AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

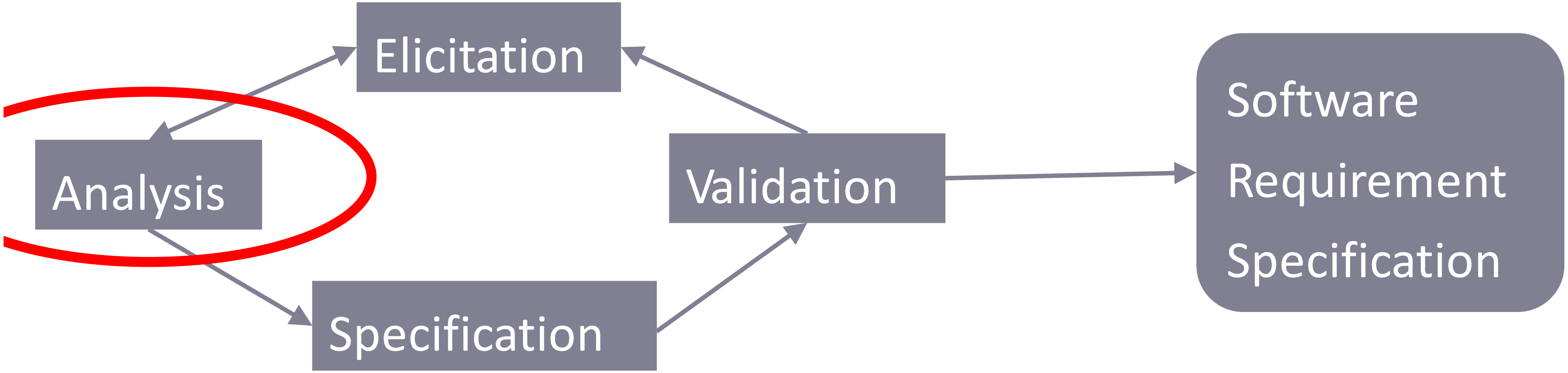
SOFTWARE ENGINEERING PRINCIPLES REQUIREMENTS ANALYSIS

STEFAN HALLESTEDE
PETER GORM LARSEN
CARL SCHULTZ

UNIVERSITET

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

PROCESS FOR CAPTURING REQUIREMENTS

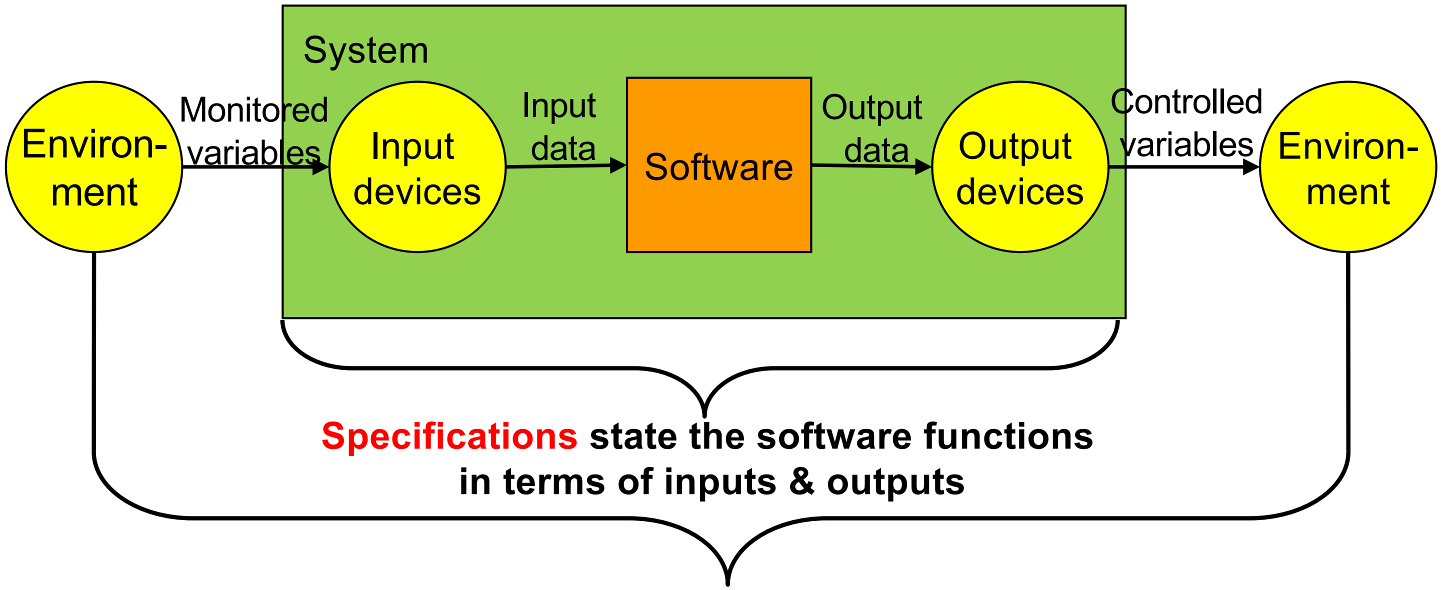


- **Elicitation**: Collecting the user's requirements
- **Analysis**: Understanding and modelling of desired behaviour
- **Specification**: Documenting the behaviour of the proposed software system
- **Validation**: Checking that the specification matches the user's requirements

AGENDA

- **Logical reasoning (A+S |- P)**
- Elevator example
- Logic
- Brief VDM introduction
- Executable models

HOW SOFTWARE ENGINEERS SEE THE WORLD

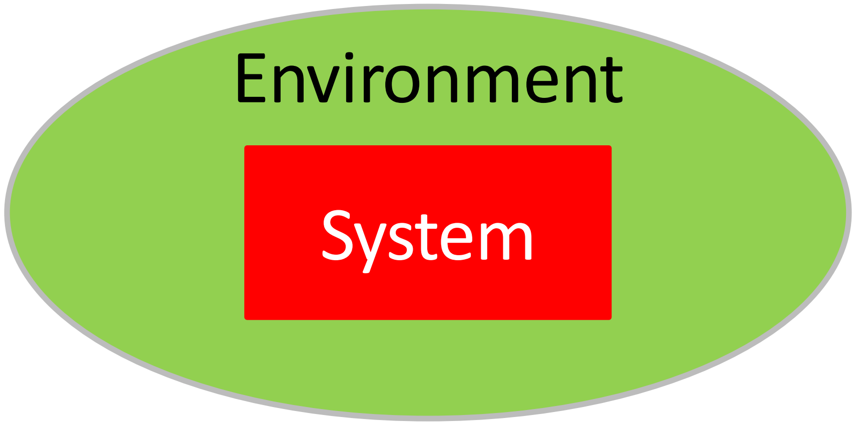


Assumptions about expections of the environment behaviour are important

Properties about the desired behaviour of the new system in the environment express a desired relationship between monitored and controlled variables

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

ASSUMPTIONS, SPECIFICATIONS AND SYSTEM PROPERTIES



Assumptions

+

Specification

Implies

Properties

About the combination

ENGINEERING ARGUMENTS

- The ability to predict the properties of a product before it is built
- Feed-forward loop from specification to product properties
- Assumptions and Specification entail Properties
- Properties are those desired for the System under Development:
  - System Requirements
  - System Constraints
- See Software for Dependable Systems: Sufficient Evidence? Daniel Jackson et al.

EXAMPLE ENGINEERING ARGUMENTS

Engineering knowledge

- Products with this kind of decomposition usually have properties P
- Since this product will have this kind of decomposition
- It will probably have properties P

Throw-away prototyping

- Since the prototype has properties P,
- And the prototype is similar to the final product
- The final product probably has properties P

EXAMPLE ENGINEERING ARGUMENTS

Model execution

› Since the model execution has properties P
› If the system implements this model exactly
› Then the system will have properties P

Model checking

› Since the state transition graph has properties P,
› If the system implements this graph correctly
› Then the system will have properties P

EXAMPLE ENGINEERING ARGUMENTS

Theorem proving

› Since the decomposition has been proved to have properties P,

› If the system correctly implements this decomposition

› Then the system will have properties P

EXAMPLE ENGINEERING ARGUMENTS

Exercise - give the following arguments:

Model exection argument

Throw-away prototyping

Model checking

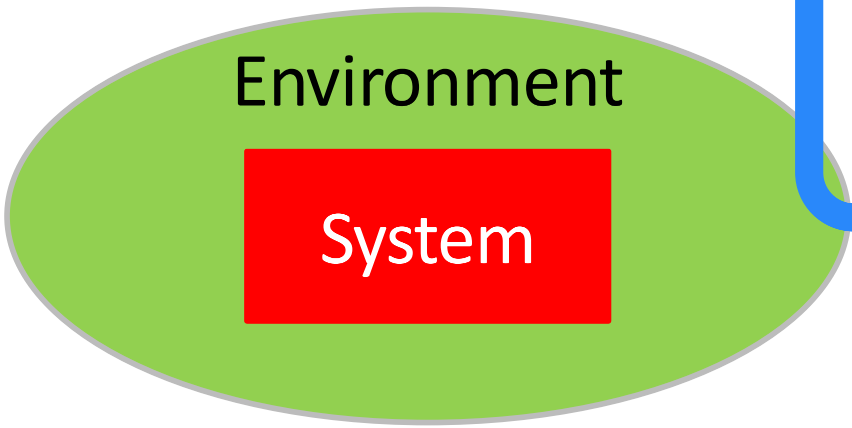
Theorem proving

Engineering knowledge

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

ASSUMPTIONS ABOUT THE ENVIRONMENT

What are assumptions?



Assumptions

+

Specification

Implies

Properties

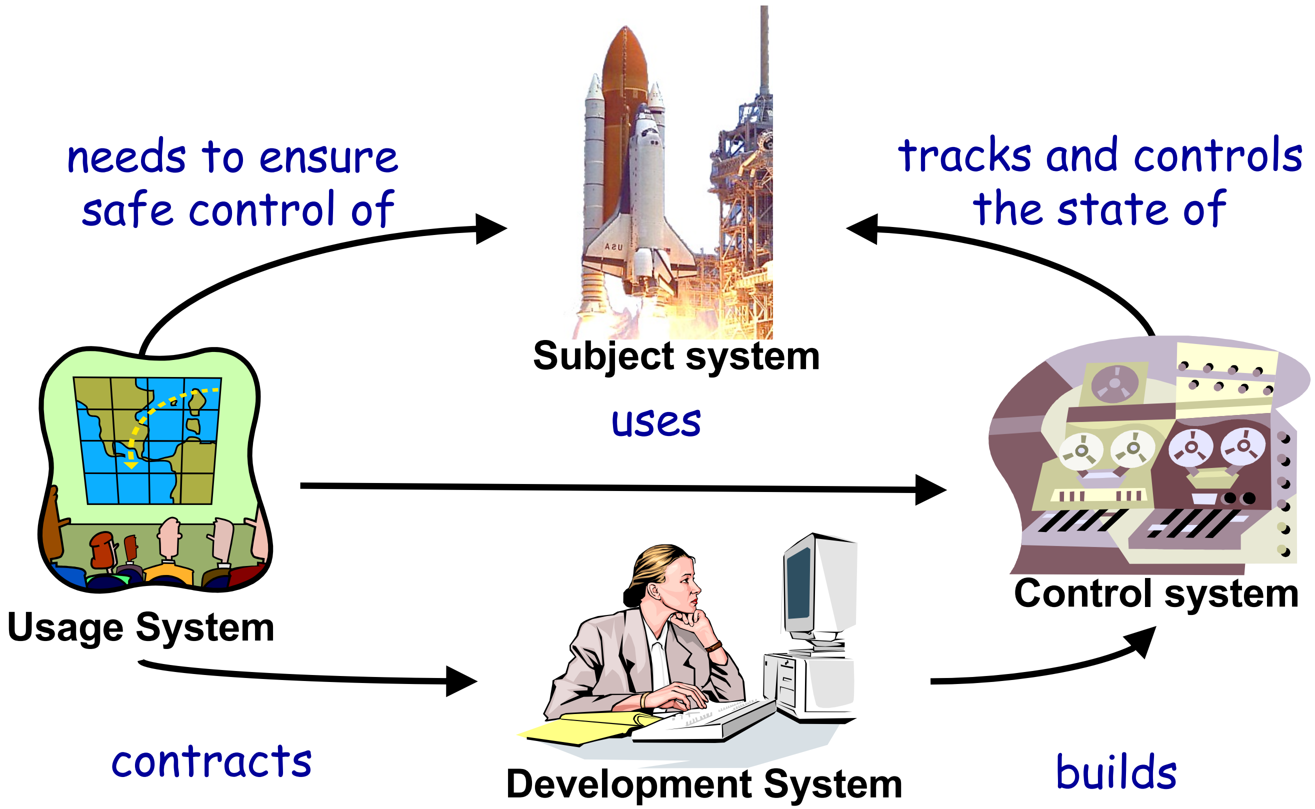
About the combination

ASSUMPTIONS ABOUT THE ENVIRONMENT

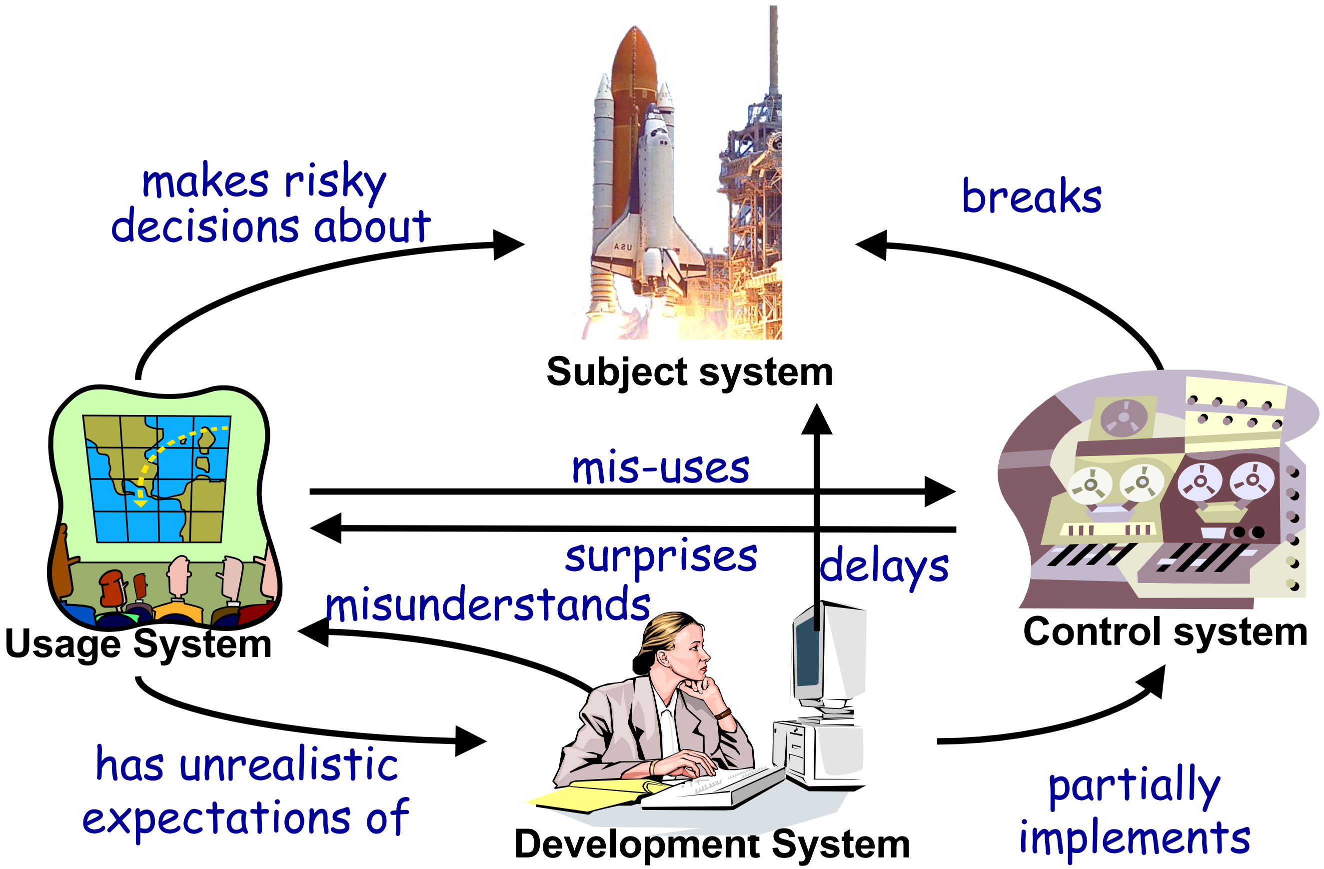
- Assumptions are statements about the environment
  - Must be true for the (stimulus, response) pair to be desirable
  - Beyond control of System under Development
- Examples of categories of assumptions
  - Laws of nature
  - Properties of devices
  - Properties of people (users, operators, …)
- Some are explicit – and some are implicit

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

HOW SYSTEMS ENGINEERS SEE THE WORLD



HOW THE WORLD REALLY IS...



WHO USES REQUIREMENTS?

- What are the uses of a software requirement specification?
- For ____________ it is a specification of the product that will be delivered, ***a contract***
- For ____________ it can be used as a basis ***for scheduling and measuring progress***
- For the ____________ it provides a specification of ***what to design***
- For ____________ it defines the range of ***acceptable implementations*** and the ***outputs that must be produced***
- For ____________ personnel it is used for ***validation, test planning, and verification***

WHO USES REQUIREMENTS?

- What are the uses of a software requirement specification?
- For customers it is a specification of the product that will be delivered, ***a contract***
- For managers it can be used as a basis ***for scheduling and measuring progress***
- For the software designers it provides a specification of ***what to design***
- For coders it defines the range of ***acceptable implementations*** and the ***outputs that must be produced***
- For quality assurance personnel it is used for ***validation, test planning, and verification***

AGENDA

- Logical reasoning (A+S |- P)
- **Elevator example**
- Logic
- Brief VDM introduction
- Executable models

Please refer to
ElevatorExample.PDF

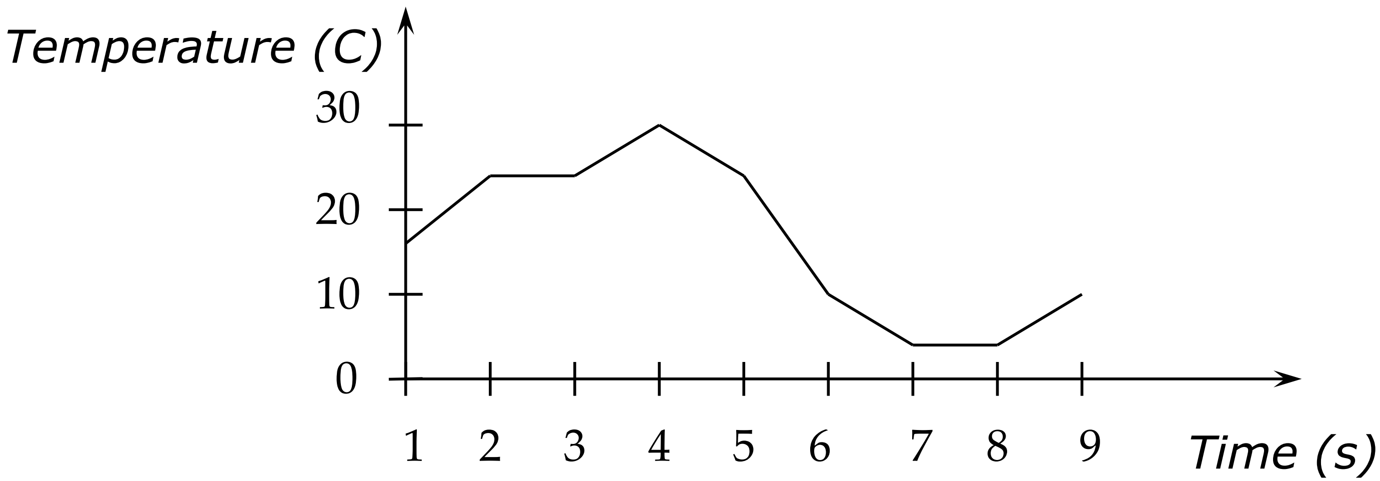
AGENDA

- Logical reasoning (A+S |- P)
- Elevator example
- **Logic**
- Brief VDM introduction
- Executable models

LOGIC

- Our ability to state invariants, record pre-conditions and post-conditions, and the ability to reason about a formal model depend on the logic on which the modelling language is based.
- Classical logical propositions and predicates
- Connectives
- Quantifiers

A TEMPERATURE MONITOR EXAMPLE



The monitor records the last five temperature readings

25	10	5	5	10

A TEMPERATURE MONITOR EXAMPLE

› The following conditions are to be detected by the monitor:

1. Rising: the last reading in the sample is greater than the first
2. Over limit: there is a reading in the sample in excess of 400 C
3. Continually over limit: all the readings in the sample exceed 400 C
4. Safe: If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.
5. Alarm: The alarm is to be raised if and only if the reactor is not safe

PREDICATES AND PROPOSITIONS

- *Predicates* are simply logical expressions. The simplest kind of logical predicate is a *proposition.*
- A *proposition* is a logical assertion about a particular value or values, usually involving a Boolean operator to compare the values, e.g.
- $3 < 27$ $5 = 9$

PREDICATES

- A predicate is a logical expression that is not specific to particular values but contains variables which can stand for one of a range of possible values, e.g.

```
x < 27
```

```
(x**2) + x - 6 = 0
```

- The truth or falsehood of a predicate depends on the value taken by the variables.

AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

PREDICATES IN THE MONITOR EXAMPLE

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

› Consider a monitor `m`. `m` is a sequence so we can index into it:

› First reading in `m`: `m.temps(1)`

› Last reading in `m`: `m.temps(5)`

› Predicate stating that the first reading in m is strictly less than the last reading:

```
m.temps(1) < m.temps(5)
```

› The truth of the predicate depends on the value of `m`.

THE RISING CONDITION

› The last reading in the sample is greater than the first

```
Monitor :: temps : seq of int
           alarm : bool

inv m == len m.temps = 5
```

› We can express the rising condition as a Boolean function:

```
Rising: Monitor -> bool

Rising(m) == m.temps(1) < m.temps(5)
```

› For any monitor m, the expression Rising(m) evaluates to true iff the last reading in the sample in m is higher than the first, e.g.

```
Rising( mk_Monitor([233,45,677,650,900], false) )

Rising( mk_Monitor([23,45,67,50,20], false) )
```

LOGICAL OPERATORS (CONNECTIVES)

We will examine the following logical operators:

- Negation (NOT)
- Conjunction (AND)
- Disjunction (OR)
- Implication (if – then)
- Biconditional (if and only if)
- Truth tables can be used to show how these operators can combine propositions to compound propositions.

NEGATION (NOT)

- Negation allows us to state that the opposite of some logical expression is true, e.g.
- *The temperature in the monitor* `mon` *is not rising:*
- `**not** Rising(mon)`

Truth table for negation:

P	¬P
true	false
false	true

DISJUNCTION (OR)

› Disjunction allows us to express alternatives that are not necessarily exclusive:

› Over limit: There is a reading in the sample in excess of 400 C

```
OverLimit: Monitor -> bool
OverLimit(m) ==
 m.temps(1) > 400 or
 m.temps(2) > 400 or
 m.temps(3) > 400 or
 m.temps(4) > 400 or
 m.temps(5) > 400
```

P	Q	PvQ
true	true	true
true	false	true
false	true	true
false	false	false

CONJUNCTION (AND)

- Conjunction allows us to express the fact that all of a collection of facts are true.
- Continually over limit: all the readings in the sample exceed 400 C

```
COverLimit: Monitor -> bool
COverLimit(m) ==
 m.temps(1) > 400 and
 m.temps(2) > 400 and
 m.temps(3) > 400 and
 m.temps(4) > 400 and
 m.temps(5) > 400
```

P	Q	P∧Q
true	true	true
true	false	false
false	true	false
false	false	false

IMPLICATION

› Implication allows us to express facts which are only true under certain conditions (“if … then …”):

› Safe: If readings do not exceed 400 C by the middle of the sample, the reactor is safe. If readings exceed 400 C by the middle of the sample, the reactor is still safe provided that the reading at the end of the sample is less than 400 C.

```
Safe: Monitor -> bool
Safe(m) ==
  (m.temps(3) > 400) =>
  (m.temps(5) < 400)
```

P	Q	P⇒Q
true	true	true
true	false	false
false	true	true
false	false	true

BIIMPLICATION

› Biimplication allows us to express equivalence (“if and only if”).
› Alarm: The alarm is to be raised if and only if the reactor is not safe
› This can be recorded as an invariant property:

```
Monitor :: temps : seq of int
           alarm : bool
inv m ==
 len m.temps = 5 and
 not Safe(m) <=> m.alarm
```

P	Q	P⇔Q
true	true	true
true	false	false
false	true	false
false	false	true

QUANTIFIERS

- For large collections of values, using a variable makes more sense than dealing with each case separately.
- `inds m.temps` represents indices (1-5) of the sample
- The "over limit" condition can then be expressed more economically as:
- **exists** i **in set inds** m.temps & m.temps(i) > 400
- The "continually over limit" condition can then be expressed using "`forall`":

```
COverLimit: Monitor -> bool
COverLimit(m) ==
 forall i in set inds m.temps & m.temps(i) > 400
```

QUANTIFIERS

Syntax:

- `forall binding & predicate`
- `exists binding & predicate`

There are two types of binding:

Type Binding, e.g.

```
x : nat
n : seq of char
```

*A type binding lets the bound variable range over a **type** (a possibly infinite collection of values).*

Set Binding, e.g.

```
i in set inds m
x in set {1,…,20}
```

*A set binding lets the bound variable range over a **finite set of values.***

UNIVERSAL QUANTIFICATION

- Universal quantification is a generalised form of conjunction
- For example, the statement “every natural number is greater than or equal to zero” is denoted by
- $\forall n$: **nat** • $n \geq 0$ (∀ is a turned-round “A”, “for All” and written as “**`forall`**” in ASCII)
- “for all n drawn from the natural numbers,
- n is greater than or equal to zero”
- This statement is equivalent to (and a lot more succinct than):
- $0 \geq 0 \wedge 1 \geq 0 \wedge 2 \geq 0 \wedge 3 \geq 0 \wedge \ldots$

QUESTIONS

Formulate the following statements using predicate logic:

1. Everybody likes Danish pastry
2. Everybody either likes Danish pastry or is a vegetarian
3. Either everybody likes Danish pastry or everybody is a vegetarian
4. Are the last two statements equivalent?

EXISTENTIAL QUANTIFICATION

› Existential quantification allows us to assert that a predicate holds for at least one value — but not necessarily all values — of a given set

› For example, the statement "there is a natural number that is greater than or equal to zero" is denoted by:

› $\exists n$: **nat** • $n \geq 0$ ($\exists$ is a turned-round "E", "there Exists" and written as "**`exists`**" in ASCII)

› "there exists an n drawn from the natural numbers such that n is greater than or equal to zero"

$0 \geq 0 \vee 1 \geq 0 \vee 2 \geq 0 \vee 3 \geq 0 \vee \ldots$

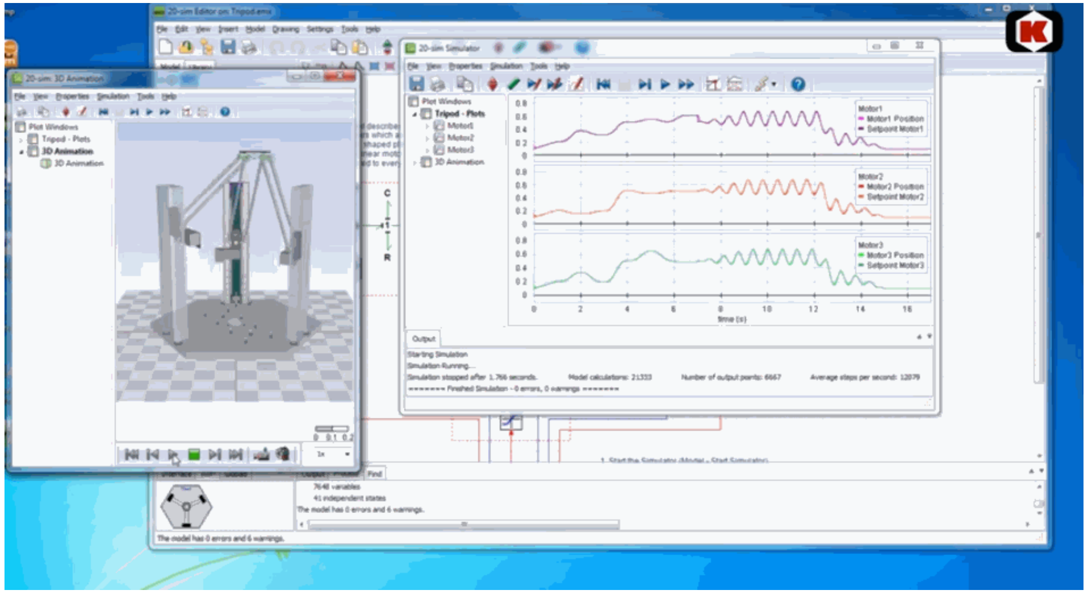
QUESTIONS

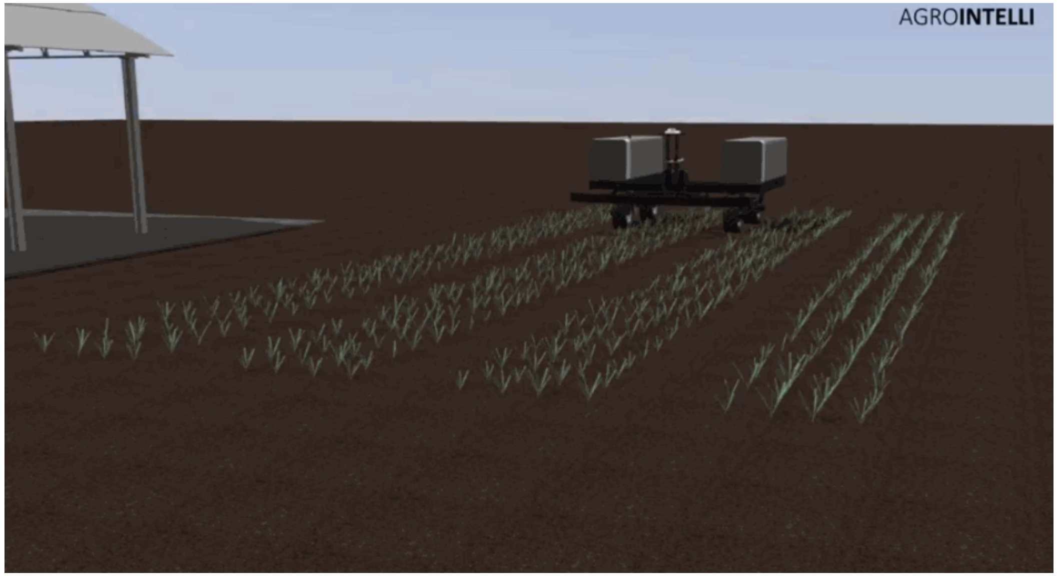
Formulate the following statements using predicate logic:

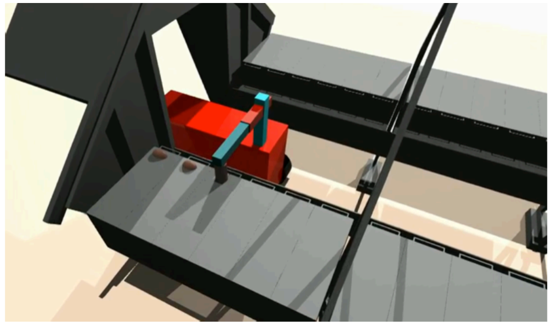
1. Somebody likes Danish pastry
2. There is somebody who either likes Danish pastry or is a vegetarian
3. Either somebody likes Danish pastry or somebody is a vegetarian
4. Are the last two statements equivalent?

AGENDA

› Logical reasoning (A+S |- P)
› Elevator example
› Logic
› **Brief VDM introduction**
› Executable models




AGROINTELLI



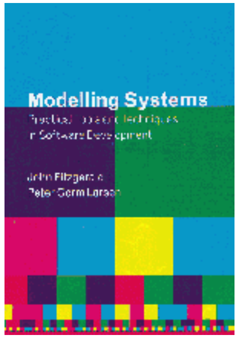
VDM BACKGROUND

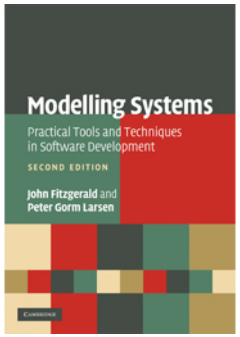
- Our goal: well-founded but accessible modelling & analysis technology
- VDMTools → Overture → Crescendo → Symphony
- Pragmatic development methodologies
- Industry applications
- VDM: Model-oriented specification language
- Extended with objects and real time.
- Basic tools for static analysis
- Strong simulation support
- Model-based test

VDM (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
  - Specification of functionality:
    - Implicit specification (pre/post)
    - Explicit specification (functional or imperative)

VDM-SL MODULE OUTLINE

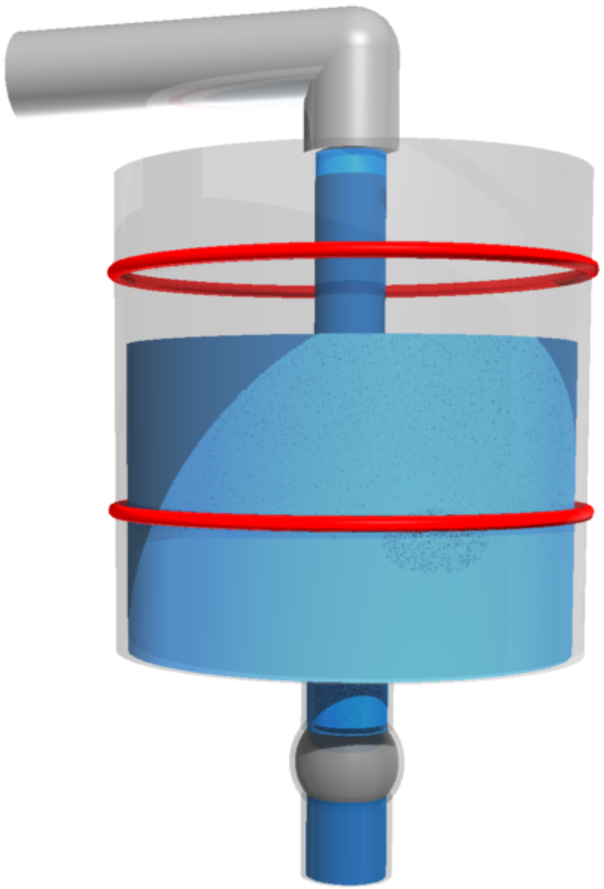





```
module <module-name>
   imports
   exports              } Interface
     ...

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

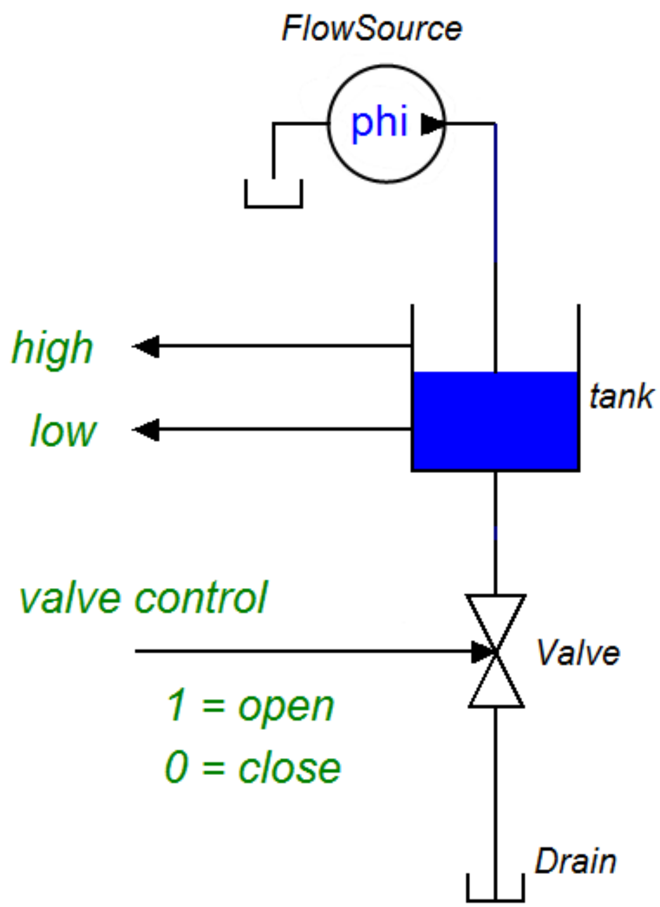
EXAMPLE: WATER TANK



$$\frac{dV}{dt} = \varphi_{in} - \varphi_{out}$$

$$\varphi_{out} = \begin{cases} \frac{\rho \cdot g}{A \cdot R} \cdot V \; if \; valve \; open \\ 0 \qquad if \; valve \; closed \end{cases}$$

EXAMPLE: WATER TANK



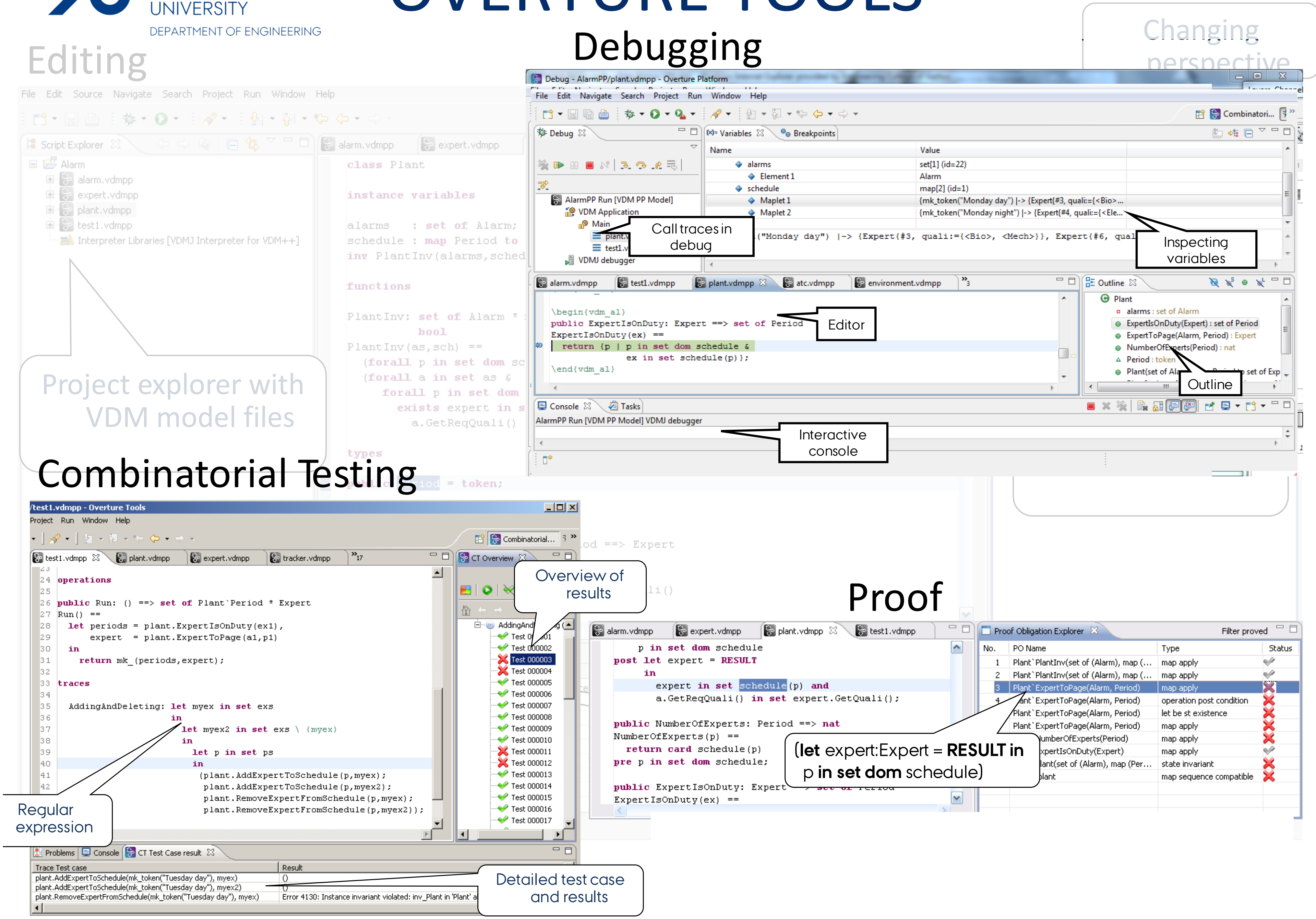
CT-side

```
class Controller
instance variables
 private i : Interface
operations
 async public Open:() ==> ()
 Open() == duration(50)
 i.SetValve(true);
 async public Close:() ==> ()
 Close() == cycles(1000)
 i.SetValve(false);
sync
 mutex(Open, Close);
 mutex(Open); mutex(Close)
end Controller
```

DE-side

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

OVERTURE TOOLS



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)
```

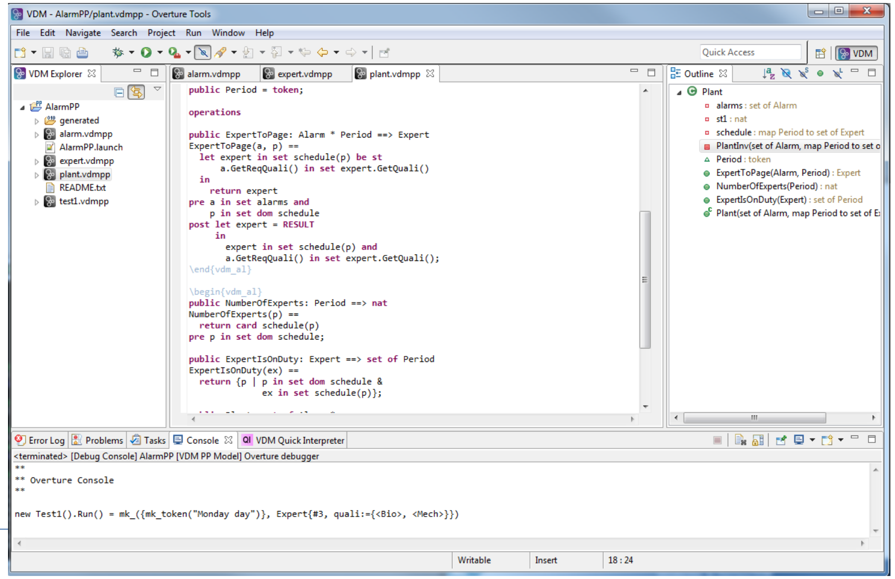
Explicit definition can be executed

- Pre-condition and post-conditions

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

THE OVERTURE TOOL

- Open-source tool for analysing VDM models
- Eclipse/Java based
- Currently in version 2.5.2 (September 11th 2017)
- Visit us at http://overturetool.org/
- Useful references
- Examples can be imported
- Language manual
- Tool users manual



AGENDA

- Logical reasoning (A+S |- P)
- Elevator example
- Logic
- Brief VDM introduction
- **Executable models**

VALIDATION TECHNIQUES

Exercise: match techniques with definitions

- automatic checks of syntax & type correctness, detect unusual features
- search the state space to find states that violate the properties we are checking.
- run the model and check outcomes against expectations
- use a logic to reason symbolically about whole classes of states at once.
- organized process of examining the model alongside domain experts

- **Inspection**
- **Static Analysis**
- **Testing**
- **Model Checking**
- **Proof**

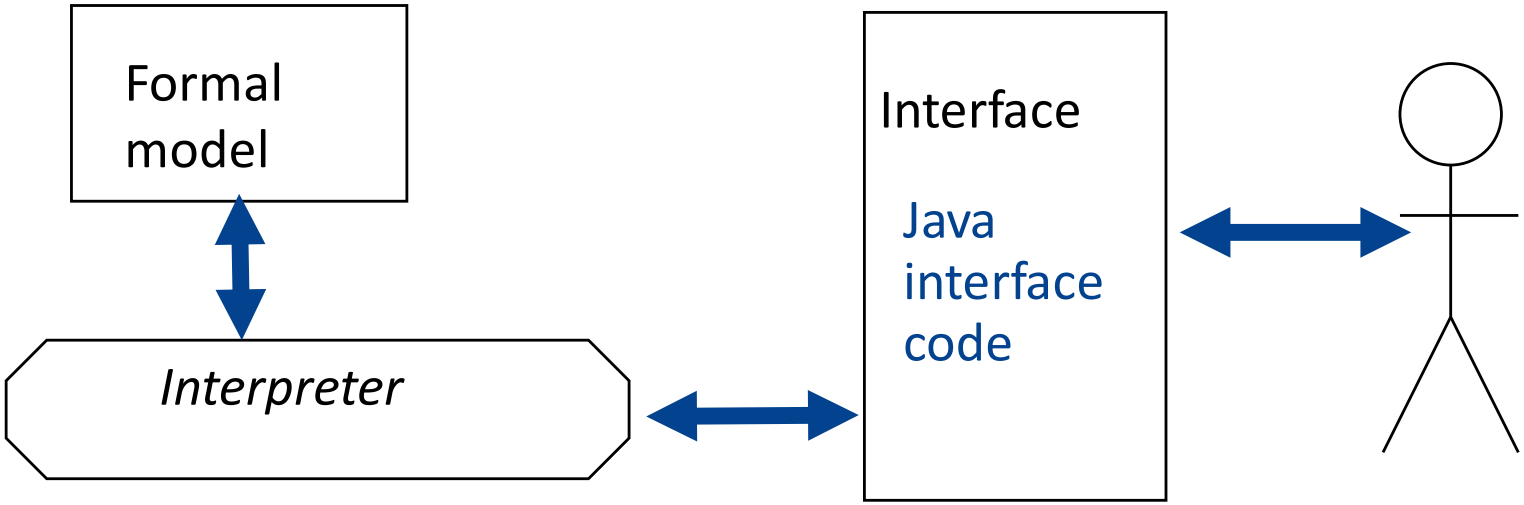
VALIDATION TECHNIQUES

- **Inspection**: organized process of examining the model alongside domain experts.
- **Static Analysis**: automatic checks of syntax & type correctness, detect unusual features.
- **Testing**: run the model and check outcomes against expectations.
- **Model Checking**: search the state space to find states that violate the properties we are checking.
- **Proof**: use a logic to reason symbolically about whole classes of states at once.

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

VALIDATION VIA ANIMATION

Execution of the model through an interface. The interface can be coded in the Java programming language



Testing can increase confidence, but is only as good as the test set. Exhaustive techniques could give greater confidence.

FUNCTION DEFINITIONS (1/2)

› Explicit functions:

```
f: A * B * … * Z -> R1 * R2 * … * Rn
f(a,b,…,z) ==
  expr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,RESULT)
```

› Implicit functions:

```
f(a:A, b:B, …, z:Z) r1:R1, …, rn:Rn
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,r1,…,rn)
```

Implicit functions cannot be executed by the VDM interpreter.

FUNCTION DEFINITIONS (2/2)

› Extended explicit functions:

```
f(a:A, b:B, …, z:Z) r1:R1, …, rn:Rn ==
  expr
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,r1,…,rn)
```

Extended explicit functions are a non-standard combination of the implicit colon style with an explicit body.

› Preliminary explicit functions:

```
f: A * B * … * Z -> R1 * R2 * … * Rn
f(a,b,…,z) ==
  is not yet specified
pre preexpr(a,b,…,z)
post postexpr(a,b,…,z,RESULT)
```

EXPLICIT FUNCTION DEFINITIONS

› A recursive function definition could look like:

```
fac: nat -> nat1
fac (n) ==
   if n > 1
   then n * fac(n-1)
   else 1
measure Id;
Id: nat -> nat
Id(a) == a
```

› Pre-conditions can also be used:

```
Division: real * real -> real
Division(p,q) ==
  p/q
pre q <> 0
```

AGENDA

› Logical reasoning (A+S |- P)
› Elevator example
› Logic
› Brief VDM introduction
› Executable models