POLITECNICO
MILANO 1863

Software Engineering 2

Alloy Part 1:

Introduction
First example: the address book
Second example: the family tree

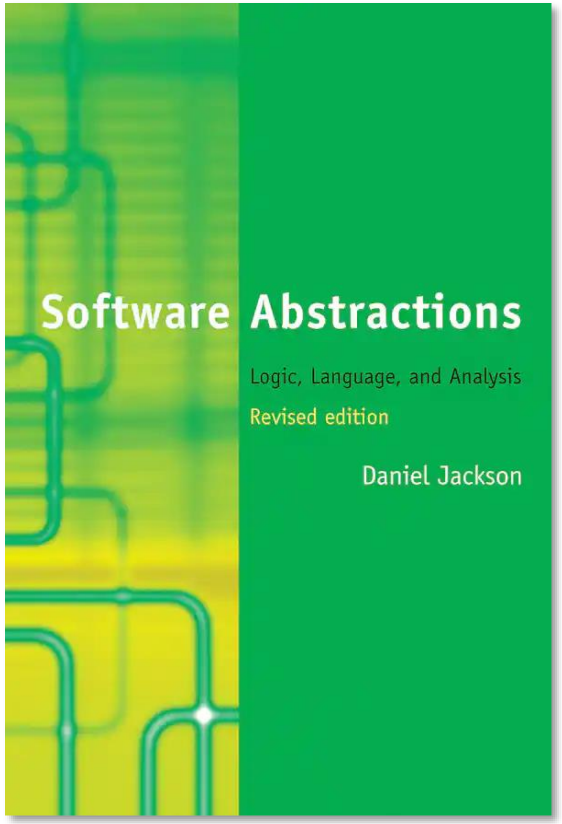
M Camilli, E Di Nitto, M Rossi

Alloy part 1

Introduction

Alloy Resources

- The book is the main resource
  - New, work-in-progress book freely available at: https://haslab.github.io/formal-software-design/
- Other material: https://alloytools.org
  - Tool
  - Documentation
  - Examples
  - Tutorial



- "Some slides are adapted from Greg Dennis and Rob Seater Software Design Group, MIT"

Alloy

- Alloy is a **formal** notation for specifying models of systems and software
  - Looks like a declarative OO language
  - But also has a strong mathematical foundation
- Alloy comes with a supporting tool to
  - **Simulate** specifications and
  - Perform **formal verification** of properties = (bounded) model checking

- Note (for former students)
  - We refer to **version 6 (last version)** of the Alloy language and tool
  - Some small, but important, modifications w.r.t. version 5
  - Some additional concepts, which will be illustrated at the end of this part

Alloy — Declarative Modeling

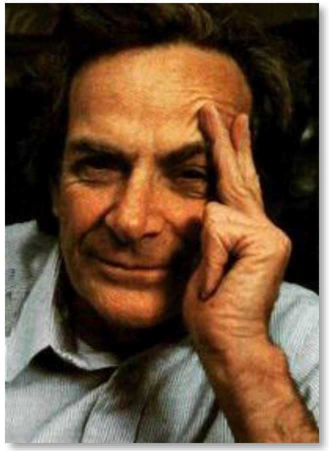
- Alloy is used for abstractions and conceptual modeling in a declarative manner
- Declarative approach to programming and modeling:
  - “declarative programming is a programming paradigm that expresses the logic of a computation without describing its control flow”
  - “describing what the program should accomplish, rather than describing how to go about accomplishing it”

Alloy — Applications for SE

- In RE Alloy can be used to **formally describe**
    - the **domain** and its **properties**, or
    - **operations** that the machine must provide
- In software design to formally model components and their interactions

Alloy — Automated Analysis

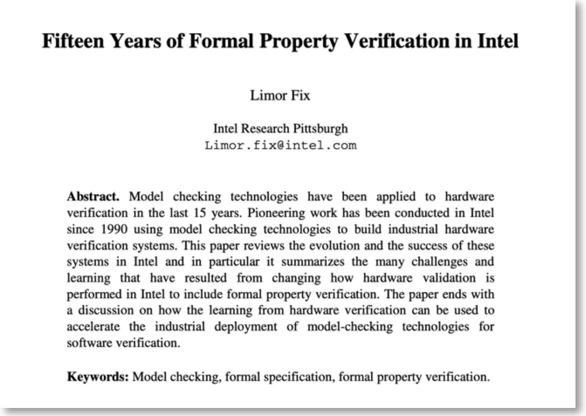
- Any real-life system has a set of properties and constraints
- **Automated formal verification** of specifications can help check whether or not the properties will be satisfied by the system and the constraints will never be violated



"The first principle is that you must not fool yourself, and you are the easiest person to fool."
~ Richard P. Feynman

Formal verification — successful stories

- **Intel** has been using model checking to verify correctness of operations implemented by CPUs



Fifteen Years of Formal Property Verification in Intel

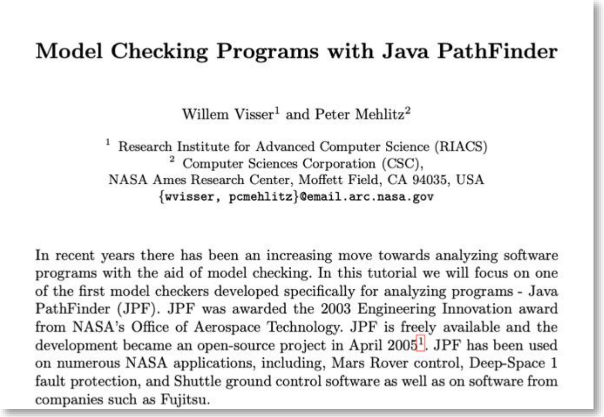
Limor Fix

Intel Research Pittsburgh
Limor.fix@intel.com

Abstract. Model checking technologies have been applied to hardware verification in the last 15 years. Pioneering work has been conducted in Intel since 1990 using model checking technologies to build industrial hardware verification systems. This paper reviews the evolution and the success of these systems in Intel and in particular it summarizes the many challenges and learning that have resulted from changing how hardware validation is performed in Intel to include formal property verification. The paper ends with a discussion on how the learning from hardware verification can be used to accelerate the industrial deployment of model-checking technologies for software verification.

Keywords: Model checking, formal specification, formal property verification.

- **NASA** has been using model checking techniques to verify executable source code used in their space missions



Model Checking Programs with Java PathFinder

Willem Visser[1] and Peter Mehlitz[2]

[1] Research Institute for Advanced Computer Science (RIACS)
[2] Computer Sciences Corporation (CSC),
NASA Ames Research Center, Moffett Field, CA 94035, USA
{wvisser, pcmehlitz}@email.arc.nasa.gov

In recent years there has been an increasing move towards analyzing software programs with the aid of model checking. In this tutorial we will focus on one of the first model checkers developed specifically for analyzing programs - Java PathFinder (JPF). JPF was awarded the 2003 Engineering Innovation award from NASA's Office of Aerospace Technology. JPF is freely available and the development became an open-source project in April 2005[1]. JPF has been used on numerous NASA applications, including, Mars Rover control, Deep-Space 1 fault protection, and Shuttle ground control software as well as on software from companies such as Fujitsu.

What is Alloy?

- First order predicate logic + relational calculus
- Carefully chosen subset of relational algebra
  - uniform model for individuals, sets and relations
  - no higher-order relations
- Almost no arithmetic
- Modules and hierarchies
- **New in Alloy 6: mutable sets and relations**

- Suitable for small, exploratory specifications
- Powerful and fast analysis tool
  - is this specification satisfiable? is this predicate true?

Alloy part 1

The AddressBook example:

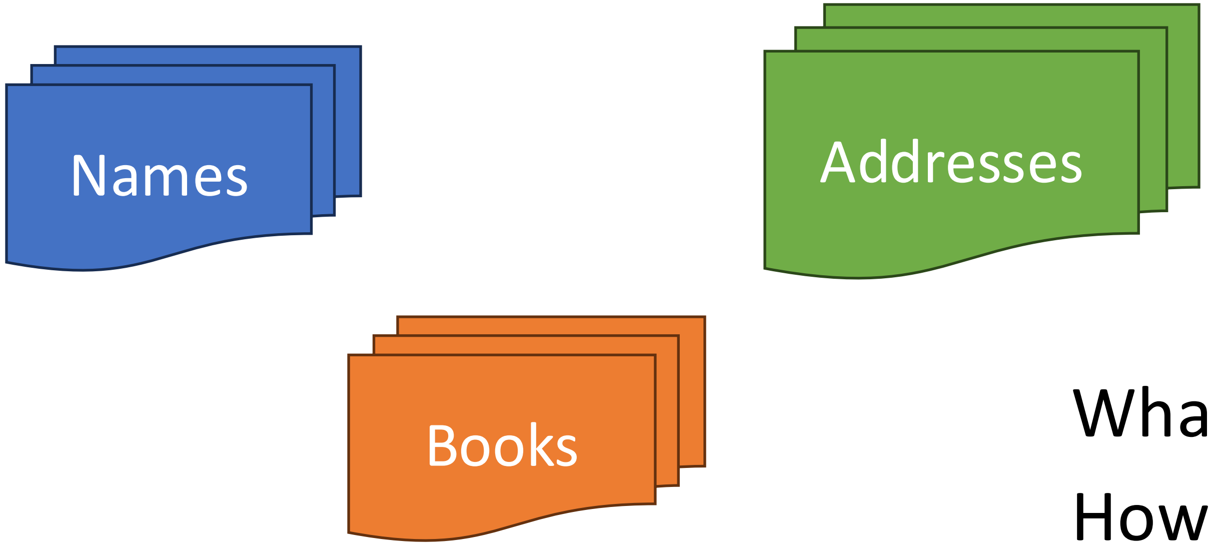
Atoms

Relations

Predicates and Assertions

Let’s Get Started!

- Suppose we are asked to model very simple address books
- Books contain a bunch of addresses linked to the corresponding names



What to model?

How to model?

AddressBook in Alloy

```
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

`set`	*any number*
`one`	*exactly one*
`lone`	*zero or one*
`some`	*one or more*

- `Name` and `Addr` are two types of entities
- `Book` is another type of entity
- `addr` links `Name` to `Addr` within the context of `Book`
    - It is a ternary relation `b -> n -> a`
- keyword **`lone`**: each `Name` can correspond to *at most one* `Addr`

Logic: relations of atoms

- Atoms are Alloy's primitive entities
  - indivisible, immutable, uninterpreted

- Relations associate atoms with one another
  - set of tuples, tuples are sequences of atoms

Relations

- Relation = set of ordered n-tuples of atoms
  - n is called the *arity* of the relation
- Relations in Alloy are typed
  - Determined by the declaration of the relation
  - Example: a relation with type
    `Person -> String`
    only contains pairs
    - whose first component is a `Person`
    - and whose second component is a `String`

Logic: Everything is a Relation

- Sets are unary (1 column) relations

```
Name = {(N0),        Addr = {(A0),        Book = {(B0),
        (N1),                (A1),                (B1)}
        (N2)}                (A2)}
```

- Scalars are singleton sets

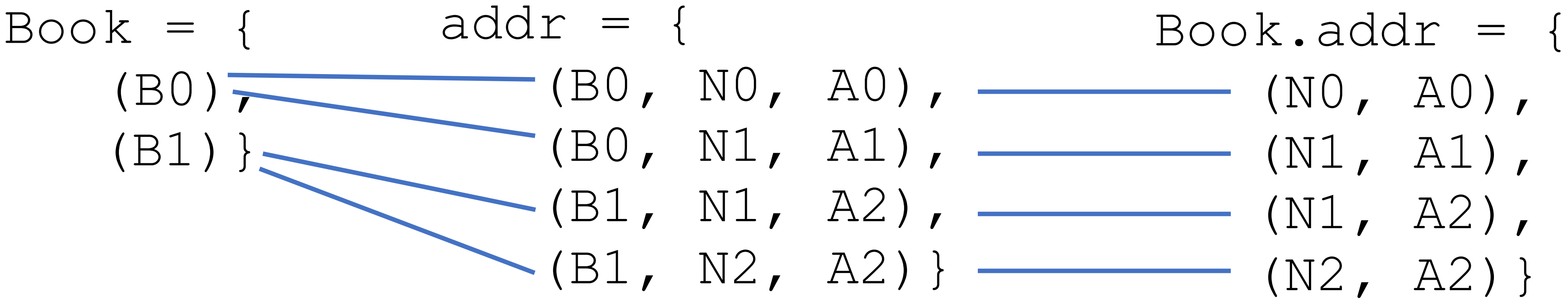
```
myName   = {(N1)}
yourName = {(N2)}
myBook   = {(B0)}
```

- Ternary relation

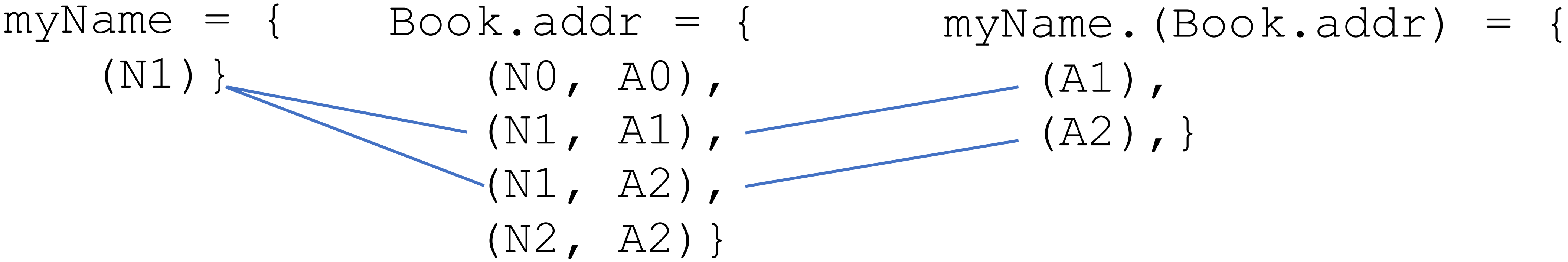
```
addr = {(B0, N0, A0),
        (B0, N1, A1),
        (B1, N1, A2),
        (B1, N2, A2)}
```

what is `myName.(Book.addr)`?

myName.(Book.addr)



```
Book = {        addr = {                 Book.addr = {
  (B0),           (B0, N0, A0),            (N0, A0),
  (B1)}           (B0, N1, A1),            (N1, A1),
                  (B1, N1, A2),            (N1, A2),
                  (B1, N2, A2)}            (N2, A2)}
```



```
myName = {    Book.addr = {      myName.(Book.addr) = {
  (N1)}         (N0, A0),           (A1),
                (N1, A1),           (A2),}
                (N1, A2),
                (N2, A2)}
```

Static Analysis

- Let’s open the Alloy tool and play a bit!
- We add an empty predicate `show` by using keyword **`pred`** to find the instances of the entities involved in the modeling
- In this case, the state exploration is limited to 3 for each entity except Book that is set to 1

```
pred show {}

run show for 3 but 1 Book
```

Static Analysis (cont.)

- Adding a constraint on the number of `(Name,Address)` relations in a given book

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

- Constraining the number of different addresses that appear in the book

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

- Can we fulfil the constraint below?

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

Dynamic Analysis
(old style, but compatible with Alloy 6)

- Let’s model some operations…
  - A predicate that adds an address and name to a book

```
pred add [b, bpost: Book, n: Name, a: Addr] {
    bpost.addr = b.addr + n -> a   }
```

Cartesian product (a pair in this case)

  - b and bpost model two versions of the book, respectively before and after the operation
- Operations can be invoked

```
pred showAdd [b, bpost: Book, n: Name, a: Addr] {
   add[b, bpost, n, a]
   #Name.(bpost.addr) > 1    }

run showAdd
```

Dynamic Analysis (cont.)

- Deleting a name from the address book

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

Assertions and counterexamples

- What happens if we run a `delete` after an `add` predicate?
  Will this take us to the initial state?

```
assert delUndoesAdd {
  all b, bpost, bppost: Book, n: Name, a: Addr |
     add[b, bpost, n, a] and del[bpost, bppost, n]
          implies b.addr = bppost.addr    }

check delUndoesAdd for 3
```

- Counterexample is a scenario in which the assertion is violated
- While checking an assertion, Alloy searches for counterexamples
- Do we find a counterexample in this case?

Resolving the Counterexample

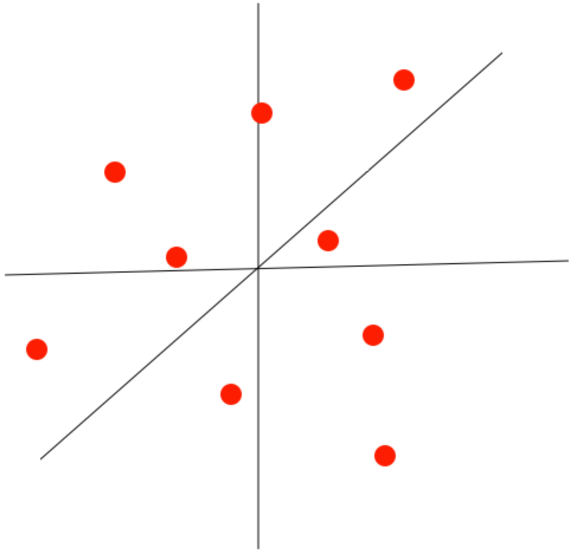
- … Yes! When `add` tries to add a name that already exists and `delete` will delete the name and the corresponding address
- Here is how we can solve the problem:

```
assert delUndoesAdd {
    all b, bpost, bppost: Book, n: Name, a: Addr |
        no n.(b.addr) and
        add[b, bpost, n, a] and del[bpost, bppost, n]
            implies
        b.addr = bppost.addr
}
```

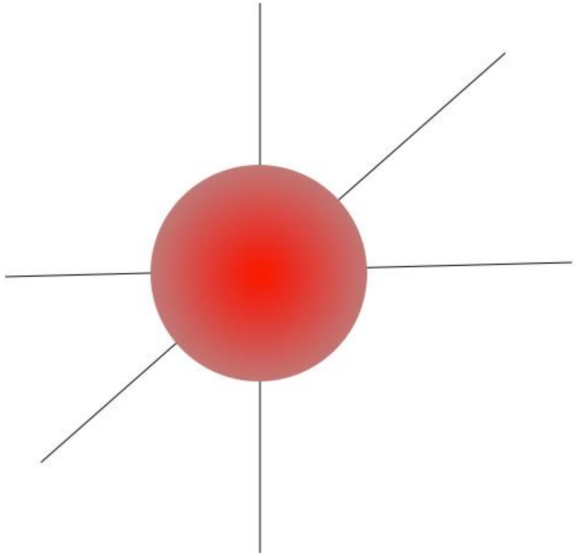
Counterexamples in Alloy

- The analysis by Alloy is always bounded to a defined scope
  - Alloy performs **bounded verification**
- This means that there is NO guarantee that the assertions always hold if no counterexample is found in a certain scope!

Testing vs Bounded Verification



testing:
a few cases of arbitrary size



bounded verification:
all cases within a certain bound

Functions: An Example

- Can we get the addresses in the book corresponding to a certain name?

```
        fun lookup [b: Book, n: Name] : set Addr {
            n.(b.addr)    }
```

- Example of usage of `lookup`

```
    assert addLocal {
        all b, bpost: Book, n1, n2: Name, a: Addr |
            add[b, bpost, n1, a] and n1 != n2
              implies
            lookup[b, n2] = lookup[bpost, n2]    }

    check addLocal for 3 but 2 Book
```

Summary of Alloy Characteristics (1)

- An Alloy document is a “source code” unit
- It may contain:
  - **Signatures**: define types and relationships
  - **Facts**: properties of models (constraints!)
  - **Predicates/functions**: reusable expressions
  - **Assertions**: properties we want to check
  - **Commands**: instruct the Alloy Analyzer which assertions to check, and how
    - A predicate is **run** to find a world that satisfies it
    - An assertion is **checked** to find a counterexample

Summary of Alloy Characteristics (2)

- Signatures, predicates, facts and functions tell how correct worlds (models) are made
  - When the Alloy analyzer tries to build a model, it must comply with them
- Assertions and commands tell which kind of checks must be performed over these worlds
  - E.g., “find, among all the models, one that violates this assertion”
- Of course, the Analyzer cannot check all the (usually infinite) models of a specification
  - You must also tell the Analyzer how to limit the search

Alloy part 1

The family tree example:

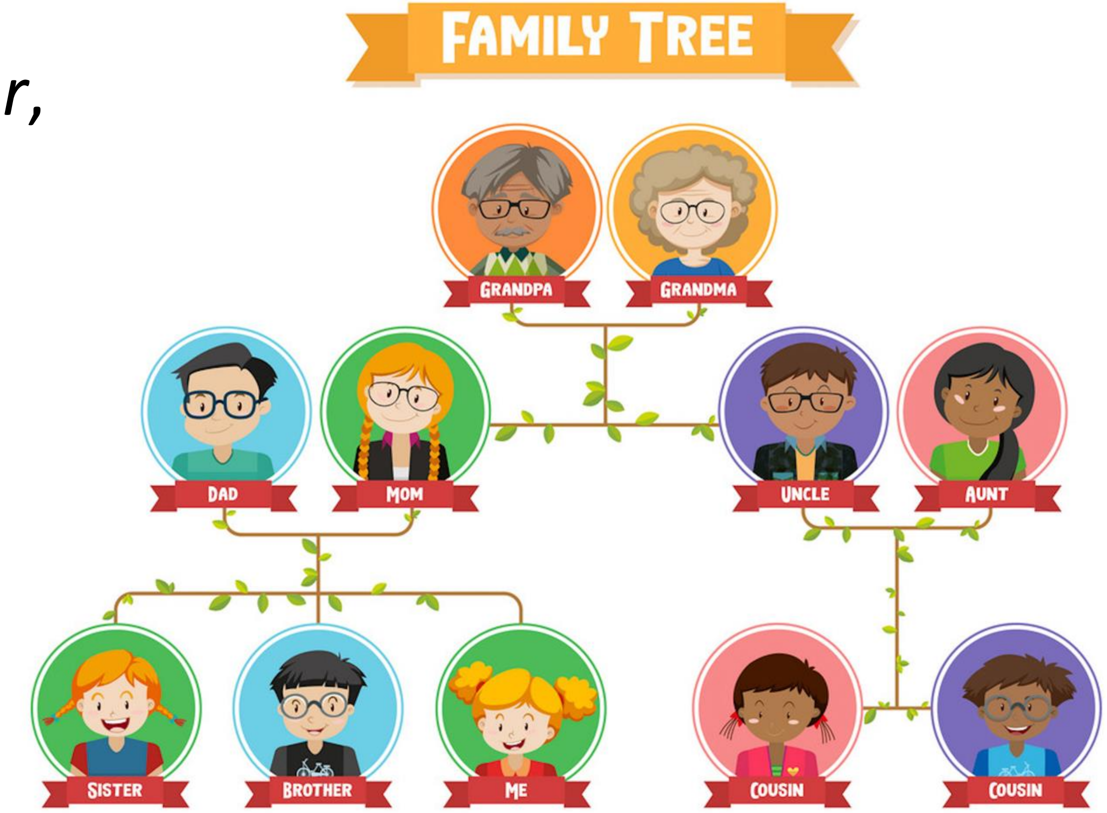
Hierarchies

Facts vs predicates

Other operators (transitive closure, transpose)

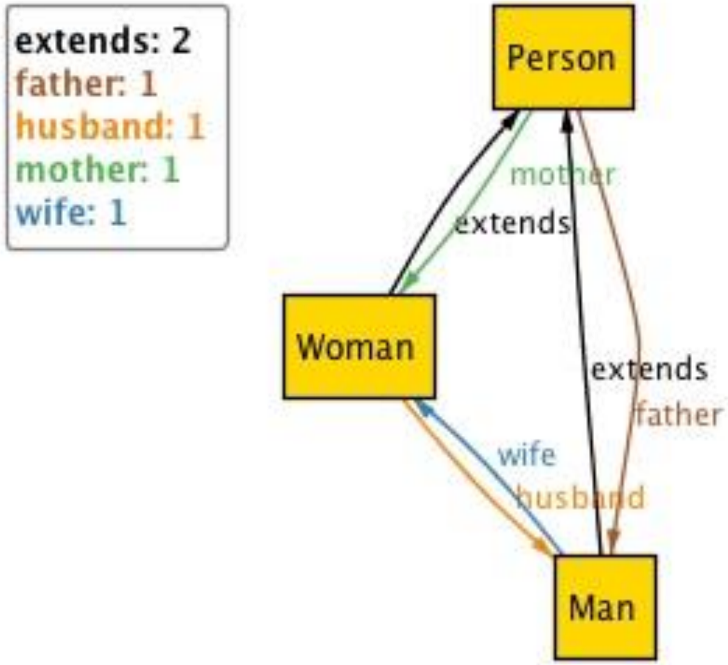
Family Relationships

- There are notions like *Person*
- There are relationships like *father*, *mother*, *wife*, …



Initial Alloy Model

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

sig Man extends Person {
        wife: lone Woman
}

sig Woman extends Person {
        husband: lone Man
}
```



Meta-model

Signatures and Fields

- Signatures (declared through keyword `sig`) represent sets of atoms
- The **fields** of a signature are relations whose domain is a subset of the signature
- The `extends` keyword is used to declare a subset of a signature
- An `abstract` signature has no elements except those belonging to the signatures that extend it
- `m sig A {}`: `m` (e.g., `one`, `some`, etc.) declares the multiplicity, the number of elements of `A`

Relations

- Relations are declared as fields of signatures:
  - `sig A {f : e}`
    - `A` is the domain
    - `e` is the range
- You can specify the multiplicity when defining a relation
  - `sig A {f : m e}`
    - The default multiplicity is **`one`**

Can you be your own Grandpa?

- Biologically not, but terminologically… let’s check this out:
  - I’m my own Grandpa!

    http://www.youtube.com/watch?v=eYlJH81dSiw
    (with family tree)

    https://www.youtube.com/watch?v=gkiOm-vmpcY&list=RDgkiOm-vmpcY
    (with muppets)

Adding Predicates and Functions

- The function `grandpas` returns the grandfathers of a given person `p`

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

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

This can be written also as
`p in grandpas[p]`

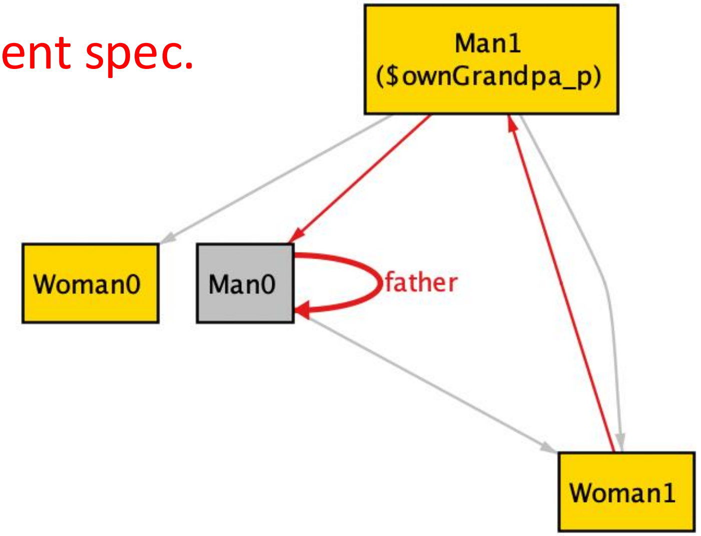
Running the Model

- The model is executed by trying to see if a predicate is valid for a set of instances of the meta-model

```
run ownGrandpa for 4 Person
```

Fixing some problems in the spec (1)

Valid model according to the current spec.



- **Issue**: Man0 is father of himself
- More in general “No one can be his/her own ancestor”

Fixing some problems in the spec (1)

- “No one can be his/her own ancestor”

```
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) …
}
```

… we need a **transitive closure** operator

Transitive closure

^ *transitive closure*

* *reflexive transitive closure*

apply only to binary relations

union

```
^r = r + r.r + r.r.r + …
*r = iden + ^r
```

the identity relation

a list

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

^next = {(N0, N1), (N0, N2), (N0, N3),
         (N1, N2), (N1, N3),
         (N2, N3)}
*next = {(N0, N0), (N0, N1), (N0, N2), (N0, N3),
         (N1, N1), (N1, N2), (N1, N3),
         (N2, N2), (N2, N3), (N3, N3)}
```

```
sig Node{
  next : lone Node
}
```

```
first = {(N0)}
rest = {(N1), (N2), (N3)}

first.^next = rest
first.*next = Node
```

Fixing some problems in the spec (2)

- “No one can be his/her own ancestor”

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

Fixing some problems in the spec (3)

- “If X is husband of Y, then Y is wife of X”

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

- We can express an equivalent constraint through the **transpose operator** (~)

Transpose operator (~)

- Consider the following example

```
Node = {(N0), (N1), (N2), (N3)}
next = {(N0, N1), (N1, N2), (N2, N3)}

~next = {(N1, N0), (N2, N1), (N3, N2)}
```

  - The transpose operator applies only to binary relations

- "If X is husband of Y, then Y is wife of X"

```
fact {
      wife = ~husband
}
```

More on facts

- A fact can contain multiple constraints

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

- The two constraints in the fact are to be considered in **and**

Are facts different from predicates?

- They are both used to express constraints

- Facts must hold **globally** for any atom and relation in the model
- Predicates have to be invoked

Adding Assertions

- Check if there is no man who is also his father
- Assertions are used to find counterexamples which are the instances of a model

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

check NoSelfFather
```

Adding more facts to the model

- **Social convention**: it is not possible that someone has a wife or husband who is also one of his/her ancestors

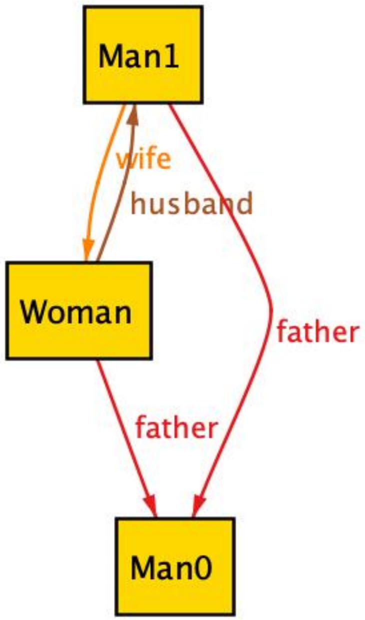
facts can be named for readability purposes

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

intersection

Adding more facts to the model

- **Other issues**: Still, wife and husband can have common ancestors!



How to avoid that wives and husbands have common ancestors?

```
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) )
}
```

- Does this solve the common ancestor problem?
- To which fact is this equivalent?
    - Answer: SocialConvention

Another solution

```
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
}
```

Cartesian product
(a pair in this case)

empty set

- Check yourself if it works

Other solution (?)

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

- `noCommonAncestors` is *stronger* than `SocialConvention3`

"Proof" that noCommonAncestor is stronger than SocialConvention3

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

Notice the declaration

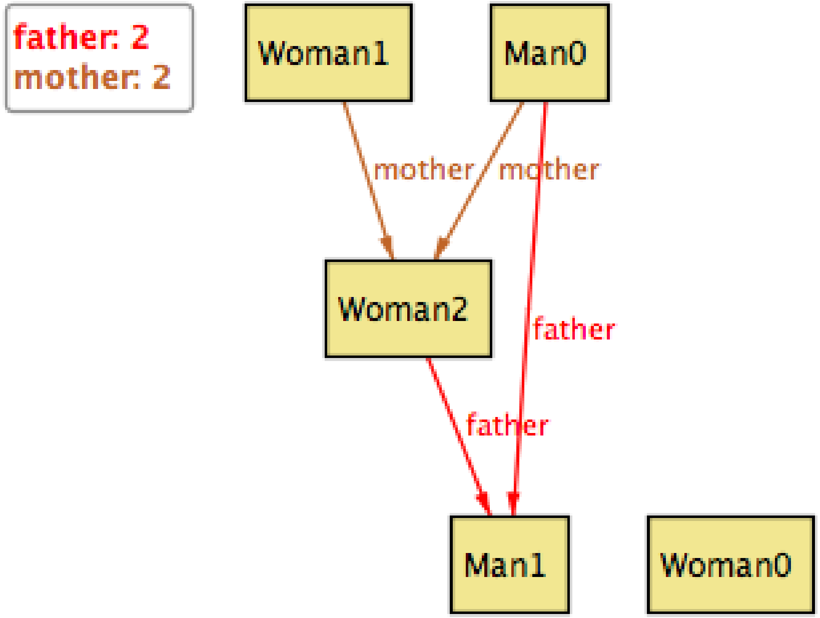
```
pred SocialConvention3 {
  no ((wife+husband) & (mother+father).~(mother+father))
}
```

"Proof" (cont.)

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


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

Still, we can have this case

Man1 is father and, at the same time, grandpa of Man0.



Does this address the problem?

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

POLITECNICO
MILANO 1863

Software Engineering 2

Alloy Part 2:

New features: mutable relations and sets

M Camilli, E Di Nitto, M Rossi

From unmutable relations…

- Let's go back to the address book example:

```
sig Name, Addr {}
sig Book {
    addr: Name -> lone Addr
}
```

- Describing the evolution of the address book following additions and deletions required to explicitly represent the address book before and after the operation, because it was unmutable:

```
pred add [b, bpost: Book, n: Name, a: Addr] {
    bpost.addr = b.addr + n -> a  }
```

… to mutable ones

- What if we can say that relation `addr` changes over time?

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

new keyword, it means the relation is **mutable**

- Now, we can have that an address book initially (first instant) is empty, then, *after* that (i.e., in the second instant) it has 1 address, then, *after* that (i.e., in the third instant) it has 2 addresses

Evolution of an address book

- We show an address book whose size increases in the first 3 instants:

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

It means "at the next instant"

- It can be also written, more succintly, with the "sequence" operator (;)

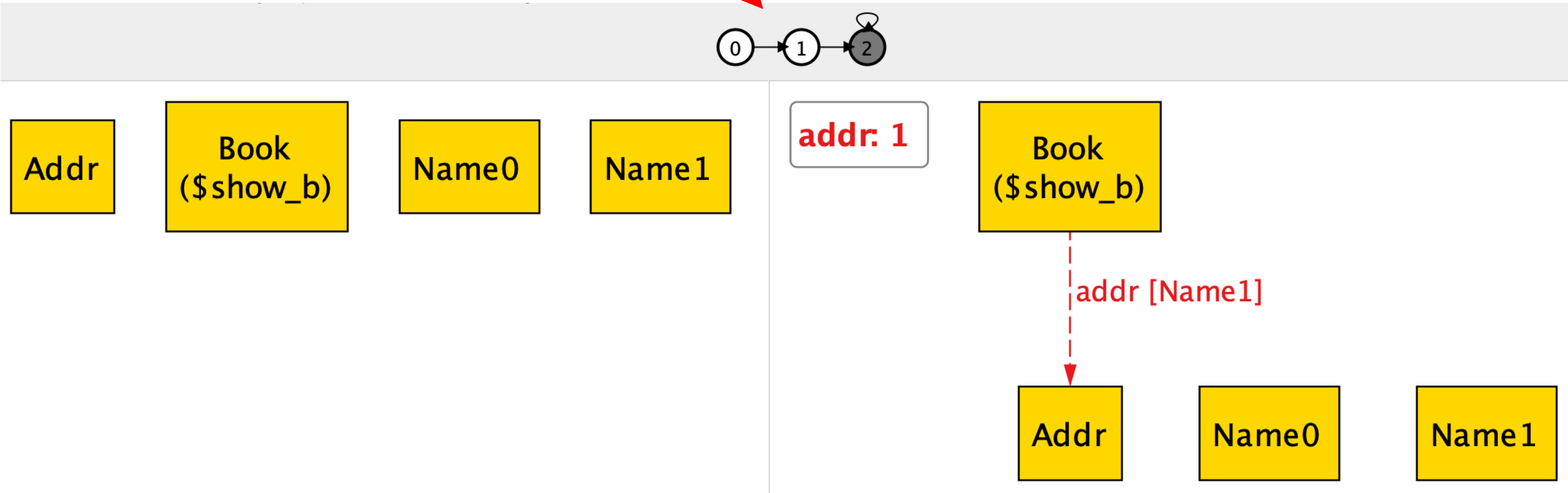
```
pred show [b: Book] {
   #b.addr = 0 ; #b.addr = 1 ; #b.addr = 2 )
}
```

Let us run the predicate

```
run show for 3 but 1 Book, 3 steps
```

We say how many instants we want to represent

These are the first 2 instants (the white ones)



New add and delete

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

It means "the next value of `b.addr`"

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

New assertion

```
assert delUndoesAdd {
    all b: Book, n: Name, a: Addr |
        no n.(b.addr) and add[b, n, a] and after del[b, n]
            implies
        b.addr = b.addr''
}
```

This means "`b.addr` after 2 instants"

```
check delUndoesAdd for 5 steps
```

- This assertion means "if in the first instant `n` does not belong to `b`, and `n` is added to `b`, and then, at the next instant (i.e., the second) `n` is removed from `b`, then address book `b` after 2 instants is the same as the initial one"

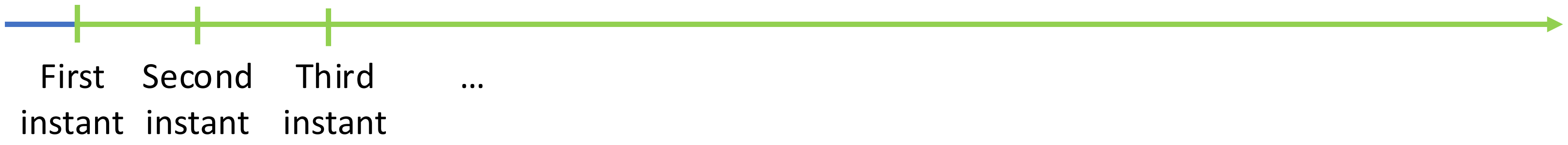
"Always"

- What if we want to say that "at all time instants, if `n` does not belong to `b` and `n` is added…"?
- We can say "always, if …":

```
assert alwDelUndoesAdd {
  all b: Book, n: Name, a: Addr |
    always ( no n.(b.addr) and (add[b, n, a] ; del[b, n])
             implies
             b.addr = b.addr'' )
}
```

new operator

always F = true in i **iff** F = true **for each** time instant k >= i



Example to illustrate temporal operators

(h/t L. Padalino, F. P. Panaccione, F. Santambrogio: github.com/lucapada/ResearchProjectAlloy6)

- Consider the following signature

```
sig Device {
    var status: DevStatus
}

enum DevStatus { Working, Broken }
```

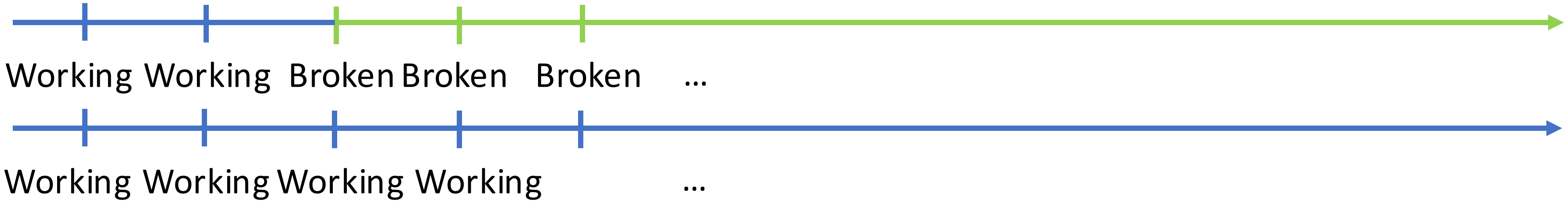
Useful abbreviation to avoid declaring the abstract signatures

Some temporal constraints

- Once broken, a device can never be repaired

```
fact NoRepair {
    all d: Device |
      always ( d.status = Broken
               implies
               after always d.status = Broken )
}
```

Examples



Working Working Broken Broken Broken …

Working Working Working Working …

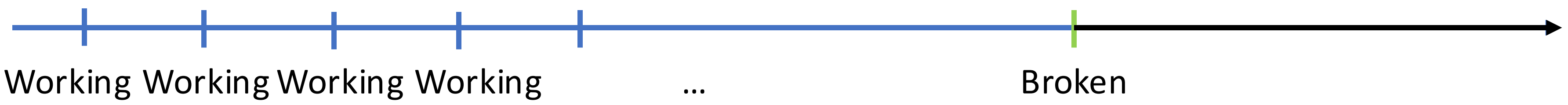
eventually operator

- If a device is working, it will eventually break

```
fact NotAlwaysFunctioning {
    all d: Device |
      always ( d.status = Working
               implies
               eventually d.status = Broken )
}
```

eventually F = true in i **iff** F = true for **some** time instant k>=i

Example



Working Working Working Working ... Broken

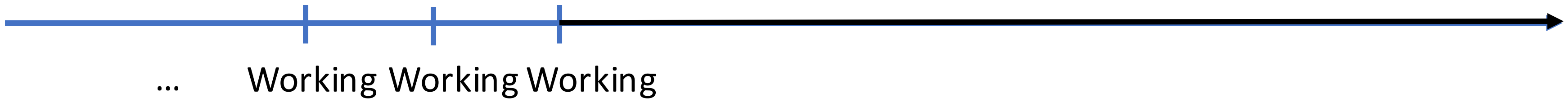
"Past" operators: historically

- If a device is working, it has been working since the beginning

```
fact NowWorkingPreviouslyWorking {
    all d: Device |
      always ( d.status = Working
               implies
               historically d.status = Working )
}
```

historically F = true in i **iff** F = true **for each** time instant k<=i

Example



"Past" operators: **before**

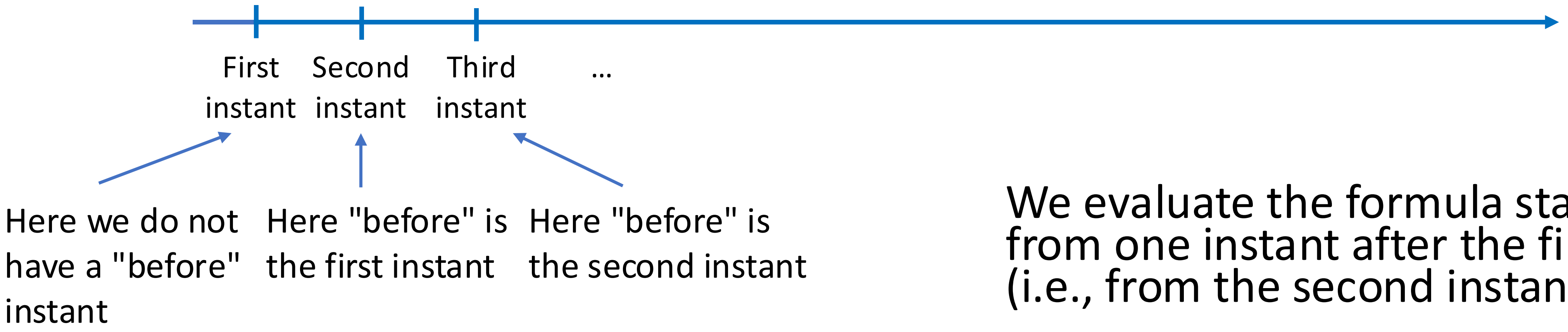
- This is equivalent to `NowWorkingPreviouslyWorking`

```
fact NowWorkingPreviouslyWorking2 {
    all d: Device |
      after always ( d.status = Working
                     implies
                     before d.status = Working )
}
```

It means "the previous instant"

Evaluating **before** in the first instant

- The first instant does not have a previous instant, so by definition **before** F in the first instant is false, no matter the formula F



First instant

Second instant

Third instant

…

Here we do not have a "before" instant

Here "before" is the first instant

Here "before" is the second instant

We evaluate the formula starting from one instant after the first one (i.e., from the second instant)

```
fact NowWorkingPreviouslyWorking2 {
    all d: Device |
      after always ( d.status = Working implies before d.status = Working )
}
```

"Past" operators: once

- Let us introduce a predicate

```
pred break[ d: Device ] {
    d.status = Working
    d.status' = Broken
}
```

- If a device is broken, it must have broken sometime in the past

```
fact IfBrokenDidBreak {
  all d: Device |
    always ( d.status = Broken  implies  once break[d] )
}
```

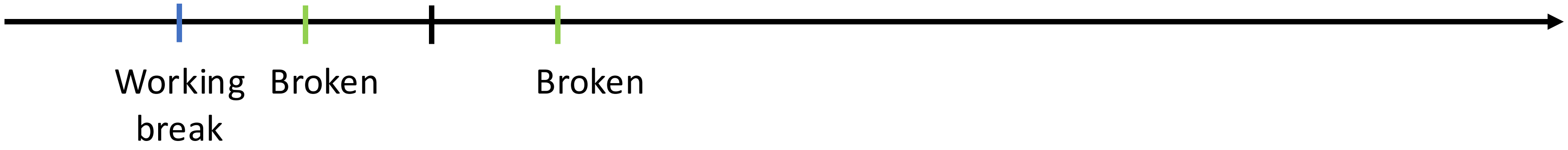
"Past" operators: once

- If a device is broken, it must have broken sometime in the past

```
fact IfBrokenDidBreak {
  all d: Device |
    always ( d.status = Broken  implies  once break[d] )
}
```

once F = true in i **iff** F = true **for some** time instant k<=i

Example



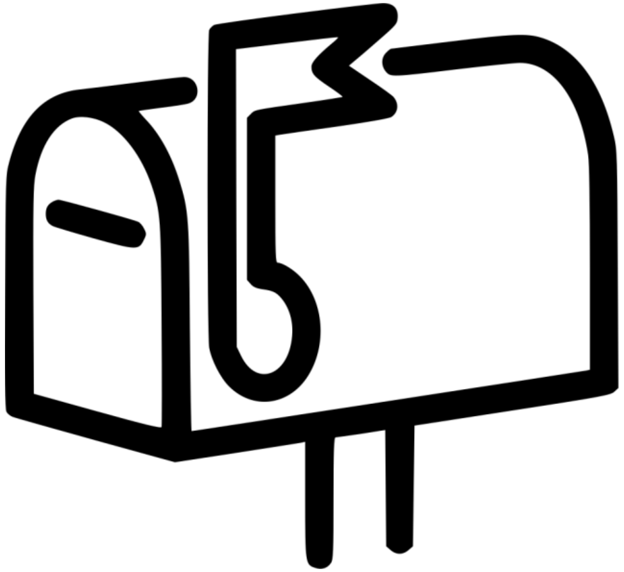
Temporal operators*: summary

FUTURE	PAST
always	historically
eventually	once
after	before

* For details and other operators, you can watch this video https://www.youtube.com/watch?v=GncsAmy29l0

Example: message deletion from mailbox

- 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



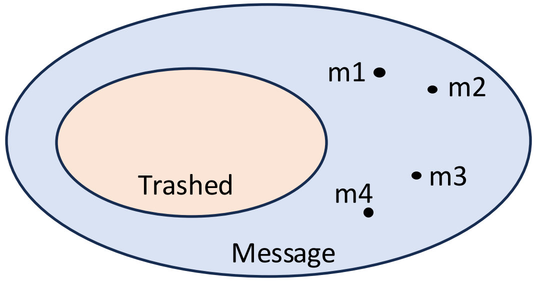
Signatures

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

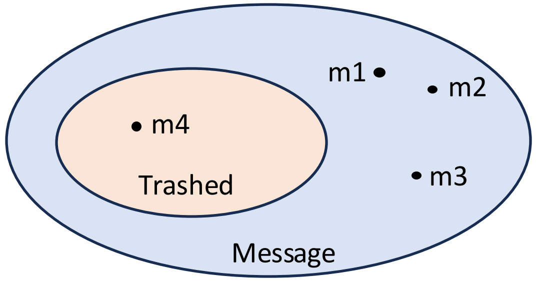
```
var sig Message {}

var sig Trashed in Message {}
```

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

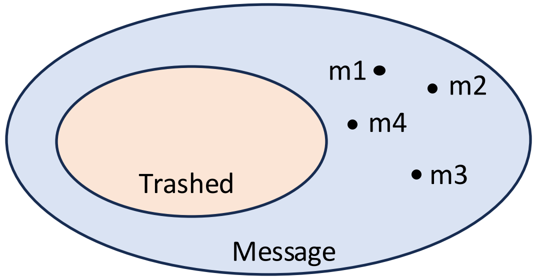
Some messages are in the `Trashed` set.
Signatures are mutable, which means that a message can be "regular", then be trashed, then be regular again, etc.



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             }
```