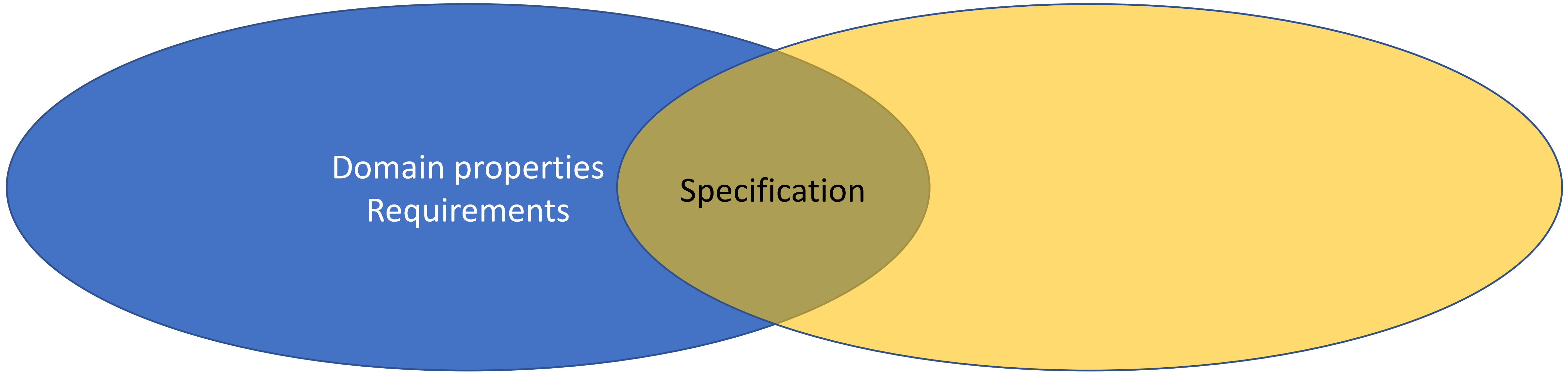
- **Module declaration**
- **Predicates/functions**: reusable expressions
- How the worlds must be structured:
  - **Signatures**: define types and relationships
  - **Facts**: further constraints
- Which properties should be true for a model:
  - **Assertions**: desired properties of the resulting models
  - **Commands**: instruct the Alloy Analyzer how to check assertions

## Facts and assertions

- Note the difference!
- Facts express **how the models of the document are:**
  - They describe the **domain properties**, i.e., the properties of the environment (e.g., reagent X explodes above some temperature)
  - They also describe the **specification**, i.e., how the machine must behave according to its design (e.g., under which conditions the system activates the cooling of the reagents tank)
- Assertions express **how we would like the models to be:**
  - They describe the **requirements** that we would like to be true (e.g., the chemical plant will never explode)
- A good specification always ensures the requirements

# In Jackson-Zave parlance…

🟦 = environment phenomena 🟨 = machine phenomena



The domain properties and the specification must imply the requirements

⬇

The facts must imply the assertions

## Writing assertions and commands

- An assertion is a (closed) predicate with the `assert` keyword
- The commands are:
  - `check` : takes an assertion and performs checking
  - `run` : takes a predicate or function and performs simulation
- Commands accept a list of **scopes** (i.e., the maximum sizes) for sets of atoms
- If checking fails it produces a **counterexample**
- If simulation succeeds it produces an **example**

## Examples of commands

```
assert noParentReflexivity { no x: Person | x in x.parents }
check noParentReflexivity for 2 but exactly 1 String

pred personWithoutParents[x: Person] { no x.parents }
run personWithoutParents for 1 but exactly 1 String

fun personParents[x: Person]: set Person { x.parents }
run personParents for 1 Date, exactly 1 String, exactly 2 Person
```

## “Bounded analysis” means…

- If checking does not find a counterexample, the assertion does not necessarily hold
- This because the counterexample could exist in a greater scope
- In general, if the scope is “big enough” it is **likely** the case that the assertion holds
- In some cases a finite scope exists allowing to perform a complete analysis (e.g., a Firewire bus has at most 63 devices)
- It is possible to perform bounded temporal analysis, not real temporal logic model checking

## The unbalance

| | |
|---|---|
| If checking fails the assertion is surely false… | …but if it succeeds, we cannot draw any conclusion |
| If simulation succeeds the Alloy document is surely consistent… | …but if it fails, we cannot draw any conclusion |

## Case study: Elevator scheduling

## The elevator scheduling problem

- A classic problem in software engineering
- See, e.g., Ghezzi, Jazayeri, Mandrioli, “Fundamentals of Software Engineering”, 2nd ed.

## Informal description (1)

- A system with n elevators must be installed in a building with m floors
- Each floor has two buttons, one to request an elevator going up, and one to request an elevator going down (the first and last floors are the obvious exceptions)
- These buttons have lights that switch on when they are pressed and switch off when
  - The elevator stops at the floor **and**
  - Either is moving in the required direction or has no outstanding request
- In this last case, if both buttons at the floor were pressed, the elevator must decide which direction to serve (and which button light to switch off) so to minimize the wait time for **both** requests

## Informal description (2)

- An elevator’s cabin has one button for each floor, to request the cabin to stop at that floor
- When a button in a cabin is pressed, a light switches on
- The light of a cabin’s button is switched off when the cabin stops at the corresponding floor
- All the requests from floors must be eventually serviced, with all floors given equal priority
- All the requests from cabins must be eventually serviced, with floors serviced sequentially in the direction of the cabin’s travel
- If the elevator has no request to serve, the cabins remain at their final destinations, waiting for further requests
- **Objective: design the elevator’s movement scheduler**

## Assumptions and modeling decisions

- We will consider the one cabin case for the sake of simplicity
- We drop the (hard? Infeasible?) requirement of minimizing the waiting time for both requests
- We do not consider real time, just transitions between different states of the system
- We model the status of buttons just with the status of the associated pending request (on, off), abstracting away pressure, lighting…
- We only consider the states where the cabin is at a floor, and abstract away the fact that the cabin might be cruising between two floors
- We do not model opening/closing of the doors

## Modelling

- Directions
  - Up and down
- Floors
  - Have an order (from the ground up to the last one)
- Cabin
  - Has a position (the current floor)
  - Moves according to the directives of the scheduler
- Scheduler:
  - Stores all the requests
  - Decides which is the next floor that the cabin must serve

## Movement and floors

```
module elevator

open util/ordering[Floor] as floors

sig Floor { }

abstract sig Move { }
one sig Up extends Move { }
one sig Down extends Move { }
```

**Polymorphic module** : defines a set of signatures/facts/etc. over another signature

**Abstract signatures** : have no atoms (inherit those of their subsignatures)

**Subsignatures** : all atoms of a subsignature are also atoms of the signature it extends

Multiplicity constraint

## The system (cabin + scheduler)

```
sig System {
  curFloor: Floor,
  curDirection: Move,
  intRequests: set Floor,
  extRequests: Floor -> Move,
  nextFloor: lone Floor
}
{
  …
}
```

← **Signature facts** : simplify writing a set of facts inherent to a signature’s atoms

# The system (cabin + scheduler)

```
sig System {
  curFloor: Floor,
  curDirection: Move,
  intRequests: set Floor,
  extRequests: Floor -> Move,
  nextFloor: lone Floor
}
facts {
  all this: System |
    …
}
```

## Representation invariant

- We need to enforce the representation invariants to exclude meaningless states
- The only invariant is disallowing “go down” requests from the ground floor and “go up” requests from the last floor (these buttons do not exist!)

# The system (cabin + scheduler)

```
…
open util/ordering[Floor] as floors
…
sig System {
  curFloor: Floor,
  curDirection: Move,
  intRequests: set Floor,
  extRequests: Floor -> Move,
  destFloor: lone Floor
}
{
  //The first and last floor have only one request button
  Down !in extRequests[floors/first[]]
  Up !in extRequests[floors/last[]]
  …
}
```

## The util/ordering module

- The `util/ordering[Sig]` module defines signatures/facts to impose a total order on `Sig`
- It also defines many functions and predicates:
  - `first[]`, `last[]`: the first / last element of `Sig`
  - `prev[s: Sig]`, `next[s: Sig]` : the previous / next element of `s`
  - `prevs[s: Sig]`, `nexts[s: Sig]` : all the predecessors / successors of `s`
  - `lt[s1, s2: Sig]`, `gt[s1, s2: Sig]` : does `s1` come before / after `s2`?
  - `lte`, `gte` : before or equal / after or equal

## The scheduler (1)

- We do not define a specific (imperative?) algorithm for the scheduler
- We exploit the power of logic to define a set of constraints that any sensible scheduling algorithm should have
- However, these constraints should not be too abstract that they cannot be considered “implementable”
- In Jackson-Zeve terms, we are writing a **specification** for scheduling

## The scheduler (2)

- The shared phenomena between the scheduler (machine) and the elevator (environment) are
  - The pending requests: `s.intRequests`, `s.extRequests`
  - The destination floor: `s.destFloor`
- Therefore, the specification for the scheduler must tell:
  - When a pending request is considered served and removed from the list of pending requests
  - How the destination floor is calculated

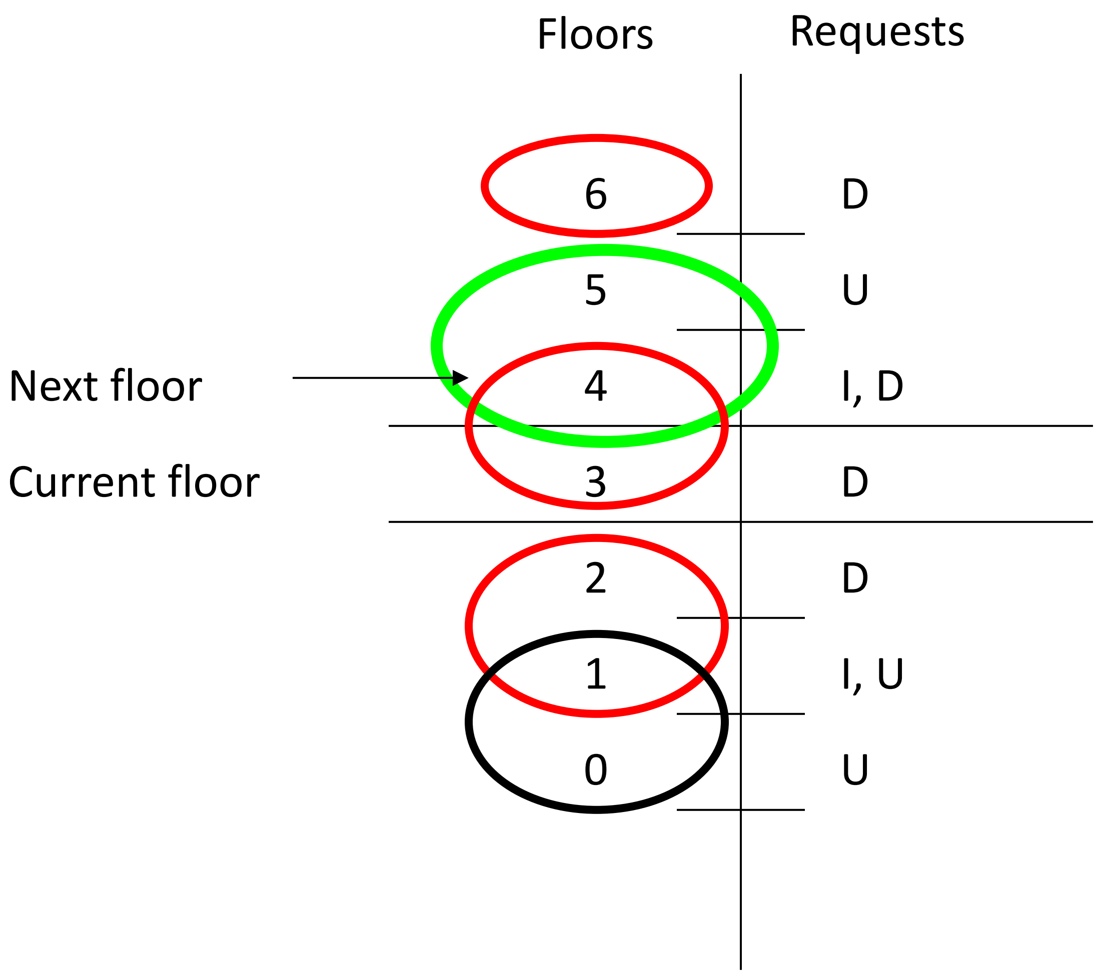
## Calculation of destination floor

- Idea: divide the pending requests in three sets with different priority
    - Top priority: requests that can be served without switching the direction of the cabin movement
    - Mid priority: requests that can be served by switching the direction of the cabin movement once
    - Low priority: requests that can be served by switching the direction of the cabin movement twice
- The destination floor is picked with some criterion from the nonempty set with highest priority

## Definition of priority sets

- If the cabin’s current direction is up:
    - Top priority:
        - Cabin requests to floors above the current one, and
        - Requests coming from floors above the current one for going up
    - Mid priority:
        - Cabin requests to floors below the current one, and
        - Request coming from floors for going down
    - Low priority:
        - Requests coming from floors below the current one for going up
- Dual if the cabin’s current direction is down

# Definition of priority sets, example



Current direction: U

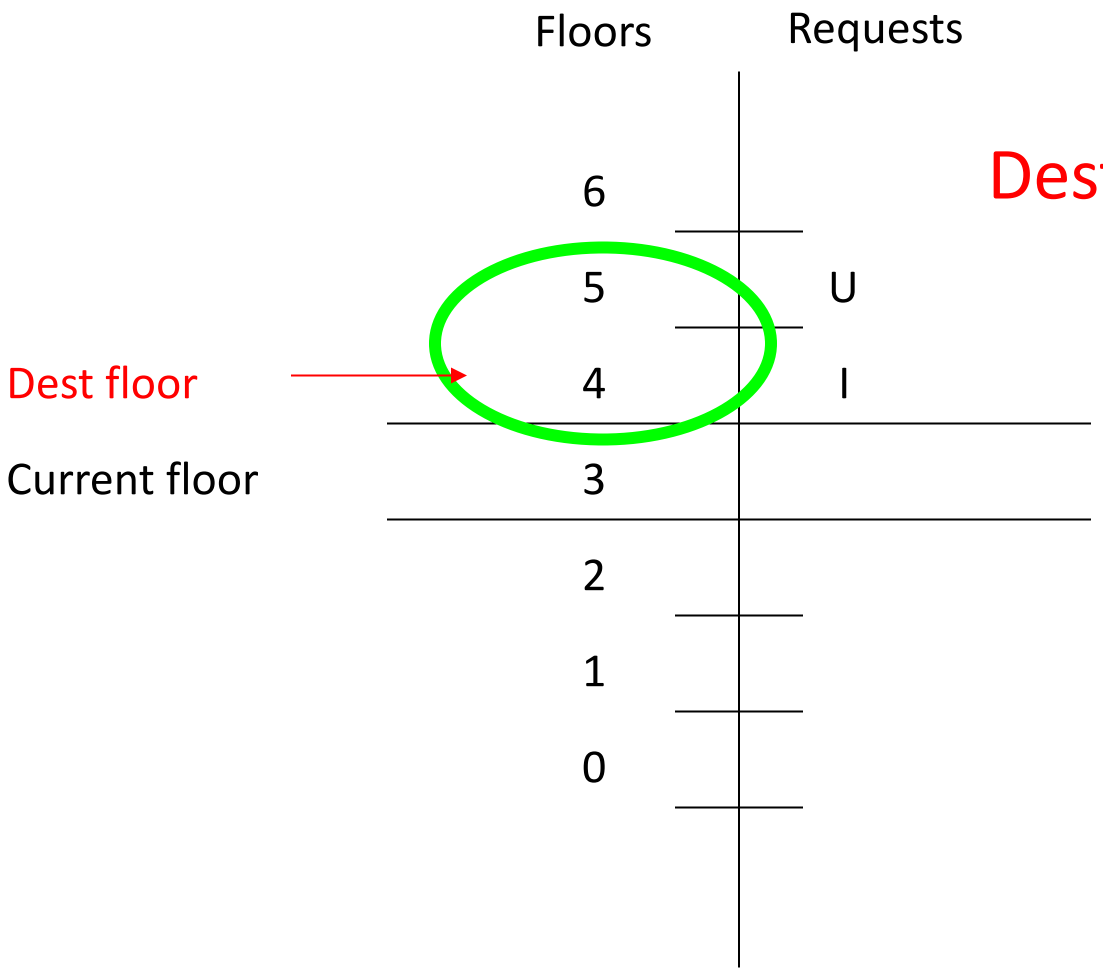
**TOP**: ≥ && (I || U)
**MID**: (< && I) || D
**LOW**: < && U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

## Picking the destination floor

- Once selected the right priority set, which floor shall we pick from it as the destination one?
- Idea: pick the one that maintains the minimal number of movements
- More precisely: pick the floor that, if served, will allow to serve the other floors in the same priority class **without switching direction** (assuming that no other requests arrive)

# Picking the destination floor, example (1)

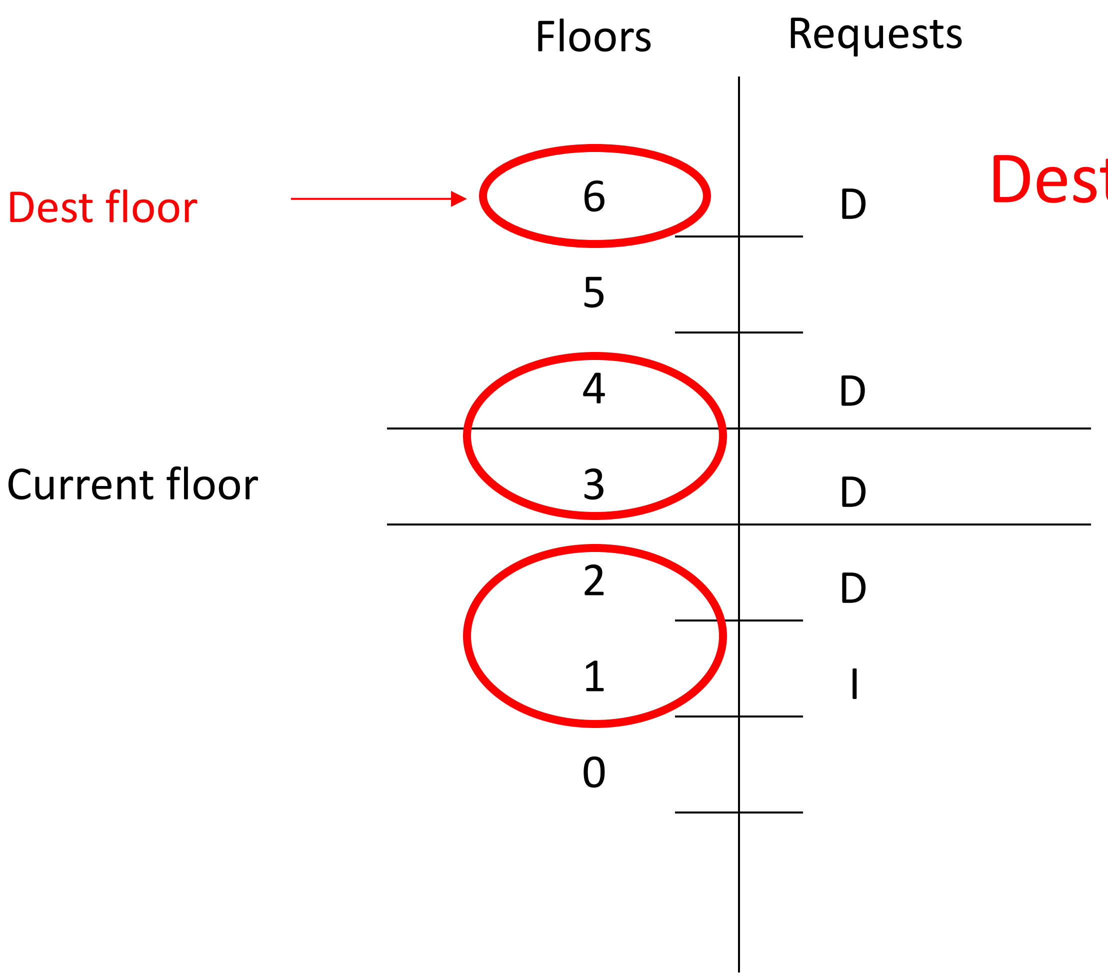


Dest floor = closest in the current direction

Current direction: U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Picking the destination floor, example (2)

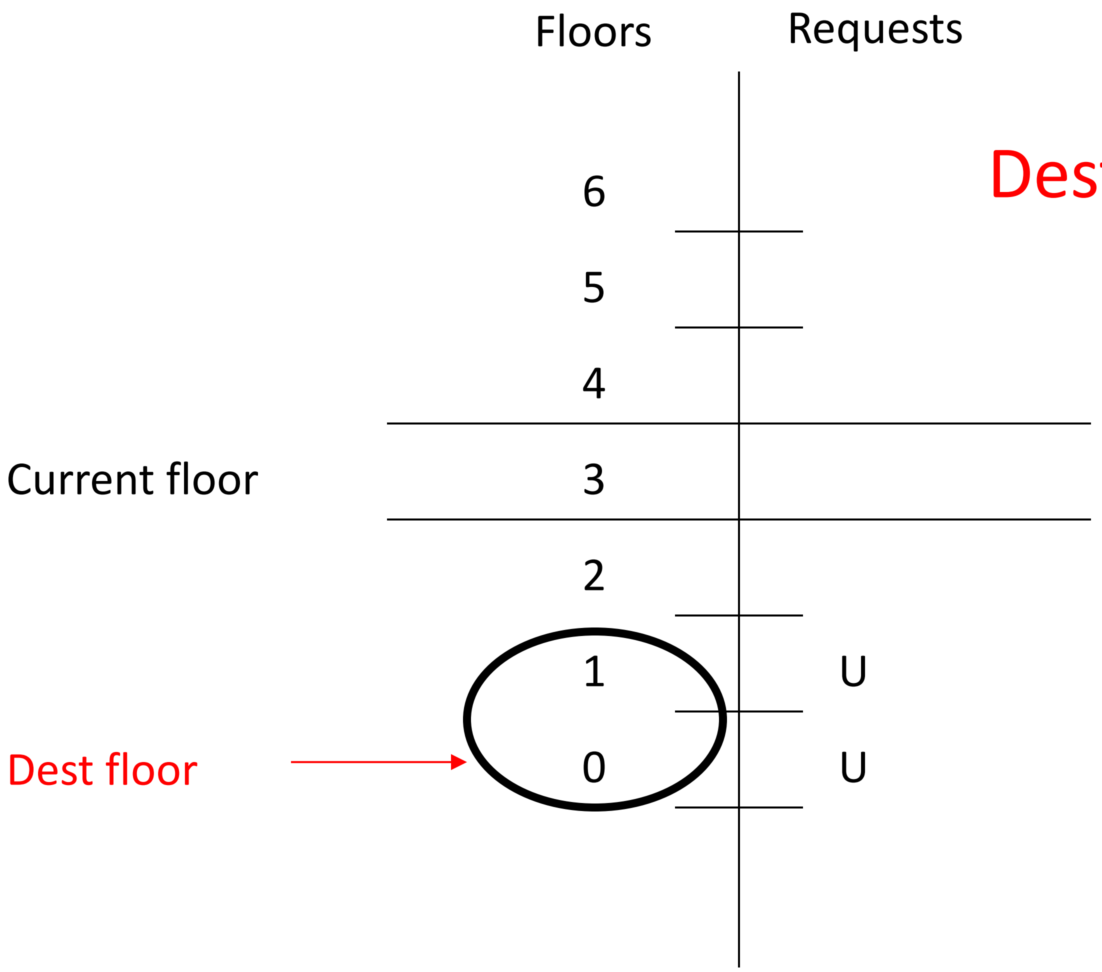


Dest floor = farthest in the current direction

Current direction: U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Picking the next floor, example (3)



Dest floor = farthest in the opposite direction

Current direction: U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

## Scheduler, calculation of destination floor (1)

```
fun priTopUp[s: System]: set Floor {
  { f: Floor | floors/gte[f, s.curFloor] &&
               (f in s.intRequests || Up in s.extRequests[f]) }
}
```

**Set comprehension** : defines a set of atoms

Alternative join notation, equivalent to `f.(s.extRequests)`

# Scheduler, calculation of destination floor (2)

```
fun priTopUp[s: System]: set Floor {
  { f: Floor | floors/gte[f, s.curFloor] &&
               (f in s.intRequests || Up in s.extRequests[f]) }
}

fun priMidUp[s: System]: set Floor {
  { f: Floor | (floors/lt[f, s.curFloor] && f in s.intRequests) ||
               Down in s.extRequests[f]
}

fun priLowUp[s: System: set Floor {
  { f: Floor | floors/lt[f, s.curFloor] && Up in s.extRequests[f] }
}

//Dual if current direction is down
```

## Scheduler, calculation of destination floor (3)

```
pred minFloor[f: Floor, fs: set Floor] {
  f in fs && (no f’: Floor | f’ in fs && floors/lt[f’, f])
}

pred maxFloor[f: Floor, fs: set Floor] {
  f in fs && (no f’: Floor | f’ in fs && floors/lt[f, f’])
}
```

## Scheduler, calculation of destination floor (4)

```
sig System {
  …
}
{
  …
  //calculates the destination floor
  curDirection = Up => (
    (some priTopUp[this] &&
      some destFloor && minFloor[destFloor, priTopUp[this]]) ||
    (no priTopUp[this] && some priMidUp[this] &&
      some destFloor && maxFloor[destFloor, priMidUp[this]]) ||
    (no priTopUp[this] && no priMidUp[this] && some PriLowUp[this] &&
      some destFloor && minFloor[destFloor, priLowUp[this]]) ||
    (no priTopUp[this] && no priMidUp[this] && no priLowUp[this] &&
      no destFloor)
  //Dual if curDirection = Down
}
```

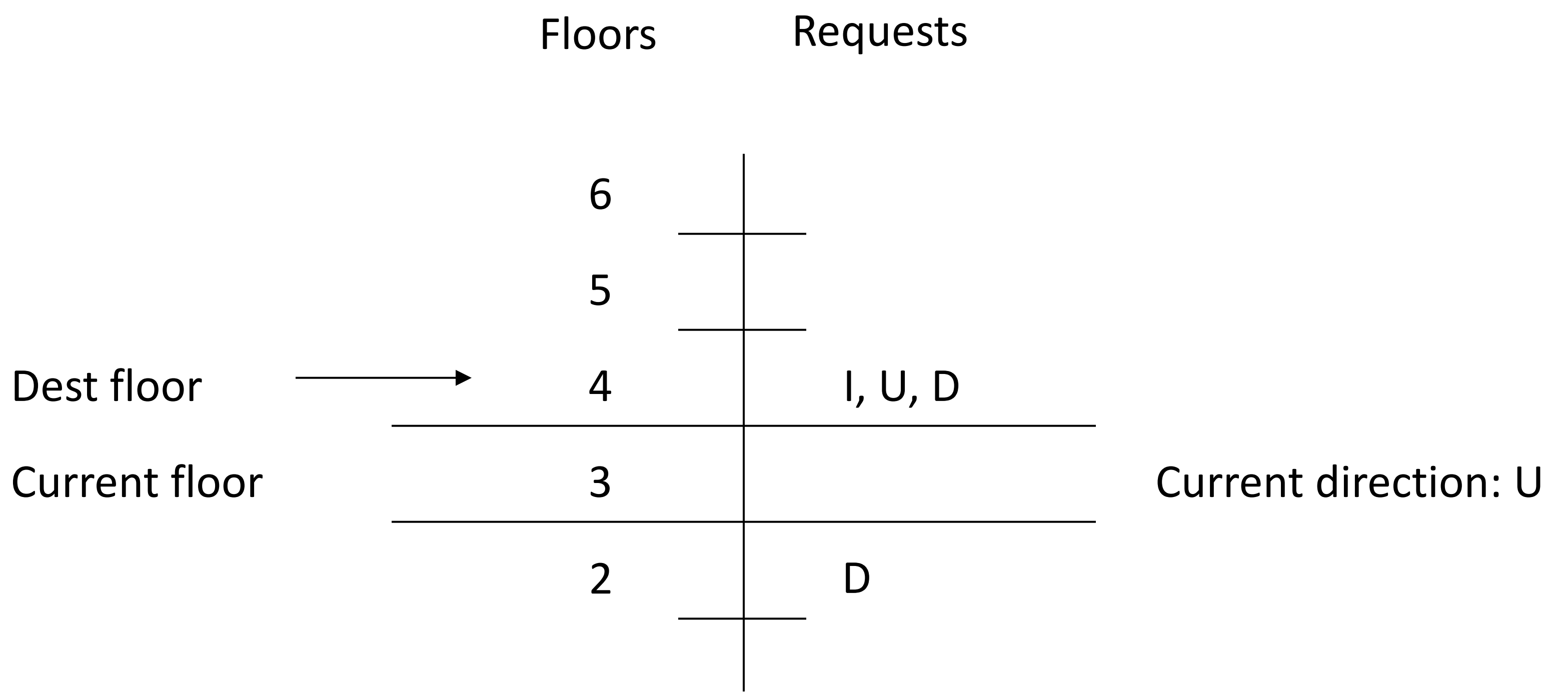
## Calculation of the served requests

- As the cabin arrives at the destination floor, the scheduler must remove all the requests that it considers served
- Note that **not** all the requests for the current floor are considered served by the scheduler:
    - If there is a pending cabin request, this is always considered served
    - If there is a pending floor request for a given direction, this is considered served only if the scheduler will move the cabin in the right direction
- Result: at most one cabin request and one floor request are considered served each time a cabin arrives at a floor

## A complication

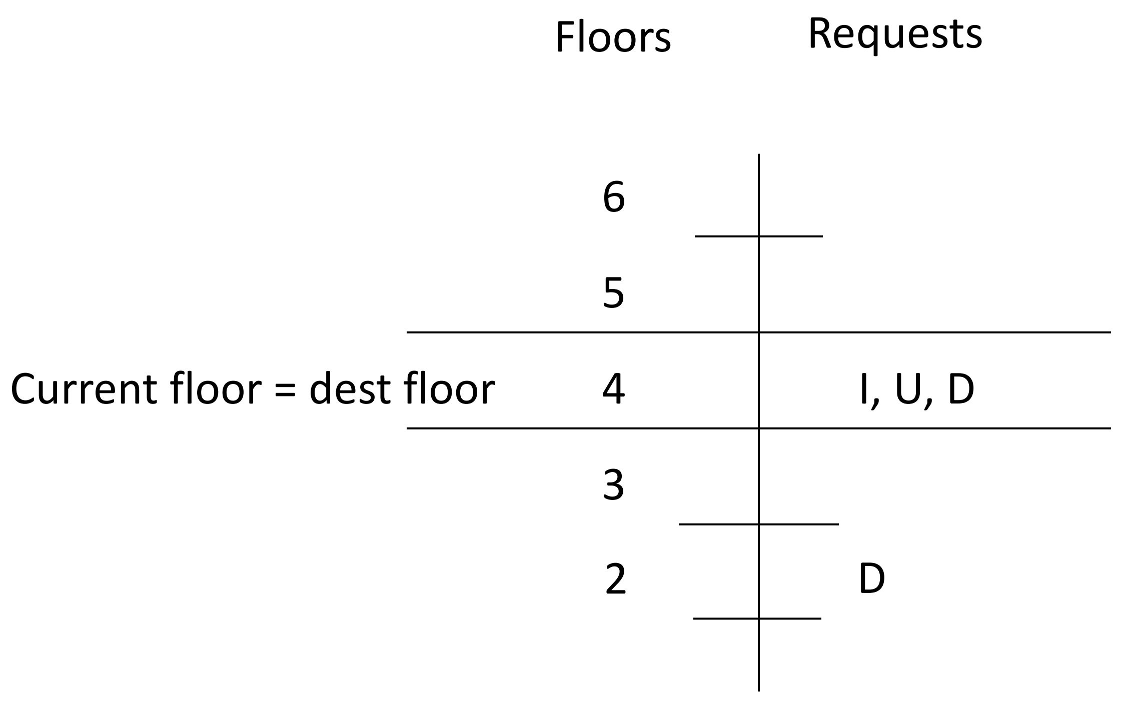
- As a request is considered served, the scheduler should recalculate the next direction to move the cabin
- But at the same time, to calculate the next direction it must know which requests have been served
- We will solve this circularity by representing calculation of served requests with state transitions (actions):
  - The system transitions to a state where the internal request (if present) is removed, and the schedule recalculated
  - If the system is stuck at the same floor, it transitions to a state where the “best” external request (if present) is removed, and the schedule recalculated again
  - If the system is still stuck at the same floor, the last external request is removed

# Served requests, example



I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example

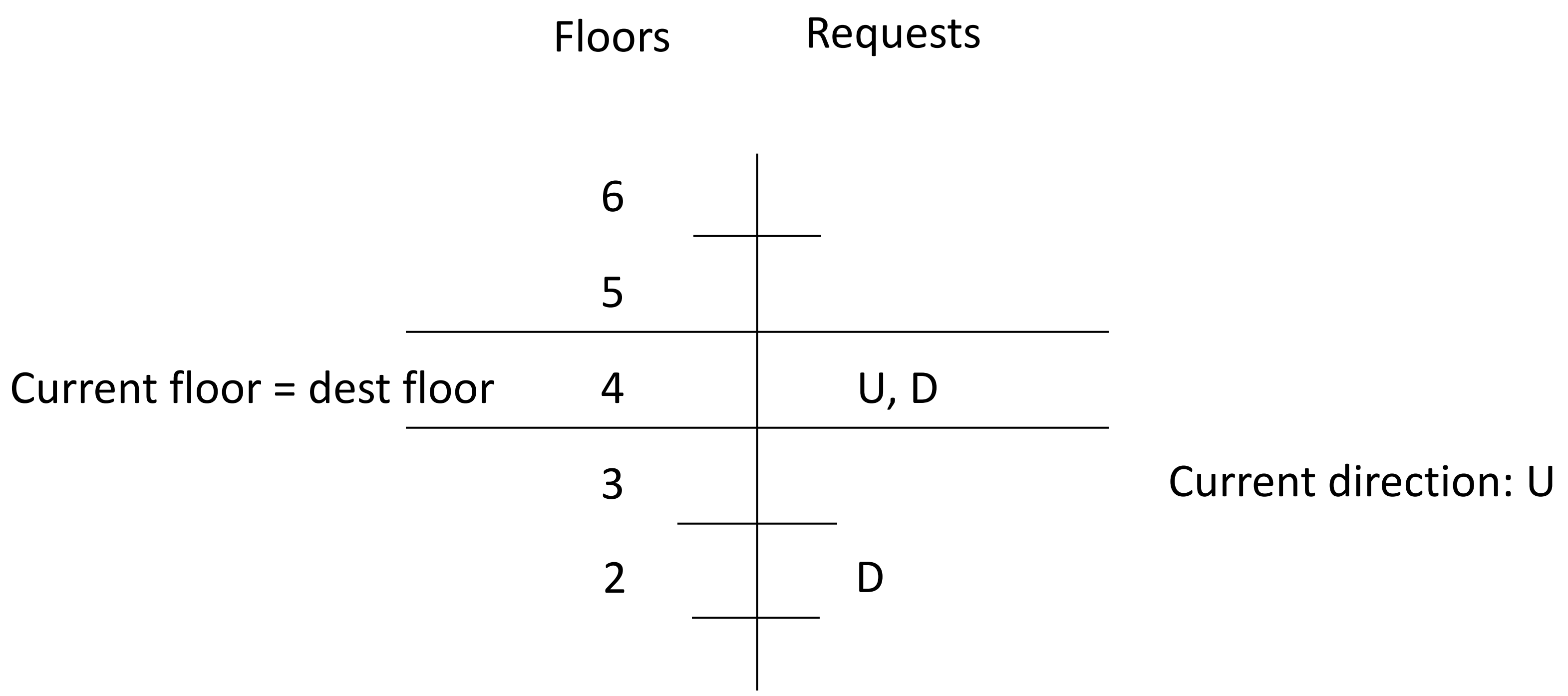


Current direction: U

Request served: I

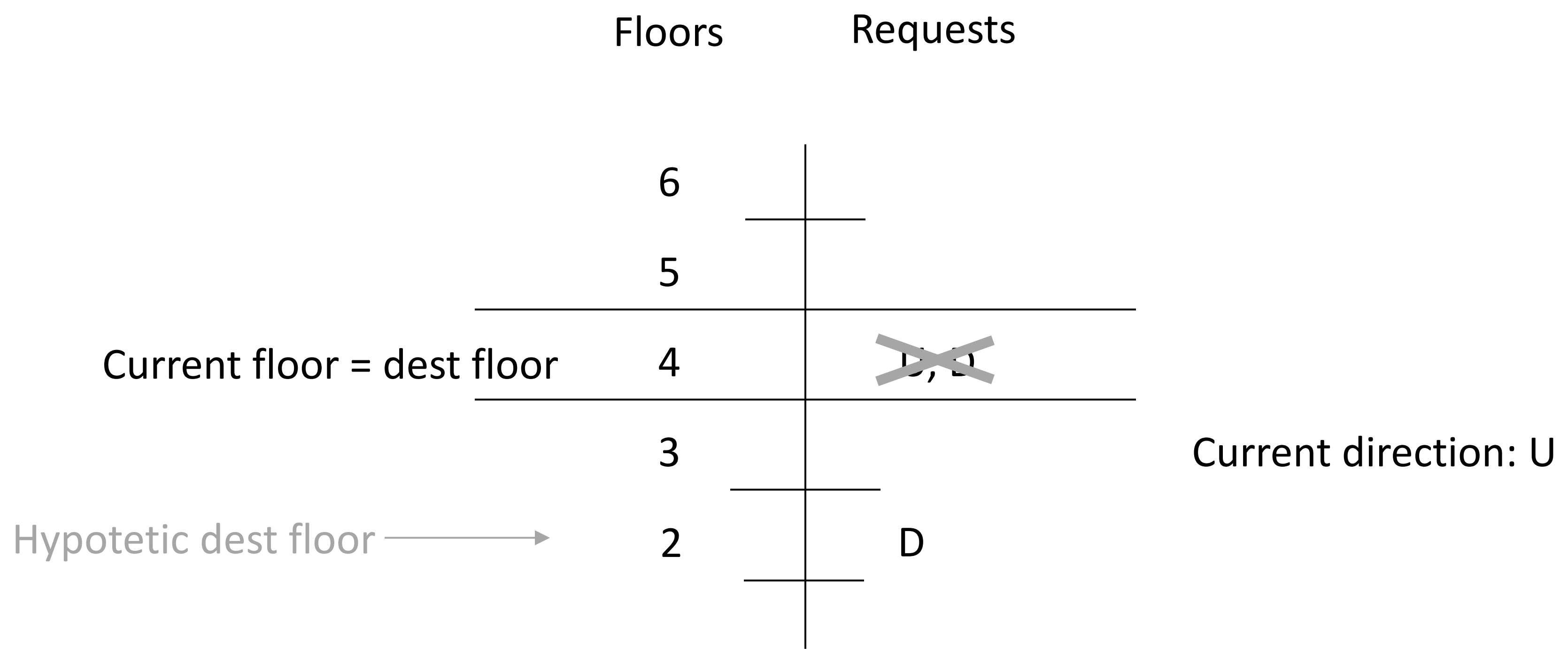
I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example



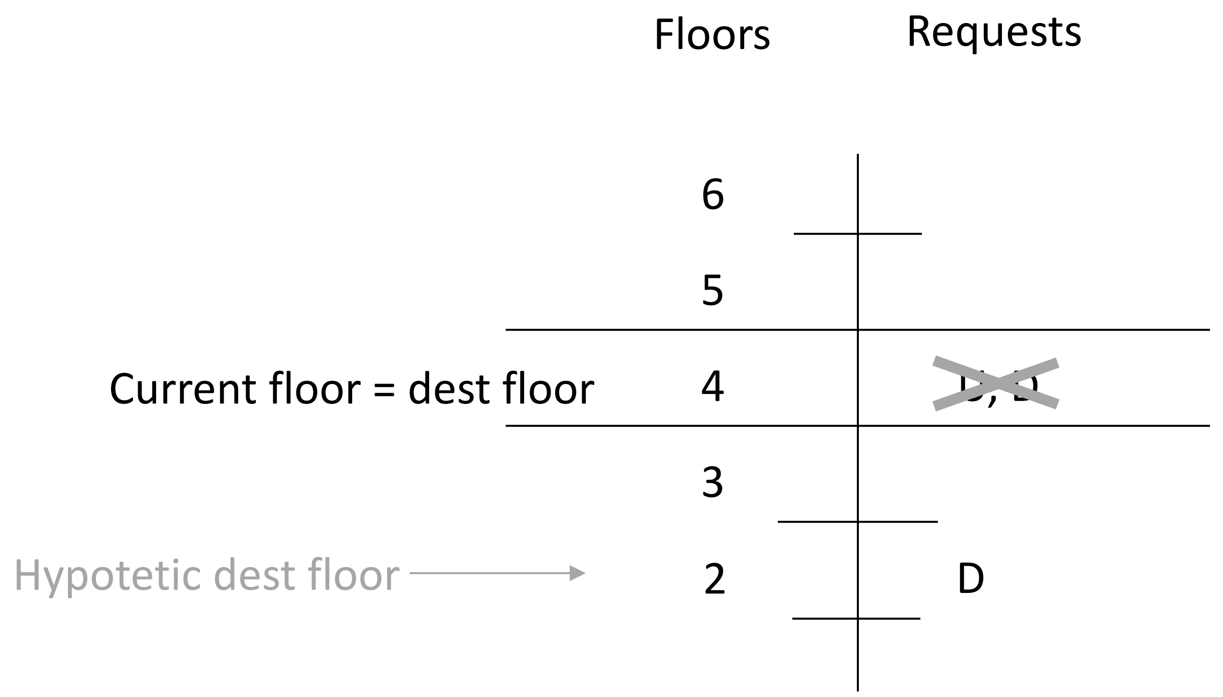
I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example



I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example

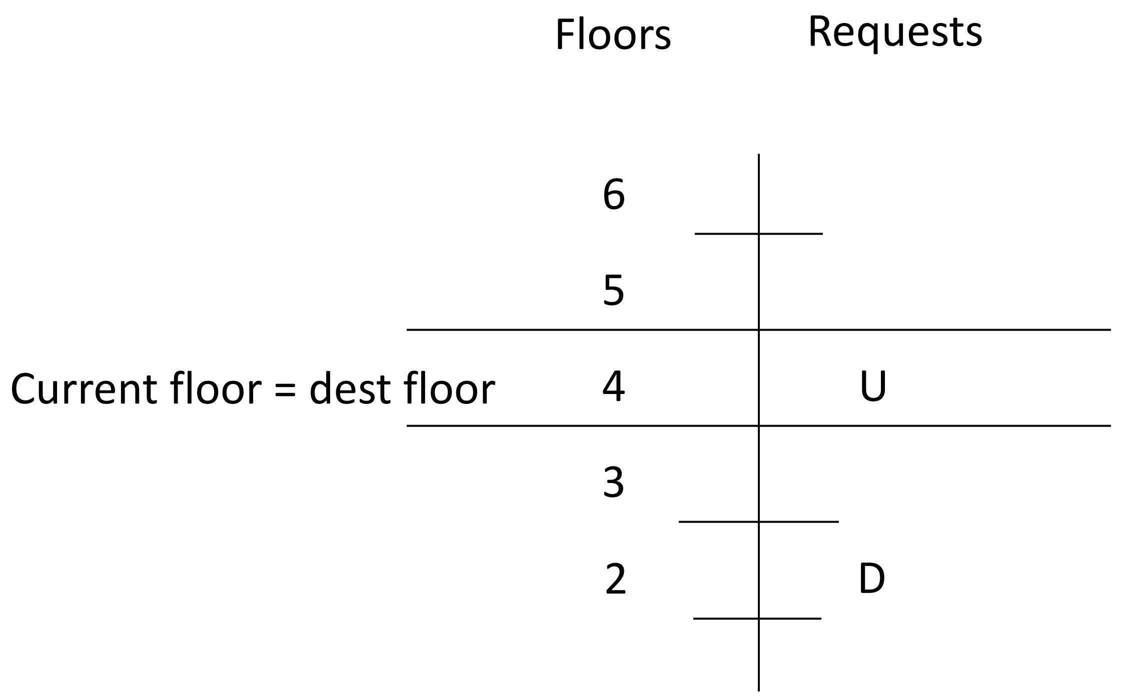


Current direction: U

Request served: D

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

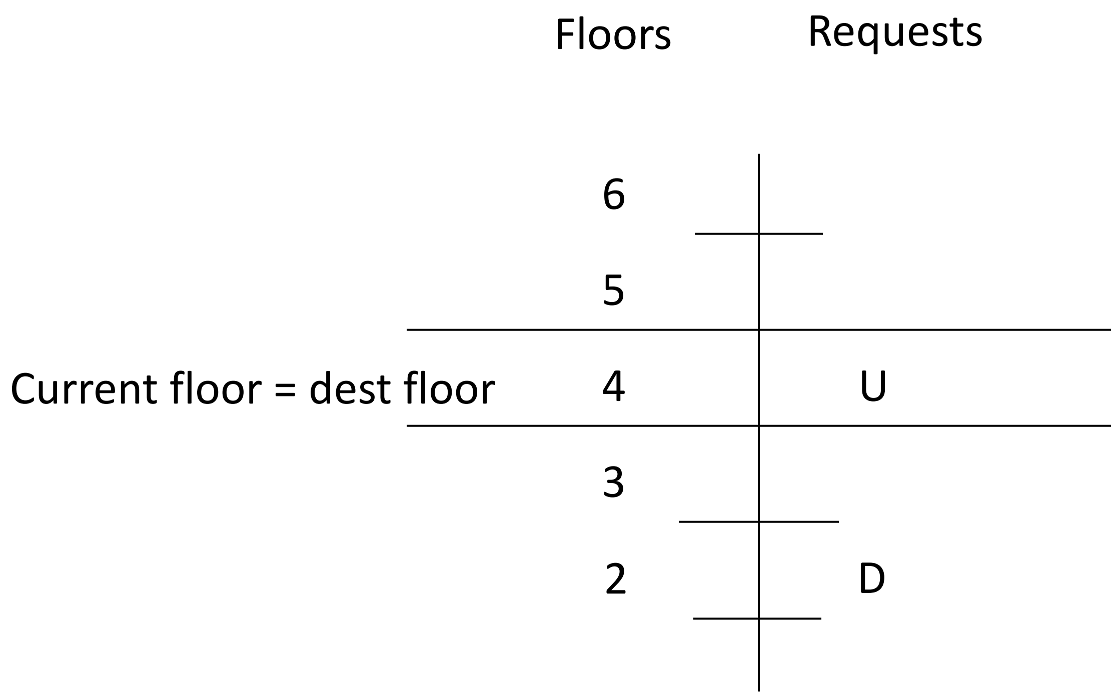
# Served requests, example



Current direction: U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example

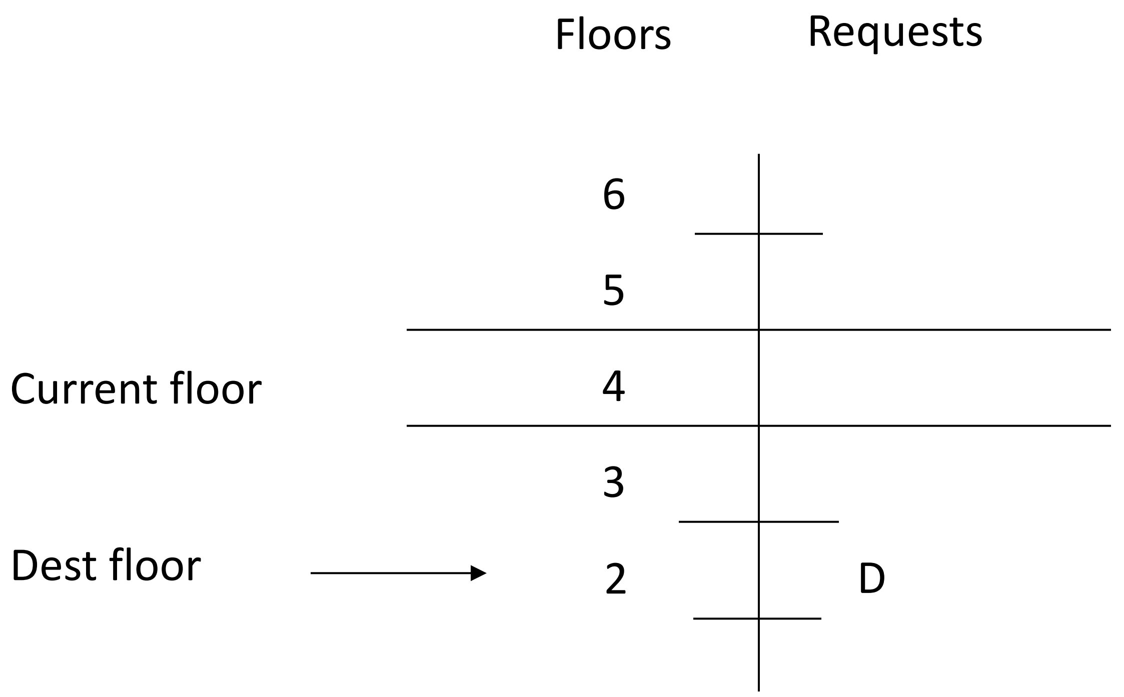


Current direction: U

Still stuck: delete U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

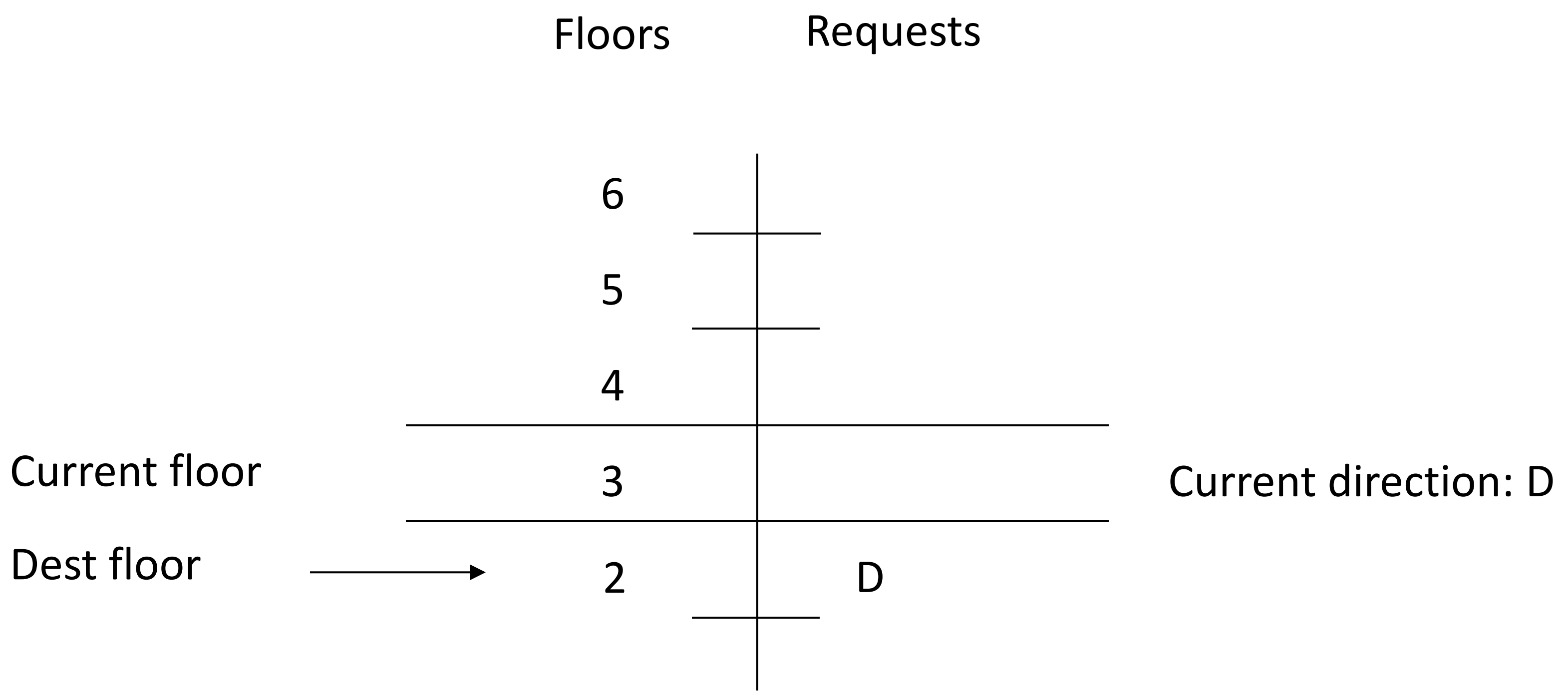
# Served requests, example



Current direction: U

I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

# Served requests, example



I = internal (cabin) request
D = external (floor) request for going down
U = external (floor) request for going up

## Some refactoring

- We need to do some refactoring to avoid repeating the same formulas more than once
- We put the calculation of the destination floor in a predicate, so it can be used to calculate the hypothetic destination floor
- For the same reason we also need to change the parameters of the auxiliary functions that define the priority sets

# Calculation of destination floor, refactored (1)

```
fun priTopUp[curFloor: Floor, intRequests: set Floor, extRequests: Floor -> Move]:
set Floor {
  { f: Floor | floors/gte[f, curFloor] &&
               (f in intRequests || Up in extRequests[f]) }
}

//Similarly for the other functions
```

## Calculation of destination floor, refactored (2)

```
pred isTheDestFloor[destFloor: lone Floor, curDirection: Move, curFloor: Floor,
                    intRequests: set Floor, extRequests: Floor -> Move] {
  curDirection = Up => (
    (some priTopUp[curFloor, intRequests, extRequests] &&
      some destFloor && minFloor[destFloor, priTopUp[…]]) ||
    (no priTopUp[…] && some priMidUp[…] &&
      some destFloor && maxFloor[destFloor, priMidUp[…]]) ||
    (no priTopUp[…] && no priMidUp[…] && some PriLowUp[…] &&
      some destFloor && minFloor[destFloor, priLowUp[…]]) ||
    (no priTopUp[…] && no priMidUp[…] && no priLowUp[…] &&
      no destFloor)
  //Dual if curDirection = Down
}

fun theDestFloor[curDirection: Move, curFloor: Floor, intRequest: set Floor,
                 extRequest: Floor -> Move]: lone Floor {
  { destFloor: Floor |
    isTheDestFloor[destFloor, curDirection, curFloor, intRequest, extRequest] }
}
```

# Calculation of destination floor, refactored (3)

```
sig System {
  …
}
{
  …
  //calculates the destination floor
  destFloor = theDestFloor[curDirection, curFloor, intRequests, extRequests]
}
```

## State transitions

- We need to represent state transitions:
  - Triggered by inputs
  - Or spontaneous
- In a noble tradition we model state transitions with predicates that relate the pre-state to the post-state
- If the transition has inputs/outputs, we add them as parameters of the predicate
- We call **actions** these kind of predicates

## Actions

- Each action may have a **precondition** and has a postcondition
- Precondition: sub-predicate identifying the pre-states in which the action may fire
- Postcondition: sub-predicate relating the post-state to the pre-state

## InternalPush, ExternalPush

- The action that correspond to the fact that someone pushed a cabin button or a floor button
- `InternalPush` has as parameter the destination floor
- `ExternalPush` has as parameters the floor where the button is, and the required direction
- The effect of the actions is just to add an item to the pending requests

# InternalPush

```
//someone pushes the button in the cabin to floor f
pred InternalPush[s, s': System, f: Floor] {
  //Precondition:
  f !in s.intRequest //no stutter

  //Postcondition:
  s'.curFloor = s.curFloor
  s'.curDirection = s.curDirection
  s'.intRequest = s.intRequest + f
  s'.extRequest = s.extRequest
}
```

# ExternalPush

```
//someone pushes the button at floor f for direction m
pred ExternalPush[s, s': System, f: Floor, m: Move] {
  //Precondition:
  (f -> m) !in s.extRequest //no stutter

  //Postcondition:
  s'.curFloor = s.curFloor
  s'.curDirection = s.curDirection
  s'.intRequest = s.intRequest
  s'.extRequest = s.extRequest + (f -> m)
}
```

## MoveToNextFloor

- This action activates spontaneously (i.e., no input) when the current floor is different from the next floor calculated by the scheduler
- The effect is to update the current floor (i.e., to move the cabin), and nothing else: Other actions calculate the new pending requests
- Note that it moves one floor at a time!

## MoveToNextFloor

```
pred MoveToNextFloor[s, s': System] {
  //Precondition
  some s.destFloor //otherwise, the cabin must stand still
  s.curFloor != s.destFloor //same

  //Postcondition:
  floors/lt[s.destFloor, s.curFloor] =>
    (s'.curFloor = floors/prev[s.curFloor] && s'.curDirection = Down)
  else
    (s'.curFloor = floors/next[s.curFloor] && s'.curDirection = Up)
  s'.intRequest = s.intRequest
  s'.extRequest = s.extRequest
}
```

# Calculation of the served requests

- The calculation of the served requests is performed by three actions:
  - `ServeIntRequest` is triggered when the cabin is at the destination floor, if there is an internal request for the floor, and removes it
  - `ServeExtRequestHypotheticalDir` is triggered after `ServeIntRequest`, if the cabin is still at the destination floor, and removes the external request in the hypothetical direction
  - `ServeExtRequestStuck` is triggered after `ServeExtRequestHypotheticalDir`, if the cabin is still at the destination floor, and removes the remaining external request (after it, the scheduler **will** calculate a different destination floor).

## ServeIntReq

```
pred ServeIntReq[s, s': System] {
  //Precondition
  s.curFloor = s.destFloor
  s.curFloor in s.intRequests //no stutter

  //Postcondition
  s'.curFloor = s.curFloor
  s'.curDirection = s.curDirection
  s'.intRequest = s.intRequest - s.curFloor
  s'.extRequest = s.extRequest
}
```

## Calculation of hypothetical direction

```
fun hypotheticalDestFloor[s: System]: lone Floor {
  theDestFloor[s.curDirection, s.curFloor, s.intRequests,
               s.extRequests - (s.curFloor -> Move)]
}

fun hypotheticalDir[s: System]: lone Move {
  some hypotheticalDestFloor[s] => (
    floors/lt[s.curFloor, hypotheticalDestFloor[s]] =>
      Up
    else
      Down)
  else
    none
}
```

# ServeExtRequestHypotheticalDir

```
pred ServeExtRequestHypotheticalDir[s, s': System] {
  //Precondition
  s.curFloor = s.destFloor
  s.curFloor !in s.intRequest //already served internal requests
  some hypotheticalDir[s]
  hypotheticalDir[s] in s.extRequests[s.curFloor]

  //Postcondition
  s'.curFloor = s.curFloor
  s'.curDirection = s.curDirection
  s'.intRequest = s.intRequest
  s'.extRequest = s.extRequest – (s.curFloor -> hypotheticalDir[s])
}
```

# ServeExtRequestStuck

```
pred ServeExtRequestStuck[s, s': System] {
  //Precondition
  s.curFloor = s.destFloor
  s.curFloor !in s.intRequest
  no hypotheticalDir[s] ||
    hypotheticalDir[s] !in s.extRequests[s.curFloor]
    //already served external request in the hypothetical direction

  //Postcondition
  s'.curFloor = s.curFloor
  s'.curDirection = s.curDirection
  s'.intRequest = s.intRequest
  s'.extRequest = s.extRequest - (s.curFloor -> Move)
}
```

## Assertions

- Is the (complex) scheduler we designed a good scheduler?
- To answer the question, we need to identify a set of **requirements** that define what a “good scheduler” is, and then verify that they hold
- In other words, we need to write a set of assertions capturing these requirements

## Bounded temporal analysis

- What if we want to write assertions on **computations**?
- We need to precisely define with a predicate what a “computation” is:
  - A sequence of states such that
  - The first state in the sequence is a suitable initialization state, and
  - Each subsequent pair of states in the sequence is related by an action (for some input)
- We can further refine the concept and write predicate for particular kinds of computations

# Bounded temporal analysis (1)

```
…
open util/ordering[System] as systems
…
pred Init[s: System] {
  //Postcondition
  s.curFloor = floors/first[]
  s.curDirection = Up
  no s.intRequest
  no s.extRequest
  no s.destFloor
}
```

## Bounded temporal analysis (2)

```
pred Step[s, s': System] {
    ( some f: Floor | InternalPush[s, s', f] ) ||
    ( some f: Floor, m: Move | ExternalPush[s, s', f, m] ) ||
    MoveToNextFloor[s, s'] ||
    ServeIntRequest[s, s'] ||
    ServeExtRequestHypotheticalDir[s, s'] ||
    ServeExtRequestStuck[s, s']
}

//A computation starting from an arbitrary state
pred ComputeFromAny[] {
  all s, s': System | s' = systems/next[s] => Step[s, s’]
}
```

## Bounded temporal analysis (3)

```
//An unconstrained computation
pred Compute[] {
  Init[systems/first[]]
  ComputeFromAny[]
}

//A computation without additional requests
pred ComputeFromAnyNoReq[] {
  ComputeFromAny[]

  //No additional requests
  all s, s': System | s' = systems/next[s] =>
    no f: Floor, m: Move |
    ( InternalPush[s, s', f] || ExternalPush[s, s', f, m] )
}
```

## Some possible requirements (“static”)

- **System is consistent**: There is at least one `System` that satisfies all the constraints
- **The destination floor is requested**: The scheduler always selects the destination floor among the pending requests
- **No destination implies no request**: If the scheduler does not select any destination floor, there is no pending request
- **The scheduler is deterministic**: The destination floor univocally depends on the current floor, the current direction, and the pending requests

## Some possible requirements (“dynamic”)

- **System has computations**: At least one computation must exist
- **A request stays scheduled until is served**: The destination floor does not change until the initial request is initially served, if no additional requests are issued in the meantime
- **The cabin does not get stuck at the destination floor**: After the cabin arrives at the destination floor, if there are other pending requests at different floors, then eventually the scheduler will select a new, different destination floor

## Liveness

- Does the designed elevator system progress, i.e., is every request eventually served?
- We have two flavors of liveness that we may want to check:
  - (easy) An issued request is eventually served, if after it is issued no other request is issued
  - (hard) An issued request is eventually served, unconditionally
- The Alloy Analyzer cannot help us prove neither, but can help us gaining some confidence that the easy one holds

## Extra

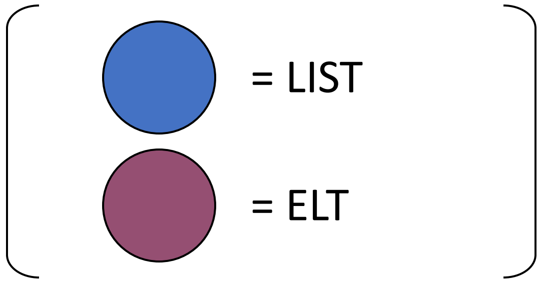
## (Non)determinism

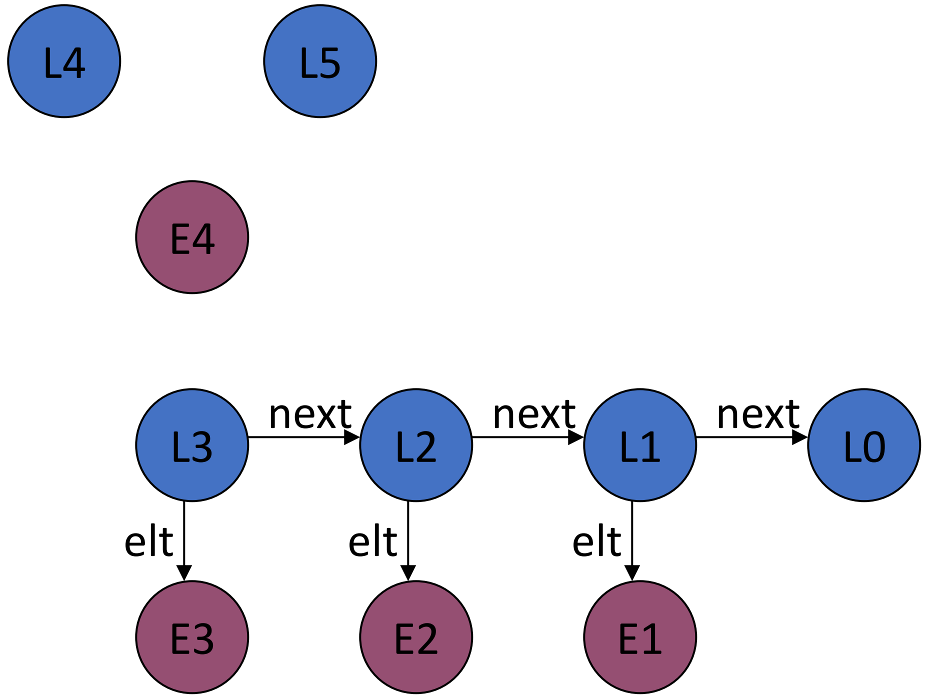
- Our Alloy document is very likely to be nondeterministic
- Deterministic = at most one result, nondeterministic = more than one
- Nondeterminism is due to at least these factors:
  - Arbitrary inputs (underspecified environment)
  - Arbitrary choice of next action (actions have overlapping preconditions)
  - Nonfunctional actions (intrinsically nondeterministic)
- But there is another, more subtle factor…

## Example (1)

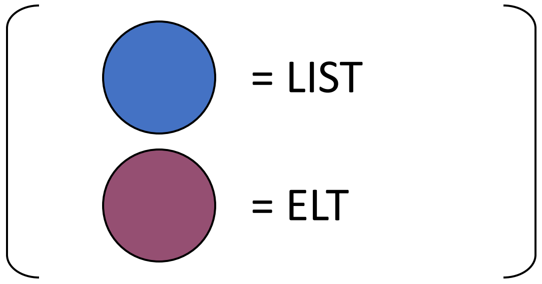
```
sig Element { }

abstract sig List { }
one sig EmptyList extends List { }
sig NonemptyList extends List {
  elt: Element,
  next: List
}

fact acyclic { all p: List | p !in p.^next }

pred cons(ls, ls’: List, elt: Element) {
  ls’.elt = elt && ls’.next = ls
}

//Can you spot the nondeterminism of cons?
```

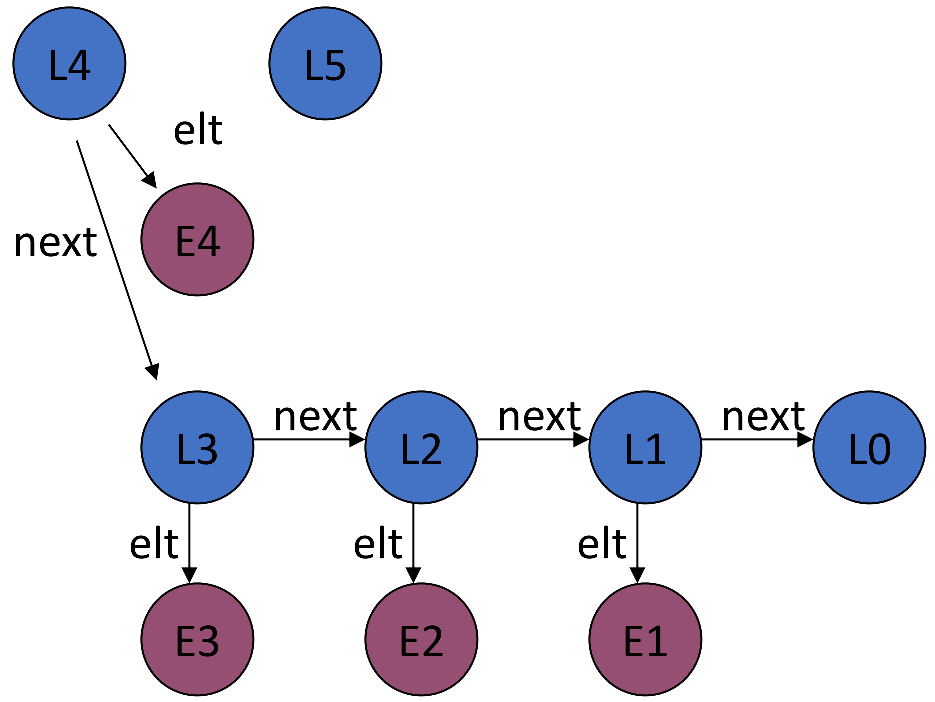
## Example (2)



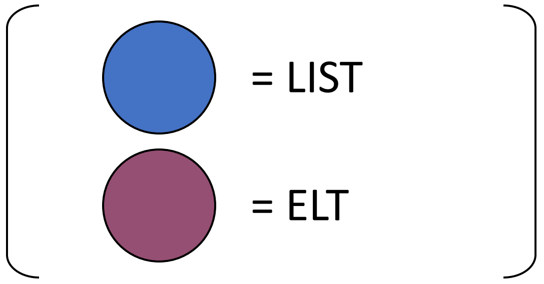


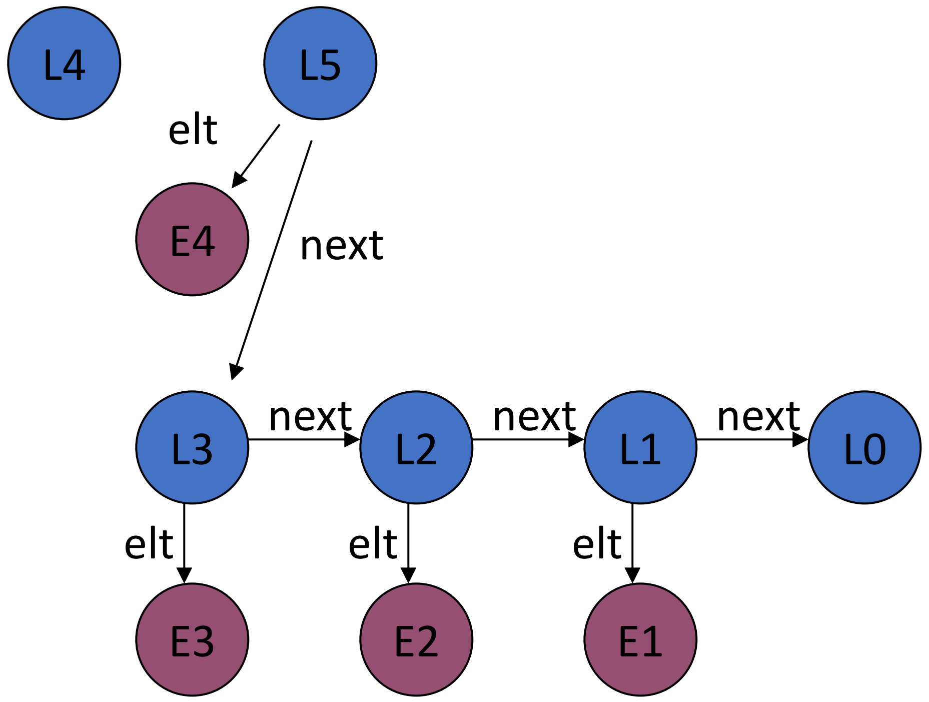
# Example (2)


= LIST
= ELT


L4
L5
elt
next
E4
L3
L2
L1
L0
next
next
next
elt
elt
elt
E3
E2
E1

## Example (2)


= LIST
= ELT


L4
L5
elt
E4
next
L3
next
L2
next
L1
next
L0
elt
elt
elt
E3
E2
E1

## Avoiding nondeterminism

- To make an Alloy document deterministic, you need to add suitable facts to it
  - Constrain the environment to specify the sequence of inputs
  - Refine actions so that their preconditions do not overlap and their postconditions are deterministic
- Avoiding the last form of nondeterminism (presence of isomorphic worlds) is not in general possible

## Stuttering

- We took special care in avoiding stuttering when defining actions
- We say that we have stuttering when it is possible that an action does not yield a post-state different from the pre-state
- In other word, for some action `act`, there are systems/inputs such that `act[s, s', inputs]` and `s.curFloor = s'.curFloor`, `s.curDirection = s'.curDirection`, …
- Stuttering yields computations that do not progress, which turn to be unduly counterexamples to assertions of the kind “eventually something happens”

## Bibliography

- For Alloy:
  - Daniel Jackson, *Software Abstractions: Logic, language, and analysis.* Revised edition. MIT Press.
- For requirements engineering:
  - Michael Jackson, The world and the machine. In *Proceedings of ICSE’95*. ACM Press.
  - Pamela Zave, Michael Jackson. *Four dark corners of requirement engineering*. ACM TOSEM, 6(1), January 1997.