2
Constructing a Model

Aims

In this chapter we aim to provide an overview of the main components of a VDM-SL model, and some insight into the process by which models are constructed. This is done by developing a model of a small alarm system from scratch.

2.1 Introduction

This chapter tells the story of how a simple formal model is developed from informally expressed requirements. The model under consideration, the call-out mechanism for a chemical plant alarm system, illustrates most of the features of VDM-SL covered in this book. Although this is a great deal of material for a single chapter, there is no need to be able to understand all the details of the language as these will be covered at a slower pace in later chapters. The intention is to introduce, albeit superficially, the concepts that will be covered in depth later, and to provide some initial guidance on how to start developing formal models using VDM-SL.

> Constructing a model is never as straightforward as it appears in the textbooks! We would not want to give the impression that a model is reached by a sequence of steps which run smoothly from start to finish: developers of models often run up against problems and have to scrap previous attempts. Experience can be the best guide here: where our work on building formal models has anything to offer the reader, we present it in distinguished text like this.

2.2 Requirements for an alarm system

This section contains the requirements for a simple alarm system for a chemical plant. The example was inspired by a subcomponent of a large alarm system developed by IFAD, a Danish high-technology company, for the local telephone company, Tele Danmark Process.

A chemical plant is equipped with a number of sensors which are able to 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. The individual requirements are labelled for reference.

R1 A computer-based system is to be developed to manage the alarms of this plant.

R2 Four kinds of qualification are needed to cope with the alarms. These are electrical, mechanical, biological and chemical.

R3 There must be experts on duty during all periods which have been allocated in the system.

R4 Each expert can have a list of qualifications.

R5 Each alarm reported to the system has a qualification associated with it along with a description of the alarm which can be understood by the expert.

R6 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.

R7 The experts should be able to use the system database to check when they will be on duty.

R8 It must be possible to assess the number of experts on duty.

2.3 Constructing a model from scratch

There is no right or wrong way to compose a model from a requirements description. However, faced with a collection of requirements, it is useful to have a rough procedure to follow which at least helps in getting a first attempt down on paper. This section presents such a procedure. However, before beginning the process of developing a model, it is vital to consider the model's *purpose*. A model is normally developed so that some analysis can be performed on it. For example, in the development of a single system, a model might be constructed in order to help determine the resource requirements for the system; to clarify the rules under which the system must operate; or to assess security or safety. The purpose for which a model is constructed determines the model's abstraction: which details will be represented and which will be ignored because they are not relevant to the analysis. When we introduce the examples in each core chapter of this book, we will try to make the purpose of the model clear. In this chapter, the purpose of the model is to clarify the rules governing the duty roster and calling out of experts to deal with alarms.

After establishing the purpose of the model, the following list of steps can be a helpful guide to its construction:

1. Read the requirements.
2. Extract a list of possible data types (often from nouns) and functions (often from actions).
3. Sketch out representations for the types.
4. Sketch out signatures for the functions.
5. Complete the type definitions by determining any invariant properties from the requirements and formalise these.
6. Complete the function definitions, modifying the type definitions if necessary.
7. Review the requirements, noting how each clause has been treated in the model.

This procedure will be followed in our analysis of the chemical plant alarm system. If any discrepancies are discovered during this process they will be noted down when they are discovered.

2.4 Reading the requirements

In constructing a model of a computing system it is necessary to find representations for the data and computations involved. While reading the requirements, it is worth noting the nouns and actions which typically may correspond to types and functions in the model.

Statements of requirements, like the example above, can contain undesirable features such as noise, silence, ambiguity, wishful thinking and misstated intention. Look out for these when developing the formal model: when imprecision is encountered, it is necessary to resolve the issue and record the resolution in order to keep track of all the decisions made during the analysis.

Examining each of the requirements listed above yields possible types and functions:

R1 This concerns the entire system and suggests that a type could be introduced to represent the various alarms which can arise. Call this type `Alarm`. The requirement also suggests that a type could be introduced to represent the overall condition of the plant. This type will be called `Plant`.

R2 This requirement mentions a collection of kinds of qualification, so a type `Qualification` is suggested. `Alarm`s are also mentioned again.

R3 This requirement mentions experts and periods of duty, suggesting types `Period` and `Expert`.

R4 Here `Qualification` and `Expert` are mentioned again.

R5 Descriptions appear to be associated with alarms, so this suggests a type called `Description`. The types `Alarm`, `Qualification` and `Expert` are mentioned again.

R6 This requirement refers to the action of finding an expert with the right qualification to handle an alarm. This suggests that a function is required to find an expert to page. Let this function be called `ExpertToPage`. The types `Alarm`, `Expert` and `Qualification` are all mentioned again.

R7 This requirement refers to the action of checking when experts are on duty, suggesting that a function is needed to perform this check. Call the function `ExpertIsOnDuty`. In addition, the type `Expert` is mentioned again.

R8 A function to return the `NumberOfExperts` on duty is suggested, and the type `Expert` is mentioned again.

2.5 Extracting possible types and functions

The systematic read through of the requirements above suggests the following list of possible types and functions:

Types	**Functions**
`Plant`	`ExpertToPage`
`Qualification`	`ExpertIsOnDuty`
`Alarm`	`NumberOfExperts`
`Period`	
`Expert`	
`Description`	

It would be reasonable to expect that functions would be needed to add and remove alarms and experts from the system, but these are not mentioned in the simplified requirements presented here. For the rest of this chapter, the list of types and functions above will suffice.

2.6 Sketching type representations

How would one go about writing the type definitions for the types identified above? Requirement *R2* states the four kinds of qualifications needed in this system. This suggests a type `Qualification`. Its definition has the following form, where "`???`" stands for some representation for the type:

```
Qualification = ???
```

The qualifications are all mentioned by name and it is simply necessary to be able to distinguish between them: the details of their representations (as strings of characters, numbers etc.) are not significant factors at this stage of development. It is therefore possible to abstract away from the particular representation of these four values (as strings, numbers etc.), but we would like to provide a name for each of them. In VDM-SL this is done by means of an enumerated type, similar to the enumerated types found in programming languages. In this case the type `Qualification` could be defined as follows:

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

This defines a type consisting of four values corresponding to the four kinds of qualification required in this system: `<Elec>`, `<Mech>`, `<Bio>` or `<Chem>`. The "<" and ">" brackets are used to indicate a special type (called a *quote type*) containing just the named value. For example, the type `<Elec>` contains just the single value `<Elec>`. The "`|`" symbol forms a *union type* from the quote types and thus constructs the new type with just four values in it. Union and quote types are described in more detail when the basic type constructors of VDM-SL are introduced (Section 5.3.1).

In Requirement *R2*, `Alarm` was identified as a potential type. Requirement *R5* states that an alarm "has a qualification associated with it". In this case, we will assume that exactly one qualification is associated with each alarm and record this assumption in the model. However, the phrase could be understood to mean that each alarm has *at least one* qualification associated with it. This is an example where the textual description might be interpreted in different ways by different readers. An advantage of building a model at this stage is that the assumption is explicitly recorded and, if necessary, can be corrected when the model is analysed and reviewed. In this case we will assume that an alarm has exactly one qualification and one description.

Requirement *R5* emphasises that the description should be made so that it can be understood by the experts. This could indicate that the description is a piece of text (a sequence of characters) which is to be sent to the pager of the selected expert. Note that there is no guarantee that the sequence of characters present here actually can be understood by an expert. Thus, at a later stage of development it would be necessary to review the "alarm text" to check that they are actually clear. The notion of clarity is subjective and beyond the scope of a model in VDM-SL.

In VDM-SL, a *record type*, similar to the record or struct in a programming language, is used to model values made up of several components. The definition of an alarm is

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

This states that an alarm is composed of two fields called `alarmtext` and `quali`. The type **`char`** used in the definition is a basic type in VDM-SL that represents characters. The keywords **`seq of`** produce a sequence type so that the `alarmtext` is represented by a sequence of characters.

Suppose a value `a` belongs to the type `Alarm`. The components can be selected from `a` using a dot notation. For example, the description of an alarm `a` is written `a.alarmtext`. Thus if `a` is of type `Alarm` then `a.alarmtext` is a sequence of characters representing the text for the alarm.

Having modelled alarms, we now consider the experts. Requirement *R4* states that an expert can have *a list of* qualifications. Whenever we encounter a requirement stating that a list should be used, we must ask ourselves how the elements of the list should be ordered. In this example, the order cannot be deduced from the requirements, so we must note this as a point to be resolved and record in the model any assumption made. In this case we assume that the order is not significant and use an unordered collection of qualifications in the form of a *set*.

In order to distinguish two experts from each other, some kind of unique identification of experts is needed. Thus the type of an `Expert` can be modelled using another record type:

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

The representation of `ExpertId` is important in the final implementation, but in this model none of the functions need be concerned with its final representation. All that is required at this level of abstraction is for the identifiers to be values that can be compared for equality. In VDM-SL, the special type representation called **`token`** is used to indicate that we are not concerned with the detailed representation of values, and that we only require a type which consists of some collection of values. This is usually the case when the functions in the model do not need to read or modify the information stored in an element of the type. For example, if we were to include a function which modifies `ExpertId`s, then we would need to model the identifiers in detail in order to be able to describe the modification. No such function is required in this model, so the representation of expert identifiers is immaterial and the completed type definition in VDM-SL is

```
ExpertId = token
```

Notice that no kind of unique identification of experts was mentioned explicitly in the requirements although requirement *R6* indicated the need for it. This is an example where an extra clause like *All experts must be uniquely identified* could be added to the requirements.

Now let us turn to the modelling of `Period` mentioned in *R3*. It is not important whether the periods are represented as character strings or reference numbers with some fixed syntax, and we cannot deduce from the requirements whether a period is composed of entire working days, hours or some other time period. In the final implementation this needs to be clarified. However, none of the functions we develop in the model here require access to the representation of `Period`, so this is not a core part of the model. It is enough to know that a period has some identifier and that it is possible to compare periods to see whether they are the same or not. Abstracting from this detail, the **`token`** type can be used once again:

```
Period = token
```

Requirement *R3* suggests that there must be a schedule relating periods to the experts who are on duty in each period. We therefore identify `Schedule` as a possible type. Note that we did not manage to identify `Schedule` as part of the initial "nouns and actions" analysis of the requirements. The "nouns and actions" approach is certainly not foolproof – it may fail to come up with types which are only implied by the text. Moreover, some types corresponding to nouns may not be needed.

For each allocated period, the schedule must record the collection of experts on duty during that period. The schedule can therefore be thought of as a *mapping* from periods to collections of experts. Given a period, we can look up the collection of experts in the mapping. Should the collections of experts be sets or sequences? We resolve this by assuming that experts are not recorded in any particular order, enabling us to use sets. If the requirements did suggest that there was a significant ordering among the experts, such as the distance that they live away from the chemical plant, we could use sequences.

A schedule is therefore represented as a mapping from periods to sets of experts. An example of such a schedule is shown schematically in Figure 2.1. The corresponding type definition in VDM-SL is as follows:

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

Figure 2.1 *A possible schedule.*

The **map** _ **to** _ notation represents mappings from the first type (here the type `Period`) to the second type (here **set of** `Expert`). Consider a schedule `sch`. The collection of periods in `sch` is called the *domain* of `sch`: in Figure 2.1 the domain has four elements. The sets of experts in the mapping are collectively called the *range* of `sch`: the range of the schedule in Figure 2.1 has three elements. If a period is in the domain, then it is part of the "allocated" schedule and points across the mapping to a set of experts on duty in this period. If a period `peri` is in the domain of `sch`, we write `sch(peri)` to represent the range element (the set of experts) in `sch` to which `peri` points. If a period `peri` has not yet been planned and assigned a collection of experts, it is not in the domain of the mapping `sch` and so `sch(peri)` is not defined.

Finally, consider Requirement *R1*. This combines the type definitions we have made into a top-level type definition modelling the entire `Plant`. This must be a record with the schedule for experts that are on duty and a collection of alarms which can be activated. We use a record type definition again:

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

The model so far

At this point, representations have been proposed for the main types which arose from a reading of the requirements. A substantial part of the VDM-SL notation has been introduced, so it is worth taking stock. Each type has been given a *representation* in terms of other types. Some of the representations have been basic types which are already part of the VDM-SL language, e.g. **token** and **char**. Others have been constructed from basic types and values, such as the union type construction for `Light`. More sophisticated record structures can be built up from components, each of which has its own type. Collections of values can be built up as sets (where the ordering of the elements is insignificant), sequences (where the ordering is significant) and mappings (which represent relationships from values of one type to values of another).

The type definitions for the chemical plant alarm system developed so far are shown below. At this point it is worth making a remark about how models are presented. Models written in a special notation such as VDM-SL should *never* be presented without comment to support the reader. Throughout this book, we intersperse the VDM-SL text with natural language explanations typeset separately. VDMTools Lite supports several different ways of providing models with comments. In the simplest form, the explanatory text and VDM-SL are both in a plain VDM file. The explanatory text can be presented in comments using the syntax shown below. Note that definitions are separated by semicolons and that comments are put on lines beginning with "--", in common with many programming languages.

```
-- Expert call-out system.
-- VDM-SL model Version 3.5
-- Date: 21 August 2008
-- Engineer: PGL

-- Overall plant contains a schedule relating each period
-- to the set of experts on duty in the period; the alarms
-- component records the alarms which can possibly arise.

Plant :: schedule : Schedule
         alarms   : set of Alarm;

Schedule = map Period to set of Expert;

Period = token;

-- Experts represented by unique identifier and a
-- set of qualifications.

Expert :: expertid : ExpertId
          quali    : set of Qualification;

ExpertId = token;

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

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

The data types are presented in a top-down order with the `Plant` type first. It is quite common to do this because data type definitions are often more readable this way. However, in VDM-SL, the names of types do not have to be defined in any particular order, as long as they are all defined somewhere in the model. The particular order of presentation is therefore left to the author of the model.

2.7 Sketching function signatures

Now it is possible to consider the functionality provided by the system, beginning with the function signatures. Recall the functions identified from the requirements (Section 2.4):

```
ExpertToPage
ExpertIsOnDuty
NumberOfExperts
```

A function takes a number of input parameters and returns a result. The *signature* of a function gives the name of the function, the types of its input parameters and the type of the result.

First consider the `ExpertToPage` function. From Requirement *R6*, it appears that this function must take an alarm as input. In order to find the appropriate expert it also needs an identification of the period and requires access to the overall plant structure in order to read the schedule. The *signature* of `ExpertToPage` lists the types of the inputs and result as follows:

```
ExpertToPage: Alarm * Period * Plant -> Expert
```

where `*` is used to separate the types of inputs from one another and `->` is used to separate the types of the inputs from the type of the result.

Requirement *R7* suggests that the function `ExpertIsOnDuty` must take an expert and a plant as inputs. The function must check when the expert is on duty according to the schedule of the plant. The assumption is made here that the order in which the periods are presented is not relevant to the model's purpose, and so the result is a *set* of periods rather than a sequence. An assumption such as this one is naturally something that must be appropriately recorded and subsequently checked with the customer in order to find out whether a particular ordering is desired. The signature of `ExpertIsOnDuty` is as follows:

```
ExpertIsOnDuty: Expert * Plant -> set of Period
```

Finally, Requirement *R8* identifies a function `NumberOfExperts`. This function must certainly take the plant as input. However, the requirement is not very precise about whether this is for a specific period. We assume that this is the case but we would need to record this imprecision and our assumption that we are dealing with a specific period here. Thus, the signature is

```
NumberOfExperts: Period * Plant -> nat
```

The result type **`nat`** is a basic type in VDM-SL. It represents the natural numbers 0, 1, 2, 3, . . .

2.8 Completing the type definitions

At this stage, when all the type definitions and function signatures have been sketched out, it is worth considering whether there are any constraints that must hold in the model at all times. Such constraints are called *invariants*. Each invariant is recorded immediately after the definition of the type to which it refers. Indeed, the invariant forms part of the type's definition so, for a value to belong to the type, it must respect the invariant.

To state an invariant formally, we express the property which all values must satisfy as a Boolean expression on a typical element of the type. As an example, consider Requirement *R4*; arguably it should be strengthened. Experts with no qualifications are not of much use in this plant. If the expert is called `ex` this can be expressed by a Boolean expression as follows:

```
ex.quali <> {}
```

This expression can be used to introduce an invariant for the type `Expert` as follows:

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

where **`inv`** is a keyword indicating the invariant definition and `ex` is a pattern matching the structure of the type definition and finally the == means is defined as.

Requirement *R3* indicates that there must be at least one expert on duty in all periods for which the system has assigned a group of experts. This is a constraint on schedules, so it should be defined as part of the `Schedule` type definition. Consider a typical schedule (called `sch`, say). This is a mapping from periods to sets of experts. The constraint is that all the sets of experts in the range of the mapping are non-empty. Put formally:

```
forall exs in set rng sch & exs <> {}
```

The Boolean expression used here is called a *quantified* expression. It states that, for all sets of experts in the range of the mapping, the set must be non-empty. Quantified expressions are often used to express properties which should hold for collections of values. This language of logical expressions will be introduced in Chapter 4.

However, this constraint is not sufficient because we would also like to ensure that the expert identifiers are unique in each set of experts in order that one could not erroneously have two experts with different qualifications having the same expert identification. Put formally:

```
forall ex1, ex2 in set exs & ex1 <> ex2 =>
                             ex1.expertid <> ex2.expertid
```

Again a quantified expression is used, but this time two experts called `ex1` and `ex2` are selected from the set `exs`. In case they are different experts their identifications must also be different. The `=>` symbol is used for implication and this will also be introduced in Chapter 4.

Including both of these constraints into the invariant for `Schedule` gives us

```
Schedule = map Period to set of Expert
inv sch ==
   forall exs in set rng sch &
      exs <> {} and
      forall ex1, ex2 in set exs &
         ex1 <> ex2 => ex1.expertid <> ex2.expertid
```

The invariant is formulated as a logical expression that limits the elements belonging to the type being defined (in this case `Schedule`) to those satisfying both the type description and the invariant.

2.9 Completing the function definitions

It has already been seen that the data descriptions in a formal model need not contain all the intricate detail found in a program's data structures. We have simplified the model by abstracting away from a number of aspects which are not important for the functionality we want to describe. Indeed, a good general principle is that a model should contain no more detail than is relevant to its analysis. A similar principle applies to the definition of functions, so VDM-SL provides facilities to give a function definition *explicitly*, stating how a result can be calculated from the inputs. However, the language also provides for *implicit* function definition, where the result is characterised, but no specific "recipe" for calculating the result is used. This section illustrates first the explicit style and later the implicit.

Let us first consider the `NumberOfExperts` function. Requirement *R8* said "*It must be possible to assess the number of experts that are on duty*". In sketching the function signatures we decided that it would be natural to take a `Period` as an input to this function. The signature is therefore as follows:

```
NumberOfExperts: Period * Plant -> nat
```

The function is defined by giving its result in terms of its inputs. Thus, we need to look up the set of experts on duty for the given period in the schedule and return the size of the set. This can be defined as follows:

```
NumberOfExperts: Period * Plant -> nat
NumberOfExperts(peri,plant) ==
  card plant.schedule(peri)
```

The **card** notation indicates the *cardinality* (or size) of a set. It is applied to the set of experts obtained by looking up the input period `peri` in the schedule component of the plant record (`plant.schedule`). The operators and expressions used here will be introduced in greater depth in later chapters.

Calling the function `NumberOfExperts` is only meaningful if the period `peri` is actually described in the plant's schedule. We record this assumption by means of a *pre-condition*, a logical expression over the input parameters saying that the function may be applied when it is true. The pre-condition can be formulated as

```
NumberOfExperts: Period * Plant -> nat
NumberOfExperts(peri,plant) ==
  card plant.schedule(peri)
pre peri in set dom plant.schedule
```

The pre-condition says that the input period `peri` must be in the set of periods which forms the domain of the schedule mapping. We do not guarantee anything about what happens if a function is applied to inputs which do not meet the pre-condition. In a model at a later stage of design, we may wish to provide an error message in the situation where the pre-condition is violated and this can be done quite easily. However, at this early stage of analysis, we prefer to abstract away from this and record the restriction that the function must not be applied in this way.

Now consider the `ExpertIsOnDuty` function required by *R7*. In the case of `ExpertIsOnDuty`, the function returns a set of all the periods where the expert is on duty. Thus, in this case we need to select all the periods in the `schedule` which have the expert `ex` as one of the experts on duty in that particular period. The full function definition for `ExpertIsOnDuty` is as follows:

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

The body of this function is expressed using a *set comprehension* expression which collects all periods `peri` (from the domain of the schedule) for which the expert `ex` is registered. Set comprehension is explained in Chapter 6.

Notice that the definition of the function `ExpertIsOnDuty` selects the `schedule` component from the `plant` value twice. It may be worth using a more general pattern in the parameter list to avoid this:

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

At the moment patterns should simply be viewed as a way to provide names to the components of the `Plant` rather than having to use the dot notation for selecting the `schedule` component all the time. The `alarms` component is not needed in the function definition, and so is represented by a "-" symbol. We will return to more explanation about patterns in Chapter 5.

From a system point of view, it should also be ensured that the experts have the computing or telephone equipment they need to allow them to check whether they are on duty. However, this model abstracts away from the communication mechanism and interface.

Finally, consider the `ExpertToPage` function. Requirement *R6*, which gives rise to the function, states "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". Note that the requirement does not mandate any method for determining *which* expert has to be found: any expert will do as long as he or she has the correct qualification: there is some *looseness* in the requirement.

A common reaction to looseness is to assume that the requirements are incomplete. While this may sometimes be the case, careful consultation with a customer often reveals that resolving the looseness would involve adding detail that is not relevant to the model's purpose and may constrain developers too early in the development process by pre-empting design decisions that should be taken later in the development process. In such cases, the looseness should be left in the model. Modelling languages such as VDM-SL provide specific facilities for retaining looseness in models.
For many software developers, retaining looseness in a model goes against the grain because it means that the model is not directly executable in the same way that programs are executable. However, models are not the same things as programs, being designed for analysis rather than execution. Successful applications of formal modelling actually seek to increase the looseness in areas of the model that should not be "pinned down".

In the alarm example, the designer of the system is free to choose a mechanism for deciding which expert to call. This suggests that an *implicit* definition may be appropriate. Recall that an implicit function definition characterises the result without giving a particular mechanism for calculating it.

It is possible to model a function by stating what properties are required of the result it returns, without indicating how the result is to be calculated. This is done by means of an *implicit* function definition. The main advantage of using implicit function definitions is the ability to state the required properties of a result without biasing a subsequent developer towards any particular way of producing the result from the input parameters. An implicit definition of `ExpertToPage` is as follows:

```
ExpertToPage(a:Alarm,peri:Period,plant:Plant) r: Expert
pre peri in set dom plant.schedule and
    a in set plant.alarms
post r in set plant.schedule(peri) and
     a.quali in set r.quali
```

Notice that no separate signature is given, but that the names and types of the inputs and results are given together in a header. A pre-condition describes a condition under which the function should be applied and a post-condition describes the result without giving a particular algorithm for calculating it. The pre-condition is similar to that of the `ExpertIsOnDuty` function except that we also require the alarm `a` to be known for the plant. The post-condition states that the expert to be paged, `r`, must belong to the collection of experts on duty in this period, and one of his or her qualifications should be the one required to deal with the given alarm. If more than one such expert is available the post-condition above does not state who should be chosen. A definition that does not narrow the result down to a single possible outcome is said to be *loose*. We will return to the subject of looseness in Chapter 6.

By this stage one problem with the model may have become apparent. The requirements have been silent about the question “How can we be sure that an expert with the required qualification exists in the required period?”. The `ExpertToPage` function cannot yield a satisfactory result if no expert is available. The situation can be resolved in one of two ways. Either the pre-condition can be made more restrictive (requiring that a suitable expert should be available before an attempt is made to find one), or an invariant can be imposed on the type `Plant` to require that, at all times, there is at least one expert for each kind of qualification. This choice, between representing constraints as pre-conditions and recording them as invariants, is commonly faced by modellers. As a general rule, it is best to model this kind of constraint as an invariant if the property should hold at all times. If the restriction is only required to hold when a function is applied, then a pre-condition is more appropriate. In this example, we will assume that for each qualification at least one expert has to be available at any time. Thus, we should modify the invariant. The modified `Plant` type definition would be

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

and the function `QualificationOK` is defined explicitly as

```
QualificationOK: set of Expert * Qualification -> bool
QualificationOK(exs,reqquali) ==
  exists ex in set exs & reqquali in set ex.quali
```

The body of this function is a Boolean expression which is **`true`** if an expert with the required qualification (`reqquali`) exists in the given set of experts `exs`. Note that the function `QualificationOK` has been written purely for notational convenience: it makes the invariant easier to read. Unlike the other functions defined so far, it is not required to be implemented in the system under development. Such functions, defined for convenience rather than implementation, are called *auxiliary functions*. The use of auxiliary functions is to be encouraged, since they improve the ease with which a model can be read and analysed.

There is an obligation (called a *proof obligation*) on the writer of function definitions to check that each function will be able to produce a result whenever its pre-condition is satisfied. Note that without the addition of the invariant to the `Plant` type we would not have been able to meet this obligation for the `ExpertToPage` function. The role of such obligations will be examined more closely when we look at validation and consistency checking in Chapter 10.

2.10 Reviewing requirements

Having completed the formal model we look through the requirements one last time to review how each clause has been considered:

R1 A computer-based system managing the alarms of this plant is to be developed. *Considered in the overall* `Plant` *type definition and the function definitions.*

R2 Four kinds of qualifications are needed to cope with the alarms. These are electrical, mechanical, biological and chemical. *That is considered in the* `Qualification` *type definition.*

R3 There must be experts on duty during all periods which have been allocated in the system. *Invariant on the type* `Schedule`.

R4 Each expert can have a list of qualifications. *Assumption: non-empty unordered set instead of an ordered list in* `Expert`.

R5 Each alarm reported to the system has a qualification associated with it and a description which can be understood by the expert. *Considered in the* `Alarm` *type definition assuming that it is precisely one qualification.*

R6 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. *The* `ExpertToPage` *function with additional invariant on the* `Plant` *type definition.*

R7 The experts should be able to use the system to check when they will be on duty. *The* `ExpertIsOnDuty` *function. Assumption that the ordering of the periods during which the experts will be on duty is not important.*

R8 It must be possible to assess the number of experts on duty. *This is the* `NumberOfExperts` *function with assumption for a given period.*

Weaknesses in the requirements document

The development of the formal model has helped to identify some weaknesses in the original requirements document. We made assumptions to resolve these, so it is important to record them. Below we list the deficiencies spotted:

- No explicit mention of unique identification of experts.
- Experts without qualifications are useless, and may not be considered "experts" at all.
- How can we be sure that an expert with the required qualification is on duty in the required period?
- Should the number of experts be relative to a period?
- Does "a qualification" mean "exactly one" or "at least one"?

Summary

In this chapter we have used a systematic approach to analyse the requirements for an alarm system. In practice, there is no right or wrong way to compose a model from a requirements description, but the following list of steps can be a helpful guide for the novice:

1. Read the requirements.
2. Extract a list of possible data types (often from nouns) and functions (often from actions).
3. Sketch out representations for the types.
4. Sketch out signatures for the functions.
5. Complete the type definitions by determining any invariant properties from the requirements and formalise these.
6. Complete the function definitions, modifying the type definitions if necessary.
7. Review the requirements, noting how each clause has been considered.

The presentation given here is a simplification of what would happen in the development of an industrial system. Depending upon the context, involvement from customers and/or local domain experts could be necessary to resolve the unclear points discovered and the assumptions made during the construction of a formal model. It is important to note, however, that the construction of a model allows us to expose and record these assumptions at an early stage in system development.

The approach to developing a model which has been presented in this chapter underlies the models developed subsequently. However, when we are presenting an example for the purpose of introducing some types or operators, we will not explicitly describe the individual steps in the process.

We have gone through the process of constructing a formal model. The language, no doubt, still seems unfamiliar. However, most of the features of the modelling language have been introduced: the main part of this book aims to help the reader develop confidence using the logic employed for writing invariants, pre-conditions and post-conditions, and familiarity with the operators on the data types (sets, mappings, sequences and records, mostly). The subsequent chapters take a much more detailed look at these, with plenty of examples. Appendix Appendix A provides an overview of all the constructs in the subset of VDM-SL used in this book and serves as a quick reference on the usage of the language's constructs.

Exercise 2.1 This exercise is based on a (fictional) technical report written by an engineer working on another part of the chemical plant alarm system and describes a first attempt at a model for the system under development. Read the report and then consider the questions which follow it. The object of this exercise is simply to increase your familiarity with models expressed in VDM-SL.

Reading and considering the report are more important than getting "correct" answers!

We have been asked to develop a detector for alerting the chemical plant's experts whenever there is a certain kind of failure in the sensors in any of the reactor vessels. When a sensor fails, it does not reply promptly to a request for data.

The detector will broadcast requests for data to all the sensors in the plant at certain times. A functioning sensor will reply to such a request within a time limit (called `maxtime`). A malfunctioning sensor will not reply within the time limit. If a malfunctioning sensor is discovered an alarm is to be raised. The qualification needed to deal with these alarms is always electrical knowledge.

It was decided to develop a system model in VDM-SL in order to clarify the rules for raising this alarm. Obviously, other kinds of failure are possible (sensors sticking on a value, for example), but we were asked only to analyse the rules for detecting this particular kind of failure, so the model abstracts away from other factors which are not material to this analysis.

The system consists of a collection of sensors, each with its own identifier. In addition, we keep a record of the times when requests are sent out and the collection of replies obtained to each request. The system is modelled as the type `System` shown below. The sensors are modelled as a set of sensor identifiers while the requests and answers from the sensors are recorded as a mapping from times (the times at which requests were issued) to sets of replies.

```
System :: sensors: set of SensorId
          answers: map Time to set of SensorReply
```

Recall that the purpose of the model is to allow analysis of the alarm-raising mechanism. The detailed representation of sensor identifiers is not a significant aspect of this, and so the **`token`** type representation is used.

```
SensorId = token
```

Also, for the purposes of the model, time is represented by a natural number denoting the number of elapsed time units since the combined sensors and alarm system were started up.

```
Time = nat
```

The sensor replies contain information about the time of the request being responded to, the sensor replying and the data it is carrying (a real number representing reactor temperature):

```
SensorReply :: request: Time
               sid : SensorId
               data: Data
```

The data types for alarms and qualifications are taken from the current model of the expert call-out system:

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

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

We begin defining the functionality with a simple function which clears the current record of requests and replies:

```
  Clear: System -> System
  Clear(sys) ==
    mk_System(sys.sensors,{|->})
```

The function `Request` models the issuing of a new request. This adds the request time to the domain of the `answers` mapping, pointing to an empty set of replies:

```
Request: System * Time -> System
Request(mk_System(sens,ans),t) ==
  mk_System(sens, ans munion {t |-> {}})
pre forall t1 in set dom ans & t1 < t
```

When sensor replies are received they must be registered in the system for the request made at the time given in the `request` field of the reply. Thus the `answers` mapping must be updated with this information:

```
Reply: System * SensorReply -> System
Reply(mk_System(sens,ans),sr) ==
  mk_System(sens,
            ans ++ {sr.request |-> ans(sr.request)
                                   union {sr}}
             )
 pre sr.request in set dom ans
```

An alarm is to be constructed if there is some request issued more than `maxtime` time units ago to which not all of the sensors have yet replied. The function `RaiseAlarm` is called with a system `sys` and the current time (`now`) as inputs. It constructs an alarm, but should be called only if there are still some outstanding responses to one of the requests. The precise definition of the alarm condition is in the function `CheckAlarm`.

```
RaiseAlarm: System * Time -> Alarm
RaiseAlarm(sys,now) ==
  mk_Alarm("Sensor Malfunction", <Elec>)
pre CheckAlarm(sys,now)
```

The function `CheckAlarm` returns a Boolean value which is **`true`** if there exists some request in the `answers` component of the system which is more than `maxtime` time units old and there is at least one sensor which has not returned a reply. Its definition will be given in the next draft of this report.

The report presented above was reviewed by several other engineers. They wrote the following comments and questions on the report. In each case, consider whether you agree with the comment or try to answer the question. Remember that it is more important to think about the model and its level of abstraction than to get a "correct" answer.

1. *Why do you represent the* `Data` *field in the* `SensorReply` *record? As far as I can see, it never gets used.*
2. *Presumably you need to make sure that, when you issue a new request, the time of the new request is later than all the other times in the system. How does the model ensure this?*
3. *What do you need the function* `Clear` *for?*
4. *I heard that the team working on the expert call-out software had changed their model of an alarm so that it includes a call-out time as well as a message and a qualification. What parts of your model would be affected by this?*

The last comment raises a significant point. If several parts of a large system are being developed simultaneously, it is helpful to have a mechanism for defining the interfaces between those subsystems, so that it is possible to check that data types and functions provided by the model of one subsystem are used correctly in the models of other subsystems. The modular structuring mechanism in VDM-SL (introduced in Chapter 12) allows for this kind of checking. It is often the case in software developments that misunderstandings about the interfaces between subsystems are responsible for considerable reworking costs after deficiencies are discovered when modules are tested together. □