Validating System Models

John Fitzgerald

Centre for Software Reliability

University of Newcastle upon Tyne

Validating System Models

- **We have learned to synthesize models**
  - **Sequential systems**
  - **Concurrent, communicating systems**
  - **Fault-tolerant distributed systems**

- ***But how do you know that your model is "right"?***
  - ***Inspection?***
  - ***Testing?***
  - ***Model checking?***
  - ***Proof?***

Gaining confidence of correctness is the subject of this course!

Validating System Models Course Content

1. The idea of validation:

- *A review of VDM-SL*
- *Validation techniques: from inspection to proof*

2. Proof in VDM:

- *Logical Frameworks*
- *Logic and Data*
- *The good news and the bad news …*

3. Supporting Validation:

- *Executable specification: abstraction versus accessibility*
- *Support tools for execution*
- *Support tools for proof*
- *A true story about validation and proof (The BNFL example)*

The idea of validation

1. Some Basic Principles

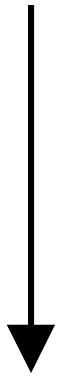
- *What is validation?*
- *System models*

2. A Review of VDM-SL

- *Model orientation: modelling data and functionality*
- *Example: the tracking manager*
- *Validation checks: internal consistency, validation conjectures*

3. Validation Techniques:

- *Technical review (inspection)*
- *Static checking*
- *Execution: testing (well, not really testing!)*
- *Model checking*
- *Proof*



More demanding - and less often done!

The idea of validation

Validation is the activity of increasing confidence that a model is consistent with informally expressed requirements.

Comparing the "formal" with the "informal".

vs.

Comparing two "formal" models.

The models could be requirements, designs, or even code.

Verification is the activity of increasing confidence that the system behaviour described by one model is consistent *with respect to another model.*

The idea of validation

What makes a good system model?

- *The model should have a clear purpose!*

- **Abstraction:** the omission of detail that is not relevant to the purpose for which the model is being constructed.

- **Rigour:** The language in which the model is expressed should be precisely defined. This permits objective, repeatable analysis of the model susceptible to machine support.

The idea of validation

VDM-SL is a model-oriented language: models consist of data and functions manipulating the defined data. A model contains:

- Data type definitions

```
Container :: fiss_mass : real
             material : <Glass> | <Metal> | <Liquid>
inv c == c.material = <Liquid> =>
         c.fiss_mass < Max_Liquid_Mass
```

Data type invariant

- Constant definitions

```
value Max_Liquid_Mass : real = 100.0
```

Type constructor (sets, sequences, mappings, records)

- State variable definitions

```
state Tracker of
  containers : map ContainerId to Container
  phases : map PhaseId to Phase
end
```

The idea of validation

- Functions

```
Consistent: Tracker -> bool
Consistent(mk_System(containers, phases)) ==
    forall ph in set rng phases &
           ph.contents subset dom containers;
```

Using operators on the basic types.

- Operations

```
Introduce(cid:ContainerId, m:real, s:Material)
ext wr containers : map ContainerId to Container
pre cis not in set dom containers
post containers = containers~ munion
                  {cid |-> mk_Container(m,s)}
```

Pre- and post-conditions (logic expressions) characterise the service to be delivered by the operation.

The idea of validation

Two definition styles for functions:

- **Explicit**

```
sumlist: seq of real -> real
sumlist(s) == if s = [] then 0
              else hd s + sum tl s
```

Gives an algorithm for the calculation of the function's result - the implemetor of the system doesn't have to use this algorithm (indeed, should not be biased by it - it need not be efficient!).

- **Implicit**

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

Gives no algorithm - merely characterises the result by giving its essential properties

The tracking manager example

A model of an architecture for tracking the movement of containers of hazardous waste as they go through reprocessing was developed by a team in Manchester Informatics with BNFL (Engineering) in 1995.

The **purpose** of the model was to establish the rules governing the movement of containers of waste which the tracking manager would have to enforce. The model was safety-related, but note that the model was built simply in order to understand the problem better, not as a basis for software development. *Models don't just have to serve as specifications.*

The tracking manager example

At the top level, the tracker holds information about containers and the phases of the plant:

```
Tracker :: containers : ContainerInfo
           phases     : PhaseInfo


ContainerInfo = map ContainerId to Container

PhaseInfo = map PhaseId to Phase

ContainerId = token

PhaseId = token

Container :: fiss_mass : real
             material : Material
```

The tracking manager example

Each phase houses a number of containers, expects certain material types and has a maximum capacity.

```
Phase :: contents : set of ContainerId
         expected_materials : set of Material
         capacity : nat
inv p == card p.contents <= p.capacity and
         p.expected_materials <> {}
```

Safety invariant put there by the domain experts.

The tracking manager example

```
State Tracker of
    containers : ContainerInfo
    phases     : PhaseInfo
inv mk_Tracker(containers,phases) ==
```

1. all of the containers present in phases are known about in the `containers` mapping.

2. no two distinct phases may have any containers in common.

3. in any phase, all the containers have the expected kind of material inside them.

```
inv mk_Tracker(containers,phases) ==
       Consistent(containers,phases) and
       PhasesDistinguished(phases) and
       MaterialSafe(containers,phases)
```

The invariants are defined as auxiliary functions.

The tracking manager example

Tracker functionality includes:

- introducing a new container to the tracker, giving its identifier and contents;
- giving permission for a container to move into a given phase;
- removing a container from a phase;
- deleting a container from the tracker.

The tracking manager example

```
Move (cid:ContainerId, ptoid, pfromid:PhaseId)
ext rd Containers: ContainerInfo
    wr phases: PhaseInfo
pre Permission(mk_Tracker(containers,phases),cid,ptoid)
post phases =
     phases~ ++ {pfromid |-> mk_Phase(...)}
             ++ {ptoid   |-> mk_Phase(...)}
```

The precondition ensures that we will respect the third part of the invariant on the tracker. In general, we need to check that the functions and operations we define respect invariants. This is an example of a ***proof obligation.***

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases),cid,dest) ==
```

The idea of validation

- ***What would we want to check in order to ensure that the tracker model meets our clients' expectations?***

- **Internal consistency:**
  - Syntax correctness
  - Type correctness
  - Unusual features (declared but not used etc.)
  - Partial operators don't get misused
  - Explicit functions respect the invariant
  - Implicit functions & operations are satisfiable

Static Checks

Proof Obligations

- **External consistency:**
  - Properties defined by the client, e.g. safety

Validation Conjectures

The idea of validation

A simple proof obligation: Domain Checking for Functions with Pre-conditions

If a function g uses a function `f:T1*…*Tn -> R` in its body, occurring as an expression `f(a1,…,an)`, then it is necessary to show

`pre-f(a1,…,an)`

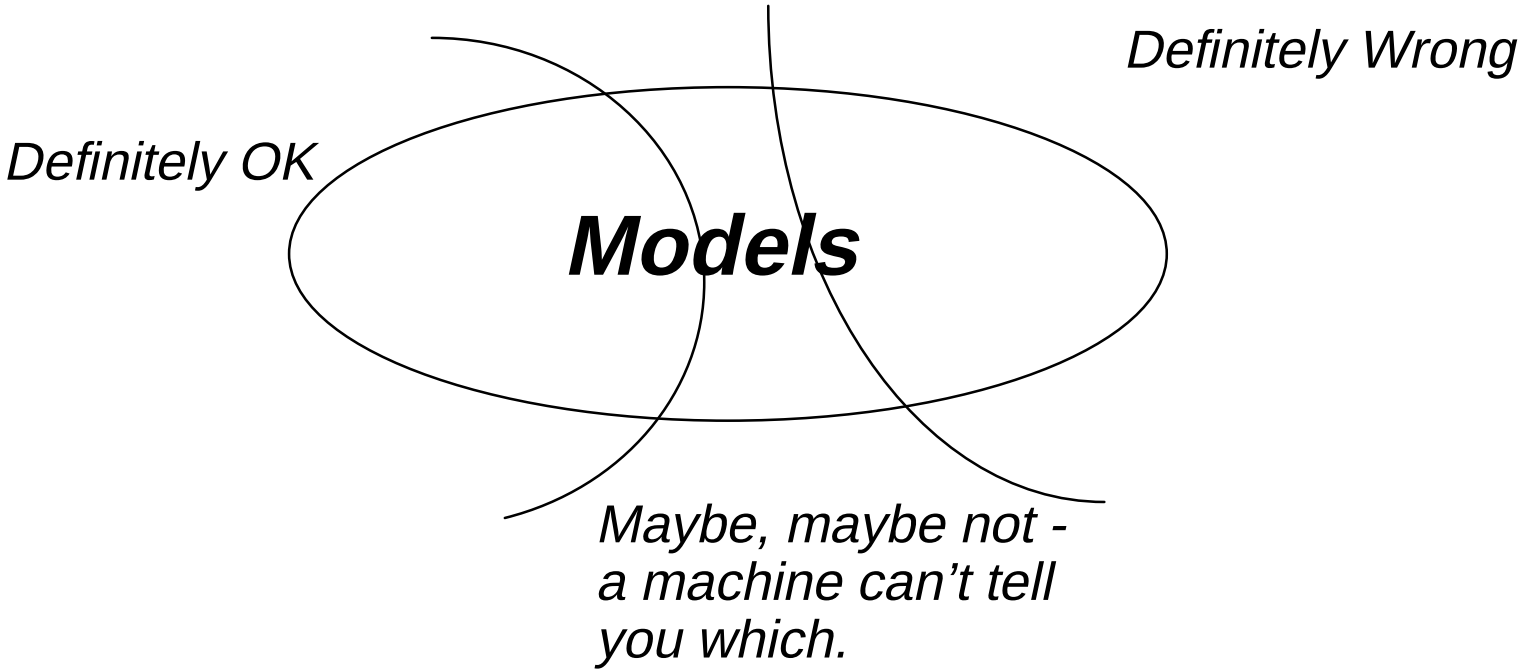
for any `a1,…,an` that can arise in this position.

```
Delete: Tracker * ContainerId * PhaseId -> Tracker
Delete(tkr,cid,source) ==
   mk_Tracker({cid} <-: tkr.containers,
              Remove(tkr,cid,source).phases)
pre pre_Remove(tkr,cid,source)
```

PO:

```
forall tkr:Tracker, cid:ContainerId, source:PhaseId &
   pre_Delete(tkr,cid,source) => pre_Remove(tkr,cid,source)
```

The idea of validation



Much of the current research in formal modelling aims to develop techniques and tools to reduce the size of the middle area by performing more and more checks automatically.

The idea of validation

An explicit function *with* a pre-condition:

```
f:T1*...*Tn -> R
f(a1,...,an) == ...
```

respects the invariant on R if, for all inputs satisfying the pre-condition, the result defined by the function body is of the correct type. Formally,

```
forall p1:T1,…,pn:Tn &
  pre_f(p1,…,pn) => f(p1,…,pn) : R
```

A function `f(a1:T1,…,an:Tn) r:R`

```
      pre ...
       ost ...
```

is ***satisfiable*** if, for all inputs satisfying the pre-condition, there exists a result of the correct type satisfying the post-condition. Formally,

```
forall p1:T1,…,pn:Tn &
   pre_f(p1,…,pn) =>
      exists x:R & post_f(a1,…,an,x)
```

- **How might we go about checking these properties?**

  - **Inspection:** organised 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.

Inspection

- "A method involving a **structured encounter** in which a group of technical personnel analyses an artifact according to a **well-specified process**. The outcome is a structured artifact that **assesses or improves** the quality of the artifact as well as the quality of the method."
- Typical roles:
  - Chairman
  - Author
  - Reader
  - Reviewers
  - Scribe
- Preparation and follow-up are vital.

Inspection

- Increasing interest in empirical assessment of inspections.
- **Defect removal efficiency** is seen as a measure of cost-effectiveness. Some combinations of inspection and testing are claimed to have achieved 99% efficiency.
- But a lack of reliable data on defect numbers in delivered products.
- 2-person teams are as effective as 4-person teams.
- Single inspection meetings are more cost-effective than multiple-meeting approaches.
- Computer-mediated inspections are a likely direction for future technical work in the area.

Static Analysis

- Parsing the source of the model (usually code) to identify anomalies.
- Derived data (e.g. flow graph) can be used to determine white-box tests (has been applied to VDM-SL models).
- No extensive empirical assessment.
- For formal models, static checks include syntax and type checking, but also automatic proof obligation generation for violation of conditions on partial operators.

Execution

- **The model must be executable!**

```
exists cid in set dom containers & …

exists cid : ContainerId & …


… post n*n = r
```

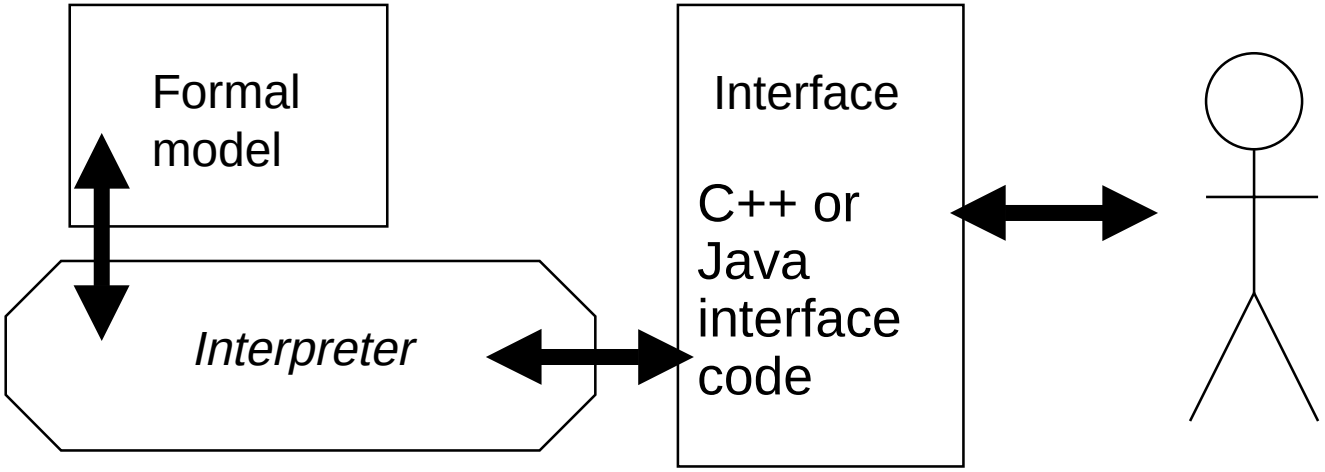
Quantification over infinite domains is problematic.

Implicit specifications can't be directly executed.

- If the model is executable, then we can run tests directly, in batch mode, with coverage analysis (done for executable VDM-SL models in the IFAD Tools).
- But validation involves domain experts who don't necessarily understand the formal model ...

The idea of validation

Animation is the execution of the model through an interface. The interface can be coded in a programming language of choice so long as a *dynamic link* facility exists for linking the interface code to the model.



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

Model checking

- **An exhaustive automated test of the state space to verify that a desired property holds in each state, or to find a counter-example.**
- The searched space must be finite!
- But infinite spaces may be searched if abstractions to a finite space can be made.
- Origins in hardware and protocol verification.
- An advantage over proof is the production of a counter example, but a disadvantage is the requirement for a tractable state space.

Levels of Proof:

- **"Textbook":** natural language supported by formulae. Justifications require human insight ("Clearly …", "By the properties of prime numbers …" etc.). Easiest style to read, but can only be checked by humans.

- **Formal:** highly structured sequences of formulae. Justifications appeal to a formally stated rule of inference (each rule can be axiomatic or itself a proved result). Can be checked by a machine. Construction very laborious, but yields high assurance (used in critical applications)

- **Rigorous:** highly structured sequence of formulae, but relaxes restrictions on justifications so that they may appeal to general theories rather than specific inference rules.

The idea of validation


```
From t:Tracker; cid:ContainerId;
     pf:PhaseId; pf in set dom t.phases;
     pt:PhaseId; pt in set dom t.phases;
     t.phases(pt).capacity = card t.phases(pt).contents

1    not(card phases(pt).contents < t.phases(pt).capacity
                                                arithmetic(h7)
2    not Permission(t,cid,pt)                 defn-Permission(1)
3    not Permission(t,cid,pt) or not pre_Remove(t,cid,pf)
                                                or-I-right(2)
4    not (Permission(t,cid,pt) and pre_Remove(t,cid,pf))
                                              not-and-I-deM(3)

Infer not pre_Move(t,cid,pt,pf)                       Fold(4)
```

The idea of validation

- **Validation** is the act of increasing confidence that a model of a system accurately reflects the client's informally expressed requirements
- A range of opportunities for validation arise in the production of a model:
  - **Internal consistency checks**
  - **Validation conjectures**
- **Techniques for validation** follow a range of different levels of assurance and at different costs.
- We also reviewed the *model-oriented* specification language of VDM.

Formal proof is seen as an ideal for validation. But what does it take to prove a property about a model?

Proof in VDM

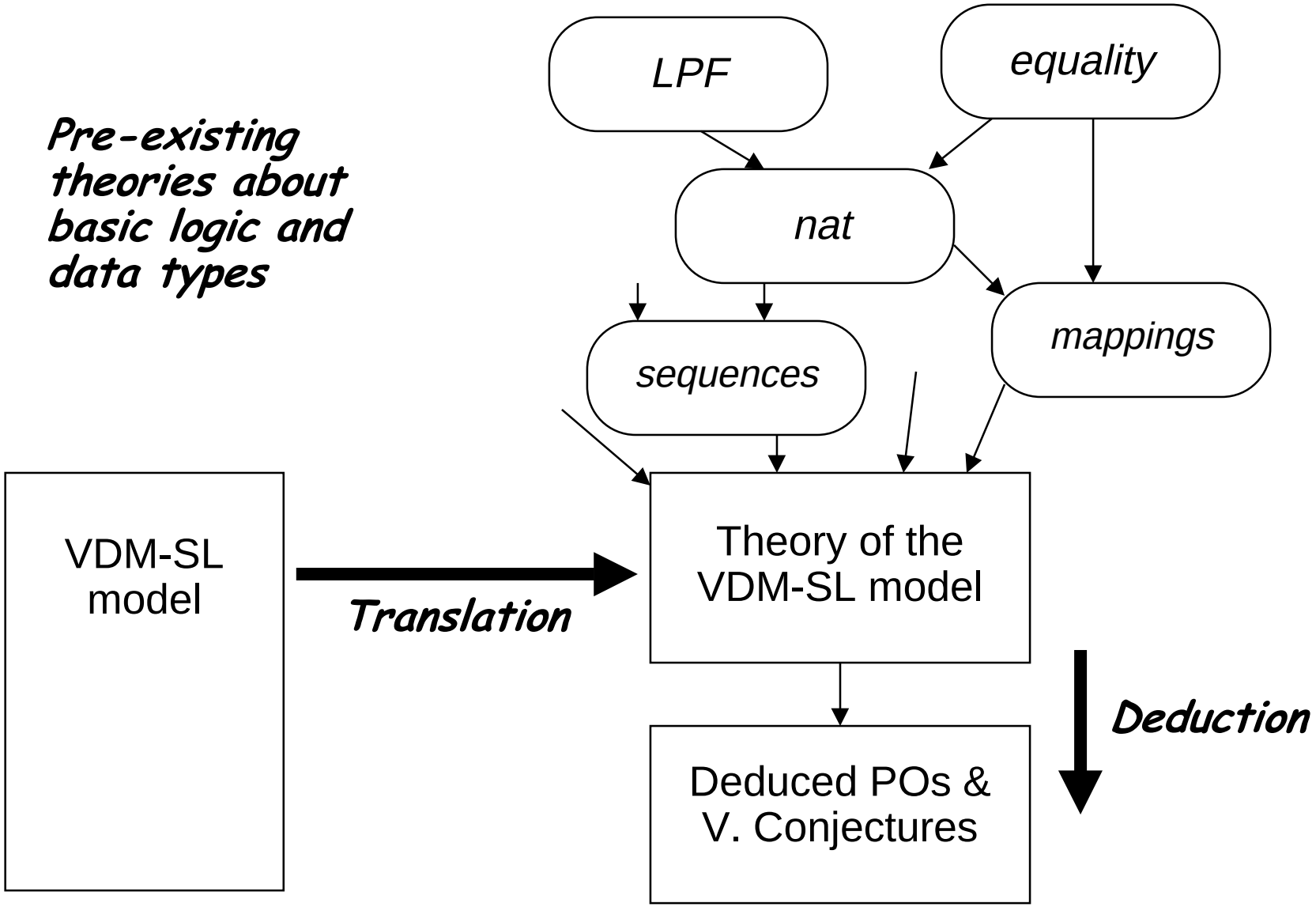
- **We examine the features needed to develop a proof theory for a modelling language:**
  - **The logical framework:** how will we represent conjectures, proofs and theorems, and collect them into useful theories?
  - **Logic and data:** how do we provide useful rules for the logic and data types in VDM?

- This has been done in practice for VDM - we will conclude by looking at what went right and what went wrong in this effort.

Proof in VDM

Logical Framework

Pre-existing theories about basic logic and data types



Proof in VDM

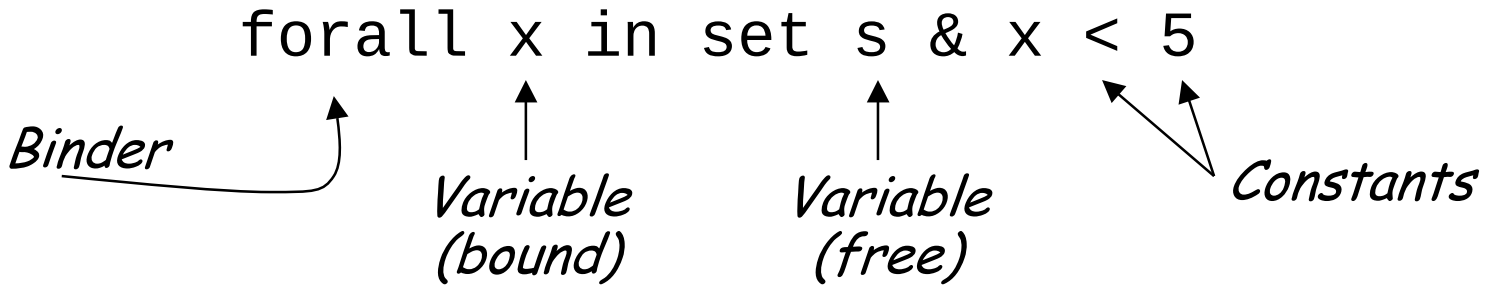
Logical Framework

In order to put together the base theories, and allow translation and deduction, we need:

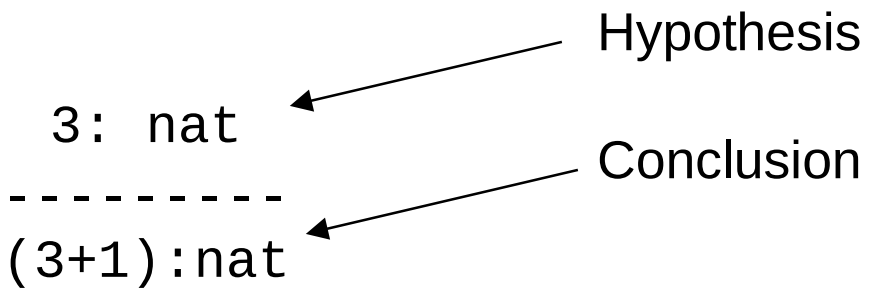
- a formal language for the expressions in the proof;
- definition of the form taken by inference rules;
- definition of a well-formed proof.

Expressions are made up from:

- **Variables,** which range over values, e.g. `v, x1, cid`
- **Constants** which represent value and type constructors, e.g. `{}` for the empty set, `[_]` for a singleton sequence, `^` for sequence concatenation, `0` for zero, `nat` for the natural numbers. Constants have an arity (the number of arguments they take, e.g. the arity of `{}` is 0, the arity of `^` is 2)
- **Binders,** which correspond to constructors that introduce and bind local variables, e.g. `forall`, `exists`, set comprehension.



Rules of Inference (Hilbert Style)



"If 3 is a natural number, then we may infer that 3+1 is a natural number.

This rule would be perfectly valid for any natural number, not just 3, so we use *metavariables* in inference rules to stand for arbitrary expressions:

```
  n: nat
---------
(n+1):nat
```

Metavariables can also take arguments:

```
a=b; P(a)
---------
   P(b)
```

Here P can take any expression with a free space, e.g. `forall x < _ & x*2 > 12`

Proof in VDM

Logical Framework

A Proof is ...

... an argument that some result follows from some hypotheses.

Consider a proof of:

```
  ns : seq of nat
------------------
[0]^ns : seq of nat
```

The proof starts as a “blank” sheet of paper with the hypotheses at the top and the conclusion at the bottom:

```
from ns : seq of nat




infer [0]^ns : seq of nat                    justify ???
```

Proof in VDM

Logical Framework

Looking in our theory of natural numbers, we find:

```
0-form  -----
        0:nat
```

No hypotheses, so this can be applied anywhere, and we can conclude:

```
from ns : seq of nat
1    0:nat                                  0-form



infer [0]^ns : seq of nat          justify ???
```

Proof in VDM

Logical Framework

Searching our theory of sequences, we find:

```
                    a:A
 singl-form  -------------
            [a]: seq of A
```

The hypothesis can be made to match line 1 if we let the metavariable a match to the expression `0` and the metavariable `A` match the expression `nat`.

```
from ns : seq of nat
1    0:nat                              0-form
2    [0]: seq of nat              singl-form(1)

infer [0]^ns : seq of nat        justify ???
```

Proof in VDM

Logical Framework

Having a final look in the theory of sequences, we find:

```
                s1: seq of A; s2: seq of A
^-form          ---------------------------
                      s1^s2: seq of A
```

The hypothesis of the rule can be made to match line 2 if we let the metavariable `s1` match to the expression `[0]`, `s2` matching `ns` and the metavariable `A` match the expression `nat`. The conclusion of the rule then matches the conclusion of the proof:

```
from ns : seq of nat
1    0:nat                              0-form
2    [0]: seq of nat                 singl-form(1)
infer [0]^ns : seq of nat             ^-form(2,h1)
```

Proof in VDM

Logical Framework

- The completion of the proof allows us to add the proven rule to the collection of theories (e.g. in the theory of sequences or in the theory of the specific model in which the sequence concatenation was used).
- *It was pretty time consuming wasn't it? We required a lot of apparatus to prove a simple rule.*
- *Proof about formal models is like that:*
  - *shallow - not requiring much insight*
  - *but high volume*

These are exactly the properties which make formal proof in validation an activity that could benefit from machine support.

Proof in VDM

Logical Framework

Natural Deduction style allows us to structure proofs better, e.g.

"In order to prove that P implies Q (P => Q), assume P and prove Q follows from it."

```
            P |- Q
deduction   ------
            P => Q
```

"If you know that either A or B holds and you want to prove C, show that C follows under asssumption of A and that C follows under assumption of B"

```
              A or B;
        A |- C; B |- C
cases   ---------------
              C
```

The "|-" (sequent) symbol captures the idea that you should perform a subproof.

Proof in VDM

Logical Framework

We can make deductions using sequent hypotheses and refer to subproofs in justifications. Here's an example.

```
                P or Q;  P|-R
one case ---------------------
                   R or Q
```

```
from P or Q;  P|-R
1    from  P
1.1        R              sequent h2 (1.h1)
     infer R or Q         or-I-right (1.1)
2    from  Q
     infer R or Q         or-I-left (2.h1)
infer R or Q              cases (h2,1,2)
```

```
                  A                          B
or-I-right --------    or-I-left --------
             A or B                     A or B
```

Proof in VDM

Logical Framework

Sometimes a constant can be defined in terms of other expressions, e.g.

```
e1 and e2 == not(not e1 or not e2)
```

We allow such definitions to be used in either direction (unfolding or folding, e.g.

```
5  not ( (A and B) or not C)
6  not ( not ( not A or not B ) or not C)   unfolding(5)
7  (not A or not B) and C                   folding(6)
```

Proof in VDM

The logic on which VDM is based is non-classical.

The Logic of Partial Functions (LPF) deals with expressions which include “misapplications” of partial operators, e.g. the expression

```
x=0 or x/x =1
```

is well-defined in LPF (and true for all numbers x).

In proof terms, this means that we do not have some rules that are present in classical propositional and predicate logic, e.g

```
Excluded_middle ----------
                e or not e
```

We distinguish defined/undefined expressions by δ:

```
δe == e or not e
```

Proof in VDM

The consequence in LPF is that many rules are qualified with a definedness hypothesis, e.g.

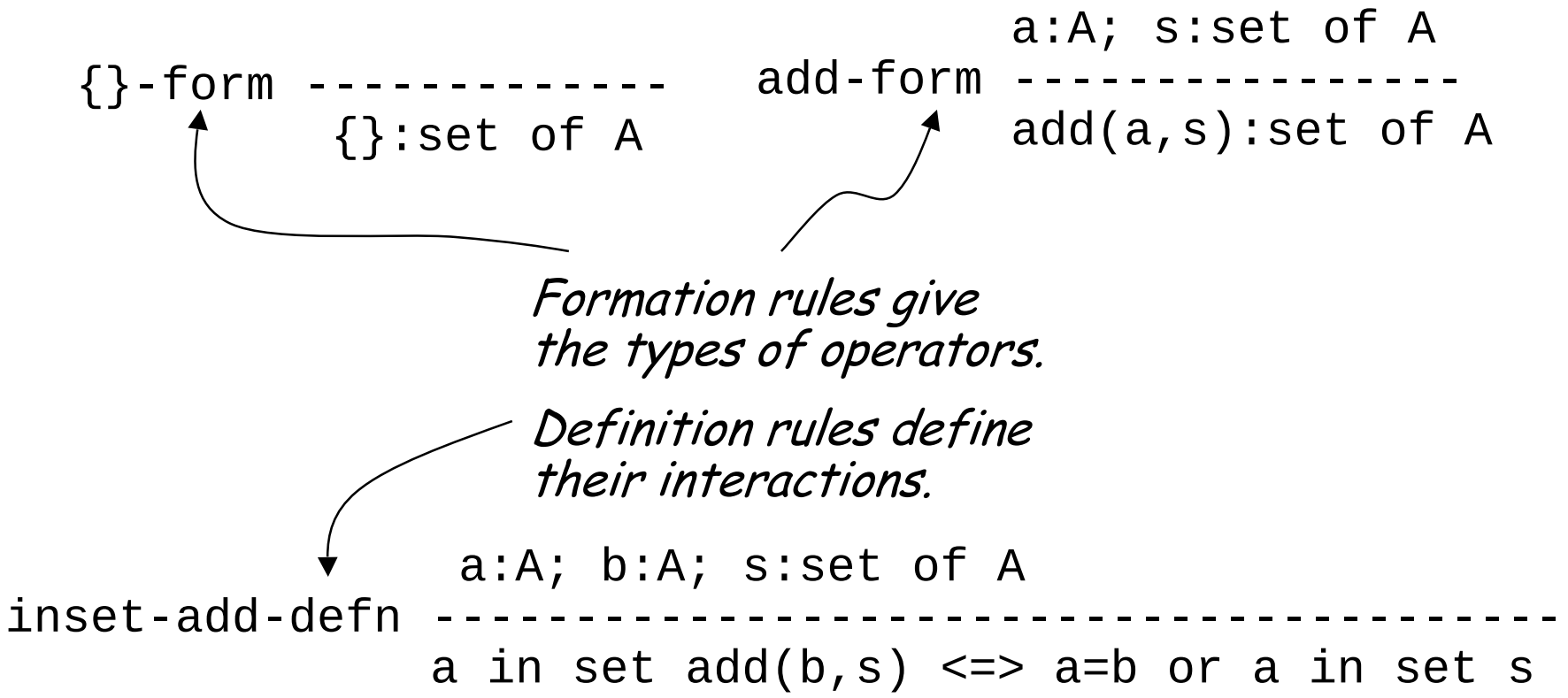
$$\text{Deduction\_Thm}\ \frac{\delta e1;\ \ e1 \vdash e2}{e1 \Rightarrow e2}$$

We must be careful when introducing binders. `exists` is treated like a large disjunction and `forall` like a large conjunction:

$$\delta\text{-exists-inherit}\ \frac{x{:}A \vdash \delta P(x)}{\delta(\text{exists } x{:}A \ \&\ P(x))}$$

Proof in VDM

For each of the major types and type constructors, we introduce a constant, e.g. `set of _` is treated as a constant of arity 1. We provide rules defining generators and basic elements, e.g.



Proof in VDM

The major collections have induction rules:

```
          s:set of A; P({});
          a:A, s':set of A, P(s'), a not in set s'
          |- P(add(a,s))
set-indn ----------------------------------------
          P(s)
```

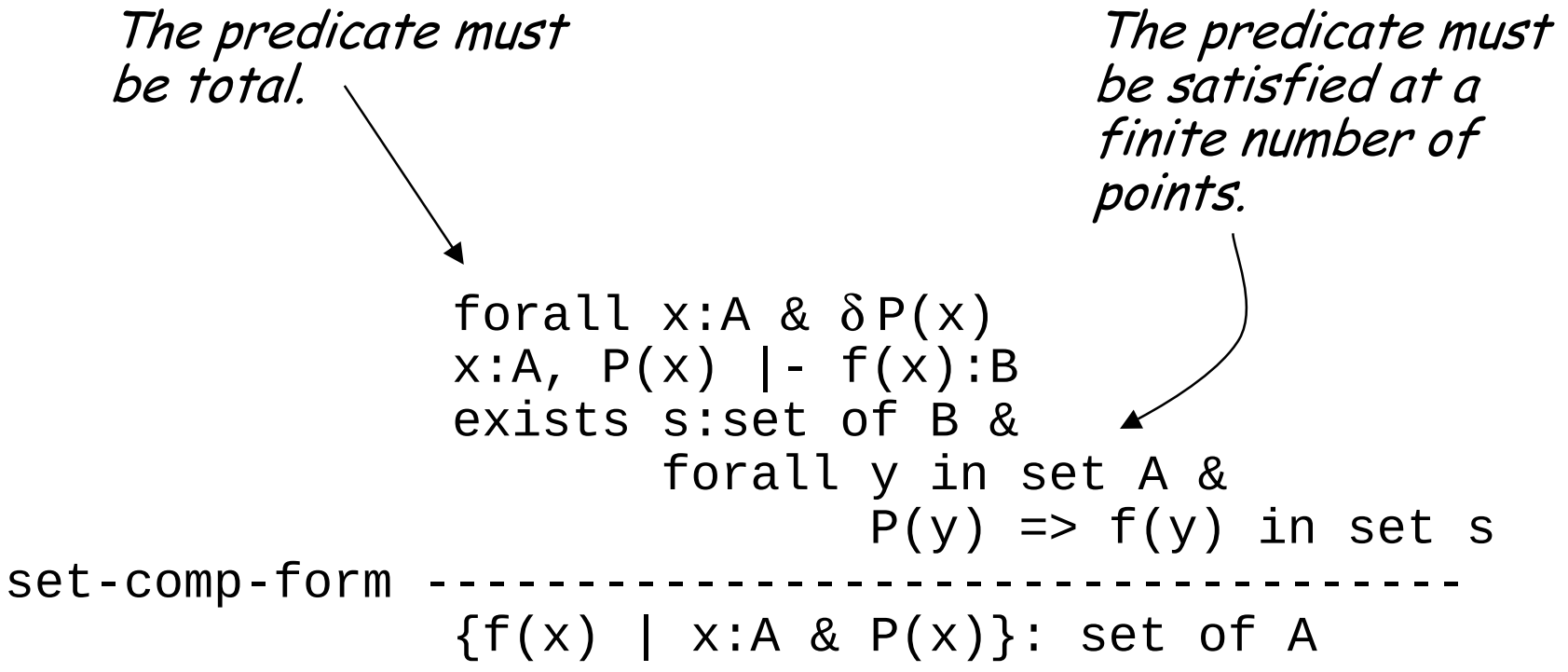
We treat the *type judgement* as a special kind of assertion, telling us that the expression is well formed.

The formation rules are therefore telling us about definedness of the expression formed in the conclusion.

This has consequences when we are defining comprehensions, which are treated as binders.

Proof in VDM

Comprehension formation rules must ensure that the comprehension is well formed:



Proof in VDM

Good & Bad Points

The axiomatisation developed for VDM-SL was based on typed predicate LPF with equality, to which we added theories for the types and type constructors of VDM.

✔ The axiomatisation is generally intuitive.

✔ It is defined in enough detail to provide for proof tools.

✔ It has been tested on case studies.

However, there are some interesting points at which it becomes hard to use:

? Arities that are not fixed, e.g. reasoning about arbitrary enumerations

[3, 5, 4, 4, 7 2]

?… and this extends to function expressions like cases

? Loose expressions, e.g.

```
let x:nat be s.t. (x**2 + 3*x + 4) in ...
```

… especially problematic in recursion.

Proof in VDM

Review

- Proof is a very powerful validation mechanism, but in order to use it, we must develop:
  - a logical framework of expressions, inference rules and a notion of proof, plus some equivalent of theory structuring.
  - theories of the base types and type constructors of the language.
- This has been done for VDM using a relatively simple logical frame.
- There are some advantages (ease of use and extension) but also some disadvantages, e.g. the overhead in handling definedness and finiteness.

We complete this course by taking a quick look at validation in practice: tool support and the lessons learned from the tracking manager case study in real life.

Supporting Validation

- **What about support tools?**
  - **Support for execution and testing**
  - **Support for proof**
- **What is it like in practice?**
  - **Lessons from the tracking manager study**

Supporting Validation — To Execute or not?

- **Only some specifications are executable.**

```
exists x: nat & P(x)

forall x: nat & P(x)

pre x > 0
post r*r = x
```

Type bindings are usually forbidden because we may have to search for an arbitrarily long time.

Implicit definitions can not be executed unless we give a putative result.

- **But many can be transformed into an executable form:**

```
exists x:nat & x < 20 and P(x)
exists x in set {1,…,20} & P(x)
```

Supporting Validation — To Execute or not?

- In practice, many implicit specifications are just “pseudo-implicit” with a post-condition of the form

```
post result = f(inputs)
```

- This leads us to believe that a functional style is not much of an overhead in many cases.

- However, this is a controversial point, and many models are better expressed as implicit operations with side effects on state variables than as referentially transparent functions.

- Some new technologies are applicable to the executability problem, especially constraint and logic programming.

Supporting Validation

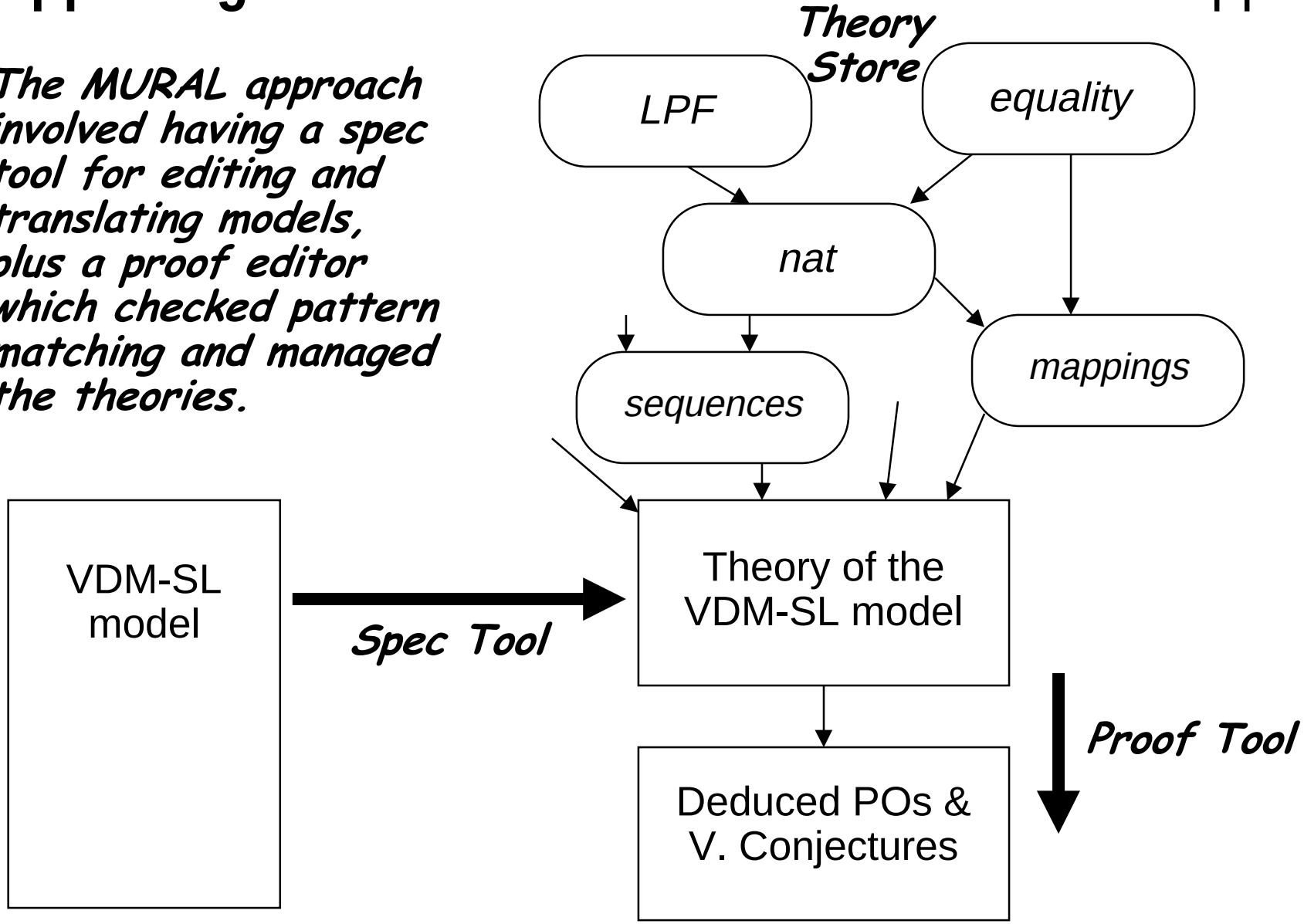
Testing Support

- **Executable definitions can be tested directly or through an accessible user interface.**
- **Tools support:**
  - batch mode testing
  - coverage analysis
  - white-box test generation
- Important not to confuse specification testing with testing of an implementation against a specification. Specification-based testing involves using the model in to generate black-box tests for the implementation. If the model is executable, it can be used as a test oracle.

Supporting Validation

Proof Support

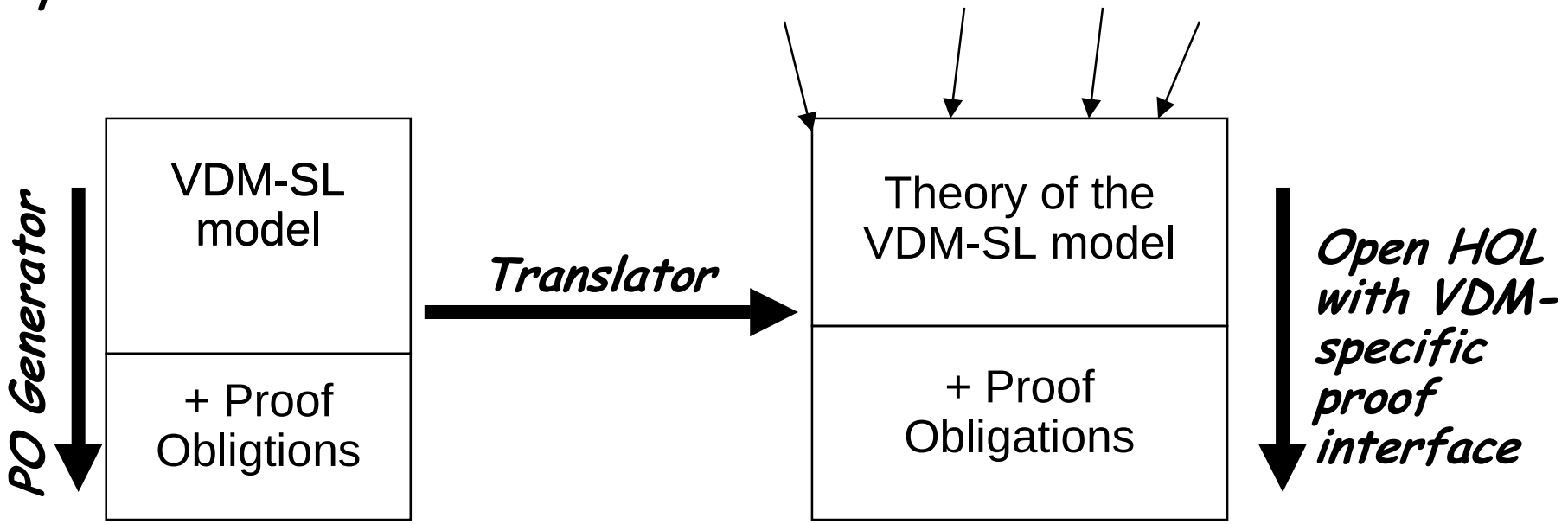
The MURAL approach involved having a spec tool for editing and translating models, plus a proof editor which checked pattern matching and managed the theories.



Supporting Validation

Proof Support

The PROSPER approach adds automation to the process.



The use of HOL adds automation but we have to translate HOL back to VDM when proofs fail. The use of "Off the shelf" theories gives us access to a wide range - but no LPF! We have to stuggle with a 2-valued interpretation of VDM!

Supporting Validation

- **Delimit system scope!** The proof of a property in a model is a proof about the model - not about the real world. Thus we can not “prove that a system is safe”. We can only prove that a model has some properties. These properties are chosen by experts in system safety. A model is only as good as the assumptions underlying it.

- **Use experience of proof during development of the model!** If a model is validated by proof, then some changes may be necessary to make the proof task more straightforward. Better to make these changes while the model is still in development!

- **Search for generic properties:** a modular specification with a generic component instantiated to the particular plan would have simplified the production of the safety case and separated validation issues of general concern from those specific to this plant.

- **Relationship to testing:** more thorough, but more demanding.

Supporting Validation

Review

- Choosing executability is a restrictive assumption.
- Some (many?) models may be transformed to executable versions with modest effort.
- Tool support for execution allows for systematic testing and animation, and the analysis of implementations using specification-based test generation.
- Proof support is available at different levels of automation. The major challenge is interacting with the user when proofs fail.
- The choice of validation strategy should have a significant effect on the way we design a system model (e.g. take advantage of potential re-use and genericity; factor validation techniques into the review cycle)

"Design for validation" applied just as much to the development of models as to the development of code!

Models of Reliable Multicast?

Properties:

(i) **Validity**: if a correct process *p* sends *m* then every correct process in *G* (including *p*) delivers *m*.

(ii) **Integrity**: for any *m*, every correct process in *G* delivers *m* at most once, and only if *m* was sent by some process in *G*.

> *Message delivered to a correct process is sent by some process in G; allows non-delivery of a message from a faulty process since the faulty sender can fail while sending the message.*

(iii) **Agreement**: if a correct *p* delivers *m* then every other correct *q* in *G* also delivers *m*.

> *Agreement property ensures that correct processes are unanimous in delivering or not delivering a message from a faulty process.*

A first attempt under some restrictive conditions ...

• no regard to process crashes or to views. The multicast itself takes place instantaneously.

• a collection of processes, indexed by identifiers.

• Characteristics of the system, such as the upper bound on transmission time, are also recorded in the state, although these are much less volatile than the processes.

```
state System of
      procs: map PId to Process
      delta: Time
end;

PId = token;
Time = nat;
```

Each process is modelled as a kind of history - a mapping from times to the sets of messages delivered at each time:

```
Process = map Time to set of Msg

Msg = token;
```

Thus, given a process, we can determine all the messages delivered by the process `p` up to a point in time `t` by the following expression:

```
dunion {p(t')|t' in set dom p & t' <= t}
```

```
RMCast1 (m:Msg, ts:Time)
ext wr procs: map PId to Process
    rd delta: Time
pre m not in set
    dunion {delivered_so_far(p,ts)|p in set dom procs}
post dom procs = dom procs~ and
     forall pid in set dom procs &
        exists tr:Time & ts < tr and tr <= ts+delta &
           let e = if tr in set dom procs(pid)
                   then procs(pid)(tr)
                   else {}
           in
           procs(pid) = procs~(pid) ++
                        {tr |-> e union {m}}
```

An executable version ...

```
RMCast1x: System * Msg * Time -> System
RMCast1x(mk_System(procs,delta),m,ts) ==

mk_System(
   {pid |-> let tr in set {ts,…,ts+delta} in
            let e = if tr in set dom procs(pid)
                    then procs(pid)(tr)
                    else {}
            in
            procs(pid) ++ {tr |-> e union {m}}
   | pid in set dom procs
   },
   delta)

pre m not in set
    dunion {delivered_so_far(p,ts)|p in set dom procs}
```