Validating Formal Models

- The Idea of Validation
- Internal Consistency
  - Levels of type checking
  - Proof Obligations
- Animation
- Systematic testing
- Proof
- Choosing a Validation Technique

The Idea of Validation

How confident can you be that a formal model accurately describes the system that the customer wanted?

• Requirements are often incomplete and ambiguous: modellers have to resolve these in unambiguous models.

• Requirements often state the client's intention incorrectly.

Validation is the process of increasing confidence that a model is an accurate representation of the system under consideration. Two aspects of this:

1. Checking internal consistency of a model.

2. Checking that the model describes the required behaviour of the system being modelled.

The Idea of Validation

If a modelling language is formal then it must have:

• a formal syntax: rules restricting the symbols in the language and saying where they can be used.

• a formal semantics: rules for determining the meaning of a model written in accordance with the formal syntax.

If the syntax is formal, then we can check it with the aid of a tool (c.f. *syntax checker* in a programming language compiler).

If the semantics is formal, then we can check at least some aspects with the aid of a tool (c.f. *type checker* in a programming language compiler).

But we can't check everything!

The Idea of Validation

The other aspect of validation is checking the accuracy with which the model records the desired system behaviour.

We will look at three approaches:

- **Animating** the model - *works well with clients unfamiliar with the modelling notation but requires a good interface.*
- **Testing** the model - *can assess coverage but limited to the quality of the tests and the model must be executable.*
- **Proving** properties of the model - *provides excellent coverage and does not require executability, but not well supported by tools.*

Internal consistency: type checking

A simple form of internal consistency checking is type checking. Consider a type checking tool working on the following extracts from function definitions:

```
Student :: …

Sid = token

Dbase = map Sid to Student

newStudent1: Sid * Student * Dbase -> Dbase
newStudent1(sid,s,db) == db ++ { sid |-> s }


newStudent2: Sid * Student * Dbase -> Dbase
newStudent2(sid,s,db) == db ^ { sid |-> s }


newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
```

Internal consistency: type checking

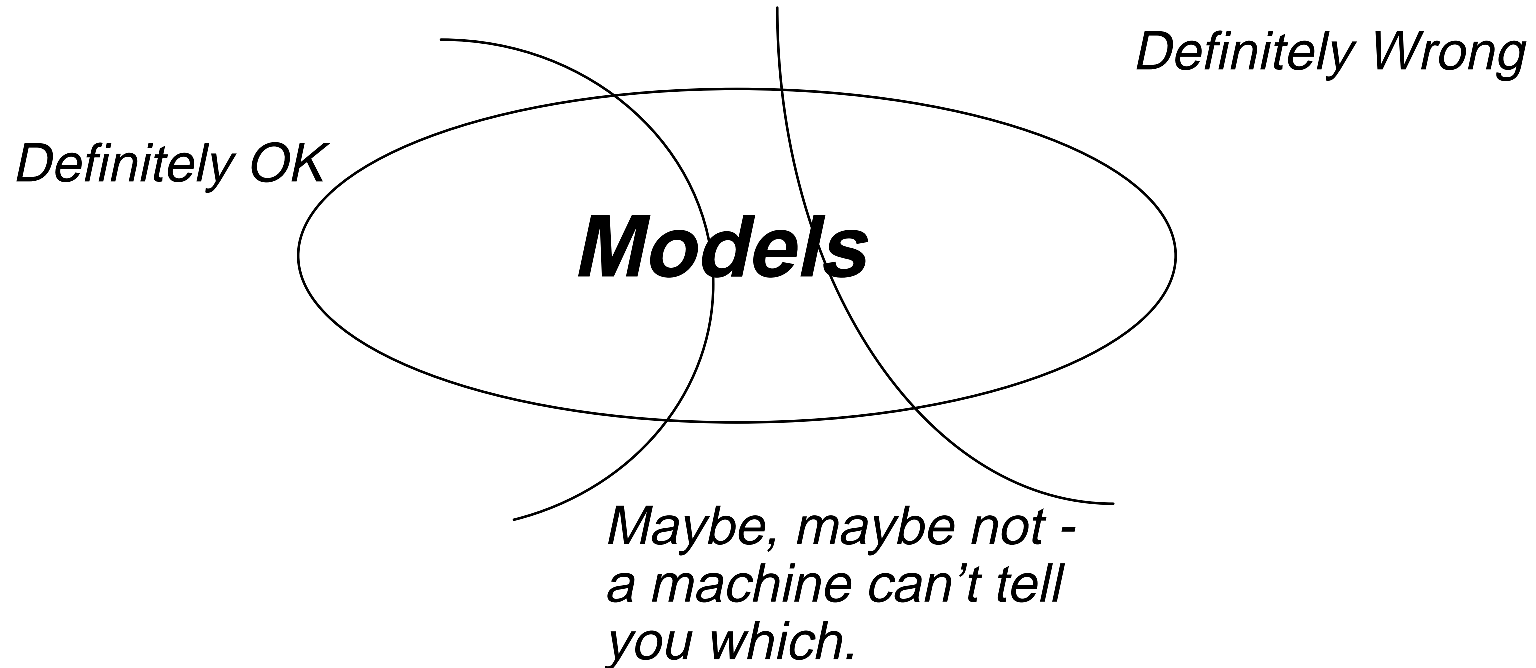
```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid in set dom db
```

We know that this is OK, but could a machine work it out? What about …

```
newStudent3: Sid * Student * Dbase -> Dbase
newStudent3(sid,s,db) == db munion { sid |-> s }
pre sid in set {s1 | s1 : Sid &
    exists y in set rng db & db(s1) = y}
```

We can't provide a completely general tool that can automatically check that all uses of operators are properly protected (programming languages have the same problem - you can't produce a general tool that can automatically statically check whether division by zero will occur unless the language is very restricted and inexpressive).

Internal consistency: type checking



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.

Internal consistency: proof obligations

If a check cannot be performed automatically, the techniques of mathematical proof are required to complete it.

The collection of all checks to be performed on a VDM model are called *proof obligations.* A proof obligation is a logical expression which must be shown to hold before a VDM-SL model can be regarded as formally internally consistent.

We look at three proof obligations on VDM-SL models:

- **Domain Checking**
- **Satisfiability of explicit definitions**
- **Satisfiability of implicit definitions**

Proof Obligations Domain Checking

Using a partial operator outside its domain of definition is usually an error on the part of the modeller. Two kinds of construct are impossible to check automatically:

- applying a function that has a pre-condition; and
- applying a partial operator.

Some definitions:

```
f:T1 * T2 * … * Tn -> R
f(a1,…,an) == ...
pre ...
```

We can refer to the precondition of `f` as a Boolean function with the following signature:

```
pre_f:T1 * T2 * … * Tn -> Bool
```

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.

Example:

```
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)
```

Proof obligation for domain checking:

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

Proof Obligations

Domain Checking for Partial Operators

Each application of a partial operator must be protected. For example, consider:

```
Introduce: Tracker * ContainerId * real * Material -> Tracker
Introduce(trk,cid,quan,mat)==
   mk_Tracker(trk.containers munion
               {cid |-> mk_Container(quan,mat)},
               trk.phases)
pre cid not in set dom trk.containers
```

The obligation is:

```
forall trk:Tracker, cid:ContainerId, quan:real, mat:Material
&
pre_Introduce(trk,cid,quan,mat) =>
  compatible(trk.containers,{cid |-> mk_Container(quan,mat)})
```

Proof Obligations

Domain Checking

Partial operators can be protected by pre-conditions (as in the Permission example) or by including an explicit check in the body of the function, e.g.

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest) ==
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity and
  containers(cid).material in set
     phases(dest).expected_materials
```

Proof obligation

```
forall mk_Tracker(containers,phases):Tracker,
       cid:ContainerId, dest:PhaseId &
       (cid in set dom containers and
        dest in set dom phases) => dest in set dom phases
```

Proof Obligations

Exercise: What is the proof obligation generated by the **highlighted** expression below?

```
Move: Tracker * ContainerId * PhaseId * PhaseId -> Tracker
Move(trk,cid,ptoid,pfromid) ==
  let pha = mk_Phase(trk.phases(ptoid).contents union {cid},
                     trk.phases(ptoid).capacity)
  in
  mk_Tracker(trk.containers,
             Remove(trk,cid,pfromid).phases ++ {ptoid |-> pha})
pre Permission(trk,cid,ptoid) and pre_Remove(trk,cid,pfromid)
```

(Highlighted expression: **trk.phases(ptoid)**)

Proof Obligations

It can be difficult to decide what to include in a pre-condition.

- Some conditions are determined by the requirements.
- Many conditions are there to guard applications of partial operators and functions.

When you write a function definition, read through it systematically, highlighting each application of a partial operator, and ensure that you have guarded against misapplication of that operator by adding a suitable conjunct to the precondition.

Proof Obligations

Satisfiability

An explicit function *without* a pre-condition defined

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

is said to be ***satisfiable*** if, for all inputs, the result defined by the function body is of the correct type. Formally,

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

An explicit function *with* a pre-condition:

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

is said to be ***satisfiable*** 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
```

Proof Obligations

For example, consider

```
Introduce: Tracker * ContainerId * real * Material ->
           Tracker
Introduce(trk,cid,quan,mat)==
   mk_Tracker(trk.containers munion
               {cid |-> mk_Container(quan,mat)},
               trk.phases)
pre cid not in set dom trk.containers
```

The satisfiability proof obligation is:

Proof Obligations

Most of the work in showing satisfiability comes in showing, not that the result returned belongs to the correct general type, but that it respects the invariant on that type.

Satisfiability of implicit function definitions

A function f defined implicitly as follows

```
f(a1:T1,…,an:Tn) r:R
pre ...
post ...
```

is said to be ***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(p1,…,pn,x)
```

Proof Obligations

Example:

```
Find(trk:Tracker,cid:ContainerId) p:(PhaseId|<NotAllocated>)
pre cid in set dom trk.containers
post if exists pid in set dom trk.phases &
               cid in set trk.phases(pid).contents
     then p in set dom trk.phases and
          cid in set trk.phases(p).contents
     else p = <NotAllocated>
```

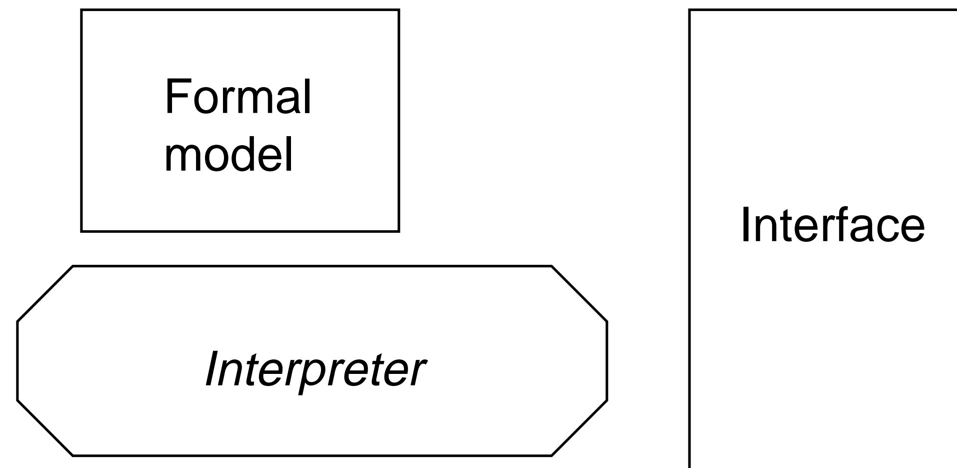
The satisfiability proof obligation is as follows:

Animation

The goal of validation is to increase confidence that a model accurately reflects the customer's intentions.

However, customers rarely understand the modelling language used, whether it is formal or not.

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.



Animation

The interface functions (in C++) have to be made known to the VDM-SL layer, This can be done in a dynamic link module which also provides a file name reference to the compiled C++ code. For example:

```
SelectFun: () -> <Intro>|<Perm>|<Move>|<Rem>|<Del>|<Stop>
```

indicates the button selected by the user

```
GetMove: () -> ContainerId * PhaseId * PhaseId
```

handle a pop-up for assigning a container to a phase

```
ShowErr: seq of char -> bool
```

show an error display

```
ShowTracker: Tracker -> bool
```

update the tracker overview

Animation

At the VDM level, a top-level “Main” function handles the selection and calls lower-level functions to handle the input:

```
Main: Tracker -> Tracker
Main(trk) ==
  cases SelectFun():
    <Intro> -> EvalIntro(trk),
    <Perm>  -> EvalPerm(trk),
    <Move>  -> EvalMove(trk),
    <Rem>   -> EvalRem(trk),
    <Del>   -> EvalDel(trk)
    <Quit>  -> trk
  end

EvalMove: Tracker -> Tracker
EvalMove(trk) ==
  let mk_(cid,ptoid,pfromid) = GetMove() in
    if pre_Move(trk,cid,ptoid,pfromid)
    then if ShowTracker(Move(trk,cid,ptoid,pfromid)
         then Main(trk)
         else trk
    else if ShowErr(“Permission for assignment not granted”)
         then Main(trk)
         else trk
```

Systematic Testing

The level of confidence gained through an animation is only as good as the particular choice of scenarios executed by the user through the interface.

More systematic testing is also possible:

define a collection of test cases

execute each test case on the formal model

compare with expectation

Test cases can be generated by hand or automatically. Automatic generation can however produce a vast number of individual test cases.

Techniques for test generation in functional programs carry over to formal models.

Systematic Testing

Executing the test:

```
Permission(mk_Tracker({|->},{|->}),mk_token(1),mk_token(2))
```

yields false. We can also tell which parts of the permission function have been exercised ("covered") by the test:

```
Permission: Tracker * ContainerId * PhaseId -> bool
Permission(mk_Tracker(containers,phases), cid, dest) ==
  cid in set dom containers and
  dest in set dom phases and
  card phases(dest).contents < phases(dest).capacity and
  containers(cid).material in set
     phases(dest).expected_materials
```

It is possible to have a tool highlight parts of the model that are not exercised by a test and use this information to devise other tests.

Validation by Proof

Systematic testing and animation are only as good as the tests and scenarios used. *Proof* allows the modeller to assess the behaviour of a the model for whole classes of inputs in one analysis.

I

n order to prove a property of a model, the property has to be formulated as a logical expression (like a proof obligation). A logical expression describing a property which is expected to hold in a model is called a *validation conjecture*.

Proofs can be time-consuming. Machine support is much more limited: it is not possible to build a machine that can automatically construct proofs of conjectures in general, but it is possible to build a tool that can check a proof once the proof itself is constructed. Considerable skill is required to construct a proof - but a successful proof gives high assurance of the truth of the conjecture about the model.

Validation by Proof

Levels of Proof:

• **“Textbook”:** argument in natural language supported by formulae. Justifications in the steps of the reasoning appeal to human insight (“Clearly …”, “By the properties of prime numbers …” etc.). Easiest style to read, but can only be checked by humans.

• **Formal:** at the other extreme. Highly structured sequences of formulae. Each step in the reasoning is justified by appealing 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.

Choosing a validation technique

- **Level of Confidence Required:**

- **Costs:**

Review

Validation: the process of increasing confidence that a model accurately reflects the client requirements.

- Internal consistency:
  - domain checking: partial ops and functions with precondition
  - satisfiability of explicit and implicit function
- Checking accuracy:
  - animation
  - testing
  - proof