AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

SOFTWARE SPECIFICATION USING VDM-SL

STEFAN HALLESTEDE
PETER GORM LARSEN
CARL SCHULTZ

UNIVERSITET

AGENDA

› **An overview of VDM**

› The process of writing a formal specification (Alarm)

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

VDM BACKGROUND

› Our goal: well-founded but accessible modelling & analysis technology

› VDMTools → Overture → Crescendo → Symphony → INTO-CPS

› Pragmatic development methodologies

› Industry applications

› VDM: Model-oriented specification language

› Extended with objects and real time.

› Basic tools for static analysis

› Strong simulation support

› Model-based test

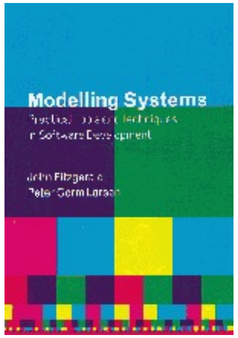
VDM

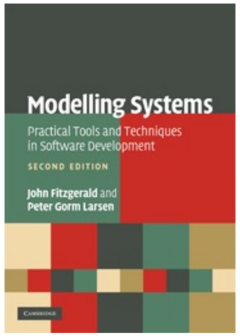
INTO-CPS

VDM (VIENNA DEVELOPMENT METHOD)

- A formal method for specification of software
- Three flavours
  - VDM-SL (Specification Language)
  - VDM++ adds object-orientation
  - VDM-RT adds real-time features (clock and deployment)
- Model-oriented specification language
  - Simple, abstract data types
  - Invariants to restrict membership
  - Specifying behaviour
    - Implicit specification (pre/post)
    - Explicit specification (functional or imperative)

VDM-SL MODULE OUTLINE






```
module <module-name>
  imports
  exports
     ...
                                   Interface

definitions
  state
  types
  values
  functions
  operations
     ...
                                   Definitions
end <module-name>
```

AGENDA

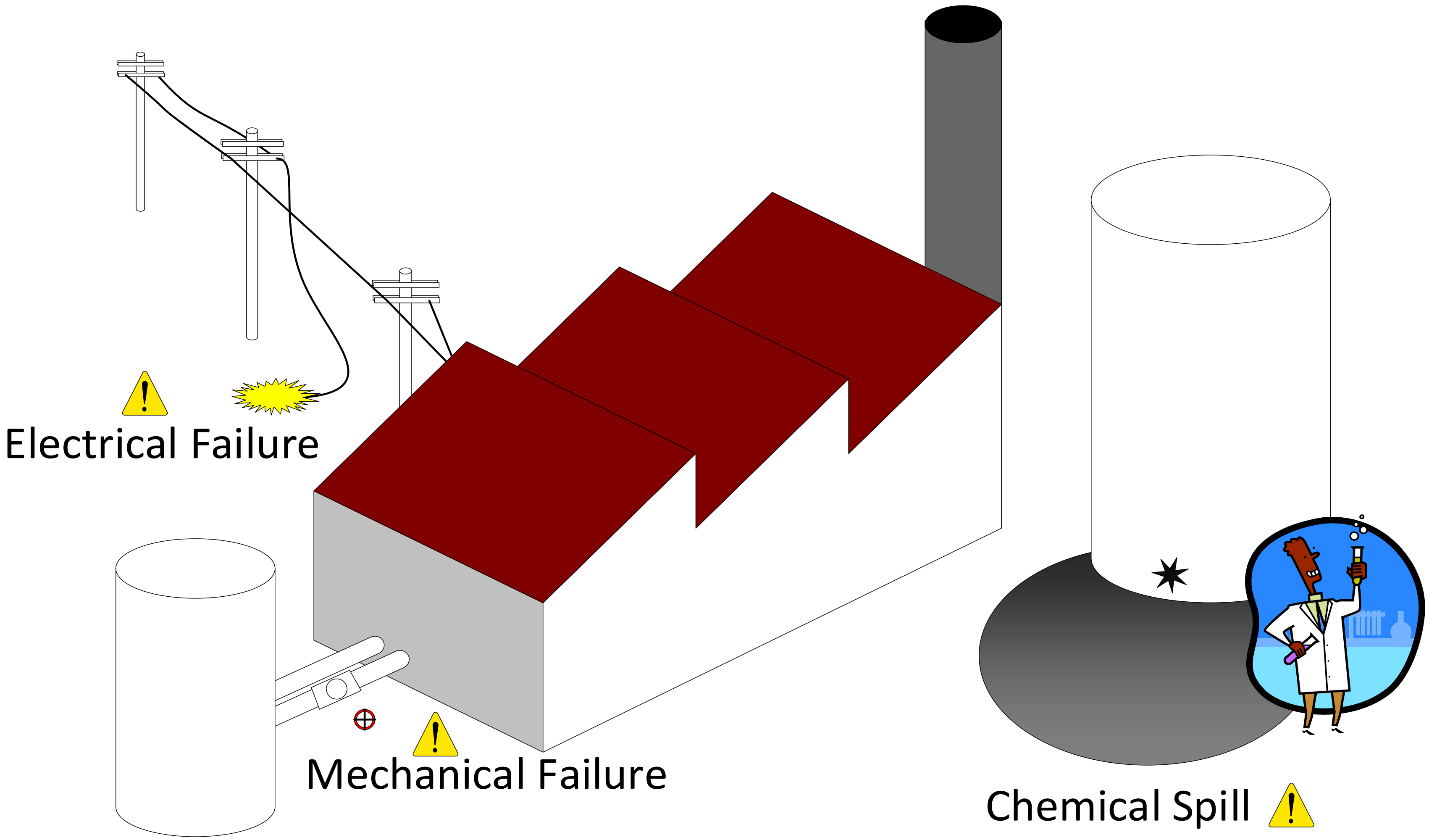
- › An overview of VDM
- › **The process of writing a formal specification (Alarm)**

STEPS TO DEVELOP A FORMAL MODEL

1. Determine the purpose of the model.
2. Read the requirements.
3. Extract a list of possible data types (often from nouns) and functions/operations (often from actions). Create a dictionary by giving explanations to items in the list.
4. Sketch out representations for the types.
5. Sketch out signatures for the functions/operations. Again, check the model's consistency in VDM.
6. Complete the data type definitions by determining potential invariant properties from the requirements and formalizing them.
7. Complete the function/operation definitions by determining pre- and post conditions and function/operation bodies, modifying the type definitions if necessary.
8. Validate the specification using systematic testing and rapid prototyping.
9. Implement the model using automatic code generation or manual coding.

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

A CHEMICAL PLANT



CHEMICAL PLANT REQUIREMENTS

1. A computer-based system is to be developed to manage the alarms of this plant.
2. Four kinds of qualifications are needed to cope with the alarms: electrical, mechanical, biological, and chemical.
3. There must be experts on duty during all periods allocated in the system.
4. Each expert can have a list of qualifications.
5. Each alarm reported to the system has a qualification associated with it along with a description of the alarm that can be understood by the expert.
6. Whenever an alarm is received by the system an expert with the right qualification should be found so that he or she can be paged.
7. The experts should be able to use the system database to check when they will be on duty.
8. It must be possible to assess the number of experts on duty.

THE PURPOSE OF THE VDM-SL MODEL

The **purpose** of the model is to clarify the rules governing the duty roster and calling out of experts to deal with alarms.

CREATING A DICTIONARY

- Potential Types (Nouns)
- **Alarm**: required qualification and description
- **Plant**: the entire system
- **Qualification** (electrical, mechanical, biological, chemical)
- **Expert**: list of qualifications
- **Period** (whatever shift system is used here)
- **System** and system database? This is probably a kind of schedule.
- Potential Functions/Operations (Actions)
- **Expert to page**: when an alarm appears (what's involved? Alarm operator and system)
- **Expert is on duty**: check when on duty (what's involved? Expert and system)
- **Number of experts on duty**: presumably given period (what's involved? operator and system)

SKETCHING TYPE REPRESENTATIONS

R2: Four qualifications: electrical, mechanical, biological and chemical.

```
Qualification = <Elec> | <Mech> | <Bio> | <Chem>
```

- The | constructs the union of several types or quote literals
- The individual quoted values are put in angle brackets <...>
- This type has four elements corresponding to the four kinds of alarm and qualification.
- Just like an enumerated type in a programming language.

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

SKETCHING TYPE REPRESENTATIONS

*R5: Each alarm has a description (text for the expert) and **a qualification**.*

It is always worth asking clients whether they mean "a" or "some" or "at least one".

```
Alarm :: alarmtext : seq of char
         quali     : Qualification
```

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

RECORD TYPES

```
Alarm :: alarmtext : seq of char
         quali     : Qualification
```

- To say that a value `v` has type `T`, we write

```
v : T
```

- So, to state that `a` is an alarm, we write

```
a : Alarm
```

- To extract the fields from a record, we use a dot notation:

```
a.alarmtext
```

- To say that `a` is made up from some values, we use a record constructor "`mk_`":

```
a = mk_Alarm("Disaster - get here fast!", <Elec>)
```

This constructor builds a record from the values for its fields

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

SKETCHING TYPE REPRESENTATIONS

R4: Each expert can have a whole list of qualifications, not just one.

*Ask the client "Did you really mean a **list**, i.e. the order in which they are presented is important?*

```
Expert :: expertId : ExpertId
          quali    : set of Qualification
```

- Sometimes requirements given in natural language do not mean exactly what they say. If in doubt, consult an authority or the client! Hence a set here rather than a sequence.
- We try to keep the formal specification as abstract as possible - we only record the information that we need for the **purpose** of the model. The choice of what is relevant and what is not relevant is a matter of serious engineering judgement, especially where safety is concerned.

SKETCHING TYPE REPRESENTATIONS

- The informal requirements give us little indication that we will need to look inside the experts' identifiers. When we need a type, but no detailed representation, we use the special symbol **`token`**.

```
ExpertId = token
```

The same is also true for the periods into which the plant's timetable is split:

```
Period = token
```

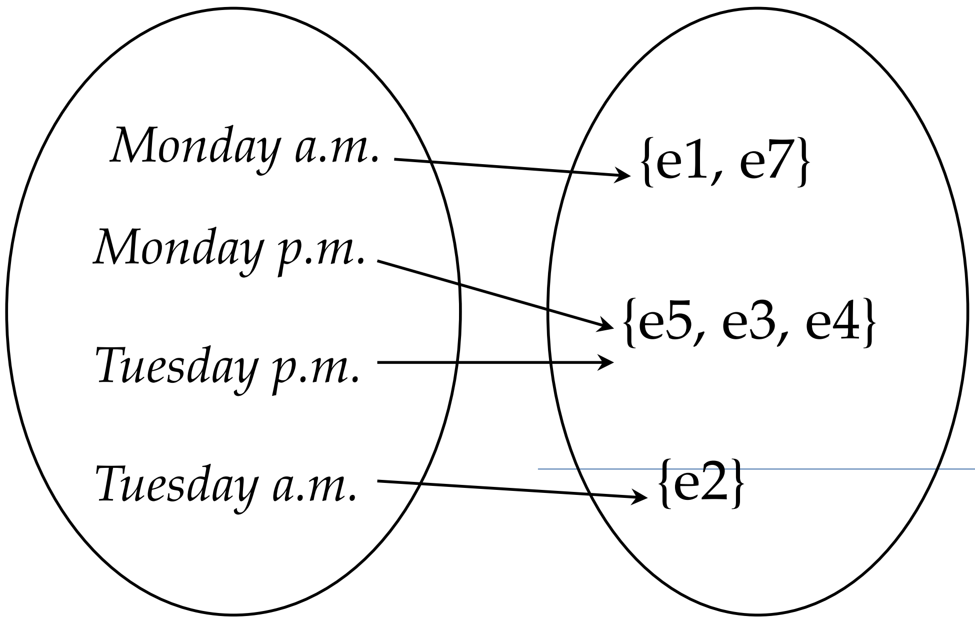
SKETCHING TYPE REPRESENTATIONS

R3: There must be experts on duty at all times.

R7: It shall be possible to check when a given expert is available.

› These requirements imply that there must be some sort of schedule relating each period of time to the set of experts who are on duty during that period:

```
Schedule = map Period to (set of Expert)
```



AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

SKETCHING TYPE REPRESENTATIONS

R1: A computer-based system is to be developed to manage expert call-out in response to alarms.

```
Plant :: sch    : Schedule
         alarms : set of Alarm
```

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

THE MODEL SO FAR - TYPE DEFINITIONS

```
Plant :: sch    : Schedule
         alarms : set of Alarm

Schedule = map Period to set of Expert

Period = token

Expert :: expertid : ExpertId
          quali    : set of Qualification

ExpertId = token

Qualification = <Elec> | <Mech> | <Bio> | <Chem>

Alarm :: alarmtext : seq of char
         quali     : Qualification
```

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

SKETCHING FUNCTION SIGNATURES

- Possible functions were: `ExpertToPage`
- `ExpertIsOnDuty`
- `NumberOfExperts`

- A function definition shows the types of the input parameters and the result in a signature:
- `ExpertToPage: Alarm * Period * Plant -> Expert`
- `ExpertIsOnDuty: Expert * Plant -> set of Period`
- `NumberOfExperts: Period * Plant -> nat`

COMPLETE TYPE DEFINITION

- Additional constraints on the values in the system which must hold at all times are called *data type invariants*.
- Example: suppose we agree with the client that experts should always have at least one qualification. This is a restriction on the type `Expert`. To state the restriction, consider a typical value `ex` of type `Expert`
- `ex.quali <> {}`
- We attach invariants to the definition of the relevant data type:

```
Expert :: expertid : ExpertId
          quali    : set of Qualification
inv ex == ex.quali <> {}
```

This is a formal parameter standing for a typical element of the type.

The body of the invariant is a Boolean expression recording the restriction on the formal parameter which represents a typical element of the type.

COMPLETE TYPE DEFINITIONS

R3: There must be experts on duty at all times.

- This is a restriction on the schedule to make sure that, for all periods, the set of experts is not empty.
- Again, we state this formally. Consider a typical schedule, called `sch`
- `forall exs in set rng sch & exs <> {}`

- Attaching this to the relevant type definition:

```
Schedule = map Period to set of Expert
inv sch == forall exs in set rng sch & exs <> {}
```

COMPLETE FUNCTION DEFINITIONS

- A function definition contains:
- *A signature*
- `NumberOfExperts: Period * Plant -> nat`
- *A parameter list*
- `NumberOfExperts(peri,pl) ==`
- *A body*
- `card pl.sch(peri)`
- *A pre-condition (optional)*
- `pre peri in set dom pl.sch`
- If omitted, the pre-condition is assumed to be **`true`** so the function can be applied to any inputs of the correct type.

COMPLETE FUNCTION DEFINITIONS

R7: It shall be possible to check when a given expert is available.

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex,pl) ==
    {peri | peri in set dom pl.sch &
            ex in set pl.sch(peri)}
```

- For convenience, we can use the record constructor in the input parameter to make the fields of the record `pl` available in the body of the function without having to use the selectors:

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex,mk_Plant(sch,alarms)) ==
```

COMPLETE FUNCTION DEFINITIONS

› The `alarms` component of the `mk_Plant(sch,alarms)` parameter is not actually used in the body of the function and so may be replaced by a `-`. The final version of the function is:

```
ExpertIsOnDuty: Expert * Plant -> set of Period
ExpertIsOnDuty(ex, mk_Plant(sch,-)) ==
  {peri | peri in set dom sch
        & ex in set sch(peri)}
```

COMPLETE FUNCTION DEFINITIONS

R6: When an alarm is raised, the system should output the name of a qualified and available expert who can then be called in.

```
ExpertToPage: Alarm * Period * Plant -> Expert
ExpertToPage(al,peri,pl) == ???
```

› Can we specify what result has to be returned without worrying about how we find it? Use an *implicit definition:*

```
ExpertToPage(al:Alarm, peri:Period, pl:Plant) r:Expert
pre    …
post   r in set pl.sch(peri) and
       al.quali in set r.quali
```

By r we are able to refer to the result

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

HAVE YOU SPOTTED A PROBLEM WITH THE SYSTEM?

- The requirements were silent about ensuring that there is always an expert with the correct qualifications available. After consulting with the client, it appears to be necessary to ensure that there is always at least one expert with each kind of qualification available.
- How could we record this in the model?

```
Plant :: sch    : Schedule
         alarms : set of Alarm
inv mk_Plant(schedule,alarms) =
  forall a in set alarms &
    forall peri in set dom schedule &
      exists ex in set schedule(peri) &
        a.quali in set ex.quali
```

FINALLY, REVIEW THE REQUIREMENTS

- *R1: system to manage expert call-out in response to alarms.*
- *R2: Four qualifications.*
- *R3: experts on duty at all times.*
- *R4: expert can have list of qualifications.*
- *R5: Each alarm has description & qualification.*
- *R6: output the name of a qualified and available expert*
- *R7: check when a given expert is available.*
- *R8: assess the number of experts on duty at a given period*

WEAKNESSES IN THE REQUIREMENTS

- Silence on ensuring that at least one suitable expert is available.
- Use of identifiers for experts was implicit.
- “List” really meant “set”.
- Silence on the fact that experts without qualifications are useless.
- “A qualification” meant “several qualifications”.

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

SUMMARY OF PROCESS

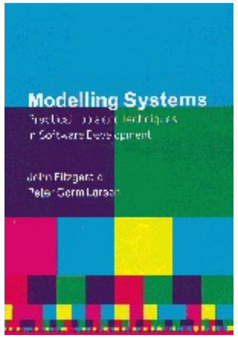
- Process of developing a model depends crucially on the statement of the model's purpose.
- VDM-SL models are based round type definitions and functions. Abstraction provided by the basic data types and type constructors and the ability to give implicit function definitions.
- **Basic types**
- **Type constructors**
- **Invariants**
- **Functions**
- **Operations**
- **State**

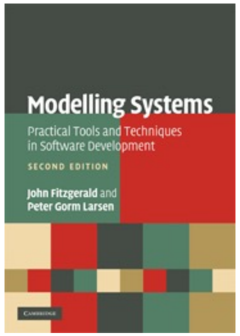
AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

AGENDA

- **An overview of VDM**
- The process of writing a formal specification (Alarm)

VDM-SL MODULE OUTLINE

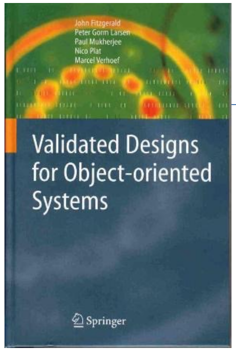





```
module <module-name>
  imports
  exports            } Interface
    ...

definitions
  state
  types
  values             } Definitions
  functions
  operations
    ...
end <module-name>
```

VDM++ CLASS OUTLINE




```
class <class-name>

  instance variables
     ...

  types
  values
  functions
  operations

  thread
     ...

  sync
     ...

  traces
     ...

end <class-name>
```

- instance variables: Internal object state
- types, values, functions, operations: Definitions
- thread: Dynamic behaviour
- sync: Synchronization control
- traces: Test automation support

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

DATA TYPES

- **Basic types**
  - Boolean
  - Numeric
  - Tokens
  - Quote types
  - Characters / String
- **Compound types**
  - Set types
  - Sequence types
  - Map types
  - Product types
  - Record types
  - Union types
  - Optional types

Type	Values
nat1	1, 2, 3, ...
nat	0, 1, 2, ...
int	..., -2, -1, 0, 1, ...
real	-12.78356, ..., 0, ..., 3, ..., 1726.34, ...

```
flag: bool := false;
flag := true
```

```
id: token := mk_token(5);
id := mk_token("Peter")
```

```
<RED>
<BLUE>
```

x = y (equality) and x <> y (inequality)

```
letter: char := 'k';
name: seq of char := "Peter"
```

```
s: set of int := {1,5,8,3};
1 in set s == true
```

```
s: seq of int := [1,5,5,8,3];
s(4) == 8
```

```
m: map int to real := {1 |-> 3.14};
m(1) == 3.14
```

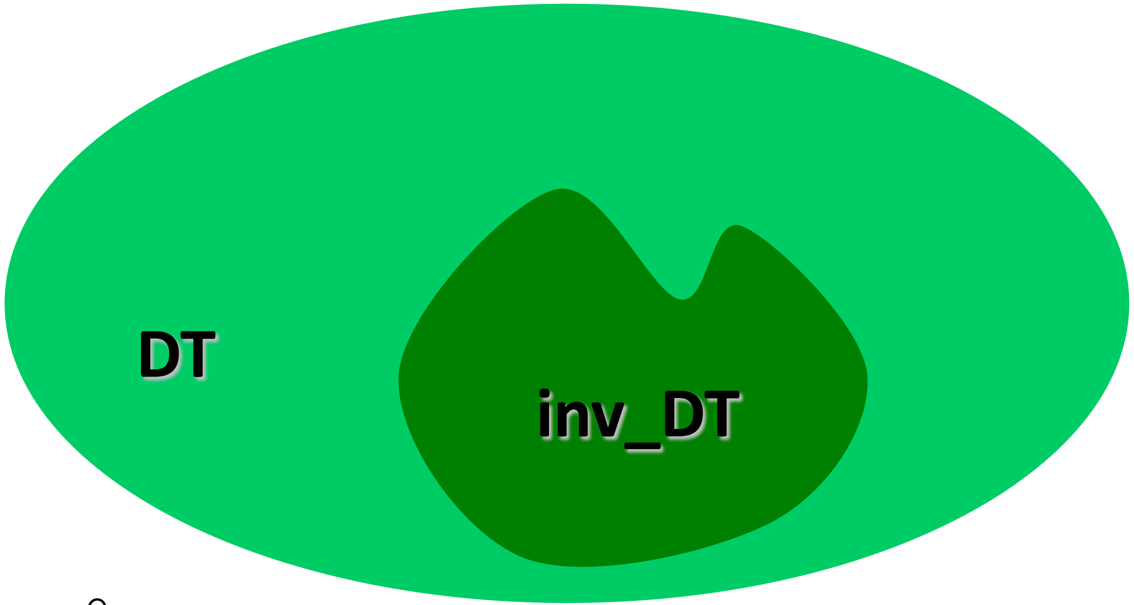
```
Pair = nat * real;
x: Pair := mk_(1, 3.14);
x.#1 == 1
```

```
Pair :: a : nat
        b : real;
x: Pair := mk_Pair(1, 3.14)
```

```
Colour = <RED> | <GREEN> | <BLUE>;
x: Colour := <RED>
```

```
Type = [nat]
```
equivalent to
```
Type = nat | nil
```

TYPE INVARIANTS




```
Even = nat
inv n == n mod 2 = 0

SpecialPair = nat * real – the first is smallest
inv mk_(n,r) == n < r

DisjointSets = set of set of  A
inv ss == forall s1, s2 in set ss &
                s1 <> s2 => s1 inter s2 = {}
```

SET TYPES

- Unordered collections of elements
- One copy of each element
- The elements themselves can any type
- e.g.
- **`set of int`**
- `{1,5,8,3};`
- `{}`

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

OVERVIEW OF SET OPERATORS

`e in set s1`	Membership ($\in$)	`A * set of A -> bool`
`e not in set s1`	Not membership ($\notin$)	`A * set of A -> bool`
`s1 union s2`	Union ($\cup$)	`set of A * set of A -> set of A`
`s1 inter s2`	Intersection ($\cap$)	`set of A * set of A -> set of A`
`s1 \ s2`	Difference (\)	`set of A * set of A -> set of A`
`s1 subset s2`	Subset ($\subseteq$)	`set of A * set of A -> bool`
`s1 psubset s2`	Proper subset ($\subset$)	`set of A * set of A -> bool`
`s1 = s2`	Equality (=)	`set of A * set of A -> bool`
`s1 <> s2`	Inequality ($\neq$)	`set of A * set of A -> bool`
`card s1`	Cardinality	`set of A -> nat`
`dunion s1`	Distr. Union ($\bigcup$)	`set of set of A -> set of A`
`dinter s1`	Distr. Intersection ($\bigcap$)	`set1 of set of A -> set1 of A`
`power s1`	Finite power set ($\mathbb{P}$)	`set of A -> set of set of A`

SEQUENCE TYPES

- Could also be called lists
- Not fixed length like Java arrays
- Ordered collections of elements
- Numbered from 1 (not 0 like Java)
- Access element with () and not [], e.g. `list(1)`
- Multiple copies of each element allowed
- The elements themselves can be any type
- e.g.
- **`seq of int`**; **`seq1 of int`** (non-empty)
- `[1,5,5,8,1,3]`
- `[]`

OVERVIEW OF SEQUENCE OPERATORS

hd l	Head	**seq1 of** A -> A
tl l	Tail	**seq1 of** A -> **seq of** A
len l	Length	**seq of** A -> **nat**
elems l	Elements	**seq of** A -> **set of** A
inds l	Indexes	**seq of** A -> **set of nat1**
l1 ^ l2	Concatenation	**seq of** A * **seq of** A -> **seq of** A
conc ll	Distr. conc.	**seq of seq of** A -> **seq of** A
l(i)	Seq. application	**seq1 of** A * **nat1** -> A
l ++ m	Seq. modification	**seq1 of** A * **map nat1 to** A -> **seq1 of** A
l1 = l2	Equality	**seq of** A * **seq of** A -> **bool**
l1 <> l2	Inequality	**seq of** A * **seq of** A -> **bool**

MAPPING TYPES

- Unordered collections of pairs of elements (maplets) with a unique relationship
- mapping keys to values
- like Python dictionary
- The elements themselves can be any type
- e.g.
- **`map int to real`**
- `{1 |-> 3, 4 |-> 7, 5 |-> 3}`
- `{|->}`

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

OVERVIEW OF MAPPING OPERATORS

dom m	Domain	(**map** A **to** B) -> **set of** A
rng m	Range	(**map** A **to** B) -> **set of** B
m1 **munion** m2	Merge	(**map** A **to** B) * (**map** A **to** B) -> (**map** A **to** B)
m1 ++ m2	Override	(**map** A **to** B) * (**map** A **to** B) -> (**map** A **to** B)
merge ms	Distr. merge	**set of** (**map** A **to** B) -> **map** A **to** B
s <: m	Dom. restr. to	**set of** A * (**map** A **to** B) -> **map** A **to** B
s <-: m	Dom. restr. by	**set of** A * (**map** A **to** B) -> **map** A **to** B
m :> s	Rng. restr. to	(**map** A **to** B) * **set of** A -> **map** A **to** B
m :-> s	Rng. restr. by	(**map** A **to** B) * **set of** A -> **map** A **to** B
m(d)	Map apply	(**map** A **to** B) * A -> B
inverse m	Map inverse	**inmap** A **to** B -> **inmap** B **to** A
m1 = m2	Equality	(**map** A **to** B) * (**map** A **to** B) -> **bool**
m1 <> m2	Inequality	(**map** A **to** B) * (**map** A **to** B) -> **bool**

SPECIFYING BEHAVIOUR

- Specifications in terms of post-conditions define a contract

```
sqrt(x: nat) r: real
post x = r * r
```

Implicit definition, not executable

- Explicit version

```
sqrt: nat -> real
sqrt(x) == Math`sqrt(x)
```

Explicit definition can be executed

- Pre-condition and post-conditions

```
sqrt: int -> real
sqrt(x) == Math`sqrt(x)
pre x > 0
post x = RESULT * RESULT
```