DD2460 Software safety and security. Lecture 4

NONDETERMINISM; FUNCTIONS AND RELATIONS IN EVENT-B MODELLING

Recall the example from the last lecture

Let us analyse the guards of two events: EXIT and REGISTER

After initialisation, all sets *registered*, *in* and *out* are empty

Hence, after initialisation only one event – REGISTER is enabled, i.e., the guards of all other events evaluate to false

However, once the sets become non-empty several (sometimes all events) become enabled

Any of the enabled events can be chosen for execution non-deterministically, i.e., we do not have control over which one will be executed next.

Each event results in a state change, i.e., models a state transition

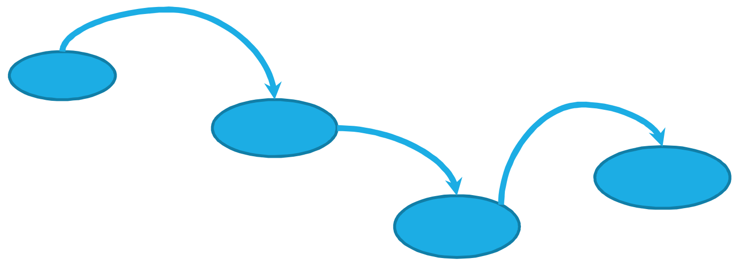
Often the choice of the next state is non-deterministic, i.e., depends on which event is chosen or which assignment is made

```
EXIT ≙                  // a student leaves the building
    any  st
    where
        grd1: st ∈ registered   // a student must be registered
        grd2: st ∈ in            // a student must be inside
    then
        act1: in := in \ {st}       // remove st from in
        act2: out := out ∪ {st} // remove st from in

    end
REGISTER ≙          // registration a new student
    any  st
    where
        grd1: st ∈ STUDENTS   // a new student
        grd2: st ∉ registered   // … that is not in the set registered yet
    then
        act1: registered := registered ∪ {st}       // add st to registered
        act2: out := out  ∪ {st}                    // add st to out
    end
```

Machine behaviour and nondeterminism

- The behaviour of an Event-B machine is defined as a **transition system** that moves from one state to another through execution of events.



- The states of a machine are represented by the different configurations of values for the variables:
  - In the student registration example, the state is defined by the variables *registered, in, out*

Machine behaviour and nondeterminism

- In any state that a machine can reach, an enabled event is chosen to be executed to define the next transition.
- If several events are enabled in a state, then the choice of which event occurs is nondeterministic.
- Also, if an event is enabled for several different parameter values, the choice of value for the parameters is nondeterministic – the choice just needs to satisfy the event guards.
  - For example, in the **REGISTER** event, the choice of value for parameter ***st*** is nondeterministic, with the choice of value being constrained by the guards of the event to ensure that it is a fresh value.
- Treating the choice of event and parameter values as nondeterministic is an abstraction of different ways in which the choice might be made in an implementation of the model.

Note on modelling order of events

To reduce non-determinism and enforce that one event is executed after another you need to mimic "a program counter": example of a simple request-response phase change (not terminating):

MACHINE SIMPLE_REQ_RESP

VARIABLES *phase*

INVARIANTS

inv1: phase ∈ PHASES

EVENTS

INITIALISATION ≙

then

act1: *phase :=REQ*

end

REQUESTING ≙

when

grd1: phase=REQ

then

- act1: phase:=RESP

end

RESPONDING ≙

when

grd1: phase=RESP

then

- act1: phase:=REQ

end

PHASES=*{REQ,RESP}*

Event order and invariant

MACHINE SIMPLE_REQ_RESP
VARIABLES *phase, letter*
INVARIANTS
inv1: phase ∈ PHASES
inv2: letter ∈ LETTERS
inv3: how letter and phase are related?
EVENTS
INITIALISATION ≙
then
act1: *phase :=REQ*
act2: *letter* $:= A$
end

REQUESTING ≙
when
grd1: ***phase=REQ***
then
- **act1:** ***letter:=B***
- **act2:** ***phase:=RESP***

end

RESPONDING ≙
when
grd1: ***phase=RESP***
then
- **act1:** ***letter:=A***
- **act2:** ***phase:=REQ***

end

PHASES=*{REQ,RESP}*
LETTERS= *{A,B}*

Event order and invariant

MACHINE SIMPLE_REQ_RESP
VARIABLES *phase, letter*
INVARIANTS
inv1: *phase* $\in$ *PHASES*
inv2: *letter* $\in$ *LETTERS*
inv3: *phase=REQ* $\Rightarrow$ *letter=A*
inv4: *phase=RESP* $\Rightarrow$ *letter=B*
VENTS
INITIALISATION ≙
then
act1: *phase :=REQ*
act2: *letter* $:= A$

end

REQUESTING ≙
when
grd1: ***phase=REQ***
then
- **act1:** ***letter:=B***
- **act2:** ***phase:=RESP***

end

RESPONDING ≙
when
grd1: ***phase=RESP***
then
- **act1:** ***letter:=A***
- **act2:** ***phase:=REQ***

end

Relations between sets

- Relation between sets is an important mathematical structure which is commonly used in expressing specifications.
- Relations allow us to express complicated interconnections and relationships between entities *formally.*

Ordered pairs

- An **ordered pair** is an element consisting of two parts:

  a ***first*** part and ***second*** part

- An ordered pair with first part $\boldsymbol{x}$ and second part $\boldsymbol{y}$ is written as:

$$\boldsymbol{x} \mapsto \boldsymbol{y}$$

- Examples:
  - $(\boldsymbol{apple} \mapsto \boldsymbol{red})$
  - $(\boldsymbol{Databases} \mapsto \boldsymbol{fall})$
  - $(\boldsymbol{115A} \mapsto \boldsymbol{30})$
  - $(\boldsymbol{Smith} \mapsto \boldsymbol{0123})$

Cartesian product

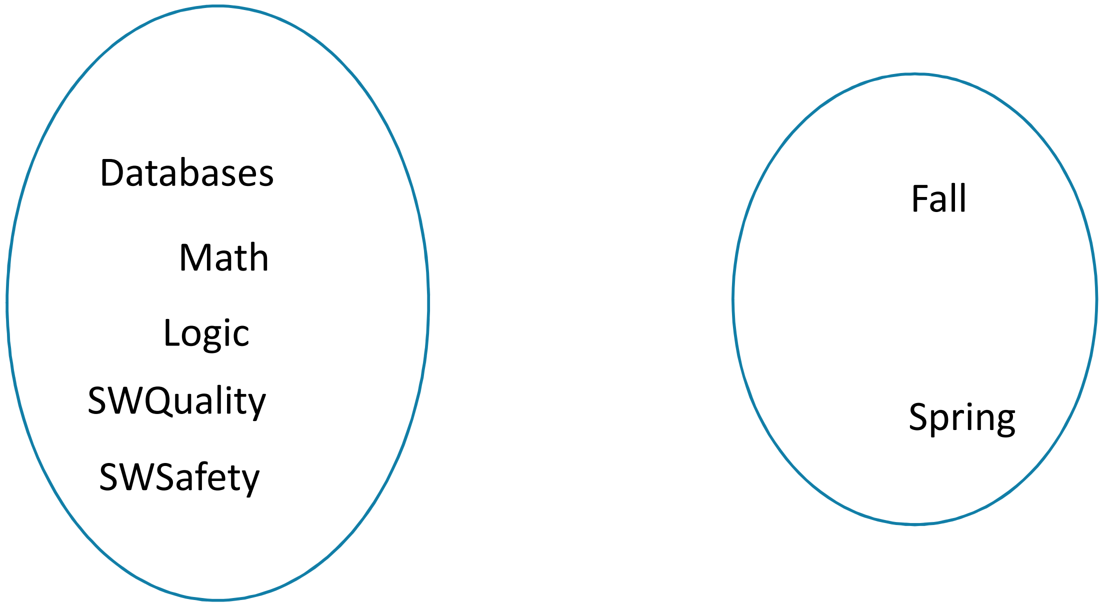
- The **Cartesian product** of two sets is

  the **set** of pairs whose first part is in $\boldsymbol{S}$ and second part is in $\boldsymbol{T}$

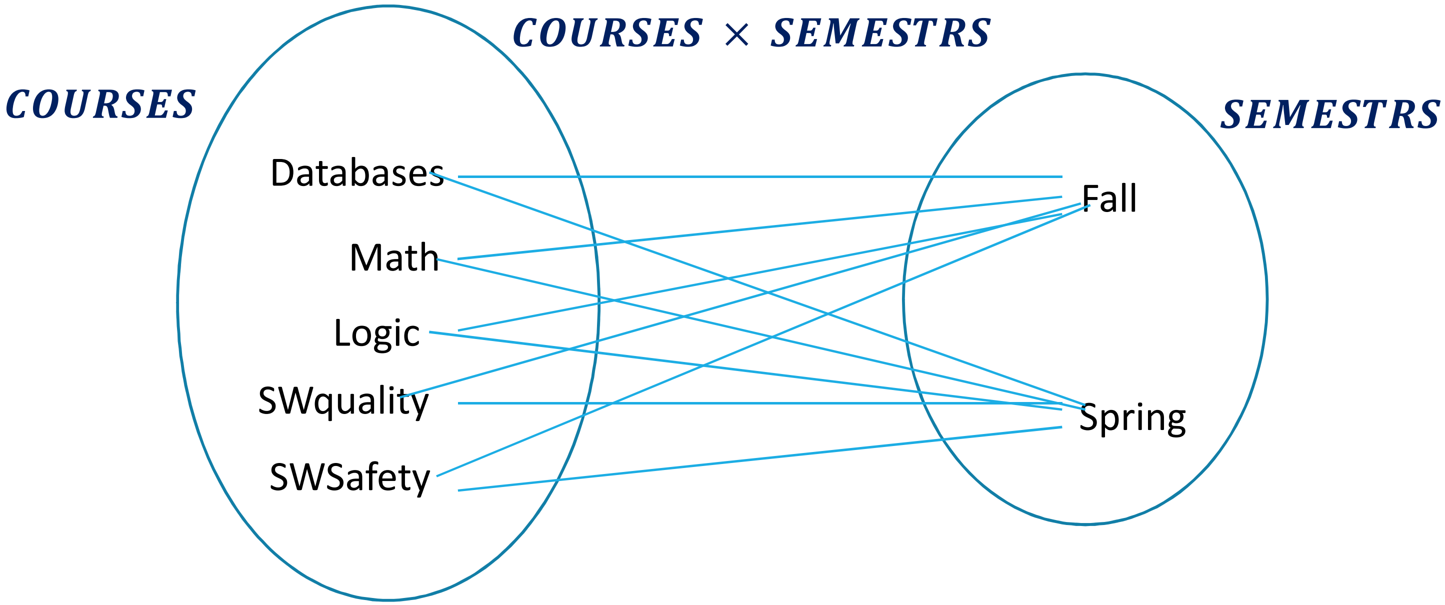
- The Cartesian product of $\boldsymbol{S}$ with $\boldsymbol{T}$ is written: $\boldsymbol{S} \times \boldsymbol{T}$

Cartesian product: example

Lets consider two sets: ***COURSES*** and ***SEMESTERS***



Cartesian product: example



Cartesian product: definition and more examples

- Defining Cartesian product:

Predicate	Definition
$\boldsymbol{x \mapsto y \in S \times T}$	$\boldsymbol{x \in S \ \wedge \ y \in T}$

- Examples:
  - $\mathbb{N} \times \mathbb{N}$ pairs of natural numbers
  - $\{1,2,3\} \times \{a, b\} = \{1 \mapsto a, 1 \mapsto b, 2 \mapsto a, 2 \mapsto b, 3 \mapsto a, 3 \mapsto b\}$
  - $\{Anna, Bill, Jack\} \times \emptyset = \emptyset$
  - $\{\{1\}, \{1,2\}\} \times \{a, b\} = \{\{1\} \mapsto a, \{1\} \mapsto b, \{1,2\} \mapsto a, \{1,2\} \mapsto b\}$
  - **card**$(\{yes, no\} \times \{a, b\}) =$ **card**$(\{yes \mapsto a, yes \mapsto b, no \mapsto a, no \mapsto b\}) = 4$

Cartesian product is a type constructor

- $\boldsymbol{S} \times \boldsymbol{T}$ is a new type constructed from types $\boldsymbol{S}$ and $\boldsymbol{T}$.
- Cartesian product is the type constructor for ordered pairs.
- Given $\boldsymbol{x} \in \boldsymbol{S}$ and $\boldsymbol{y} \in \boldsymbol{T}$ we have $\boldsymbol{x} \mapsto \boldsymbol{y} \in \boldsymbol{S} \times \boldsymbol{T}$

- Examples:
  - $4 \mapsto 7 \in \mathbb{Z} \times \mathbb{Z}$
  - $\{2, 3\} \mapsto 4 \in \mathbb{P}(\mathbb{Z}) \times \mathbb{Z}$
  - $\{2 \mapsto 1, 3 \mapsto 3, 4 \mapsto 5\} \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$

Sets of ordered pairs

A simple database can be modelled as a set of ordered pairs:

$$\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\}$$

$\boldsymbol{studentCourses}$ has type

$$\boldsymbol{studentCourses} \in \mathbb{P}(NAMES \times COURSES)$$

Relations

- A **relation** R between sets $\boldsymbol{S}$ and $\boldsymbol{T}$ expresses a relationship between elements in $\boldsymbol{S}$ and elements in $\boldsymbol{T}$:
  - A relation is captured simply as a set of ordered pairs $(\boldsymbol{s} \mapsto \boldsymbol{t})$ with $\boldsymbol{s} \in \boldsymbol{S}$ and $\boldsymbol{t} \in \boldsymbol{T}$.
- A relation is a common modelling structure so Event-B has a special notation for it:

$$\boldsymbol{S} \leftrightarrow \boldsymbol{T} = \mathbb{P}\,(\boldsymbol{S} \times \boldsymbol{T})$$

- We can write then

$$\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\}$$

as

$$\boldsymbol{studentCourses} \in NAMES \leftrightarrow COURSES$$

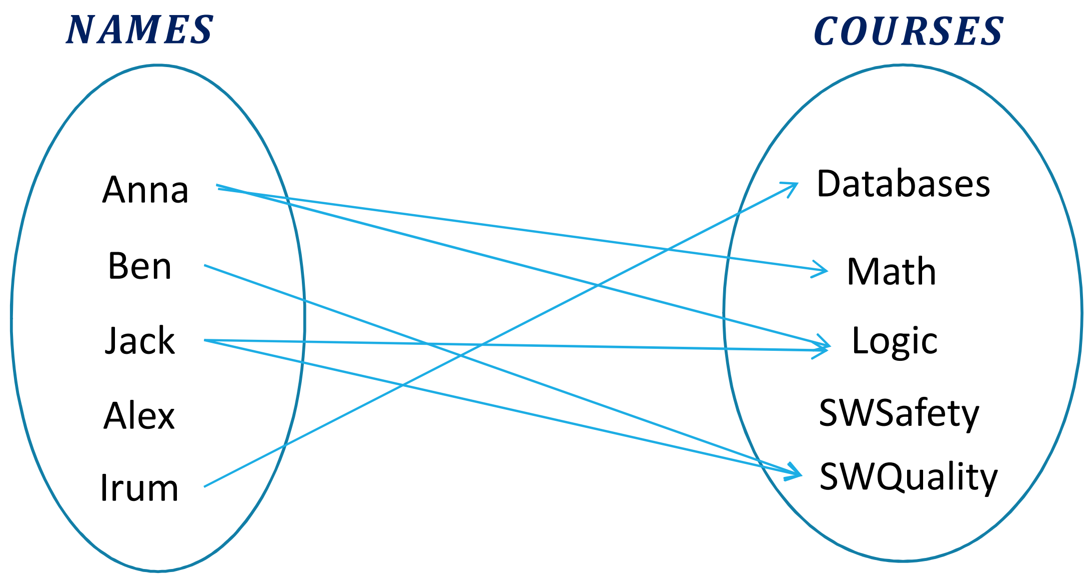
- Do not confuse the arrow symbols:

$\leftrightarrow$ combines two sets to form a set;

$\mapsto$ combines two elements to form an ordered pair.

Domain and range

$$\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\}$$



$\boldsymbol{NAMES} = \{Anna, Ben, Jack, Alex, Irum\}$ $\boldsymbol{COURSES} = \{Databases, Math, Logic, SWSafety, SWQuality\}$

Domain

- The **domain** of a relation $\boldsymbol{R}$ is the **set** of first parts of all the pairs in $\boldsymbol{R}$, written $\boldsymbol{dom}(\boldsymbol{R})$

Predicate	Definition
$\boldsymbol{x} \in \boldsymbol{dom}(\boldsymbol{R})$	$\exists\, \boldsymbol{y}.\ \boldsymbol{x} \mapsto \boldsymbol{y} \in \boldsymbol{R}$

$$\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\},$$

$$\boldsymbol{NAMES} = \{Anna, Ben, Jack, Alex, Irum\}$$

then

$$\boldsymbol{dom}(\boldsymbol{studentCourses}) = \{Anna, Ben, Jack, Irum\}$$

Range

- The **range** of a relation $\boldsymbol{R}$ is the **set** of <u>second</u> parts of all the pairs in $\boldsymbol{R}$, written $\boldsymbol{ran}(R)$

Predicate	Definition
$\boldsymbol{y} \in \boldsymbol{ran}(\boldsymbol{R})$	$\exists\, \boldsymbol{x} \,.\, \boldsymbol{x} \mapsto \boldsymbol{y} \in \boldsymbol{R}$

$$\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\}$$

$$\boldsymbol{ran}(\boldsymbol{studentCourses}) = \{Logic, SWQuality, Databases, Math\}$$

Relational image definition

- Assume $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{S}$
- The **relational image** of set $\boldsymbol{A}$ under relation $\boldsymbol{R}$ is written $\boldsymbol{R}[\boldsymbol{A}]$

Predicate	Definition
$\boldsymbol{y} \in \boldsymbol{R}[\boldsymbol{A}]$	$\exists\, \boldsymbol{x}.\ \boldsymbol{x} \in \boldsymbol{A} \land \boldsymbol{x} \mapsto \boldsymbol{y} \in \boldsymbol{R}$

- Set of all elements in ***ran(R)*** that has elements of ***set A*** as the first elements of their pairs

Relational image examples

- $\boldsymbol{studentCourses} = \{Anna \mapsto Logic, Ben \mapsto SWQuality, Jack \mapsto SWQuality, Irum \mapsto Databases, Anna \mapsto Math, Jack \mapsto Logic\}$

  $\boldsymbol{studentCourses}[\{Anna, Ben\}] = \{Logic, SWQuality, Math\}$

- $\boldsymbol{courseLecturer} = \{Brown \mapsto Math, Jacson \mapsto Informatics, Brown \mapsto Statistics, Jons \mapsto Databases\}$

  $\boldsymbol{courseLecturer}[\{Brown\}] = \{Math, Statistics\}$

Partial functions

- Special kind of relation: each domain element has <u>at most one</u> range element associated with it.
- To declare $\boldsymbol{f}$ as a partial function:

$$\boldsymbol{f} \in \boldsymbol{X} \mathrel{+\!\!\!\rightarrow} \boldsymbol{Y}$$

- It is said to be partial because there may be values in the set $\boldsymbol{X}$ that are not in the domain of $\boldsymbol{f}$
- Each domain element is mapped to <u>**one**</u> range element:

$$\boldsymbol{x} \in \boldsymbol{dom}(\boldsymbol{f}) \quad \Longrightarrow \quad \boldsymbol{card}(\boldsymbol{f}[\{\boldsymbol{x}\}]) = \mathbf{1}$$

- Usually formalised as a uniqueness constraint

$$\boldsymbol{x} \mapsto \boldsymbol{y}_1 \in \boldsymbol{f} \land \boldsymbol{x} \mapsto \boldsymbol{y}_2 \in \boldsymbol{f} \quad \Longrightarrow \quad \boldsymbol{y}_1 = \boldsymbol{y}_2$$

Function Application

We can use functional application for partial functions

- If $x \in dom(f)$, then we write $f(x)$ for the unique range element associated with x in f.
- if $x \notin dom(f)$, then $f(x)$ is undefined.
- if $card(f[\{x\}]) > 1$, then $f(x)$ is undefined.

Name	Expression	Meaning	Well-definedness
Function application	$f(x)$	$f(x) = y \Leftrightarrow x \mapsto y \in f$	$f \in X \pfun Y \land x \in dom(f)$

Examples

$NAMES = \{Anna, Ben, Jack, Alex, Irum\},\ MNUMBERS = \{0123, 1230, 2301, 3012\}$

$\boldsymbol{studentNumber1} = \{Anna \mapsto 0123, Ben \mapsto 1230, Irum \mapsto 3012\}$

$\boldsymbol{studentNumber2} = \{Anna \mapsto 0123, Ben \mapsto 1230, Jack \mapsto 2301, Jack \mapsto 3012\}$

- $\boldsymbol{studentNumber1} \in NAMES \nrightarrow MNUMBERS$

$\boldsymbol{studentNumber1}(Ben)$=1230

$\boldsymbol{studentNumber1}(Jack)$ is undefined

- $\boldsymbol{studentNumber2} \notin NAMES \nrightarrow MNUMBERS$

$\boldsymbol{studentNumber2}(Jack)$ is undefined

Domain Restriction

- Given relation $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{S},$ the **domain restriction** of $\boldsymbol{R}$ by $\boldsymbol{A}$ is written

$$\boldsymbol{A} \lhd \boldsymbol{R}$$

- Restrict relation $\boldsymbol{R}$ so it only contains pairs whose <u>first</u> part is in the set $\boldsymbol{A}$ (keep only those pairs whose first element is in A)
- Example:

$$\boldsymbol{fruitColor} = \{green \mapsto grape, yellow \mapsto banana, red \mapsto apple\}$$

$$\{red, pink\} \lhd \boldsymbol{fruitColor} = \{red \mapsto apple\}$$

Domain Subtraction

- Given $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{S}$ the domain subtraction of $\boldsymbol{R}$ by $\boldsymbol{A}$ is written

$$\boldsymbol{A} \lhd \boldsymbol{R}$$

- Remove those pairs from relation $\boldsymbol{R}$ whose first part is in the set $\boldsymbol{A}$ (keep only those pairs whose first element NOT in A)

- Example:

$$\boldsymbol{fruitColor} = \{green \mapsto grape, yellow \mapsto banana, red \mapsto apple\}$$

$$\{red, pink\} \lhd \boldsymbol{fruitColor} = \{green \mapsto grape, yellow \mapsto banana\}$$

Range Restriction

- Given $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{T}$ the range restriction of $\boldsymbol{R}$ by $\boldsymbol{A}$ is written

$$\boldsymbol{R} \rhd \boldsymbol{A}$$

- Restrict relation R so the it only contains pairs whose <u>second</u> part is in the set $\boldsymbol{A}$ (keep only those pairs whose second element is in $\boldsymbol{A}$)

- Example:

$$\boldsymbol{fruitColor} = \{green \mapsto grape, yellow \mapsto banana, red \mapsto apple\}$$
$$\boldsymbol{fruitColor} \rhd \{grape, pear\} = \{green \mapsto grape\}$$

Range Subtraction

- Given $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{T}$ the range subtraction of $\boldsymbol{R}$ by $\boldsymbol{A}$ is written

$$\boldsymbol{R} \mathbin{⩥} \boldsymbol{A}$$

- Remove those pairs from relation $\boldsymbol{R}$ whose <u>second</u> part is in the set $\boldsymbol{A}$ (keep only those pairs whose second element NOT in $\boldsymbol{A}$)

- Example:

$$\boldsymbol{fruitColor} = \{green \mapsto grape, yellow \mapsto banana, red \mapsto apple\}$$

$$\boldsymbol{fruitColor} \mathbin{⩥} \{grape, banana\} = \{red \mapsto apple\}$$

Domain and range, restriction and subtraction: summary

Assume $\boldsymbol{R} \in \boldsymbol{S} \leftrightarrow \boldsymbol{T}$ and $\boldsymbol{A} \subseteq \boldsymbol{S}, \boldsymbol{B} \subseteq \boldsymbol{T}$

Predicate	Definition	Name
$\boldsymbol{x} \mapsto \boldsymbol{y} \in A \lhd R$	$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \land \boldsymbol{x} \in A$	Domain restriction
$\boldsymbol{x} \mapsto \boldsymbol{y} \in A \ntriangleleft R$	$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \land \boldsymbol{x} \notin A$	Domain subtraction
$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \rhd B$	$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \land \boldsymbol{y} \in B$	Range restriction
$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \ntriangleright B$	$\boldsymbol{x} \mapsto \boldsymbol{y} \in R \land y \notin B$	Range subtraction

Relation and function

Function is a special case of relation: at most one element from the range correspond to each element in the domain

Any operation applicable to a relation or a set is also applicable to a function

- domain and range of a function, range restriction, etc.

If $\boldsymbol{f}$ is a function , then $\boldsymbol{f}(\boldsymbol{x})$ is the result of function $\boldsymbol{f}$ for the argument x.

Function Overriding

- Override the function f by the function g :

$$f \oplus g$$

- Function f is updated according to g (Override: replace existing mapping with new ones)
- f and g must be partial functions of the same type

Function overriding definition

- Definition in terms of function override and set union

$$\boldsymbol{f} \oplus \{\boldsymbol{a} \mapsto \boldsymbol{b}\} = (\{\boldsymbol{a}\} \lhd \boldsymbol{f}) \cup \{\boldsymbol{a} \mapsto \boldsymbol{b}\}$$

$$\boldsymbol{f} \oplus \boldsymbol{g} = (\boldsymbol{dom}(\boldsymbol{g}) \lhd \boldsymbol{f}) \cup \boldsymbol{g}$$

- Examples:

$\boldsymbol{studentNumber} = \{Anna \mapsto 0123, Ben \mapsto 1230, Jack \mapsto 2301, Irum \mapsto 3012\},$

$\boldsymbol{g} = \{Ben \mapsto 5555\}$

$\boldsymbol{studentNumber} \oplus \boldsymbol{g} = \{Anna \mapsto 0123, Ben \mapsto 5555, Jack \mapsto 2301, Irum \mapsto 3012\}$

$\boldsymbol{g1} = \{Ben \mapsto 5555, Anna \mapsto 1111\}$

$\boldsymbol{studentNumber} \oplus \boldsymbol{g1} = \{Anna \mapsto 1111, Ben \mapsto 5555, Jack \mapsto 2301, Irum \mapsto 3012\}$

Total Functions

- A **total function** is a special kind of partial function. Declaration of f as a total function

$$f \in X \longrightarrow Y$$

- This means that f is well-defined for every element in X, i.e., $f \in X \longrightarrow Y$ is shorthand for

$$f \in X \nrightarrow Y \ \land \ dom(f) = X$$

Total injective function

Function called total **injective** (or **1-1)**, if for every element $\boldsymbol{y}$ from its range there exists only one element $\boldsymbol{x}$ in the domain and $\boldsymbol{dom(f) = X}$. Declaration $\boldsymbol{f}$

$$\boldsymbol{f \in X \rightarrowtail Y}$$

- Example*:*

$\boldsymbol{username \in USERS \rightarrowtail UNAMES}$

Every user in a system has one unique user name.

Total surjective function

Function called **surjective**, denoted as

$$\boldsymbol{f} \in \boldsymbol{X} \twoheadrightarrow \boldsymbol{Y}$$

if its range is the whole target and $\boldsymbol{ran}(\boldsymbol{f}) = \boldsymbol{Y}$.

- **Example**

$\boldsymbol{f}$ $-$"*attends school*"
$\boldsymbol{f} \in STUDENTS \twoheadrightarrow SCHOOLS$

- No school without students (full set $SCHOOLS$ is covered).

Bijective function

Function is **bijective**, if it is total, injective and surjective:

$$\boldsymbol{f} \in \boldsymbol{X} \rightarrowtail\!\!\!\!\twoheadrightarrow \boldsymbol{Y}$$

- Example

"Married to" – is **bijective** function,

$\boldsymbol{X}$ - set of "married man"

$\boldsymbol{Y}$ - set of "married woman"

Example: printer access for students

The system tracks the permissions that students have with regard to the printers available at the university network.

- A system should support adding a permission for a student in order to get an access to a particular printer and removing a permission.
- A system should support removing a student's access to all printers at once.
- A system should support giving the combined permissions of any two students to both of them.

Requirements document

R1. There is a finite number of students at the university

R2. There is a finite number of printers at the university network

R3. A student might have or might have not a permission to use one or several printers

R4. A permission can be added to a student

R5. A permission to use a certain printer can be removed from a student

R6. A permission to use all the printers can be removed from a student

R7. The system should be able to give a combined permission to any two students

Printer access

- Permissions are naturally expressed as a *relation* between students and printers, so the machine makes use of a variable whose type is relation.
- Since the machine will have to keep track of changing permissions, it will make use of a *variable* ***access*** whose type is a *relation* between *STUDENTS* and *PRINTERS*.
- As permissions are added or removed, the variable will be updated to reflect the information.

Printer access: context

```
CONTEXT PrinterAccess_c0
SETS STUDENTS   R1
     PRINTERS   R2
AXIOMS
     axm1: finite(STUDENTS)   R1
     axm2: finite(PRINTERS)   R2
     axm3: STUDENTS≠ Ø        R1
     axm4: PRINTERS≠ Ø        R2
END
```

Printer access: machine

```
MACHINE PrinterAccess_m0
SEES PrinterAccess_c0
VARIABLES access
INVARIANTS
      inv1: access ∈ STUDENTS ↔ PRINTERS   R3
EVENTS
  INITIALISATION ≙
      begin
            act1: access := ∅
      end
...
```

Model events

```
ADD ≙    R4
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
    then
        act1: access:=access ∪ {st ↦ pr}
    end
BLOCK ≙    R5
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
        grd3: st ↦ pr ∈ access
    then
        act1: access:=access \{st ↦ pr}
    end
```

Model events

•Domain subtractions: remove those pairs from relation access whose first part is in the set **{st}**(keep only those pairs whose first element is NOT st)

BAN ≙ $\boxed{\text{R6}}$

any *st*
where
grd1: *st* ∈ *STUDENTS*
then
act1: *access*:= $\{st\} \lhd$ *access*
end

UNIFY ≙ $\boxed{\text{R7}}$

any *st1 st2*
where
grd1: *st1* ∈ *STUDENTS*
grd2: *st2* ∈ *STUDENTS*
then
act1: *access*:= *access* ∪ $(\{st1\} \times access[\{st2\}]) \cup (\{st2\} \times access[\{st1\}])$
end

END

•Relational image: Set of all elements in ***ran(access)*** that has ***st2*** as the first elements of their pairs

Model events

•Cartesian product with singleton set will produce a set of pairs with st1 as the first elements and printers of the second student as the second element

BAN ≙ R6

any *st*

where

grd1: $st \in STUDENTS$

then

act1: $access := \{st\} \lhd access$

end

UNIFY ≙ R7

any *st1 st2*

where

grd1: $st1 \in STUDENTS$

grd2: $st2 \in STUDENTS$

then

act1: $access := access \cup (\{st1\} \times access[\{st2\}]) \cup (\{st2\} \times access[\{st1\}])$

end

END

Printer access rules

- Assume that we want to restrict the number of printers that a student can have access to.

For example, a student can use no more than 3 printers.

We have to reflect this new functionality in our model.

Model events: modification of ADD event

```
ADD ≜
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
        grd3: ??? // we have to specify new condition here
    then
        act1: access:=access ∪ {st ↦ pr}
    end
```

Model events: modification of ADD event

```
ADD ≙
    any st pr
    where
        grd1: st ∈ STUDENTS
        grd2: pr ∈ PRINTERS
        grd3: card({st}◁access) < 3        // new guard
    then
        act1: access:=access ∪ {st ↦ pr}
    end
```

// We restrict a domain of ***access*** *relation by a set containing one element student* ***st****, i.e.,* $\{st\} \lhd$ ***access***. As a result of this operation we get a set of pairs, whose the first element is ***st***. Then by ***card*** operator we count a number of such pairs. Thus, we get a number of printers that this particular student ***st*** has access to.

Model events: modification of UNIFY event

Similarly, we have to modify the event **UNIFY.**

However, the new guard here will be rather complex:

- *Informally:* we have to check, if, after the Unify operation, two students still will have access to no more than 3 printers.

This means that the following property should be defined as a model invariant (and, consequently preserved during events execution):

$$\forall\, st.\ st \in \boldsymbol{dom}(\boldsymbol{access}) \Rightarrow \boldsymbol{card}(\{st\} \lhd \boldsymbol{access}) \leq 3$$

More examples

- ***Every person is either a student or a lecturer. But no person can be a student and a lecturer at the same time.***

$STUDENTS \subseteq PERSONS, LECTURERS \subseteq PERSONS$

$LECTURERS \cup STUDENTS = PERSONS$

$LECTURERS \cap STUDENTS = \emptyset$

- ***Only lecturer can teach course***

$e.g., \ CourseLecturer \ \in COURSES \leftrightarrow LECTURERS$

More examples

- ***Every* course is given by *at most one* lecturer**

$CourseLecturer \in COURSES \longrightarrow LECTURERS$ // total function

- ***A lecturer has to teach at least one course and at most three courses***

$CourseLecturer \in COURSES \longrightarrow LECTURERS \wedge \boldsymbol{ran}(CourseLecturer) = LECTURERS$

$\wedge (\forall\, l.\ \boldsymbol{card}(CourseLecturer \rhd \{l\}) \leq 3))$

Range restriction: results in a set of pairs whose second element is *l*

Comment on Initialisation event

MACHINE *CoursesRegistration_m0*
SEES *CoursesRegistration_m0*
VARIABLES *access*
INVARIANTS
    inv1: $CourseLecturer \in COURSES \longrightarrow LECTURERS$
    ….
EVENTS
 INITIALISATION ≙
    begin
        act1: $CourseLecturer := \emptyset$ *// wrong! Since* $CourseLecturer$ *defined as a total function*
    end

inv1 invariant should be preserved upon **INITIALISATION** event.

BUT Rodin prover will fail to prove that since upon substitution $CourseLecturer$ by $\emptyset$, it will have to prove that $\emptyset \in COURSES \longrightarrow LECTURERS$. But it is wrong!

Simple example: seat booking system

The system allows a person to make a seat booking. Specifically:

- A system should support booking a seat by only one person;
- A system should support cancelling of a booking.

Modelling seat booking system in Event-B

- In the static part of our Event-B model – context - we will introduce required sets: *SEATS* and *PERSONS* as well as required axioms.
- In the dynamic part of the model – machine – we will define (specify) operations by events **BOOK** and **CANCEL**, correspondingly.
- We introduce a variable ***booked_seats*** whose type is a *partial function* on the sets *SEATS* and *PERSONS.*
- ***booked_seats*** keeps a track on booked seats and persons make their booking.
- Since booking of a seat can be done or cancelled, the variable ***booked_seats*** will be updated by the events **BOOK** or **CANCEL** to reflect this.

Seat booking system

We define a context **BookingSeats_c0** as follows

CONTEXT
    BookingSeats_c0
SETS
    PERSONS
    SEATS
AXIOMS
    axm1: finite(SEATS)
    axm2: finite(PERSONS)
    axm3: SEATS$\neq \emptyset$
    axm4: PERSONS$\neq \emptyset$
END

Machine BookingSeats_m0

```
MACHINE  BookingSeats_m0
SEES BookingSeats_c0
VARIABLES
        booked_seat
INVARIANTS
        inv1: booked_seat ∈ SEATS ⇸ PERSONS
// this variable is defined as a partial function (every seat can be
occupied by only one person, but not every seat from the set SEATS
is booked yet)
EVENTS
INITIALISATION  ≙
        then
                act1: booked_seat := ∅  // empty set
        end
BOOK ≙            //booking a seat
        any person  seat
        where
                grd1: person ∈ PERSONS     // take any person
                grd2: seat ∈ SEATS             // we take any seat ...
                grd3: seat ∉ dom(booked_seat)   // ... that is free
        then
                act1: booked_seat := booked_seat ∪ {seat ↦ person}
        end

CANCEL ≙           // cancelation of booking
        any person seat
        where
                grd1: seat ↦ person ∈ booked_seat     // any pair
                                                  from booked_seat
        then
                act1: booked_seat := booked_seat \ {seat ↦ person}
                // delete this pair from booked_seat
        end
END
```

Wrap-up

We have reviewed the ways to define the order of events and how it is related to invariant definition

This is important to keep in mind while defining safety invariant in assignment 1B

We have reviewed the notions functions and relations and their use in specification

We rely on the definitions of these mathematical structures to specify various aspects of system behaviour

We have reviewed the basic ideas of defining requirements document and tracing requirements in the specification