1
Introduction

Aims

The aim of this chapter is to provide a motivation for studying the modelling of computing systems by discussing the challenges of developing correct software. On completion of this chapter, the reader should be aware of the main concepts to be presented in the book and know where to find the relevant material in the text.

1.1 Software

Software is pervasive, error-prone, expensive to develop and, as an engineering medium, extraordinarily seductive. Its seemingly infinite flexibility, increasing power and the absence of physical characteristics, such as weight, make it an ideal medium in which to express complex models which might not exist at all were it not for software. As a result, software is often developed for applications which are critical either to an enterprise's mission or to the quality of life of those with whom the system interacts.

Challenged by the variety and scale of software applications, the participants in the 1968 NATO Conference on Software Engineering foresaw a discipline of software development with a sound scientific basis [Naur&69]. Over the last 40 years, there is little doubt that enormous advances have been made in our ability to control software development. However, software projects continue to suffer from serious difficulties which can lead to the delivery of faulty goods that are over budget and behind schedule.

The rapid increase in processor power has naturally led to increasing demands being made on software and its developers. Software is almost always developed as part of a larger system involving computing hardware, special systems such as sensors and actuators, human-computer interfaces and human beings. However, the long lead-times associated with the production of special items of hardware mean that additional functionality caused by changes in customers' requirements is often realised in software because that medium is seen as more flexible.

A comparison between software engineering and the engineering in other media, whether mechanical, fluid, chemical or electronic, is difficult because of the different characteristics of those media. However, there is little doubt that software engineers can still learn from other more mature engineering disciplines.

1.2 Modelling and analysis

One of the major differences between software engineering and other forms of engineering is that the other disciplines have a longer tradition of constructing abstract models of the product in the early stages of development. Such models serve as a proving ground for design ideas and as a communication medium between engineers and customers. As a result of modelling, engineers can avoid errors which might otherwise only become obvious in the very late stages of development, when expensive commitments have been made to materials and designs. There are two aspects of these models which are crucial to their successful use: abstraction and rigour.

Engineering models are abstract in the sense that aspects of the product not relevant to the analysis in hand are not included. For example, an aeronautical engineer investigating the aerodynamics of an aircraft design may model the air flow over the surfaces (mathematically or in a wind tunnel) because air flow is a dominant design parameter. The model is unlikely to include the user interface of the cockpit instruments. Similarly, human factors engineers who design cockpit instruments model the cockpit, not the aerodynamics of the wing surfaces. The choice of which aspects of a system should be included in the model (its level of abstraction) is a matter of engineering skill.

Perhaps the most significant property of a system model is its suitability for analysis. The purpose of such an analysis is to provide an objective assessment of a model's properties. For example, a model of a new design of bridge might be used to assess the design's ability to withstand physical stresses with acceptable risk of collapse. This contrasts with the more subjective analysis of a review or inspection in which the outcome may depend on the consultants carrying out the job. It is also important to be able to repeat the assessment on alternative models and to be able to perform as much as possible of the analysis mechanically, in order to minimise the risk of subjectivity and error as well as the required human effort. To obtain this level of objectivity, mechanisation and repeatability, mathematics is often used in the analysis. Indeed, many system models exist only as mathematical constructions and not as physical entities at all.

How do these concepts of system modelling transfer to the development of

computing systems, and in particular to software? A wide range of modelling techniques has been developed for computing systems, including pseudo-code, natural language, graphical and mathematical notations. Ultimately, a computer program could be seen as a model of the system which is to be provided: an executable model which meets all the relevant user requirements, or at least a large enough subset of them to make the product acceptable. Although a wide variety of modelling techniques is available, comparatively few provide the combination of abstraction and rigour which could bring the benefits of early detection of errors and increased confidence in designs.

This book describes well-established modelling techniques which combine abstraction with rigour. We will introduce the elements of a modelling language that can be combined effectively with existing software engineering techniques, opening up the possibility of improved analysis in early development stages.

Models expressed very abstractly in the early stages of system development would normally be treated as *specifications* of a system. If the models instead are described at a lower level of abstraction later in the process, they will normally be called *designs*. The borderlines between specification, design and implementation are not clearly defined and the modelling and analysis techniques discussed in this book are not confined to any particular stage of software development. We will therefore tend to avoid loaded terms such as "specification" and use the general term "model" to refer to the system descriptions we develop. Nevertheless, we focus on requirements analysis and early design stages because these are the phases in which the application of modelling is most beneficial.

1.3 This book

This book is concerned with the construction of abstract models for computing systems. The notation used to describe such models is a subset of the ISO standardised language VDM-SL [ISOVDM96]. The VDM-SL notation supports abstraction in a variety of ways which will be introduced in the text. The rigour of the language lies mainly in its definition in the ISO standard, which is extremely thorough and detailed. Indeed, the language is referred to as a *formal* modelling language because its syntax and the meaning of models expressed in the language are so thoroughly defined. This formality allows analyses to be carried out consistently because the formal definition of the meaning of the language constructs leaves little or no room for interpretations to differ between support tools or practitioners.

This book is about the practical exploitation of modelling techniques. Our pragmatic approach is realised in three ways. First, we make extensive use of a

tool, VDMTools, which has a strong record of industrial application and which is available for free download. Most exercises in the book are designed to provide training in both modelling concepts and tool support. Second, we use concrete examples to motivate the introduction of language features. The majority of these examples are derived from real models developed by industrial engineers or in close collaboration with industry. Third, the validation approach we use in this book exploits testing rather than proof, which we see as a more advanced technique to be used when occasion demands, but which does not form a part of the initial training offered here.

1.4 VDM-SL

The *Vienna Development Method* (VDM) is a collection of techniques for modelling, analysing and developing sequential software systems. VDM's modelling language, which we use as the main vehicle in this book, is commonly known as *VDM-SL* (the VDM Specification Language). It is one of the most widely used modelling languages, having been applied to the construction of a variety of software systems. Its name refers to its origin in IBM's Vienna Development Laboratory [Jones99].

The VDM-SL notation is fixed in an ISO standard [ISOVDM96]. It permits both abstraction from the data structures to be used in the final implementation of the system and also algorithm abstraction, whereby one can state *what* a function should do without having to provide detail on *how* it should work.

The analysis techniques for models in VDM-SL covered in this text include static checking of syntax and type-correctness of models, and animation of models by execution. Mathematical proof can be used to show internal consistency of models and to show that less abstract models are faithful to more abstract models of the same system. However, this book does not seek to cover the proof techniques, instead focussing on the construction of models and analysis by the other means mentioned above.

1.5 The structure of a VDM-SL model

This section provides a very brief introduction to the structure of a VDM-SL model through the example of a simple air traffic control system which monitors the movements of aircraft in the airspace around an airport. In what follows, we are not concerned with notational detail, but only with the general form and content of the model. As each main feature of the model emerges, a reference is given to the relevant part of the book. The process of deriving

a VDM-SL model from a collection of customer requirements is discussed in Chapter 2.

A VDM-SL model, like many programs, is structured around descriptions of data and functionality. Data are described through a collection of type and value (constant) definitions; functionality is described through function definitions. Each kind of definition is considered in turn below.

1.5.1 *Modelling data in VDM-SL*

Data are mainly modelled by means of *type definitions*. A type is a collection of values which might arise in the model of a system. In our air traffic control model, for example, we could have types to represent the positions of aircraft, their latitude, longitude and altitude. A type definition gives a representation to a type. For example, the author of the air traffic control model could choose a representation for latitudes. In VDM-SL, the modeller would write the following[1]:

```
Latitude = ???
```

How could a latitude be represented? A latitude is usually a number between -90 and $+90$, assuming a representation in degrees. The modelling language VDM-SL provides the modeller with a collection of basic types from which to build representations of new types such as `Latitude`. The basic types include collections of values such as the natural numbers, integers, real numbers, Boolean values and characters. The basic types from which models can be constructed are introduced in Chapter 5. In this example, the representation of a latitude could be as a real number. This would be written as follows:

```
Latitude = real
```

However, it is still necessary to record the restriction that the latitude must be between -90 and $+90$. Such additional restrictions on the values included in a type are recorded by means of *invariants*. An invariant is a property which must always be true of values in a certain type. If a type has an invariant associated with it, the invariant is stated as a Boolean expression on a typical element of the type. Thus, the following type definition defines a type called `Latitude` which contains all real numbers from -90 up to, but not including, $+90$. It does this by describing the property that a typical latitude `lat` is greater than or equal to -90 and less than or equal to $+90$. Thus:

[1] In this book, VDM-SL models will be presented in a `typewriter` typeface.

```
Latitude = real
inv lat == lat >= -90 and lat < 90
```

The Boolean expression defining the invariant on the type `Latitude` is written and interpreted using the logic of VDM-SL. This logic, which incorporates operators like `and`, `or` and `not`, is introduced in Chapter 4.

In VDM-SL there is no limit to the precision or size of the numbers in the basic type `real`. This is an example of abstraction from computer systems where a limit will exist. When the precision or maximum size of a number is a significant factor in the model, this is expressed by defining a new type (e.g. one called `LongReal`) which respects the relevant restrictions.

Types such as `Latitude` can be used in the representations of other, more complex, types. For example, aircraft positions can be modelled by a type `AircraftPosition` defined as follows:

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

Here `AircraftPosition` is represented as a *record type*. A record type is similar to a record or struct type in other programming languages such as Ada or C++. Its elements are values which are each made up of several components. This definition says that an aircraft position will consist of three things: a latitude, which is a value from the type `Latitude`; a longitude, which is a value drawn from the type `Longitude`; and an altitude modelled by a value drawn from the type `Altitude`. Records, and other ways to construct more complex types from simple types, are introduced in Chapter 5.

Often it is necessary to model more elaborate data than just single numbers, characters or records. Frequently we will need to model collections such as sets of values, sequences of values, or mappings from values of one collection to values of another. These three kinds of data type (sets, sequences and mappings) are central to the construction of models in VDM-SL – so much so that they each warrant an entire chapter later in the book (Chapters 6 to 8). As an example of their use, we could here define a model of the flight information on a radar screen as a mapping from aircraft identifiers to aircraft positions. The mapping can be thought of as a table associating aircraft identifiers with positions. In VDM-SL, we would make the following type definition:

```
RadarInfo = map AircraftId to AircraftPosition
```

The type `AircraftId` would be defined elsewhere in the model. In a programming language there are many different ways of implementing this data structure (e.g. pointer-based tree structures, arrays, hash tables) but, early in

the development process, we may not be interested in precisely which structure should be chosen. If the model has been constructed in order to analyse, for example, the possibility of a "near miss" between aircraft being alerted, the space efficiency of the data structure used to model the mapping is not a dominant consideration, and so does not form part of the abstract model. At a later development stage, this issue may become significant and so could form part of a model used for space efficiency analysis. The use of an abstract modelling language like VDM-SL in the early stages of design naturally encourages one to think in this way, in terms of *what* concepts are needed in the model and not *how* they are to be implemented.

Recursive data structures are common in software, and are especially significant in representing computer programs. Chapter 9 deals with such structures including trees and graphs, and the recursive functions that can be used to construct and traverse them.

1.5.2 *Modelling functionality in VDM-SL*

Given a collection of type definitions modelling system data, the system's behaviour is modelled by means of functions and operations defined on the data model. For example, two functions of interest in the air traffic control system might be functions to add a new aircraft, with its position, to the radar information and a function to choose an aircraft to which landing permission is to be granted.

The first function, to add a new aircraft which has just been detected by the system, could be modelled as follows. Suppose the function is to be named `NewAircraft`. When a function is defined, the types of its inputs and result are given. In this case, the inputs are some radar information, the identifier of the new aircraft and the position of the new aircraft. The result returned will be a radar information mapping, the same as the input mapping, but with the new aircraft and its position added. The types of the input and result are given and the action performed by the function is described on some input parameters. The function definition so far would be written as follows:

```
NewAircraft: RadarInfo * AircraftId * AircraftPosition ->
             RadarInfo
NewAircraft(radar,airid,airpos) == ???
```

The "`???`" contains the body of the definition: an expression showing the value of the result in terms of the input values supplied. In this case the result is just the input mapping `radar` with `airid` and `airpos` added to it so that `airid` maps to `airpos`. The details of the expression in the body of the function are

not of concern here: they will be dealt with in full in later chapters. In VDM-SL this is written as follows:

```
NewAircraft: RadarInfo * AircraftId * AircraftPosition ->
             RadarInfo
NewAircraft(radar,airid,airpos) ==
  radar munion {airid |-> airpos}
```

The `NewAircraft` function should not be applied to add just any aircraft to the radar information. It should only be applied when the newly detected aircraft is indeed new, i.e. it does not already occur in the mapping. To record a restriction on the circumstances in which a function may be applied, a *pre-condition* is used. A pre-condition is a Boolean expression which is true only for those input values to which the function may be applied. In this example, the pre-condition must state that the input `airid` is not already in the input mapping `radar`. Again, the details of the expression are not important and will be discussed in later chapters.

```
NewAircraft: RadarInfo * AircraftId * AircraftPosition ->
             RadarInfo
NewAircraft(radar,airid,airpos) ==
  radar munion {airid |-> airpos}
pre airid not in set dom radar
```

The function definition given here is abstract. There is no information about the details of how the new information is to be added to the mapping. It simply states that the new identifier and position are to be added.

In some cases, it is possible to have an even more abstract function definition. An *implicit* function definition, rather than stating what the result of the function should be, simply characterises the result by saying what properties are required of it. This technique is often valuable where we do not wish to have to give an algorithm for calculating a result. For example, we may not be interested in modelling the algorithm which is used to select an aircraft for landing, but we do need a function which describes the selection of an aircraft, because we will wish to model its removal from the radar information once the aircraft has landed. In this case, we do not give a result expression, but instead give a *post-condition*.

A post-condition is a logical expression which states how the result is to be related to the inputs. For this example, the function `SelectForLanding` takes the radar information as input and returns an aircraft identifier as the result. The post-condition states that the result is an identifier from the mapping, but goes no further in suggesting how the result is chosen. A pre-condition is recorded to assert that this function should only be applied when the radar information

is non-empty, i.e. there are some aircraft identifiers to choose from. Again, the details of the expressions used will be dealt with in later chapters. The function definition is as follows:

```
SelectForLanding(radar:RadarInfo) aircraft:AircraftId
pre  dom radar <> {}
post aircraft in set dom radar
```

In a VDM-SL model, each function is self-contained in that the only data which can be referred to in the function body are the input parameters. In particular, the function body has no access to any global variables. However, in this example it is likely that many of the functions would need to access and possibly update the radar information. There is a clear sense in which the radar information is persistent, and the functions merely make changes to part of that radar information. The functions in this case might be better expressed as procedures with side-effects on global variables. For such situations it is possible to describe the persistent data in a *state definition* and record the modifications to the state as operations. This state-based modelling style is the subject of Chapter 11.

The example used in this section does not reflect the size and complexity of a realistic computing problem. Indeed, any course on modelling must use examples which fit into the textbook space available. However, Chapter 12 deals with techniques for structuring large-scale models by splitting them into manageable, and sometimes re-usable, modules.

1.6 Analysing a model

If a model is intended to form a basis for the development of software, it is important to have confidence both in its internal consistency and in the accuracy with which it records the customer's original requirements. Checking internal consistency involves ensuring that the language syntax has been correctly followed and that the functions can indeed be calculated on the values provided as inputs (this is done by ensuring that operators are applied to values of the correct types). A range of more demanding checks can also be performed: for example, making sure that a function definition does not allow the invariant on a data type to be violated. In addition, the model can be tested by supplying sample input values and calculating the results of applying the functions to these.

The process of increasing confidence in a model is called *validation* and is described in detail in Chapter 10. When inconsistencies are discovered, or the model does not correspond to expectations in some way, the model can be modified at this early stage in the development of the computing system, before going

on to detailed design. Early identification and resolution of errors prevents their propagation into detailed design and code and subsequent, late and expensive, correction. The issues surrounding the use of modelling in the commercial development process are the subject of Chapter 13.

Summary

- Software developers have a difficult task, because of the complexity of the systems they build and the characteristics of the material out of which they are built. Nevertheless, some useful lessons can be learned from other engineering disciplines. One of these is the value of models of systems in the early stages of development.
- If used in the early stages of software development, models can ease communication between developers and between developers and clients. They can help identify deficiencies in understanding and in requirements and thus help to reduce rework costs in later development stages.
- To be useful, a model should be abstract (so that it is not too complex) and rigorously defined, so that objective and repeatable analyses can be performed.
- Formulating an abstract model using a notation with a fixed syntax and semantics enables machine support and provides a communication medium without ambiguity. Such a formal definition of a notation also provides a means of resolving disputes about the meaning of a model.
- VDM-SL is an ISO-standard modelling language which has a formal definition. It is part of a collection of techniques for analysing models and developing software from them. In this book we will concentrate only on the system modelling and analysis aspects.
- A model in VDM-SL contains definitions of the data and functionality of a system.
- The data is represented through types which are built from simple basic types such as characters and numbers. Values of these basic types can be grouped into elements of more elaborate types such as records, sets, sequences and mappings. Types may be restricted by invariants. Abstraction is obtained, where required, by allowing data values of arbitrary size and precision. A modelling language such as VDM-SL allows the modeller the freedom to choose a level of abstraction appropriate to the analysis in hand. In contrast, a model in a traditional programming language may have to contain machine-specific details not relevant to the model or analysis.