

Constructing a Model

A guided tour through a model in VDM-SL

Deriving a Model

The Chemical Plant Alarm System:

- Requirements
- Data Types and invariants
- Functions and pre-conditions

The example is derived from a subcomponent of a large alarm and callout system developed by IFAD, a Danish high-technology firm for the local telephone company Tele Danmark Process.

The contents of a model in VDM-SL

Type Definitions, e.g.

```
Altitude = real
inv alt == alt >= 0

Position :: lat   : Latitude
           long  : Longitude
           alt   : Altitude
```

Function definitions, e.g.

```
Move: Id * Position * ATCSysyem -> ATCSysyem
Move(id, pos, sys) == ... expression ...
pre ... expression ...
```

The contents of a model in VDM-SL

Data Types built from basic types (`int`, `real`, `char`, `bool` etc.) using type constructors (`sets`, `sequences`, `mappings`, `records`).

Newly constructed types can be named and used throughout the model.

A **data type invariant** is a Boolean expression that is used to restrict a data type to contain only those values that satisfy the expression.

Functions define the functionality of the system. Functions are referentially transparent - no side-effects and no global variables. In cases where it is intuitive to have global variables, a different **operational style** of modelling is used.

A **pre-condition** is a Boolean expression over the input variables that is used to record restrictions assumed to hold on the inputs.

The contents of a model in VDM-SL

Data abstraction is provided by the unconstrained nature of the data types in VDM-SL. Sets, sequences and mappings, although finite, are unbounded.

Function abstraction, when required, is provided by **implicit specification**.

```
SquareRoot(x:nat)r:real
```

```
pre  x >= 0
```

```
post r*r = x
```

Post-conditions are Boolean expressions relating inputs and outputs. Post-conditions are used when we do not wish to explicitly define which output is to be returned, or where the explicit definition would be too concrete.

Deriving a Formal Model from Scratch

- No right or wrong way to construct a formal model from a requirements description.
- Always begin by considering a model's **purpose**, as this guides abstraction decisions during development.
- Following steps:
 1. *Read the requirements.*
 2. *Extract a list of possible data types (often from nouns) and functions (often from verbs/actions).*
 3. *Sketch out representations for the data types.*
 4. *Sketch out signatures for the functions.*
 5. *Complete type definitions by determining invariants.*
 6. *Complete the function definitions, modifying data type definitions if required.*
 7. *Review the requirements, noting how each clause has been treated in the model.*

Requirements for the Alarm Example

A chemical plant has monitors which can raise alarms in response to conditions in the plant. When an alarm is raised, an expert must be called to the scene. Experts have different qualifications for coping with different kinds of alarm.

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

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

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

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

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

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

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

R8: It shall be possible to assess the number of experts on duty at a given period

Purpose of the model ...

To clarify the rules governing the duty rota and the calling out of experts in response to alarms.

Aside: We often find in professional practice that the purpose for which a model is to be developed is only rarely made clear. Yet it is this purpose which should govern the choice of abstractions made in the development of the model and hence the success, ease of use etc. of the model itself.

Possible data types and functions

Types

Functions

Sketching type representations

Enumerated types

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.

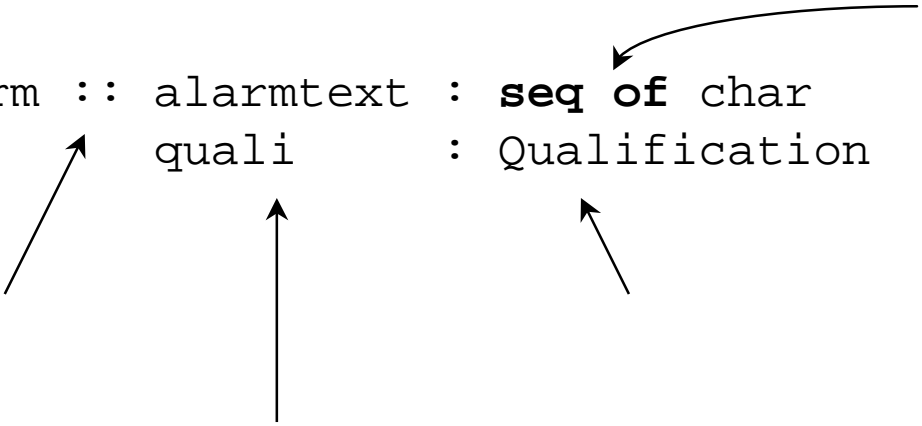
Sketching type representations

Record types

*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
```



Sketching type representations

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*

Sketching type representations

Set types

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 model 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

Token types

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

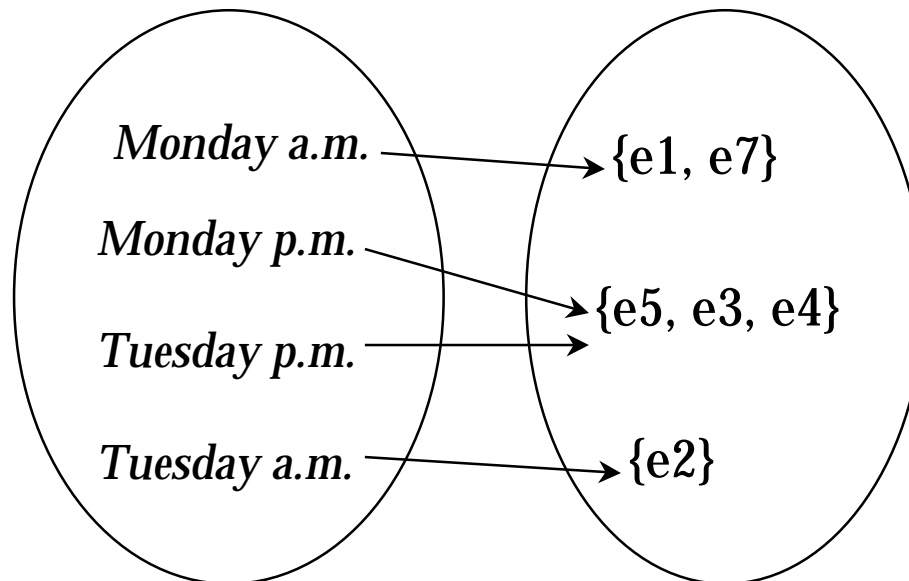
Mapping types

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)



Sketching type representations

The whole plant

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
```

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
```

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

Data type invariants

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

Data type invariants

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(per, pl) ==
```

A body

```
card pl.sch(per)
```

A pre-condition (optional)

```
pre per 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) ==
    {per | per in set dom pl.sch &
          ex in set pl.sch(per)}
```

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, -)) ==
```


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,per,pl) == ???
```

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

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



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
```

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”.

Summary

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: