Chapter 5

The SysML notation

Words are but symbols for the relations
of things to one another and to us;
nowhere do they touch upon absolute truth.

Friedrich Nietzsche (1844–1900)

5.1 Introduction

This chapter describes the nine SysML diagrams. Following this introduction, the terminology used throughout the chapter is explained and the structure of SysML diagrams is discussed. This is followed by a discussion of *stereotypes* and then of the SysML meta-model, which forms the basis of this chapter. Following this, each of the nine diagrams is described in turn. For each diagram type there is a brief introduction, a discussion of the diagram elements through its meta-model and notation, examples of how to use the diagram and a summary.

5.1.1 Diagram ordering

So far, we have looked at two of the diagrams in some detail when *block definition diagrams* and *state machine diagrams* were used to illustrate structural and behavioural modelling in Chapter 4; these diagrams are shown again in this chapter for the sake of completeness and also to introduce the meta-model using diagrams that are already well known.

The chapter first covers the *structural diagrams* and then the *behavioural diagrams*. Within these groupings there is no significance in the ordering of the diagrams. They are simply presented in, what is from the author's point of view, a logical order. Therefore, the various parts of this chapter may be read in any order.

5.1.2 The worked example

When discussing each of the SysML diagrams in the sections that follow, they will be discussed using an example System taken from the world of escapology. The System consists of an escapologist who is placed in a rectangular coffin, which is then placed into a hole. Concrete is pumped into the hole, under computer control, until the hole is full. The escapologist has to escape from the coffin and the concrete-filled hole before his breath runs out. Figure 5.1 shows the set-up for the escape.

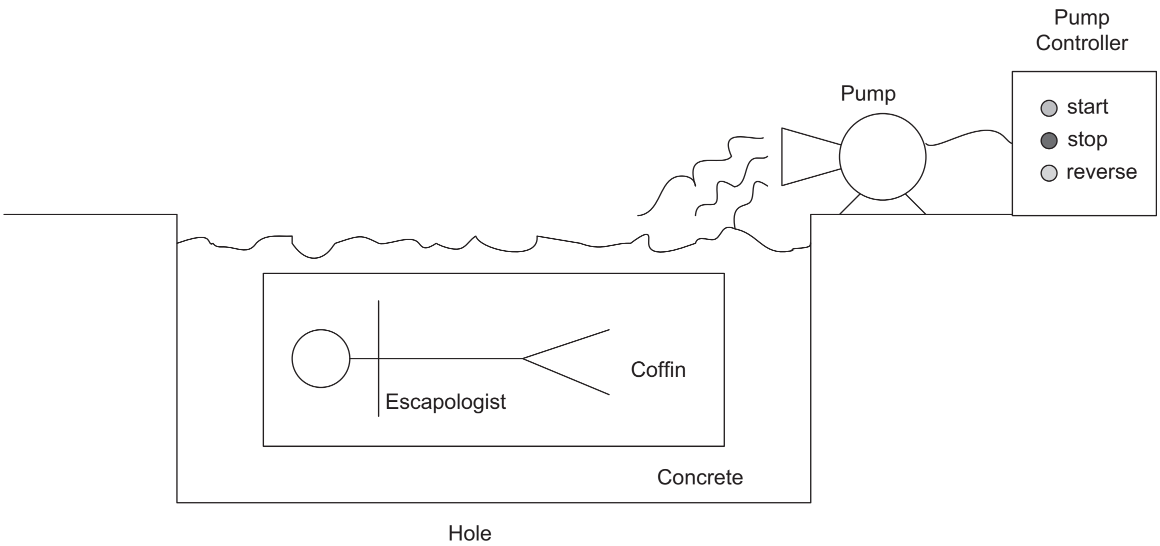

Figure 5.1 The coffin escape stunt

This is a classic escapology stunt that has been performed by many people. It is also a dangerous one, and escapologists have lost their lives performing it because the System Requirements and constraints were not properly understood or evaluated. One such performer was Joe Burrus who died 30 October 1990 when the weight of the concrete crushed the coffin he was in. This example is a socio-technical System that includes hardware, software, People and Process. It lends itself readily to the use of all of the SysML diagrams. What is more, it is not an example based around a library, an ATM or a petrol pump. The literature is already too full of such examples.

5.2 The structure of SysML diagrams

Each diagram in the SysML has the same underlying structure, which is intended to provide a similar appearance for each, as well as making cross-referencing between diagrams simpler. The structure of each diagram is shown in Figure 5.2.

The diagram in Figure 5.2 shows that each 'diagram' is made up of one or more 'graphic node' and one or more 'graphic path'. Each 'graphic path' relates together one or two 'graphic node'. Examples of graphic nodes include *blocks* on *block definition diagrams* and *states* on *state machine diagrams*. Examples of graphic paths include: *relationships* on *block definition diagrams* and *control flows* on *activity diagrams*.

The text '«stereotype»' on the *blocks* is an example of ... a *stereotype*. *Stereotypes* are a mechanism by which the SysML can be extended. Indeed, the

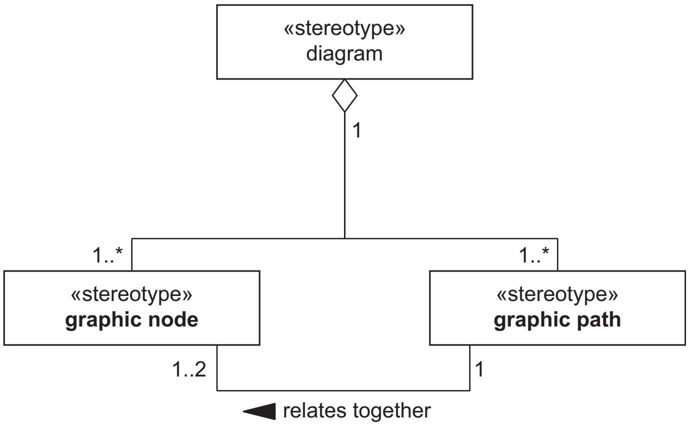

Figure 5.2 Structure of each SysML diagram

SysML itself is defined using *stereotypes* on the underlying unified modelling language (UML). *Stereotypes* are discussed in Section 5.3.

5.2.1 Frames

Any SysML diagram must have a graphic node known as a *frame* that encapsulates the diagram in order to make identification of, and navigation between, diagrams simpler. *Frames* have a defined format. This format, along with other guidelines for the use of *frames*, is described in detail in Chapter 6. Examples of *frames* will be seen around all the diagrams in the Examples subsections for each of the SysML diagrams in the following sections.

5.3 Stereotypes

Stereotypes provide a way to extend the SysML. They represent a powerful way to define new SysML elements by tailoring the SysML to your needs.

In order to use *stereotypes* effectively, it is first necessary to be able to spot one within a model. Visually, this is very simple, as *stereotypes* are indicated by enclosing the name of the *stereotype* within a set of double chevrons. Indeed, the SysML block itself contains the «block» *stereotype*.

Figure 5.3 shows two example *stereotypes*: «testCase» applied to a *block* (here representing a Scenario) and «validate» applied to a *dependency*.

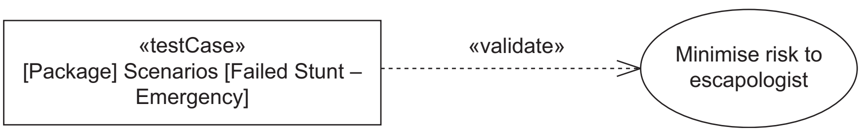

Figure 5.3 Example stereotypes

A *dependency*, represented by a dashed line with an open arrowhead, can be considered to be the weakest of the SysML relationships since it simply shows that there is some kind of (usually) unspecified relationship between the connected diagram elements. *Dependencies* are not named and cannot have any *multiplicities* associated with them. SysML makes use of a number of stereotyped *dependencies*, particularly in the *requirement diagram* and *use case diagram*, as described in Sections 5.5.5 and 5.5.9. In Figure 5.3, a new *stereotype* is used, one not found in the standard SysML, in order to show that a *test case* validates a *use case*. Note that «testCase» is a SysML stereotype and that the camel case naming is part of the SysML.

Stereotypes can be defined for any of the standard SysML elements. Unfortunately, the method by which *stereotypes* are defined varies from SysML tool to tool. However, a common diagrammatic method of defining a *stereotype*, found in many tools, is shown in Figure 5.4.

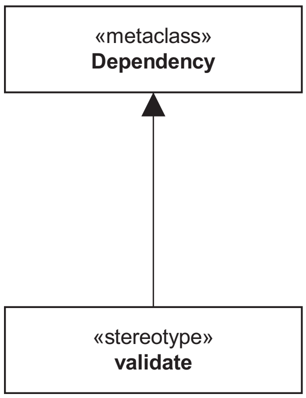

Figure 5.4 Defining a stereotype

The diagram in Figure 5.4 shows the definition of the «validate» *stereotype*. The diagram shows two *blocks*, 'Dependency' and 'validate', which are related together by a special type of *specialization/generalization* known as an *extension*. An *extension* is used specifically when defining *stereotypes*. An *extension* is represented graphically by a filled-in triangle – very similar to the *specialisation/generalisation* symbol.

The new *stereotype* to be defined, in this case 'validate', is shown in a *block*, which is itself stereotyped «stereotype». The SysML element that is being stereotyped, in this case a *dependency*, is shown in a *block* containing the «metaclass» *stereotype*. The two *blocks* are then connected with an *extension* relationship. This shows that the «validate» *stereotype* can be applied to a *dependency* and, as defined in Figure 5.4, only a *dependency*. In addition to the graphical definition, it is considered good modelling practice to provide a textual description of the *stereotype* that describes its intended use.

The diagram in Figure 5.4 can be generalised to give a rubber stamp version that forms the basis of the definition of any *stereotype*. Such a diagram is given in Figure 5.5.

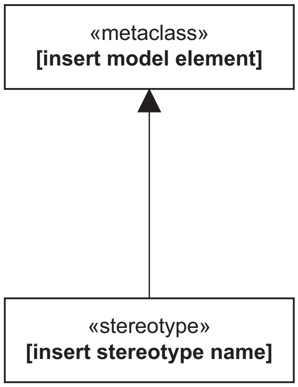

Figure 5.5 "Rubber stamp" diagram for stereotype definition

To use this diagram simply replace indicated text. For example, if it a modeller wanted to be able to apply the *stereotype* «ethernet» to an *association* on a *block definition diagram*, then start with Figure 5.5 and simply replace '[insert stereotype name]' with 'ethernet and '[insert model element]' with 'Association', giving the diagram as shown in Figure 5.6.

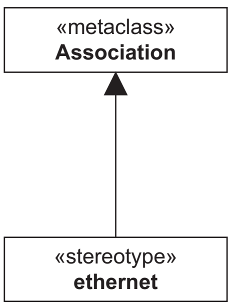

Figure 5.6 Another example of stereotype definition

When defining *stereotypes*, SysML also allows information to be associated with the *stereotype*. These *properties* are known as *tags* and they are defined as *properties* of the *stereotype block*. An example is given in Figure 5.7.

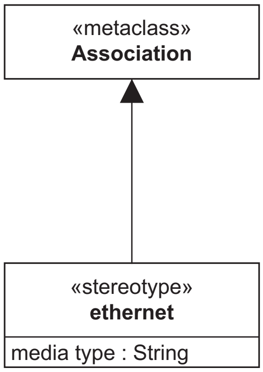

Figure 5.7 Stereotype with tag definition

The «ethernet» *stereotype* in Figure 5.7 has been extended through the definition of the 'media type' *tag*, intended to be used to show the type of ethernet being used. When the «ethernet» *stereotype* is applied to an *association* then a value can be given to any *tags* defined for that *stereotype*. These *tags* are then shown in a comment, as in the example in Figure 5.8.

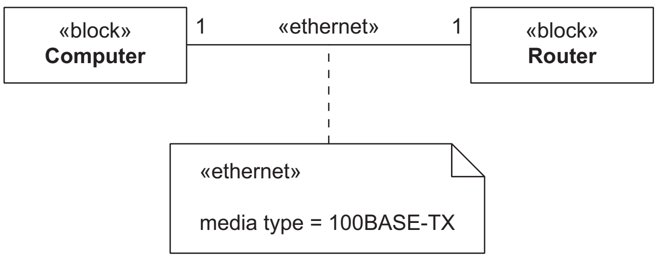

Figure 5.8 Example of stereotype usage with tags shown in comment

Note that not all SysML tools show *tags* in this way. For example, some tools show *tags* along with the stereotype as in Figure 5.9.

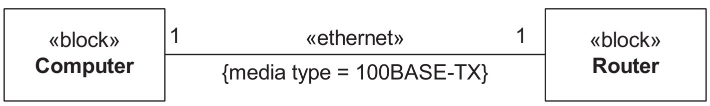

Figure 5.9 Example of stereotype usage with tags shown as part of stereotype

Each *tag* is shown with its value on a separate line underneath the *stereotype*. It is enclosed in curly braces. If a *stereotype* has multiple *tags*, then each will be displayed on a separate line.

5.4 The SysML meta-model

The SysML specification defines SysML in terms of the underlying UML on which SysML is based, and is done so using UML via the SysML meta-model. This is a model, in UML, of the SysML.

This chapter presents a partial meta-model for each of the nine SysML diagrams. In keeping with the use of UML in the SysML specification, UML *class diagrams* have been used to produce the SysML meta-model diagram throughout this chapter. These diagrams are the same as would be produced if using SysML *block definition diagrams*, and therefore can be read as SysML *block definition diagrams*. Thus, it *would* be possible to model the SysML using the SysML if desired.

The SysML meta-model itself is concerned with the modelling elements within the SysML, how they are constructed and how they relate to one another. The full UML meta-model on which SysML is based is highly complex and, to someone without much SysML (or UML) experience, can be quite impenetrable. The meta-models presented in this book show highly simplified versions of the actual meta-model in order to aid communication and to group different aspects of the model according to each diagram – something that is not done in the actual meta-model.

5.5 The SysML diagrams

This section describes each of the nine SysML diagrams, beginning with the five *structural diagrams* and concluding with the four *behavioural diagrams.*

5.5.1 Block definition diagrams

This section introduces what is perhaps the most widely used of the nine SysML diagrams: the *block definition diagram*. The *block definition diagram* was introduced in Chapter 4 in order to illustrate structural modelling and this section expands upon that information, covering more of the syntax and showing a wider range of examples, which are all taken from the escapology example that runs throughout this chapter.

Block definition diagrams realise a structural aspect of the model of a System and show what conceptual things exist in a System and what relationships exist between them. The things in a System are represented by *blocks* and their relationships are represented, unsurprisingly, by *relationships*.

5.5.1.1 Diagram elements

Block definition diagrams are made up of two basic elements: *blocks* and *relationships*. Both *blocks* and *relationships* may have various types and have more detailed syntax that may be used to add more information about them. However, at the highest level of abstraction, there are just the two very simple elements that must exist in the diagram. A *block definition diagram* may also contain different kinds of *ports* and *interfaces*, together with *item flows*, but at their simplest will just contain *blocks* and *relationships*.

Blocks describe the types of things that exist in a System, whereas *relationships* describe what the relationships are between various *blocks*.

Figure 5.10 shows a high-level meta-model of *block definition diagrams*.

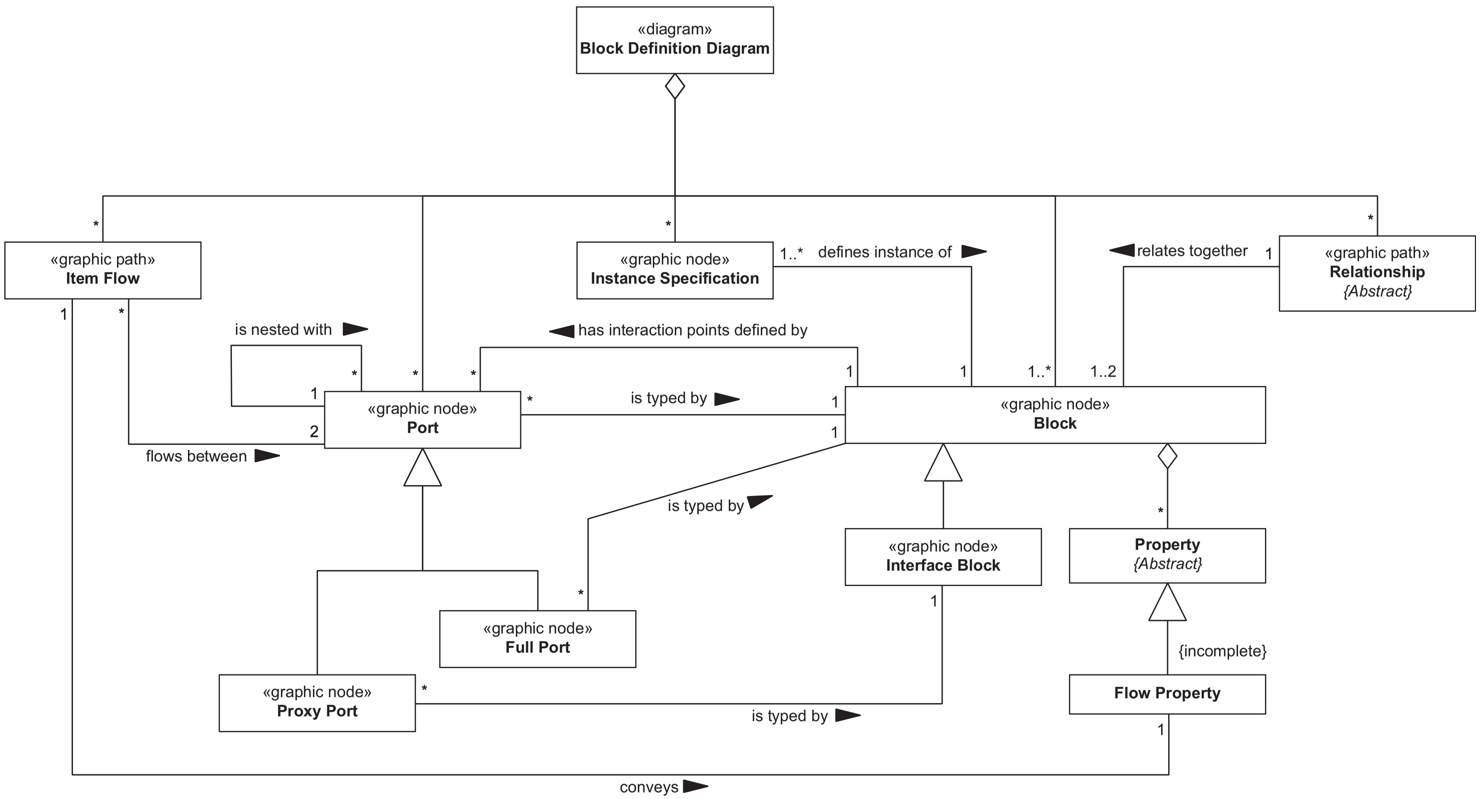

Figure 5.10   Partial meta-model for the block definition diagram

From Figure 5.10 we can see that a 'Block Definition Diagram' is made up of one or more 'Block', zero or more 'Relationship', zero or more 'Port', zero or more 'Item Flow' and zero or more 'Interface Specification'.

Each 'Relationship' relates together one or two 'Block'. Note that the multiplicity on the 'Block' side of the *association* is one or two, as it is possible for a 'Relationship' to relate together one 'Block' – that is to say that a 'Block' may be related to itself. A special kind of *block* is the 'Interface Block', used specifically to define Interfaces. An 'Instance Specification' defines an *instance* (real-world examples) of a 'Block'. Many such *instance specifications* may be defined for a 'Block'.

A 'Block' has interaction points defined by zero or more 'Port'. Each 'Port' is typed by a 'Block' and can be nested with zero or more other 'Port'. A 'Port' can be specialised further through two main sub-types:

- 'Full Port', used to represent an interaction point that is a separate element of the model. That is, a *full port* can have its own internal *parts* and behaviour.
- 'Proxy Port', used to represent an interaction point that identifies features of its owning *block* that are available to other, external *blocks*. They are not a separate element of the model and therefore do not specify their own internal *parts* and behaviour. Any such features and behaviour that they make available are actually those of its owning *block*. A 'Proxy Port' only be typed by an 'Interface Block'.

Neither *full ports* nor *proxy ports* have to be used. If it is unclear, when modelling, whether a *port* needs to be a *full port* or a *proxy port*, then leave it as a plain *port*. The decision whether to change to a *full* or *proxy port* can be made later as the model evolves.

Used in conjunction with the 'Port' is the 'Item Flow', which flows between two 'Port' and which conveys a 'Flow Property', a type of 'Property' of a 'Block' that is described below.

Each 'Block' is made up of zero or more 'Property', zero or more 'Operation' and zero or more 'Constraint' as shown in Figure 5.11.

The diagram in Figure 5.11 shows the partial meta-model for *block definition diagrams* showing the elements of a *block.* There are four types of 'Property':

- 'Part Property', which is owned by the 'Block'. That is, a *property* that is intrinsic to the *block* but which will have its own identity. A *part property* can be wholly owned by its parent *block* or may be shared between multiple parent *blocks*.
- 'Reference Property', which is referenced by the 'Block', but not owned by it.
- 'Value Property', which represents a 'Property' that cannot be identified except by the value itself, for example numbers or colours.
- 'Flow Property', which defines elements that that can flow to or from (or both) a *block*. They are mainly used to define the elements that can flow in and out of *ports* and all *item flows* that flow between *ports* are typed by *flow properties*.

Both an 'Operation' and a 'Property' (with the exception of a 'Flow Property') can be marked as being a 'Feature'. A *feature* is a *property* or *operation* that a *block* supports for other *blocks* to use (a 'Provided Feature') or which it requires other

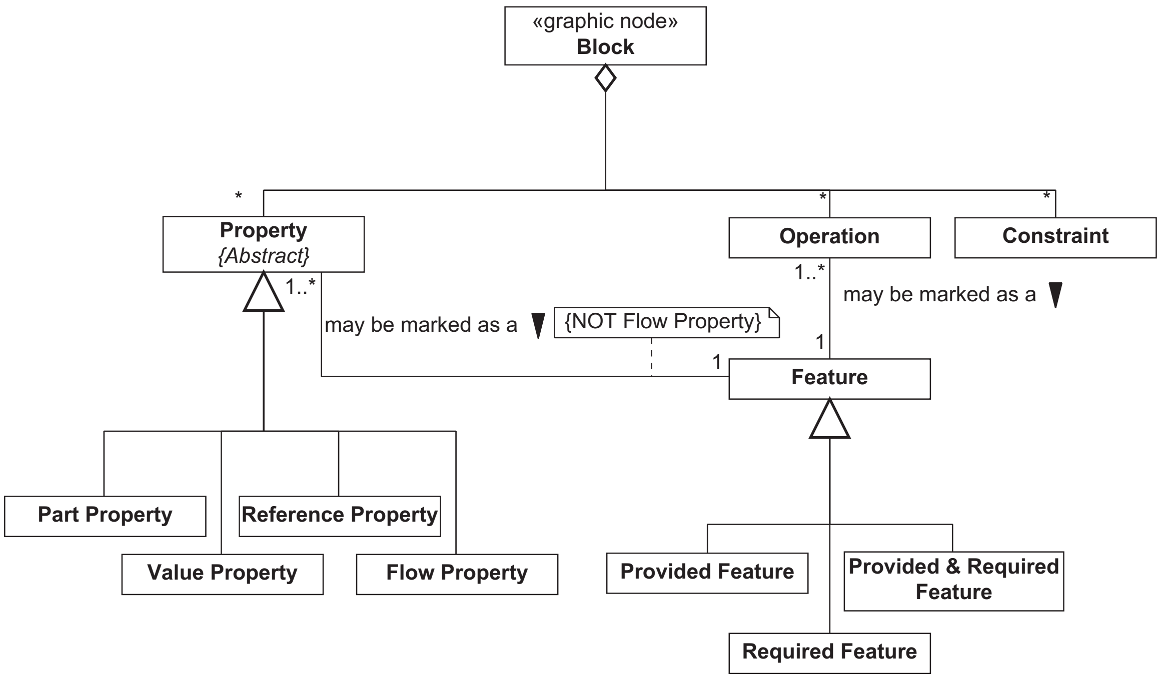

Figure 5.11 Partial meta-model for the block definition diagram showing block elements

blocks to support for its own use (a 'Required Feature'), or both (a 'Provide & Required Feature').

The differences between the first three types of *property* can be confusing. An example will help and is illustrated in Figure 5.12.

The *block definition diagram* in Figure 5.12(a) models the structure of the Coffin Escape stunt and the reader is directed to Figure 5.14 for a description of the notation. The diagram shows that the 'Coffin Stunt' is composed of a 'Reservoir', a 'Coffin', a 'Pump', a 'Hole', a Pump Controller', a 'Fluid' and an 'Escapologist'. The 'Fluid' has a 'Density', which will be represented as 'kg/m^3' (representing kilograms per cubic metre). The 'Fluid' is pumped into the 'Hole' via the 'Pump' and is supplied from the 'Reservoir'. Note the use of *role names* at the ends of the *composition* and *association relationships.*

The 'Density' is simply a number – it does not have any individual identity – and is therefore treated as a *value* property.

The 'Reservoir', 'Coffin', 'Pump', etc., are all intrinsic parts of the 'Coffin Escape'. That is, they can be thought of as having their own identity but form elements of the 'Coffin Escape'. Therefore, they are modelled as *part properties*, which is shown using *composition*. If a *part* can be an element of more than one owning *block* at the same time, then *aggregation* would be used rather than *composition*.

The 'Fluid' is not part of the 'Hole' or the 'Reservoir'. It is pumped into the former and supplied by the latter. It has its own identity. For this reason, it is related

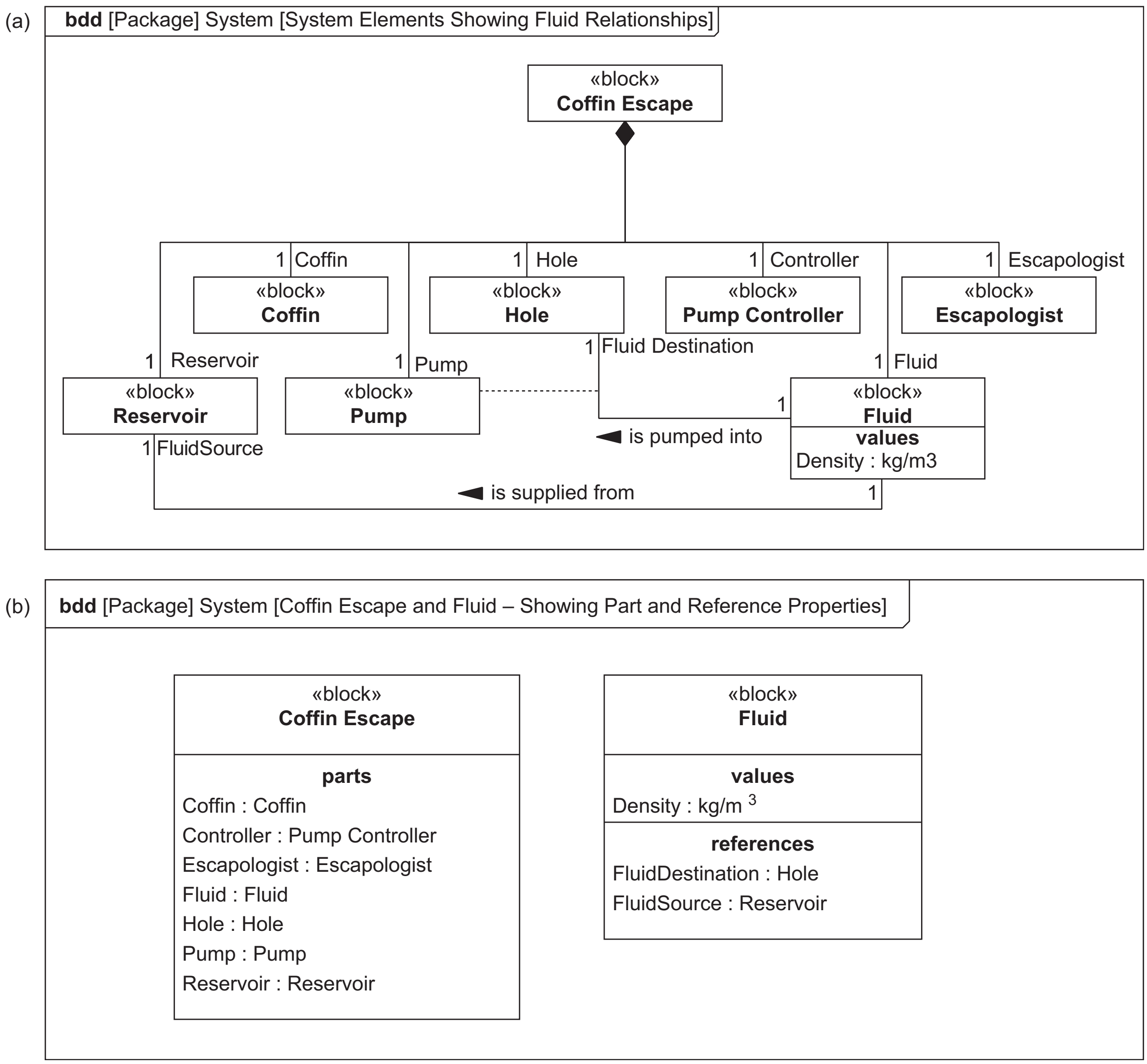

Figure 5.12 Types of property – alternative representations

to 'Hole' and to 'Reservoir' through *associations*. Any *block* related to another through an *association* can be considered to be a *reference* property of the *block* it is related to.

The nature of such *relationships* and the types of *property* they represent can be seen clearly in the *block definition diagram* in Figure 5.12(b). This shows exactly the same information but in a different format that uses named *property compartments* rather than via graphical paths and nodes. This shows how the various graphical representations can be rendered into a textual format. There are two things to note. First, the *role names* on the *relationships* are used to name the *properties* when displayed in *property compartments*. Second, in the case of *reference properties*, the *association name* ('is supplied from' or 'is pumped into' in the example above) does not form part of the information in the *property compartment*, which is a loss of information. The *property* compartment notation is more compact than the full *composition* and *association* notation, although perhaps not as clear; useful perhaps when producing summary diagrams.

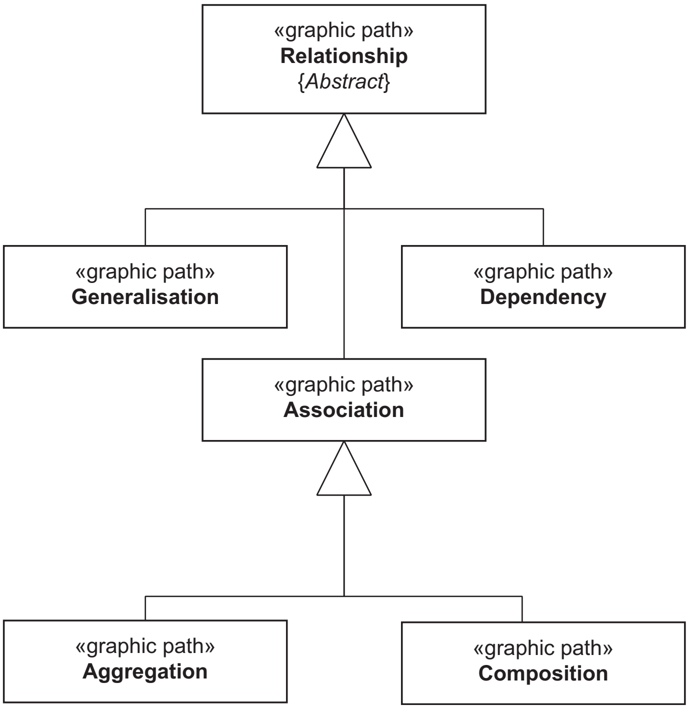

Figure 5.13 Partial meta-model for the block definition diagram showing types of relationship

Continuing our breakdown of the meta-model for the block definition diagram, there are three main types of 'Relationship' as shown in Figure 5.13:

- 'Association', which defines a simple *relationship* between one or more *blocks*. There are also two *specialisations* of 'Association' known as 'Aggregation' and 'Composition', which show *shared parts* and *owned parts* respectively, as discussed earlier in this section.
- 'Generalisation', which shows a 'has types' *relationship* that is used to show parent and child *blocks*.

- 'Dependency', which is used to show that one *block* (often referred to as the client) somehow depends on another *block* (often referred to as the supplier) such that a change to the supplier may impact the client. 'Dependency' can be considered to be the weakest of the *relationships* since it simply shows that there is some kind of (usually) unspecified *relationship* between the connected *blocks*.

A summary of the notation used in the *block definition diagram* is shown in Figure 5.14.

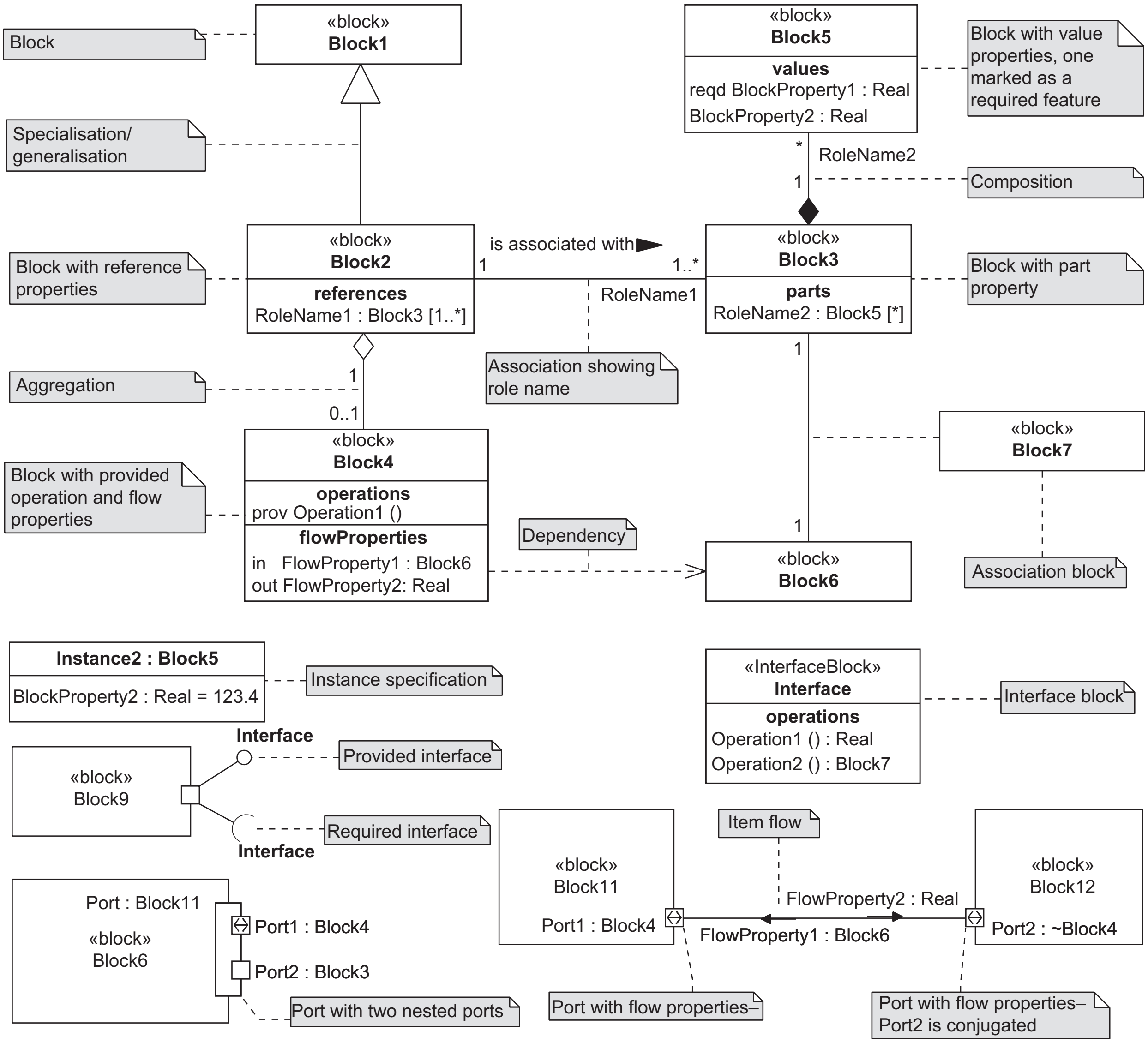

Figure 5.14 Summary of block definition diagram notation

The diagram in Figure 5.14 shows the graphical symbols used to represent elements in a *block definition diagram*. The basic symbol is the *block*, which is represented by a rectangle. Rectangles are also used to show other types of element in the SysML, so it is important to be able to differentiate between a *block* rectangle and any other sort of rectangle. A *block* rectangle will simply contain a single name, with no colons. It will also contain the *stereotype* «block».

When *properties*, *operations* and *constraints* are present, these are shown in *compartments* drawn underneath the block name, with the *properties*, *operations* and *constraints* contained within. Each of these *compartments* will be labelled to show what they contain, and the *property compartments* will be further sub-divided to show *part*, *reference*, *value* and *flow properties*.

Any *properties* or *operations* that are *features* are prefixed as shown in Table 5.1.

Table 5.1 Prefixes used with features

Type of feature	Prefix
Required	reqd
Provided	prov
Provided & Required	provreqd

Flow properties have their direction indicated with prefixes as shown in Table 5.2.

Table 5.2 Prefixes used with flow properties

Direction of flow property	Prefix
In	in
Out	out
In & out	inout

The *interfaces* are defined using special *blocks* that are *stereotyped* «interface» and which usually only have *operations*, but no *properties*. The *operations* represent the services provided by a *block* (or *port*) that has that *interface* as a *provided interface*, or the services required by a *block* (or *port*) that has it as a *required interface*. *Provided* and *required interfaces* can be shown graphically using a ball or cup notation respectively, labelled with the name of the *interface* and attached to the *block* or *port*. See for example 'Block9' in Figure 5.14.

Ports are shown as small squares (or rectangles) straddling the edge of the *block*. They can be labelled to give the *port* a name and to identify the *block* that types the *port.* For example, in Figure 5.14 'Block11' has a *port* with the name

'Port1', which is typed by 'Block4'. *Full* and *proxy ports* are indicated by placing the «full» or «proxy» *stereotype* next to the *port*.

Ports that have *flow properties* contain a small arrow showing the direction of the flow (whether into the *port*, out of the *port*, or both). See 'Port1' on 'Block11' in Figure 5.14 for an example of a *port* with *flow properties* that go both into and out of the *port*.

If a *port* has some *flow properties* that flow in and some that flow out, then when connected to another *port* it is necessary to show that these flows need to be shown in the opposite direction. For example, look again at 'Port1' on 'Block11'. This *port* is typed by 'Block4', which has two *flow properties*: 'FlowProperty1' flows in and 'FlowProperty2' flows out. This means that 'Port1' has the same *flow properties*, since it is typed by 'Block4'. However, now consider 'Port2' on 'Block12'. This is connected to 'Port1' on 'Block11' and, therefore, will have 'FlowProperty1' flowing out and 'FlowProperty2' flowing in; the opposite way round to how they have been specified in 'Block4'.

How do we resolve this? The answer is to make 'Port2' on 'Bock12' a *conjugated port*. This is indicated by the tilde "~" prefixing the name of the *block* typing the *port*: 'Port2: ~Block4'. The tilde reverses all the ins and outs prefixing the *flow properties* in the *block* that it prefixes. So, as far as 'Port2' is concerned, it has two *flow properties*: 'FlowProperty1', which flows out and 'FlowProperty2', which flows in. As the directions on the two ends now match up correctly, the *ports* can be connected and the flows shown using *items flows*.

Item flows are represented by a labelled triangle or a solid arrow attached to an *association*. The *item flow* can have a name by which it can be identified and is also labelled with the *property* that is transferred. This latter may appear at first to be redundant, as *item flows* connect *ports* that themselves are typed. However, SysML allows the modeller to differentiate between what may be transferred and what is transferred. The *type* of a *port* shows what may be transferred, with the *type* of an *item flow* showing what is transferred. However, the type of the *item flow* must be related to the type of the *port* by a specialisation–generalisation relationship. An example of this is given in the following section.

Instance specifications have a compartment that shows the *name* of the *instance specification* (so that multiple *instance specifications* of the same type can be differentiated) and the *block* that it is an instance of. This is underlined. For example, in Figure 5.14 there is an *instance specification* labelled 'Instance2 : Block5'. This *instance specification* has a name, 'Instance2' and is an instance of 'Block5'. An additional *compartment* can be shown, in which *properties* of the typing *block* may be given values for this instance. In this example, the *property* 'BlockProperty2' is given the value '123.4'.

5.5.1.2 Examples

This section presents some examples of *block definition diagrams* and related diagramming elements. Further examples will be found in the case study in Chapter 13.

Figure 5.15 shows the main structural elements for the Coffin Escape Stunt. It shows that there is a 'Coffin Escape' that is composed of a 'Reservoir', a 'Coffin',

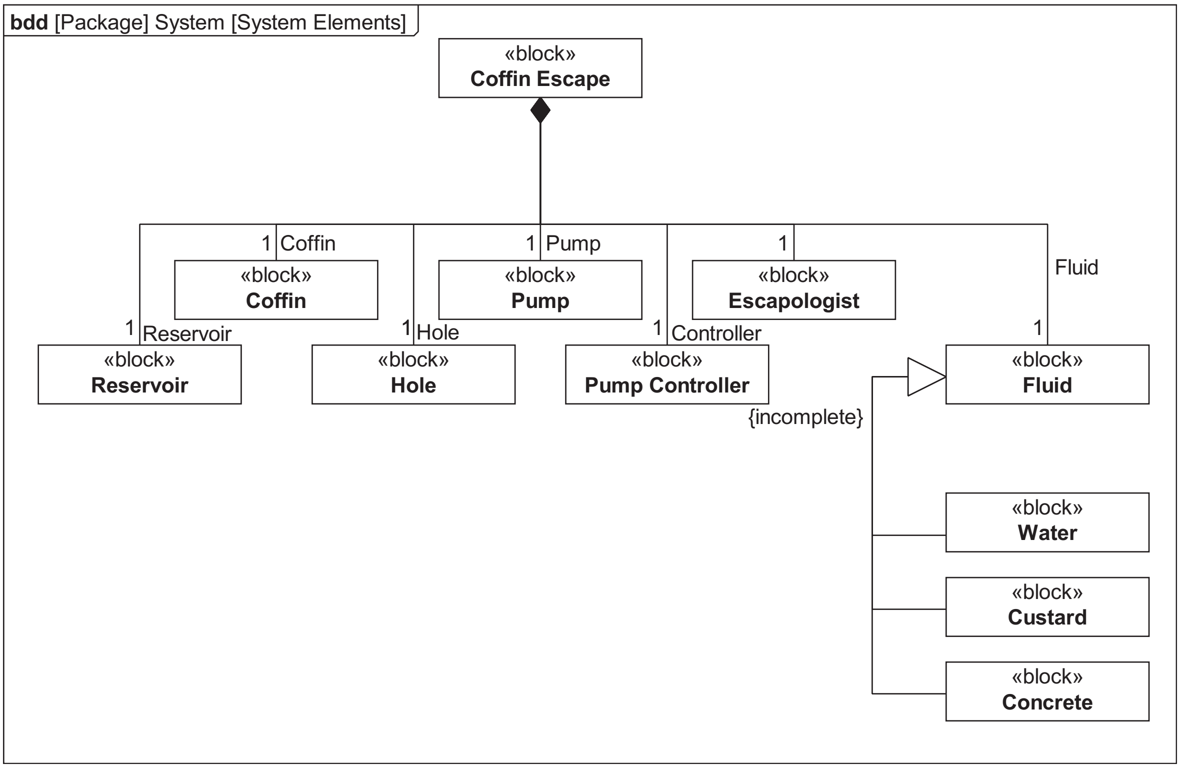

Figure 5.15 Example block definition diagram showing main structural elements of the Coffin Escape Stunt

a 'Hole', a 'Pump', an 'Escapologist' and a 'Fluid'. Three types of 'Fluid' are defined: 'Water', 'Custard' and 'Concrete'. The use of the {incomplete} *constraint* indicates that there may be additional types of 'Fluid' that are not shown in this diagram.

Note that there are no *properties* or *operations* defined for any of the *blocks* on the diagram, nor any *relationships*. This has been done deliberately in order to keep the diagram simple. This information is shown on additional *block definition diagrams*, starting with the one shown in Figure 5.16, which expands on the definition of 'Fluid'.

In Figure 5.16 the definition of 'Fluid' and its *sub-types* is expanded in order to show that 'Fluid' has a *value property* named 'Density'. Since 'Water', 'Custard' and 'Concrete' are all *sub-types* of 'Fluid' they inherit this *property*. SysML allows *value properties* to be given default values, as shown here.

Properties and *operations* of some of the other *blocks*, along with the *relationships* between them, are shown in Figure 5.17.

Figure 5.17 shows a lot more information about the various System Elements that make up the Coffin Escape System. We can see that an 'Escapologist' escapes

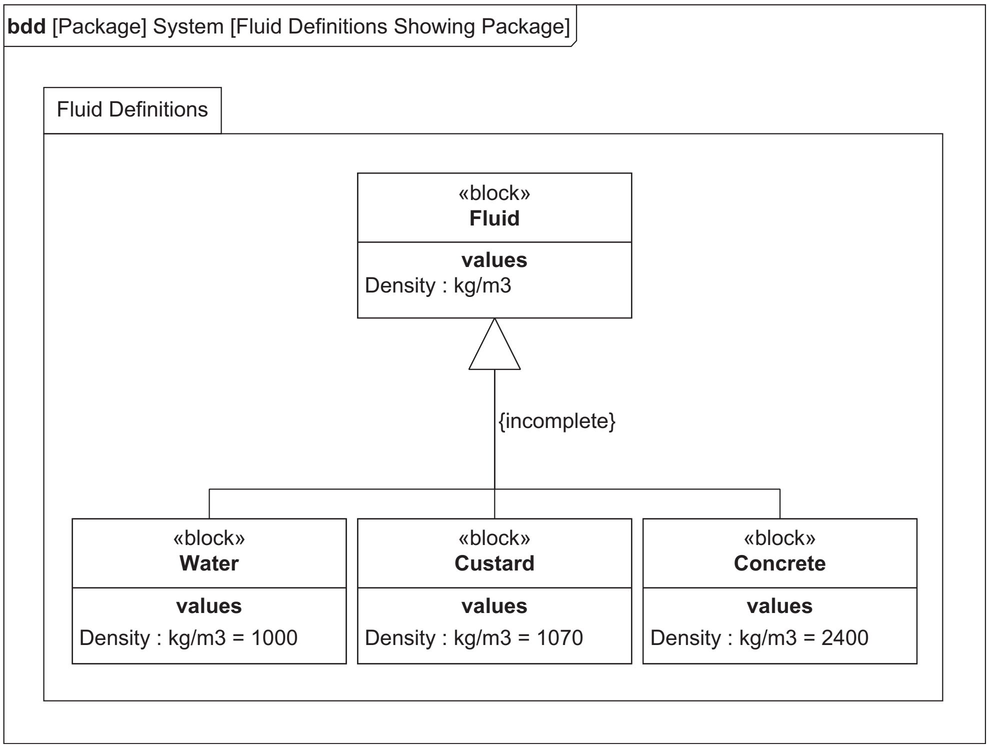

Figure 5.16 Example block definition diagram showing block properties with default values

from a ‘Coffin’ that is placed in the bottom of a ‘Hole’. A ‘Pump Controller’ controls a ‘Pump’. ‘Fluid’ is pumped into the ‘Hole’ via the ‘Pump’. This latter aspect of the model is captured through the use of an *association block*: the ‘Pump’ *block* is connected to the *association* between ‘Fluid’ and ‘Hole’ with a dashed line, making ‘Pump’ an *association block*. It is a *block* in its own right, but adds information to the *association.* A maximum of one *block* can act as an *association block* on any given *association*.

Many of the *blocks* have *value properties* that help to define them further and ‘Pump’ has a number of *operations* that show the behaviour that it can carry out. Both ‘Hole’ and ‘Pump’ have two *ports* defined. These *ports* have been shown using a *port compartment* and shown textually rather than graphically. This has been done simply to reduce visual clutter on the diagram.

It is worth considering these *ports* in a little more detail. ‘Pump’ has a *port*, ‘pOut’, that is typed by the *block* ‘FluidFlow’ (see Figure 5.18).

This ‘FluidFlow’ *block* defines a single *flow property*: ‘out fluid : Fluid’. This says that elements typed by ‘FluidFlow’ will have a single *flow property*, of type ‘Fluid’, flowing out of them. This agrees with the definition of the *port* ‘pOut’, since this *port* has the *out flow direction* prefixed. ‘Pump’ has another *port* defined, ‘pIn’.

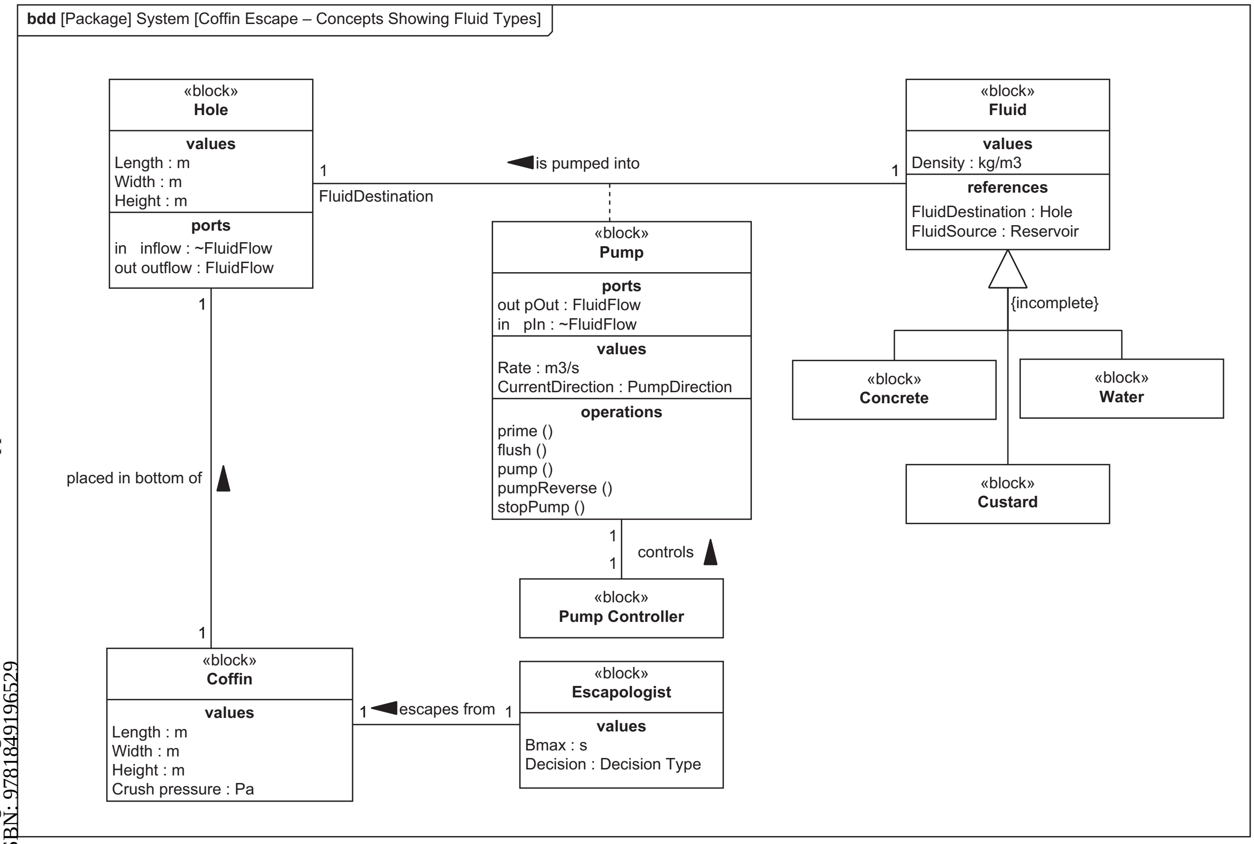

Figure 5.17 Example block definition diagram showing properties, operations and relationships

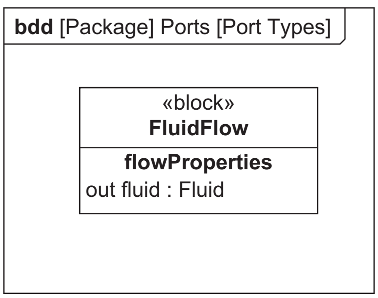

Figure 5.18 Example block definition diagram defining type of ports through use of flow properties

The intention is that this *port* takes in whatever *flow properties* are defined by 'FluidFlow'. However, if it was defined as 'in pIn : FluidFlow' then we would have a consistency issue. The *port* is marked with an *in flow direction* but the *flow property* in its *type* has an *out flow direction*. The solution is to make 'pIn' a *conjugated port*. This has been done through the use of a tilde in the definition: 'in pIn : ~FluidFlow'. The directions of the *flow properties* defined in 'FluidFluid' are now reversed as far as 'pIn' is concerned. A similar discussion holds for the *ports* of 'Hole'. The notation and use of *conjugated ports* perhaps makes more sense when they are shown connected together. An example will be shown in Section 5.5.2.

A final point to make about Figure 5.17 concerns the *reference compartment* in the 'Fluid' *block*. This shows two *reference properties*. Remember that these correspond to *associations* that the *block* is involved in. One of these is shown on the diagram, as can be deduced via the *role name* 'FluidDestination' on the *association* between 'Fluid' and 'Hole'. The other *reference property* corresponds to an *association* that is not shown. We can deduce from the *reference property* that 'Fluid' has an *association* with a *block* called 'Reservoir' and that the *role* that 'Reservoir' plays in the *association* is that of 'FluidSource'. For completeness, this association is shown explicitly in Figure 5.19.

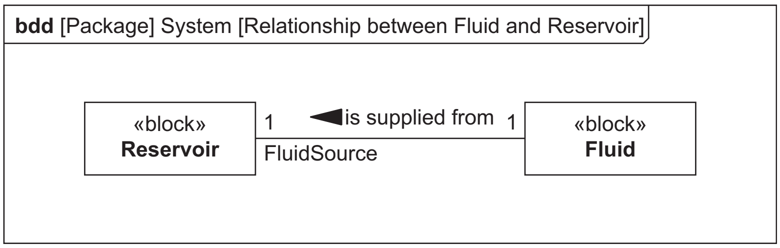

Figure 5.19 Example block definition diagram showing a reference property explicitly as an association

Figure 5.19 illustrates an important point when modelling: don't be afraid to limit what you show on a diagram. SysML tools make the consistent creation of diagrams quick and easy, provided of course that they are a robust and sharp tool (see Chapter 16 for a discussion of tools). If information is best omitted from one diagram, then do so. You can always create another diagram that does show the information.

As two final examples of a *block definition diagram* in this section, consider Figures 5.20 and 5.21.

The *blocks* in Figure 5.20 do not represent items of hardware or software or material, etc., but rather they represent Source Elements for Need Descriptions, produced as part of a requirements engineering activity. The diagram *frame* uses the *frame tag* 'SEV' to show that this *block definition diagram* is being used as a Source Element View. For a discussion of model-based requirements engineering and the ACRE Framework from which the concept of a Source Element View is taken, see Chapter 9.

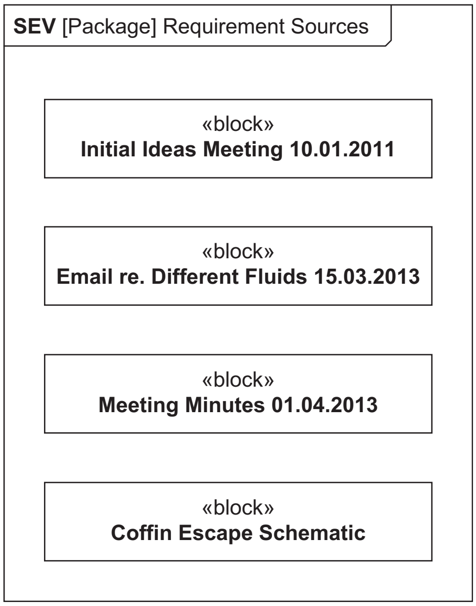

Figure 5.20 Example block definition diagram used to model Source Elements of Requirements

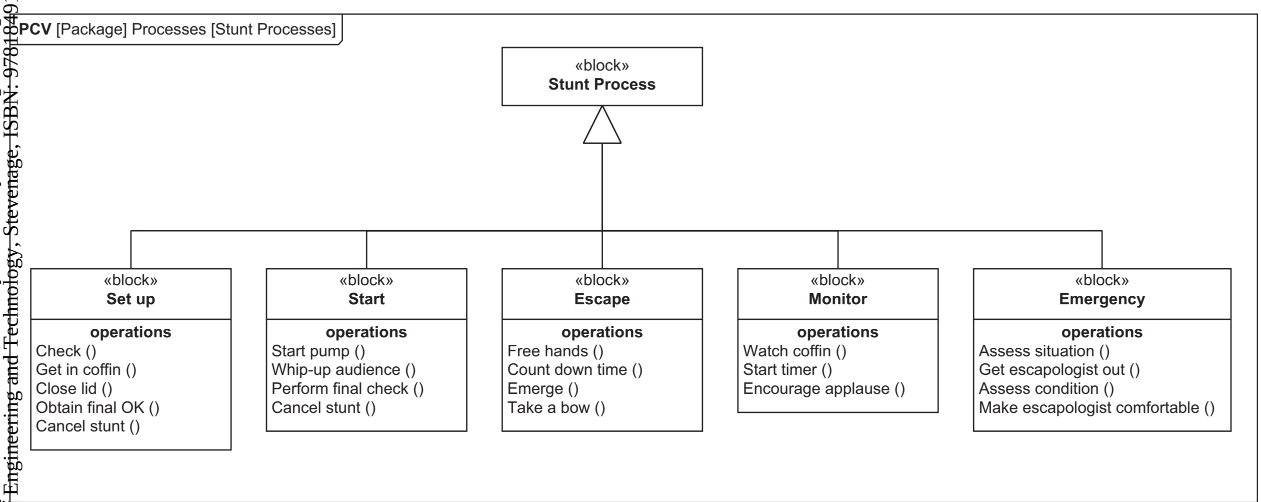

Figure 5.21 Example block definition diagram used to model Processes

The *blocks* in Figure 21 also do not represent items of hardware or software or material, etc., but rather they represent Processes, produced as part of a Process modelling activity. The diagram *frame* uses the *frame tag* ‘PCV’ to show that this *block definition diagram* is being used as a Process Content View. For a discussion of a model-based approach to Process modelling and the "seven views" Framework from which the concept of a Process Content View is taken, see Chapter 7.

5.5.1.3 Summary

Block definition diagrams can be used to model just about anything and form the backbone of any SysML model. *Block definition diagrams* are perhaps the richest in terms of the amount of syntax available and, as with all the meta-models in this chapter, the one given for *block definition diagrams* is incomplete. For example, it could be extended to include extra detail that can be added to relationships, such as *role names* and *qualifiers*.

The main aim of the *block definition diagram*, as with all SysML diagrams, is clarity and simplicity. *Block definition diagrams* should be able to be read easily and they should make sense. A diagram that is difficult to read may simply indicate that there is too much on it and that it needs to be broken down into a number of other diagrams. It may also be an indication that the modelling is not correct and that it needs to be revisited. Another possibility is that the diagram is revealing fundamental complexity inherent in the System, from which lessons may be learned.

Another fundamental point that must be stressed here is that *block definition diagrams* are not used in isolation. They will form the main structural aspect of a System but must be used in conjunction with the other eight SysML diagrams to provide structural and behavioural views of a System. These diagrams are described in the rest of this chapter.

5.5.2 Internal block diagrams

Internal block diagrams are used to model the internal structure of a *block* (hence the name). By using an *internal block diagram*, in which *compositions* and *aggregations* are implicitly represented by the containment of *parts* within the owning *block* or within other *parts*, an emphasis may be put on the logical relationships between elements of the *composition*, rather than the structural breakdown itself. This adds a great deal of value, as it forces the modeller to think about the logical relationship between elements, rather than simply which *blocks* are part of which other *blocks*.

5.5.2.1 Diagram elements

The basic element within *an internal block diagram* is the *part* that describes *blocks* in the context of an owning *block*. An *internal block diagram* identifies *parts* and their internal structures, showing how they are connected together through *ports* and showing the *item flows* that flow between *parts.*

The diagram in Figure 5.22 shows the partial meta-model for the *internal block diagram*. It can be seen that a 'Internal block diagram' is made up of one or more 'Part', zero or more 'Port' and zero or more 'Binding Connector' and zero or more 'Item Flow'.

A 'Port' defines an interaction point for a 'Part', just as they do for *blocks* (see Section 5.5.1.1) and again come in two types: 'Full Port' and 'Proxy Port'. A 'Part'

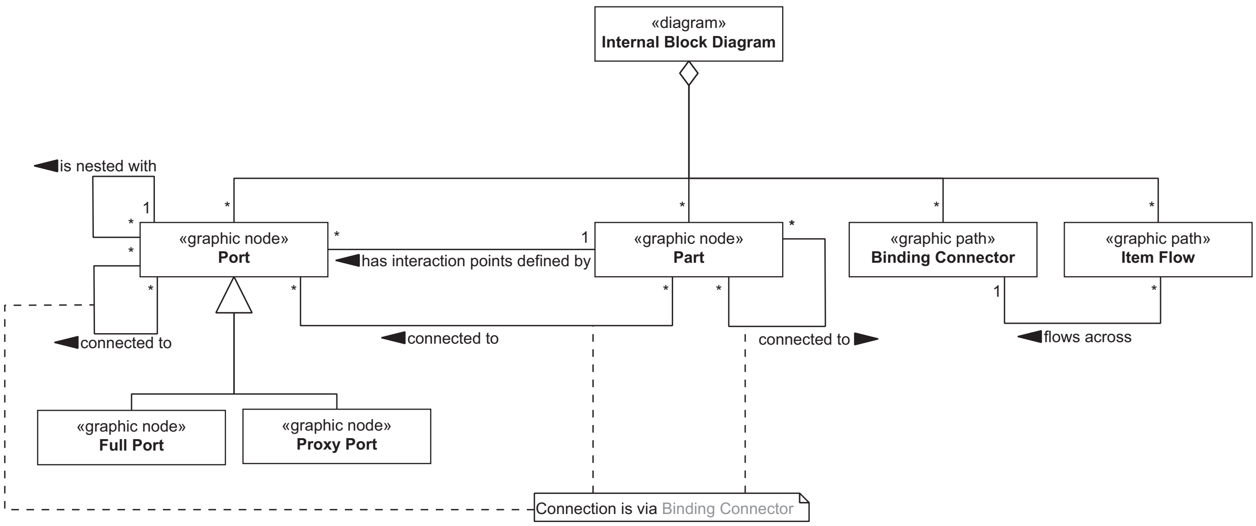

Figure 5.22 Partial meta-model for the internal block diagram

can be directly connected to zero or more 'Part' via a 'Binding Connector'. This connection may also be from a 'Part' to the 'Port' on another 'Part'. A 'Port' may also be connected to zero or more 'Port'. An 'Item Flow' can flow across a 'Binding Connector'.

The intention in the SysML specification seems to be that these connections should be shown only on an *internal block diagram*, with a *block definition diagram* showing the *ports* on a block, but not the connections between them. For this reason the *block definition diagram* meta-model in Section 5.5.1.1 omits such connection possibilities, but the authors see no reason why the same types of connection should not be shown on a *block definition diagram*.

The diagram in Figure 5.23 shows the notation used on an *internal block diagram*. Much of the notation is the same as can be found on a *block definition diagram* and will not be discussed further. However, some notational and usage points do need discussion, namely:

- The relationship between *internal block diagrams* and *block definition diagrams*, and hence that of *parts* and *blocks*
- The notation for *parts*
- *Shared parts*

Before looking at the notation for *parts*, let us first consider the relationship between *internal block diagrams* and *block definition diagrams*, and hence that of *parts* and *blocks*. The first thing to say is that an *internal block diagram* is owned by a *block*. It is used, when a *block* is composed of other *blocks*, to represent that composition in an alternative fashion and to allow the modeller to concentrate on

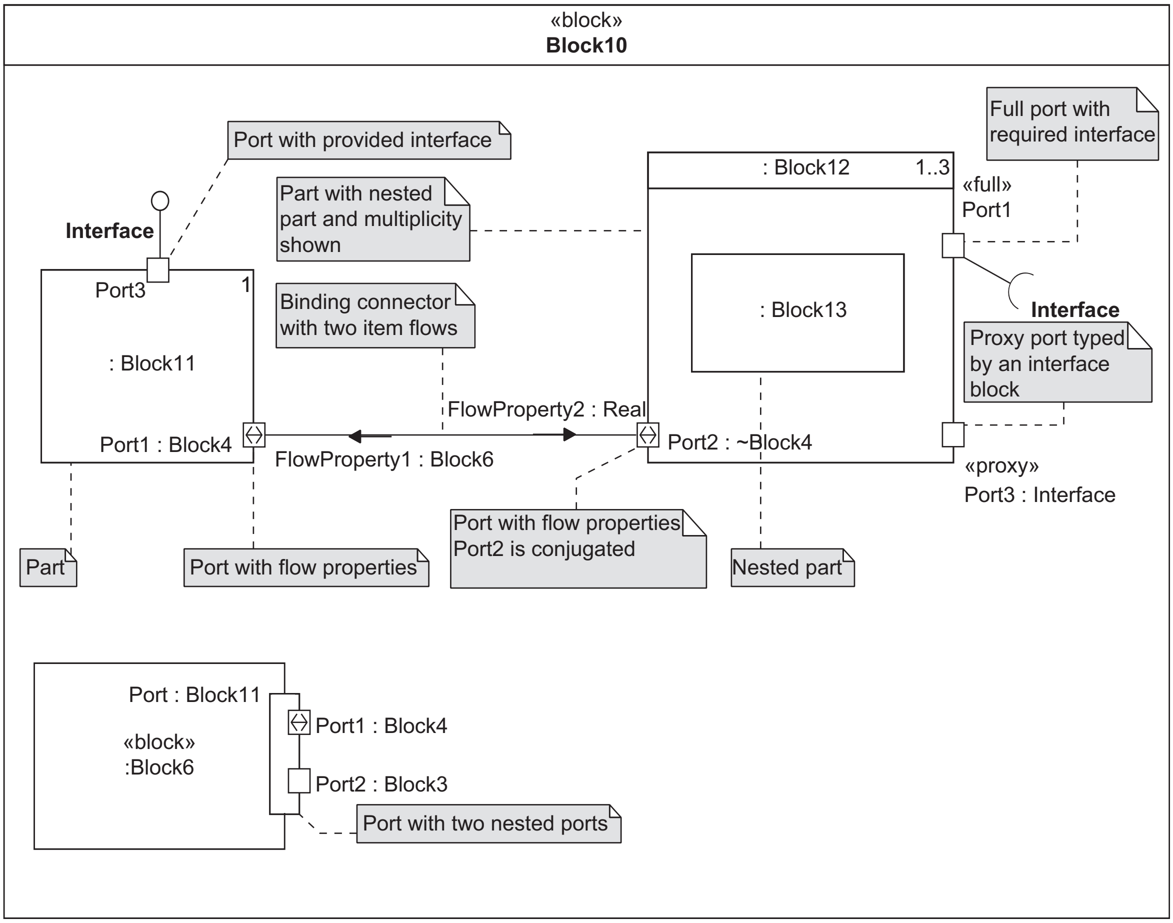

Figure 5.23 Summary of internal block diagram notation

the connections between the *blocks* rather than on the composition. From a *block* that is decomposed into sub-blocks it is possible to automatically create an *internal block diagram* for that *block* and, indeed, many SysML tools will do this for you. The *internal block diagram* in Figure 5.23 has been created for ‘Block10’, based on the *block definition diagram* in Figure 5.24.

The *internal block diagram* in Figure 5.23 is owned by ‘Block10’ and can be thought of as being inside, or internal to (hence the name) ‘Block10’. ‘Block10’ is shown as containing *block* with the *blocks* that it is composed of shown as *parts*. (Note that the *ports* in Figure 5.23 could have also been shown in Figure 5.24 but have been omitted for clarity). So, a *block* that is composed of *sub-blocks*, as detailed on a *block definition diagram*, can have its internal structure modelled on an *internal block diagram* owned by the *block*. The *blocks* that it is composed of are shown as *parts* on the *internal block diagram.*

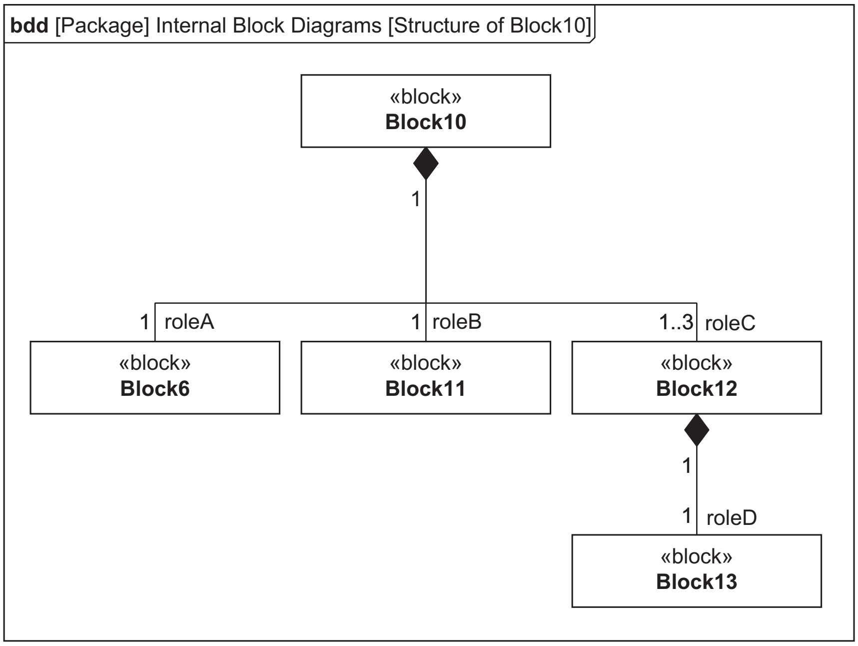

Figure 5.24 Example block definition diagram used to show its relationship to internal block diagram

This brings us on to the second point, namely the notation used for *parts*. *Parts* are represented using a rectangle that contains the name of the *part*. The name has the form:

Part name : Type Name

The *part name* serves as an identifier for the *part* and the *type name* shows the *block* that the *part* is a type of. The *part name* can be omitted if distinction between different *parts* of the same type is not necessary. The *type name* can also be omitted, but this is less common. In Figure 5.23 each *part* has its *part name* omitted and its *type name* shown. Where the *block* involved in a composition has a *role name* associated with it, then the *part name* is usually directly related to the *role name*. See Figure 5.25.

Figure 5.25 is another *internal block diagram* for 'Block10' from Figure 24, but this time with all *ports*, *interfaces* and *connectors* omitted. Also, on this diagram *part names* are shown and their relationship to the *role names* in Figure 5.24 can also be seen.

Two other points can also be seen in both Figures 5.23 and 4.24. First, if a *part* has a *multiplicity* greater than one, this is shown in the top right corner of the *part*. This can be seen for the *part* typed by 'Block12' in both diagrams, where the *multiplicity* '1..3' is shown in the top right corner of the *part*. Second, *parts* can also

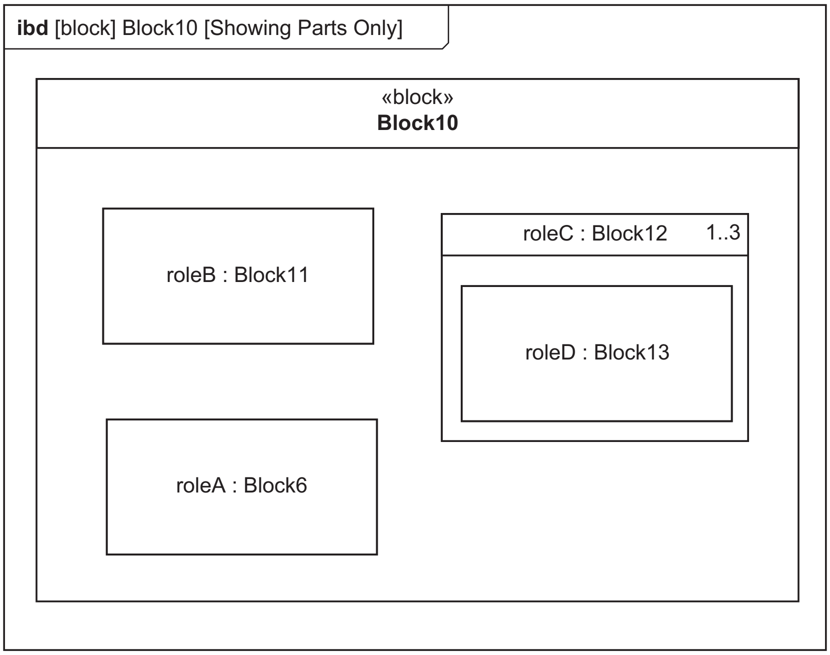

Figure 5.25 Example internal block diagram used to show its relationship to block definition diagram

be nested and this can be seen for the *part* 'roleD : Block13', which is shown inside the *part* 'roleC : Block12'. This corresponds to the *composition relationship* between 'Block12' and 'Block13'. This *composition relationship* also means that 'Block12' could have its own *internal block diagram*, which would have a single *part*, 'roleD : Block13'. For completeness, refer Figure 5.26.

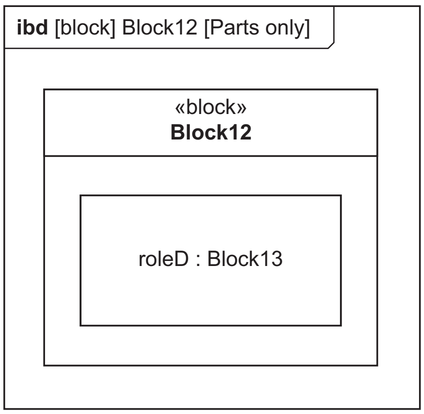

Figure 5.26 Example internal block diagram for Block12

Finally, let us consider *shared parts*. As was discussed briefly in Section 5.5.1.1, the decomposition of a *block* can be shown on a *block definition diagram* using a *composition* or an *aggregation*. A *block* may be wholly owned by its parent *block* (shown using *composition*) or may be shared between multiple parent *blocks* (shown using *aggregation*). The use of *composition* or *aggregation* has an effect on the way that *parts* are shown. An example will help.

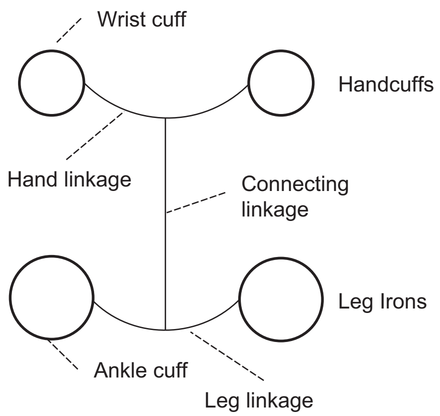

Figure 5.27 Example system schematic showing owned and shared parts

The non-SysML diagram in Figure 5.27 shows the restraints worn by the escapologist as part of the Coffin Escape Stunt: a set of handcuffs and a set of leg irons joined by a connecting linkage. The structure can be modelled using a *block definition diagram* as shown in Figure 5.28.

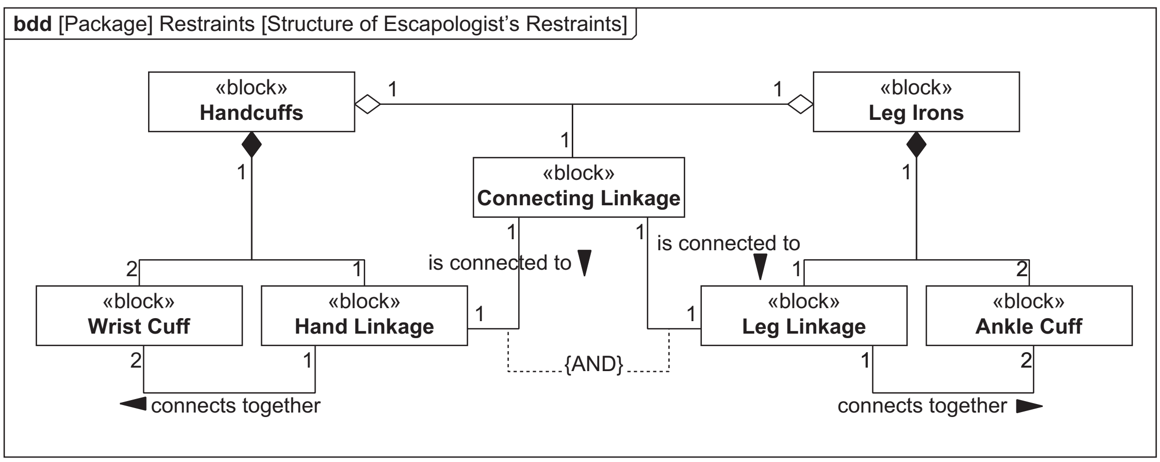

Figure 5.28 Example block definition diagram showing modelling of owned and shared parts

The restraints consist of a set of 'Handcuffs' composed of two 'Wrist Cuff' connected together by a 'Hand Linkage' and a set of 'Leg Irons' composed of two 'Ankle Cuff' connected together by a 'Leg Linkage'. The 'Hand Linkage' and the 'Leg Linkage' are connected together by a 'Connecting Linkage'. Since the 'Wrist Cuff' and 'Hand Linkage' are only part of the 'Handcuffs', *composition* is used. Similarly for the 'Ankle Cuff' and 'Leg Linkage'. However, the 'Connecting Linkage' is shared between both the 'Handcuffs' and the 'Leg Irons'. For this reason, *aggregation* is used. This has a direct effect on the notation used in the *internal block diagrams* for the 'Handcuffs' and the 'Leg Irons'. The *internal block diagram* for the 'Handcuffs' is shown in Figure 5.29. That of the 'Leg Irons' would be similar.

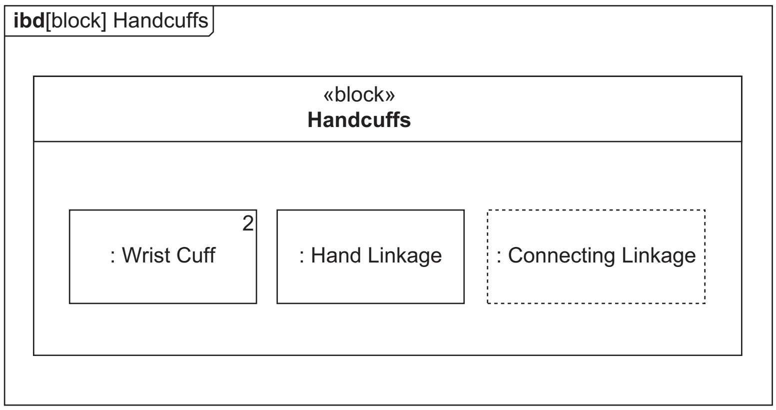

Figure 5.29 Example internal block diagrams showing owned and shared parts

The difference in notation for *shared parts* can be seen in the *internal block diagram* in Figure 5.29. A *shared part* is shown with a dashed outline. Note also the *multiplicity* in the top right of the 'Wrist Cuff' *part.* Also, note that it is not possible to tell from this diagram what else the 'Connecting Linkage' is shared with. The *block definition diagram* in Figure 5.28 is needed for this.

5.5.2.2 Examples

This section presents some examples of *internal block diagrams* and related diagramming elements. Further examples will be found in the case study in Chapter 13.

Having defined the structure of the 'Coffin Escape' stunt in section 5.5.1 on *block definition diagrams* (see Figures 5.15 and 5.17), an *internal block diagram* can be used to explore the interfaces between the System Elements of the 'Coffin Escape'. This has been done in Figure 5.30.

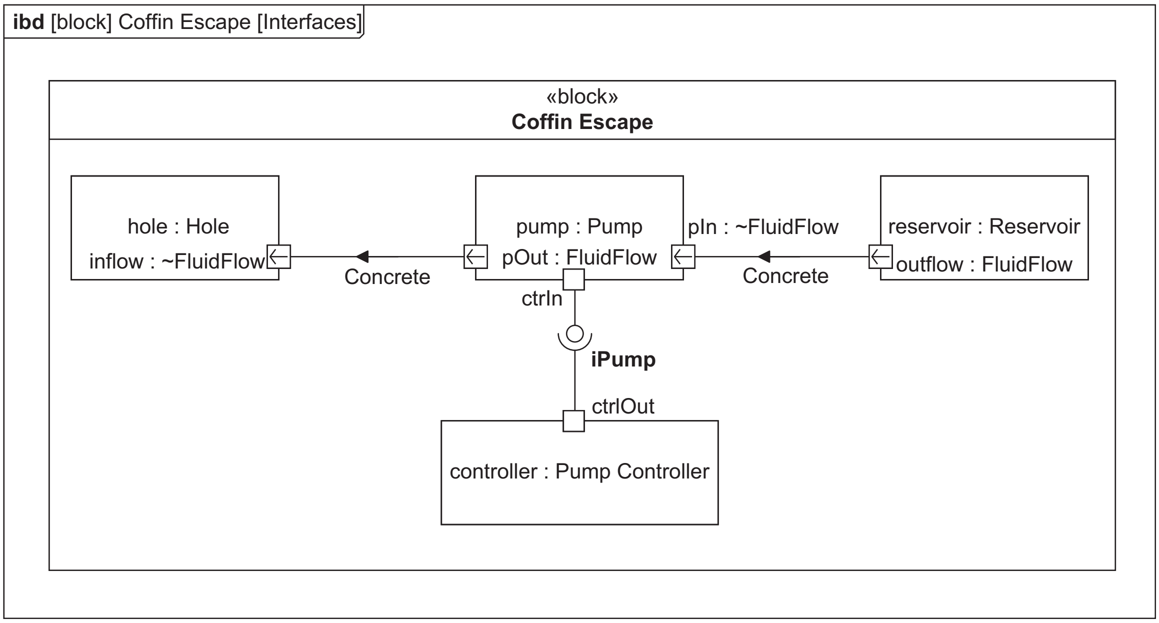

Figure 5.30 Example internal block diagram showing main Interfaces of the Coffin Escape Stunt

Figure 5.30 shows an *internal block diagram* for the 'Coffin Escape' *block.* The *parts* shown on this diagram can be populated automatically from the structural information, shown using *composition*, in Figure 5.15. Note, however, that the 'Escapologist', 'Coffin' and 'Fluid' *blocks* have not been shown in Figure 5.30. This is because, as indicated in the *diagram frame*, this *internal block diagram* has been produced to show interfaces between the main system elements. This again reinforces the point that in SysML you should be producing diagrams for a specific purpose. You do not have to try to show everything on a single diagram, nor should you try to.

Whereas Figure 5.17 implicitly indicated the various *ports* and their connections, through the use of *port compartments* on the *blocks*, these connections have been made explicit in Figure 5.30. This is a very common use of the *internal block diagram*.

There are two points worth discussing further on this diagram. The first concerns the nature of the two *item flows* on the diagram and the second that of the interface between the 'Pump Controller' and the 'Pump'.

In Figure 5.17 there are a number of *ports* defined on the 'Pump' and 'Hole' blocks. Each of these is typed by the 'FluidFlow' *block* that has a single *flow property* typed by the 'Fluid' *block*. 'Reservoir' has a similar *port* but it is not shown in Figure 5.17. In Figure 5.30 these *ports* have been connected together with *binding connectors* carrying *item flows*. The 'outflow' *port* of 'Reservoir' sends an *item flow* to the 'pIn' *port* of 'Pump', which in turn sends and *item flow* from its 'pOut' *port* to the 'inFlow' *port* of 'Hole'. The direction of each of the *item flows* honours the direction of each *port* and of the *flow property* defined by 'FluidFlow'.

However, whereas 'FluidFlow' defines a *flow property* of type 'Fluid', the *item flow* shows 'Concrete' flowing between the *ports*. Do we have an inconsistency here? The answer is no, because 'Concrete' is a type of 'Fluid', as can be seen in Figure 5.15. This is an important point, and one that makes *item flows* and *flow properties* useful. It is possible, through *flow properties*, to define the type of things that can flow between *ports* and keep this at a rather general level of abstraction (e.g. 'Fluid'). Then, through *item flows*, it is possible to show what actually does flow in a particular usage of the various *blocks*. Although the 'Pump' modelled in Figures 5.15 and 5.17 can pump a number of types of 'Fluid', when it is being used in the 'Coffin Escape', as shown in Figure 5.30, it will be used to pump 'Concrete'. The *type* of the *item flow* has to be the same as, or a *sub-type* of, the *type* of its defining *flow property*. The *flow property* is of *type* 'Fluid' and the *item flow* is of *type* 'Concrete', which is a *sub-type* of 'Fluid', so this is allowed.

The second point to discuss is the interface between the 'Pump Controller' and the 'Pump'. This connection is explicitly shown in Figure 5.30, where the 'Pump' has a *provided interface* of *type* 'iPump' and where the 'Pump Controller' has a *required interface* of the same *type*. These are shown connected together and the *type* of the *interface*, 'iPump', is also shown. This *interface* has not yet been defined. Its definition is made on a *block definition diagram* using an *interface block*, as shown in Figure 5.31.

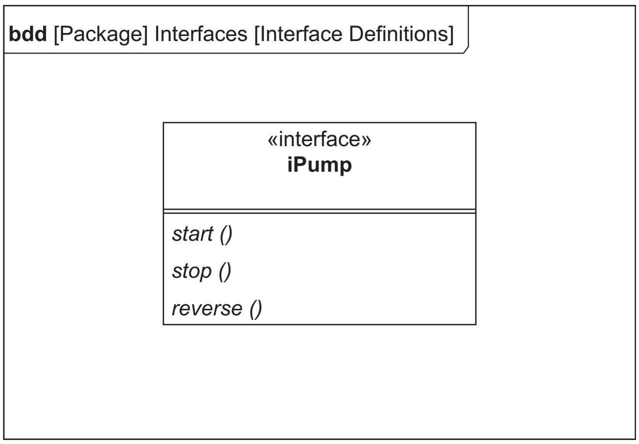

Figure 5.31 Example block definition diagram defining the iPump interface

Figure 5.31 defines a single *interface*, 'iPump', which has three *operations* 'start()', 'stop()' and 'reverse()'. In the SysML model from which this diagram is taken, each of these three *operations* would have a full description of their expected behaviour, both in text, as part of their definition in the 'iPump' *interface block*, and possibly also in SysML using an *activity diagram*. Although Figure 5.31 only defines a single *interface block*, there is no reason why other *interface blocks* could not be defined on the same diagram; the SysML does not require them to be defined on separate diagrams.

When connecting a *required interface* to a *provided interface* it is important that the *types* of the *interfaces* match (i.e. that they are defined by the same *interface block*). Actually, there is a little more flexibility allowed: the *type* of the *provided* interface must be the same as, or a *sub-type* of, the *type* of the *required interface*. This works because when a *sub-type* is defined, the *sub-type* can add additional *operations* but cannot remove any. Consider Figure 5.32.

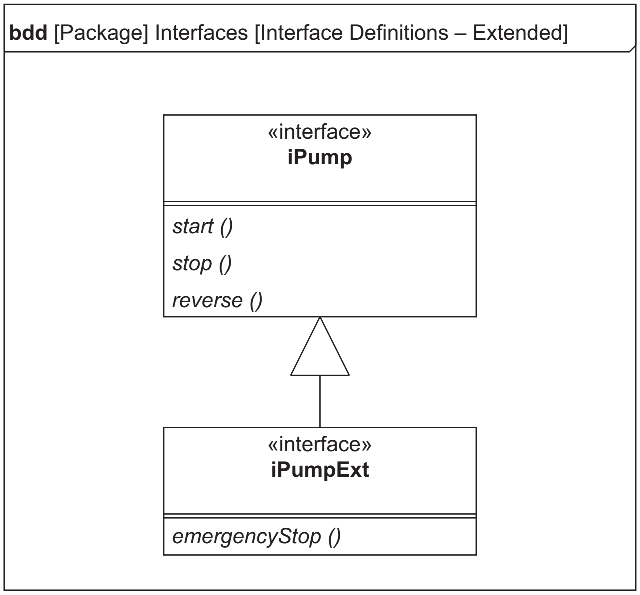

Figure 5.32 Example block definition diagram showing extended iPump interfaces

A new *interface,* 'iPumpExt', is defined in Figure 5.32. This defines a new *operation*, 'emergencyStop()'. Since 'iPumpExt' is a *sub-type* of 'iPump' it also inherits all three *operations* that are defined for 'iPump'.

Now imagine that 'Pump' in Figure 5.30 has a *provided interface* that is of *type* 'iPumpExt' rather than 'iPump'. The *required interface* on 'Pump Controller' can still be connected to this *provided interface* because 'iPumpExt' provides all the *operations* that 'iPump' did (and that are required by the 'Pump Controller'), plus one more. It happens that 'Pump Controller' will never require the use of this additional *operation*, which is okay.

However, if the *required interface* on 'Pump Controller' was of *type* 'iPumpExt' and the *provided interface* on 'Pump' was of *type* 'iPump', then the connection could not be made. This is because 'Pump Controller' requires the use of the 'emergencyStop()' *operation* defined on 'iPumpExt'. However, this is not present in the 'iPump' *interface* provided by 'Pump'.

Internal block diagrams can also be used with *association blocks*, since an *association block* is, in effect, simply a *block* connected to an *association*. This is useful when the *association block* is being used to model a connector between two physical System Elements, as shown in Figure 5.33.

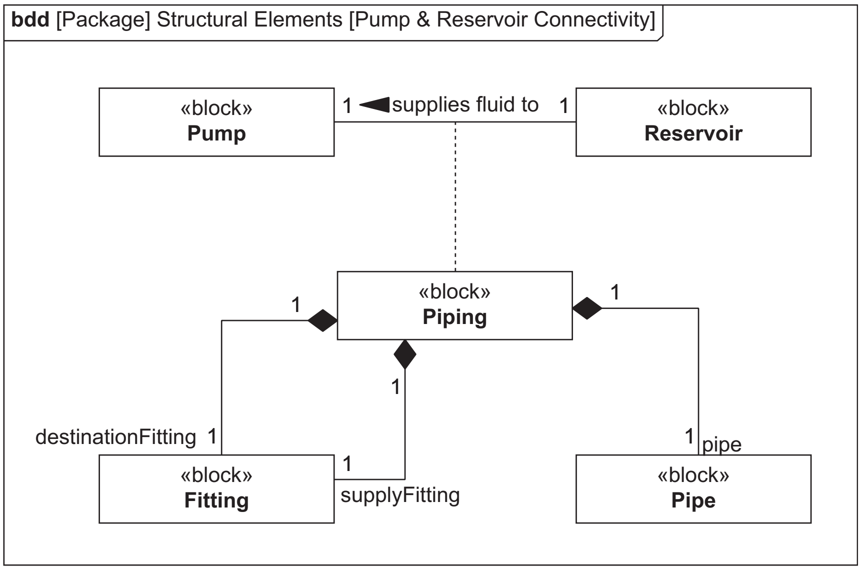

Figure 5.33 Example block definition diagram showing connectivity using an association block

In Figure 5.33, the connectivity between the 'Reservoir' and the 'Pump' is modelled using an *association block*, 'Piping', which is composed of a length of 'Pipe' and two 'Fitting', one for each end. The way that the 'Piping' is assembled is modelled using an *internal block diagram*, shown in Figure 5.34.

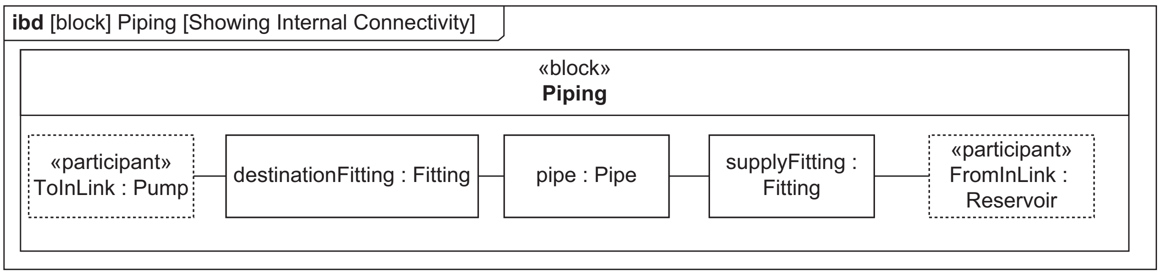

Figure 5.34 Example internal block diagram showing structure of Piping

In Figure 5.34 the *parts* from which 'Piping' is composed are shown connected using *binding connectors*. The diagram also shows two *shared parts*: 'FromInLink : Reservoir' and 'ToInLink : Pump'. These *shared parts* actually represent the ends of the *association* between 'Reservoir' and 'Pump' for which 'Piping' acts as an *association block*.

As a final example, consider Figure 5.35 that shows two *internal block diagrams* that concentrate on the 'Power Supply Unit' used in the Coffin Escape Stunt to power the 'Pump'.

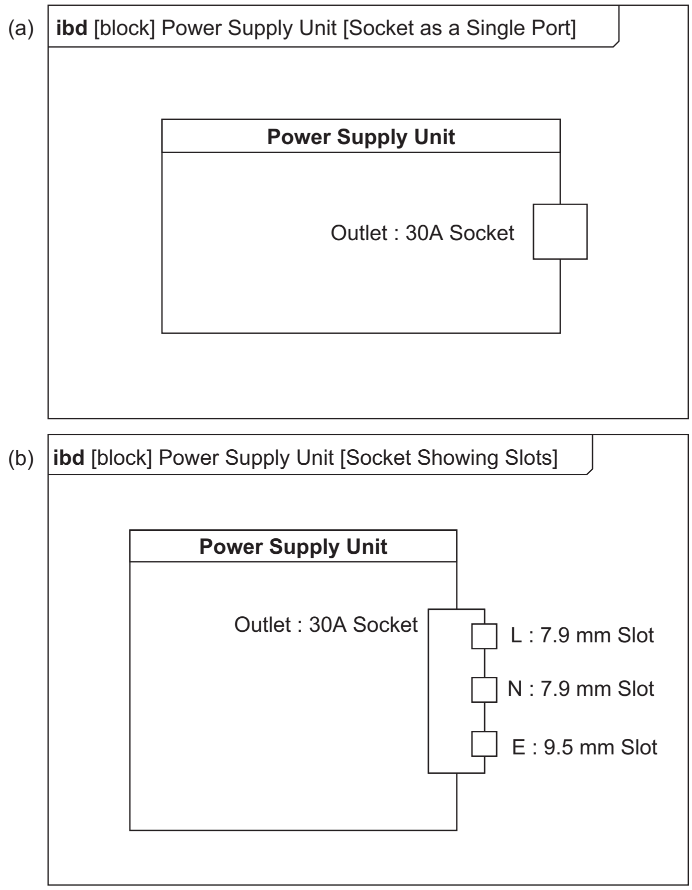

Figure 5.35 Example internal block diagram showing nested ports

In the *internal block diagram* (Figure 5.35(a)) the 'Power Supply Unit' is shown as having a 30A outlet, modelled as a *port* with the name 'Outlet : 30A Socket'. Nothing of the structure of the socket is shown here. This is fine, as long as this is the level of abstraction that is needed in the model. The *port* can be connected to another *port* representing a 30 A plug, for example, and a single *item flow* defined that connects them representing the transfer of AC current at 30 A.

However, it might be the case that the socket (and any associated plug) needs to be modelled at a lower level of abstraction. This is done in the *internal block diagram* in Figure 5.35(b), where the three slots making up the socket are shown explicitly using three *nested ports*. The 30A plug could be modelled in the same way, showing each pin using a *nested port*. Each pin and slot could then be connected individually, with the high-level *item flow* decomposed into three separate *item flows*, one connecting each pin and slot pair. This is left as an exercise for the reader.

5.5.2.3 Summary

The *internal block diagram* is very strongly related to the *block definition diagram*, using *parts* to show the structure of a complex *block*. This allows the emphasis of the diagram to be placed more on the logical relationships between elements of the *block*, rather than identifying that they are actually elements of a particular *block* (using relationships such as *aggregation* and *composition*). The way that the various *parts* are connected, through the use of *ports*, *interfaces* and *binding connectors*, and the items that flow between *parts*, through the use of *item flows*, can also be shown. The diagram also allows a distinction to be made between *parts* that are wholly owned by a parent *block*, and those that are *shared parts*, which are shared among multiple *blocks.*

5.5.3 Package diagrams

The *package diagram*, as the name implies, identifies and relates together *packages*. *Packages* can be used on other diagrams as well as on the *package diagram*; in both cases the concept of the *package* is the same – each *package* shows a collection of *diagram elements* and implies some sort of ownership. *Packages* can be related to each other using a number of different *dependency relationships*.

5.5.3.1 Diagram elements

The syntax for the *package diagram* is very simple and can be seen in Figure 5.36.

The diagram in Figure 5.36 shows the partial meta-model for the 'Package Diagram'. It can be seen that there are two main elements in the diagram – the 'Package' and the 'Dependency'. There is one type of 'Dependency' defined – the 'Package Import'. The 'Package Import' has two types, the 'Public Package Import' and the 'Private Package Import'.

The graphical notation for the *package diagram* is shown in Figure 5.37.

The diagram in Figure 5.37 shows that there are really only two symbols on the diagram: the graphical node representing a *package* and the graphical path representing a *dependency*.

A *package* is represented by a rectangle with a smaller tag rectangle on the top left-hand edge. This is similar to the folder icon that can be seen in Windows systems and, indeed, has a very similar conceptual meaning. The name of the *package* can either be shown in the tag (as seen here) or, in the case of long names, will often be shown inside the main rectangle.

The *dependency* may appear as an unadorned, regular *dependency*, or may appear with one of two *stereotypes* – «import» or «access» – representing a *public package import* or *private package import* respectively.

A *package import* (of either type) means that the *package* being pointed to (target) is imported into the other *package* (source) as part of the *source package*,

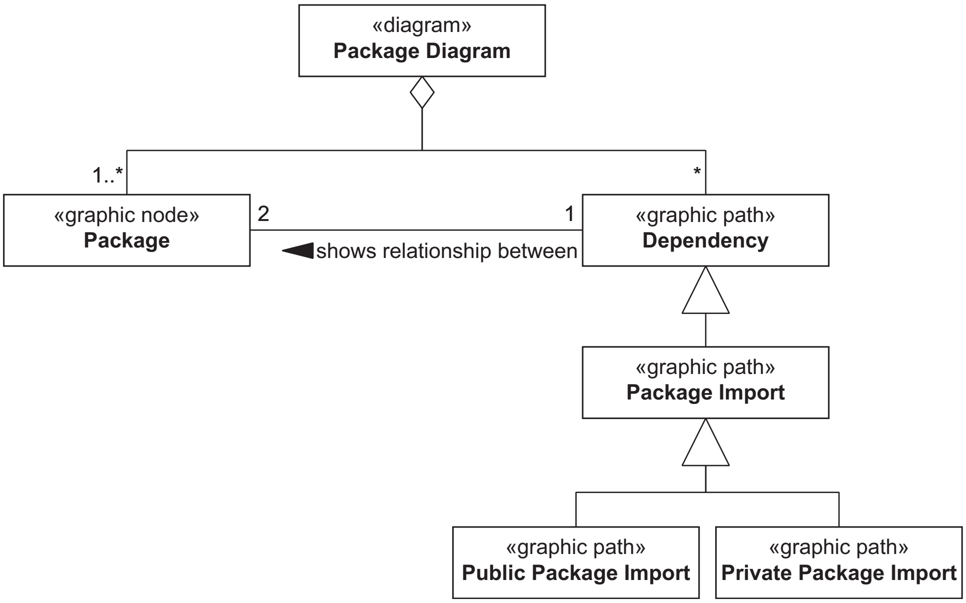

Figure 5.36 Partial meta-model for the package diagram

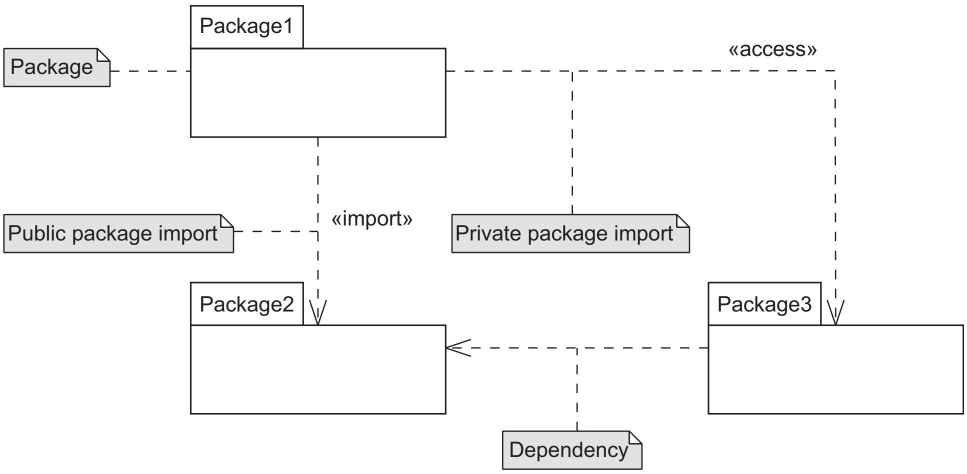

Figure 5.37 Summary of package diagram notation

but with the *target package* remaining its own *package*. Any name clashes are resolved with the *source package* taking precedence over the *target package*.

Public package import and *private package import* differ in the *visibility* of the information that is imported. What does this mean? Consider the two examples in Figure 5.38.

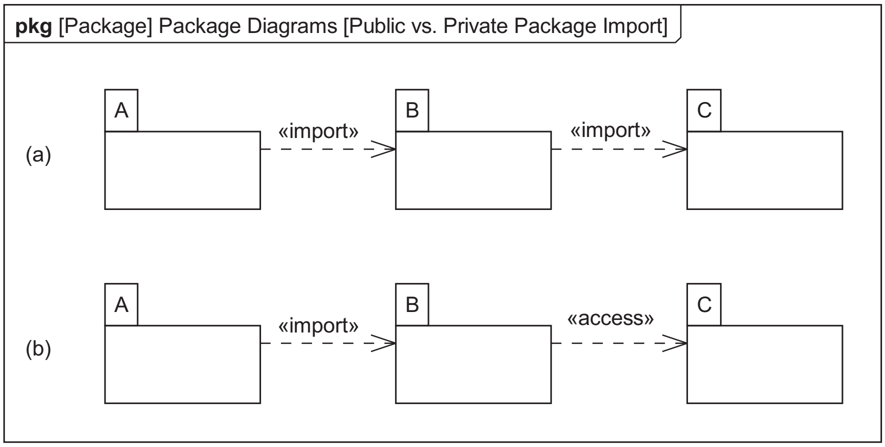

Figure 5.38 Importing packages using «import» and «access»

In example (a) *package* 'B' imports the contents of *package* 'C' using a *public package import*. *Package* 'A' then imports the contents of *package* 'B' using a *public package import*. Since 'A' has imported 'B' and 'B' has publicly imported 'C', *package* 'A' can also see the contents of *package* 'C'.

In example (b) *package* 'B' imports the contents of *package* 'C' using a *private package import*. *Package* 'A' then imports the contents of *package* 'B' using a *public package import*. Since 'A' has imported 'B' and 'B' has *privately* imported 'C', *package* 'A' cannot see the contents of *package* 'C', although it can see the contents of *package* 'B'.

Packages are used to structure a model in exactly the same way the folders (directories) organise files on a computer. Figure 5.39 helps to show how this is achieved.

The diagram in Figure 5.39 shows that a 'Package' is made up of a number of 'Packageable Element'. In the SysML, almost anything can be enclosed within a *package*, so only a few examples are shown here (indicated by the *{incomplete} constraint*). Note that a 'Package' is itself a 'Packageable Element' and thus a *package* can contain other *packages*.

5.5.3.2 Examples

Package diagrams are typically used to show model structure and *relationships* within a model at a very high level. *Packages* are often also shown on other SysML diagrams to provide information on where in a model the diagram elements can be found. Some examples are given in Figure 5.40.

The diagram in Figure 5.40 shows three *packages* from the escapology stunt. Part of the model for this stunt represents the Life Cycle Model for the Project. This is contained in the 'Life Cycle Model' *package*. This *package* makes use of the Students Managing Projects Intelligently (STUMPI) Processes, contained in the

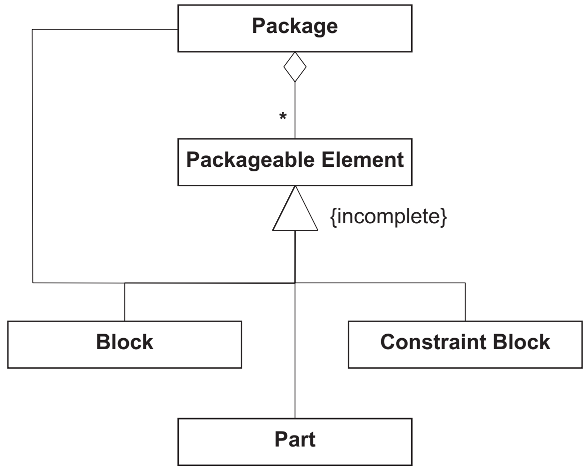

Figure 5.39 Relationships between package diagram elements and the rest of the SysML

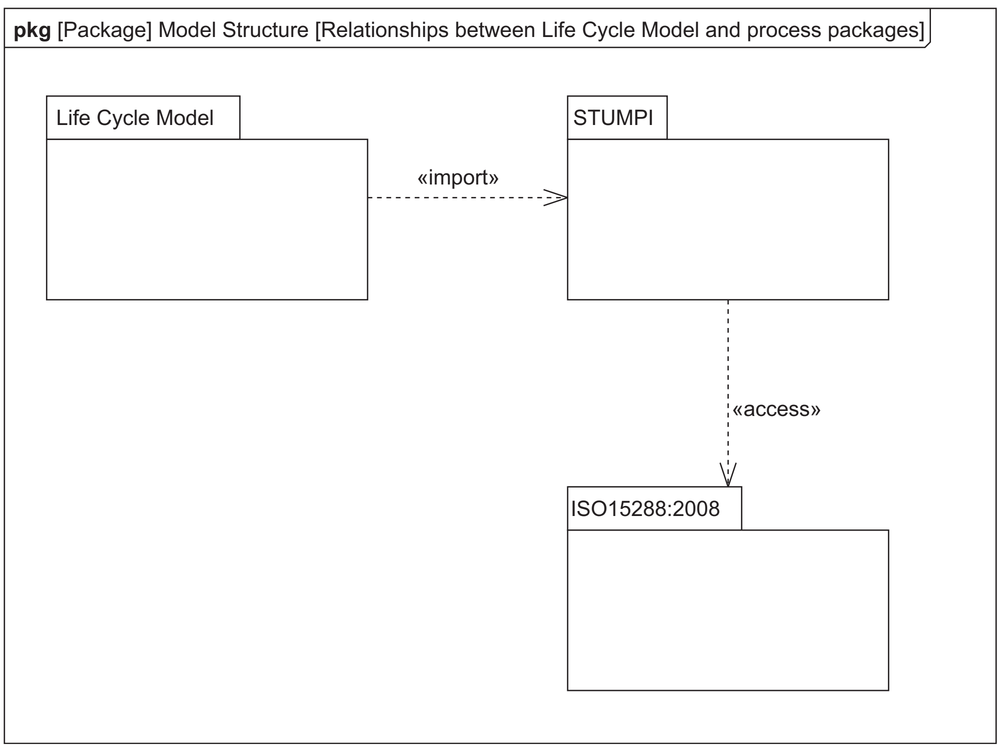

Figure 5.40 Example package diagram showing relationships between model packages

'STUMPI' *package*. Information in this *package* is visible inside the 'Life Cycle Model' *package*, as indicated by the *public package import dependency*. The STUMPI Processes themselves make use of the ISO15288:2008 process model, contained in the 'ISO15288:2008' *package* and imported using a *private package import dependency*. This means that the contents of 'ISO15288:2008' are visible within 'STUMPI' but not visible within 'Life Cycle Model'.

Packages are often shown on other diagrams. An example of this is shown in Figure 5.41.

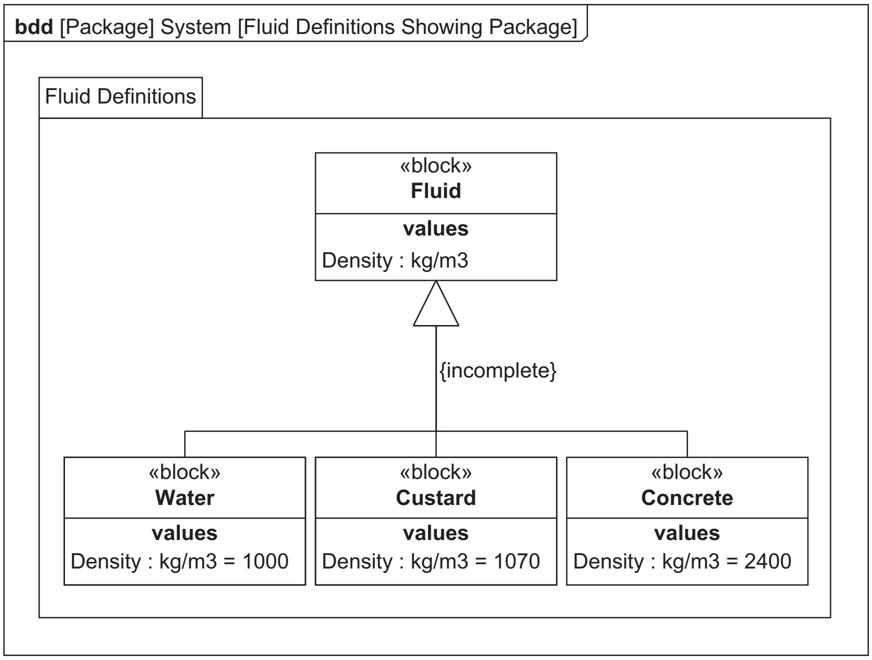

Figure 5.41 Example block definition diagram showing a package

Figure 5.41 shows a *block definition diagram* that is displaying a number of different types of 'Fluid'. From the *diagram frame* it can be seen that the *block definition diagram* is located in a *package* named 'System'. The diagram also shows a *package* named 'Fluid Definitions' surrounding the 'Fluid' *block* and its three *sub-types*. This has been done to make it explicit to the reader of this diagram that the 'Fluid', 'Water', 'Custard' and 'Concrete' *blocks* are not contained directly in the 'System' *package* but rather can be found in the 'Fluid Definitions' *package* within the 'System' *package*.

In practice, *package diagrams* are not that widely used. The use of *packages* on other diagrams is more common where it is useful for the modeller to be able to make explicit the location within a model of the diagram elements appearing on a diagram.

5.5.3.3 Summary

Package diagrams are useful for showing aspects of a model's structure where it is necessary to make clear how one *package* uses information from another (essentially how one *package* depends on another).

Packages are used within a SysML tool to structure a model. They can also be shown on any SysML diagram to indicate where particular diagram elements can be found in the model. However, such use must be tempered with the need to maintain readability of a diagram. *Packages* should be used in this way when necessary, but not as a matter of course lest the diagrams become too cluttered to be readable.

5.5.4 Parametric diagrams

The SysML *constraint block* and associated *parametric diagram* allow for the definition and use of networks of constraints that represent Rules that constrain the properties of a System or that define rules that the System must conform to.

5.5.4.1 Diagram elements

Parametric diagrams are made up of three main elements, *constraint blocks, parts* and *connectors* as shown in Figure 5.42.

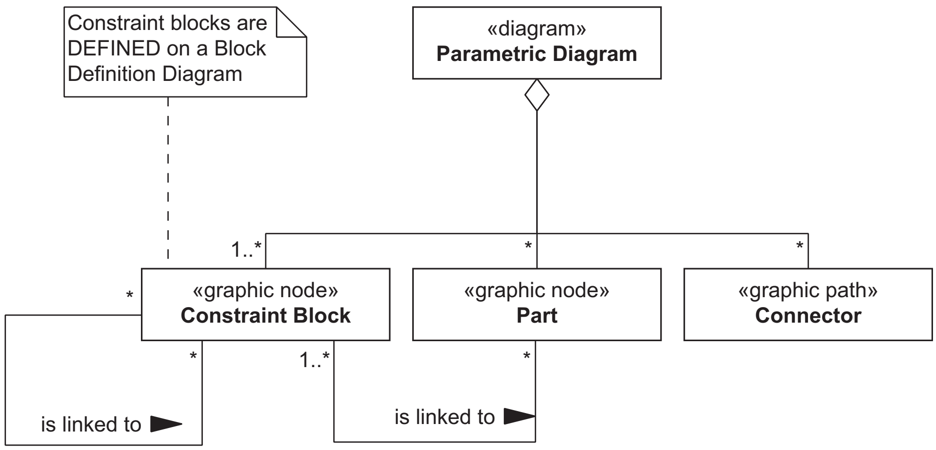

Figure 5.42 Partial meta-model for the parametric diagram

Figure 5.42 shows the partial meta-model for *parametric diagrams*. From the model it can be seen that a 'Parametric Diagram' is made up of one or more 'Constraint Block', zero or more 'Part' and zero or more 'Connector'. Zero or more 'Constraint Block' can be connected to zero or more 'Constraint Block' and one or more 'Constraint Block' can be connected to zero or more 'Part'. Although used on a 'Parametric Diagram', a 'Constraint Block' is defined on a 'Block Definition Diagram'.

There are two aspects to parametric constraints in SysML: their definition and their usage. The notations for both aspects are show in Figures 5.43 and 5.45 respectively.

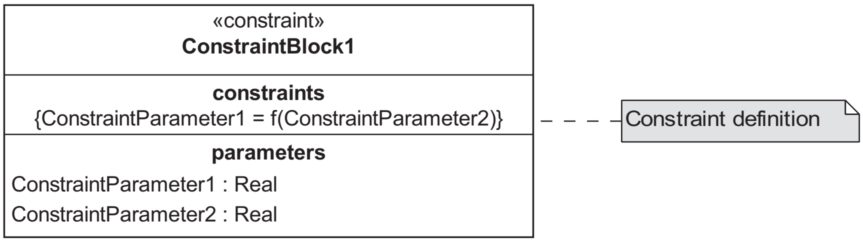

Figure 5.43 Summary of parametric diagram notation – definition of constraint block

A *constraint block* is defined using a *block* with the «constraint» *stereotype* and is given a name by which the constraint can be identified. The *constraint block* has two compartments labelled 'constraints' and 'parameters'. The *constraints compartment* contains an equation, expression or rule that relates together the *parameters* given in the *parameters compartment*. Figure 5.43 defines a *constraint block* called 'ConstraintBlock1' with two *parameters* 'ConstraintParameter1' and 'ConstraintParameter2', both of which are defined to be of type 'Real'. These *parameters* are related together by the expression 'ConstraintParameter1 = f(ConstraintParameter2)', with 'f' representing a function taking 'ConstraintParameter2' as a *parameter.*

Such *constraint blocks* are defined on a *block definition diagram*. A concrete example of a *constraint block* can be seen in Figure 5.44.

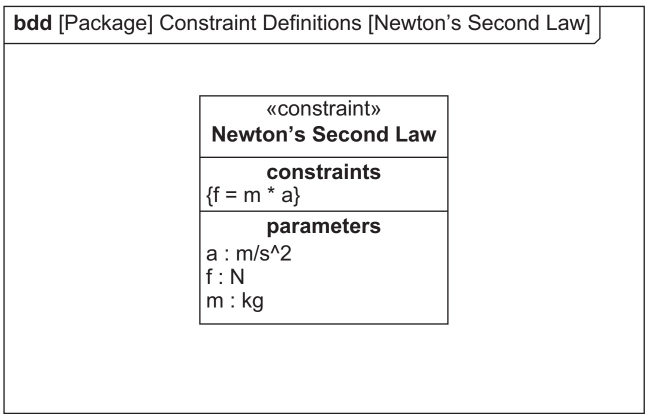

Figure 5.44 Example block definition diagram showing constraint block definition

The example in Figure 5.44 defines a *constraint block* called 'Newton's Second Law' that relates the three *parameters* 'f', 'm' and 'a' given in the *parameters compartment* by the equation 'f = m * a', as shown in the *constraints compartment*.

Although *constraint blocks* are defined on *block definition diagrams*, it is convention that such definitions are not mixed with regular *blocks* on the same diagram.

Once *constraint blocks* have been defined they can be used any number of times on one or more *parametric diagrams*, the notation for which is shown in Figure 5.45.

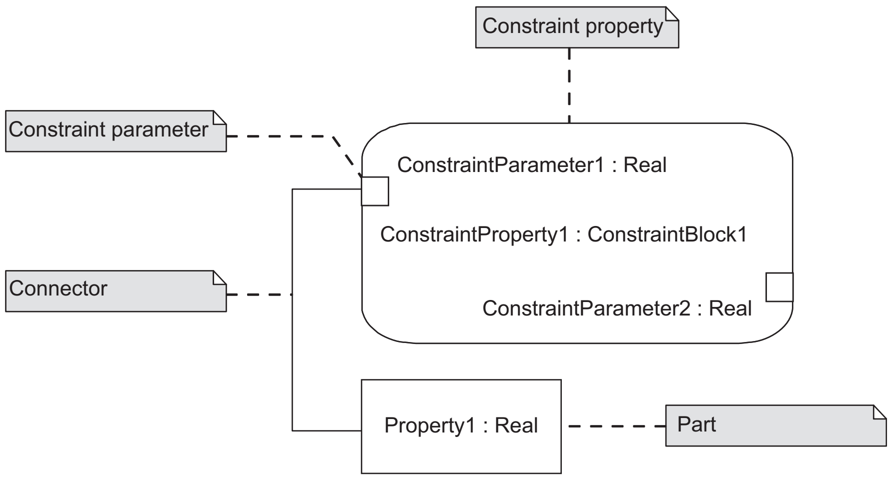

Figure 5.45 Summary of parametric diagram notation – use of constraint block

Each *constraint block* can be used multiple times on a *parametric diagram*. The use of a *constraint block* is shown as a round-cornered rectangle known as a *constraint property*. Each *constraint property* is to be named thus:

Name : Constraint Name

This allows each use of a *constraint block* to be distinguished from other uses of the same *constraint block*. In Figure 5.45 a single *constraint block*, 'ConstraintBlock1', is being used and it has been given the name 'ConstraintProperty1'.

Small rectangles attached to the inside edge of the *constraint property* represent each *constraint parameter*. These are named and their names correspond to the *parameters* defined for the *constraint block* in its definition.

These *constraint parameters* provide connection points that can be connected, via *connectors*, to other *constraint parameters* on the same or other *constraint properties* or to *block properties*. When connecting a *constraint parameter* to a *block property*, this *block property* is represented on the diagram by a rectangle known as a *part*. In Figure 5.45 a single *part* is shown, with the name 'Parametric Constraints Diagram.Block1.Property1'. This shows that this is the 'Property1' *property* of the *block* 'Block1' in the *package* 'Parametric Constraints Diagram'. *Packages* are used to structure SysML models as discussed in the previous section.

In Figure 5.45, the *part* 'Parametric Constraints Diagram.Block1.Property1' is connected to 'ConstraintParameter1'. There is nothing connected to 'ConstraintParameter2' and therefore the diagram is incomplete.

5.5.4.2 Examples

This section presents some examples of *parametric diagrams* and related diagramming elements.

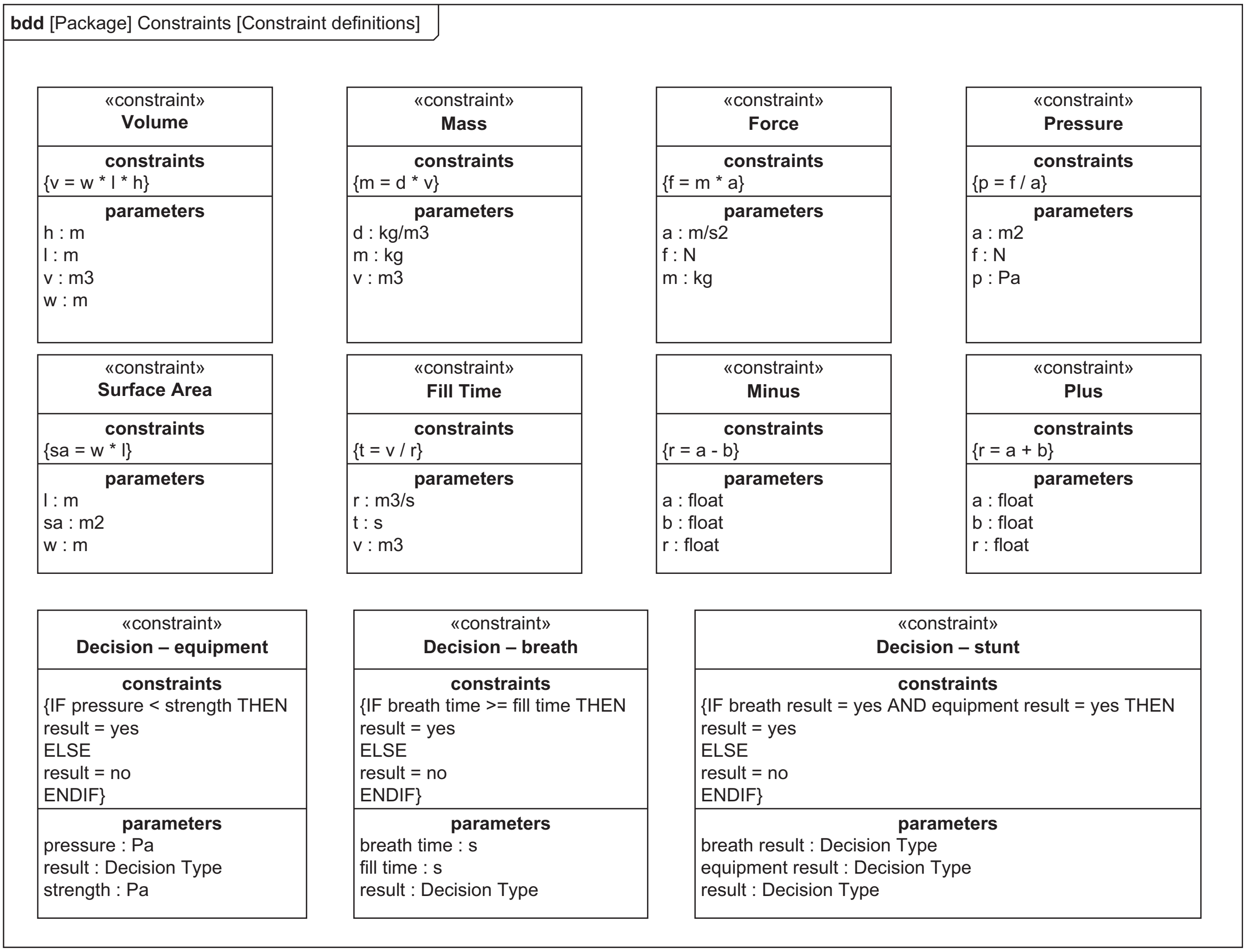

Figure 5.46 Example block definition diagram showing definition of parametric constraints

Figure 5.46 shows a number of definitions of *constraint blocks* that are defined for the Coffin Escape Stunt used as the source of examples for this chapter. As noted previously such *constraint blocks* are actually defined on a *block definition diagram*, and also as noted previously, good modelling practice has been followed with *constraint blocks* being kept separate from normal SysML *blocks*.

It can also be observed that the eight *constraint blocks* on the top two rows of the diagram are all general constraints that could be used on a number of projects, whereas the three *constraint blocks* on the bottom row are all specific to the particular System being considered (in this case the Concrete Coffin Escape). For this reason, a better way to organise them would be to split them out onto two separate diagrams and perhaps even two separate *packages* within the model in order to maximise reuse and decouple generic constraints from solution specific ones.

Another observation that can be made is that there are three different types of constraint defined:

- Constraints representing physical laws or other formulae, such as the definitions of ‘Force’ or ‘Pressure’.
- Constraints representing mathematical and logical operators that make it easier for other constraints to be connected together in a constraint usage network, such as the definitions of ‘Plus’ and ‘Minus’.
- Constraints representing decisions (heuristics) rather than calculation-type constraints, evaluating input parameters against some criteria and returning a result, which could be, for example, a ‘yes/no’, ‘true/false’ or ‘go/no-go’. The three ‘Decision’ *constraint blocks* in Figure 5.46 are examples.

If so desired, the SysML stereotyping mechanism could be used to explicitly mark the *constraint blocks* as one of these three types, as shown in Figure 5.47. This can be done in order to convey extra information about the constraints, perhaps useful if *constraint blocks* and *parametric diagrams* are to be implemented in a tool such as Simulink.

From the point of view of modelling best practice, it would probably be better to split Figures 5.46 and 5.47 into two diagrams, with the top two rows of *constraint blocks* on one diagram and the bottom row on another. From a SysML point of view there is nothing wrong with the diagrams. However, the bottom row differs from the others in that all the *constraint blocks* defined in that row happen to be specific to the Coffin Escape Stunt System, whereas those on the top two rows are general-purpose definitions that could be reused for other Systems.

An example *parametric diagram* showing the *constraint blocks* defined in Figure 5.46 being used is shown in Figure 5.48. This diagram shows the *constraint blocks* being used to determine a go/no-go decision for the escapologist based on various system properties. That is, the *parametric diagram* is being used to help validate a *use case*, namely ‘Minimise risk to escapologist’. This can be seen in the *callout note* showing that the diagram *traces* to that *use case*.

A better relationship from this diagram to the *use case* would be a *verify relationship*, with the *parametric diagram* marked as a *test case*, since that is essentially the role that it is playing here: the *parametric diagram* determines a go/no-go decision based on the other system parameters that test whether the *use case* can be met or not. However, SysML does not allow *parametric* diagrams to be marked as *test cases*, and so a simple *trace relationship* has been used. For a discussion of the various types of *traceability relationships* and the concept of a *test case*, see the following section on the *requirement diagram*.

A convention adopted by the authors, but not part of SysML, is to draw such *parametric diagrams* with an implied left to right direction. In Figure 5.48 the *parametric diagram* is drawn as though the ‘result’ *constraint parameter*, connected to the ‘Decision’ *property* of the ‘Escapologist’ *block*, is the output of

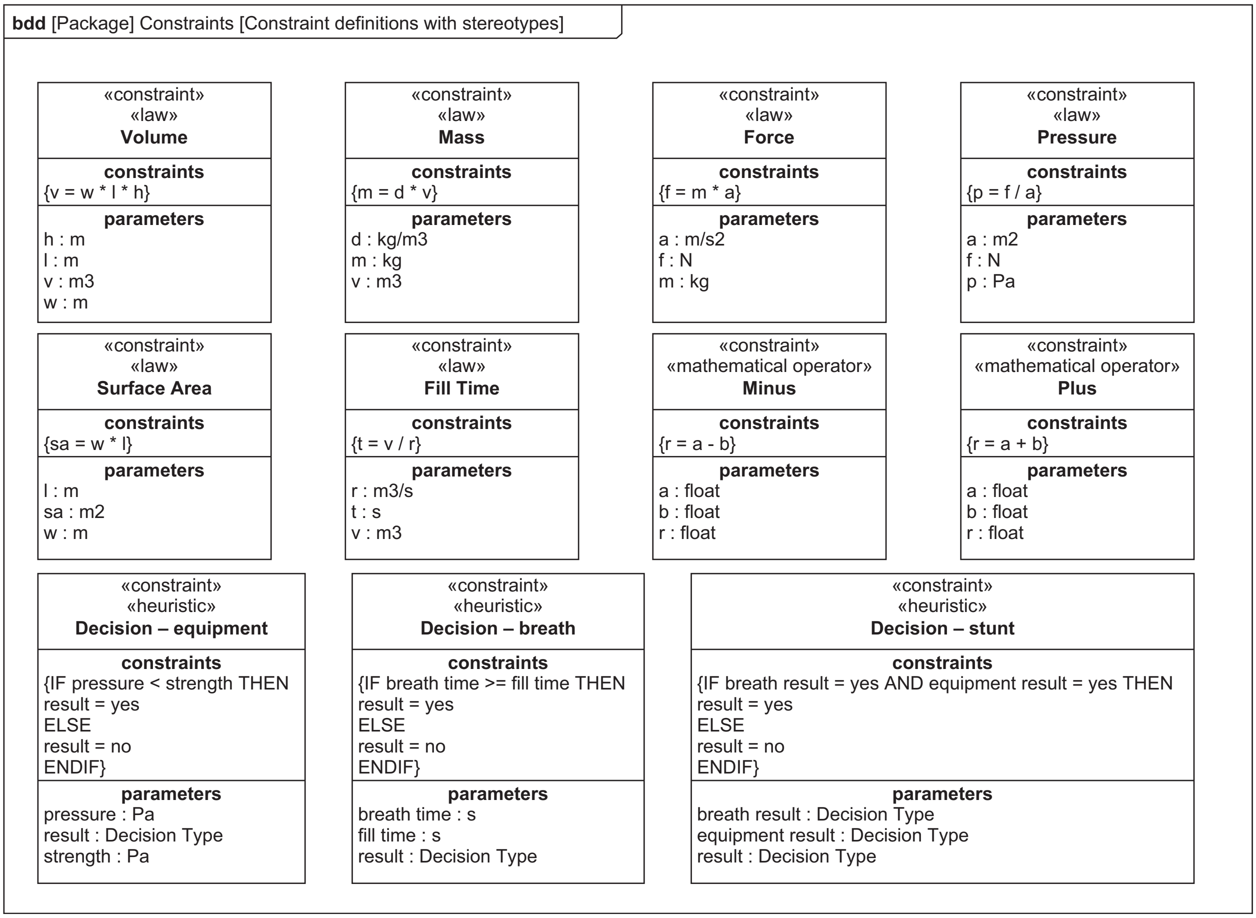

Figure 5.47 Example block definition diagram showing parametric constraints with stereotypes showing type

the diagram. Similarly, the *constraint parameters* are arranged around each *constraint property* with ‘inputs’ on the left and ‘outputs’ on the right. This is done as an aid in thinking about and constructing the diagram and, indeed, reflects the purpose of the diagram.

However, one could think about going ‘backwards’ through Figure 5.48: we could use ‘Escapologist.Bmax’ and ‘Pump.Rate’ to determine the maximum volume of concrete that can be pumped before the escapologist runs out of breath, and hence the maximum volume of the hole. If the hole is just a little longer and wider than coffin (i.e. we can set values on ‘Hole.Length’ and ‘Hole.Width’) then knowing the maximum volume of the hole would allow the height of the hole to be determined. Perhaps this usage would be used by the safety officer to calculate the hole size. If so then it could be redrawn and linked to the appropriate *use case* as shown in Figure 5.49.

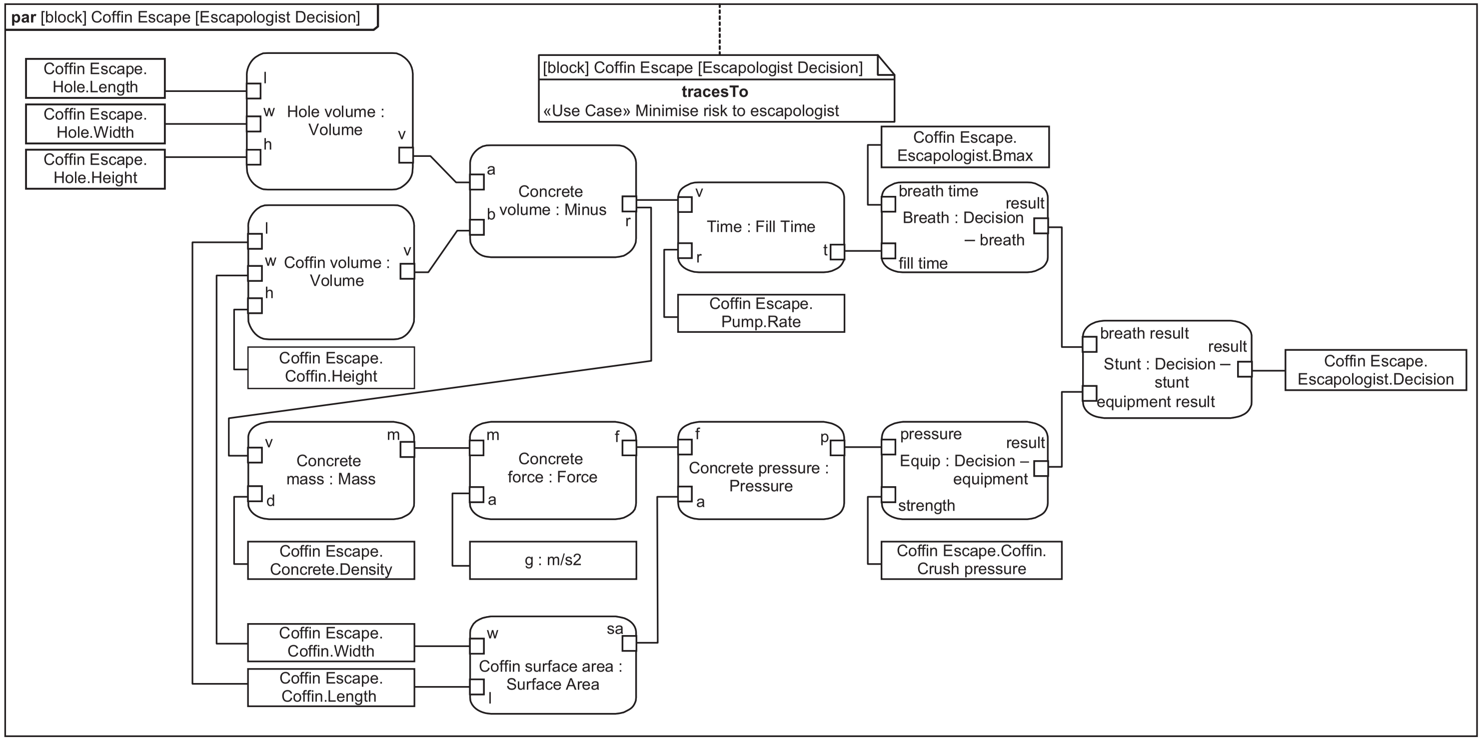

Figure 5.48 Example parametric diagram for determining go/no-go decision

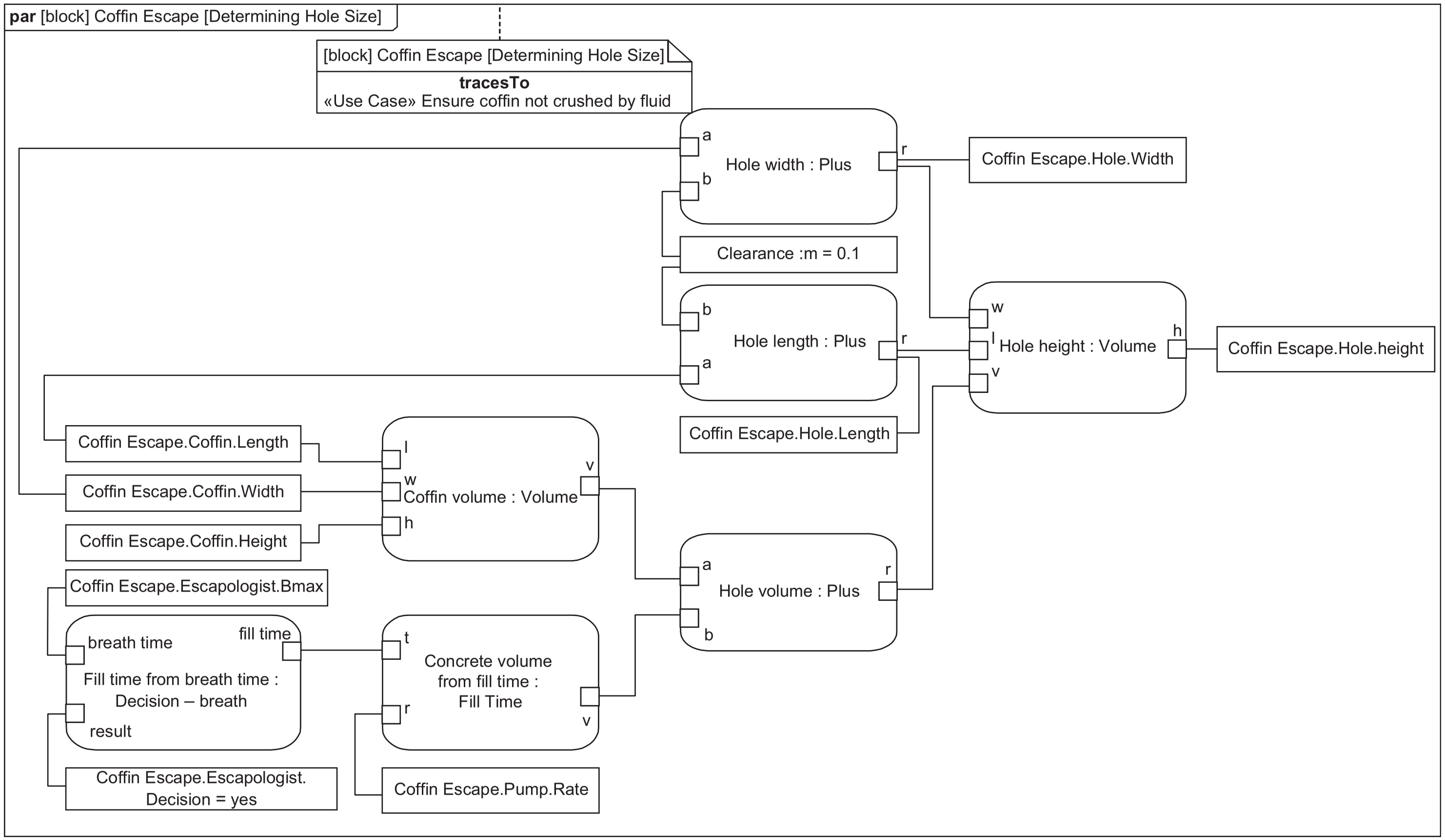

Figure 5.49 Example parametric diagram to determine hole size

Parametric constraints can also be nested, that is they can be grouped into higher level *constraint blocks* that make use of existing *constraint blocks*. Consider the three *parametric constraints* in the top left of Figure 5.48 that are used to calculate the amount of concrete needed to fill the space in the hole above the coffin. These three constraints can be grouped into a 'HoleFillVolume' *constraint block*. First we define the new *constraint block* as shown in Figure 5.50.

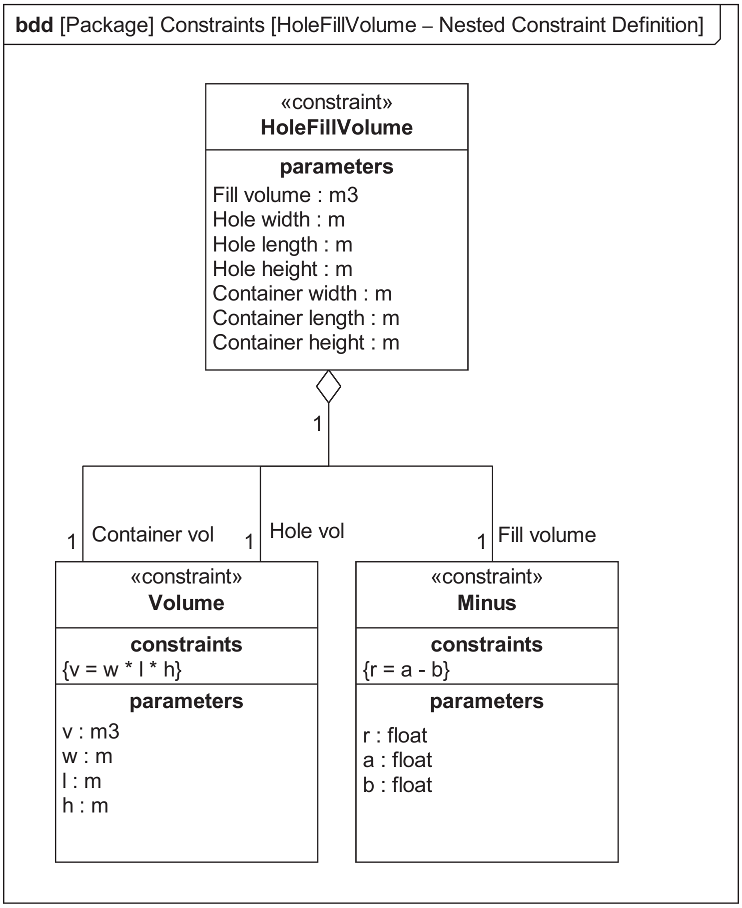

Figure 5.50 Example block definition diagram showing how higher level constraints can be constructed for the Coffin Escape Stunt

'HoleFillVolume' is defined as being made up of two 'Volume' *constraint blocks* and one 'Minus' *constraint block* and has a number of *parameters* defined. Note the use of *role names* to distinguish the role that each *constraint block* plays. Also note the use of *aggregation* rather than *composition* in the definition of 'HoleFillVolume'. This was chosen since the 'Volume' and 'Minus' *constraint blocks* are not restricted to being only *parts* of the 'HoleFillVolume' *constraint* *block*, but can also form *parts* of other *constraint blocks*, that is they can be *shared parts* and hence the use of *aggregation.*

It can also be seen that the actual *constraint expression* is not defined on this diagram. For this we need a special *parametric diagram* that shows how the component *constraint blocks* are used. This is shown in Figure 5.51; this *parametric diagram* is needed to fully define this nested constraint and must be considered as part of the definition.

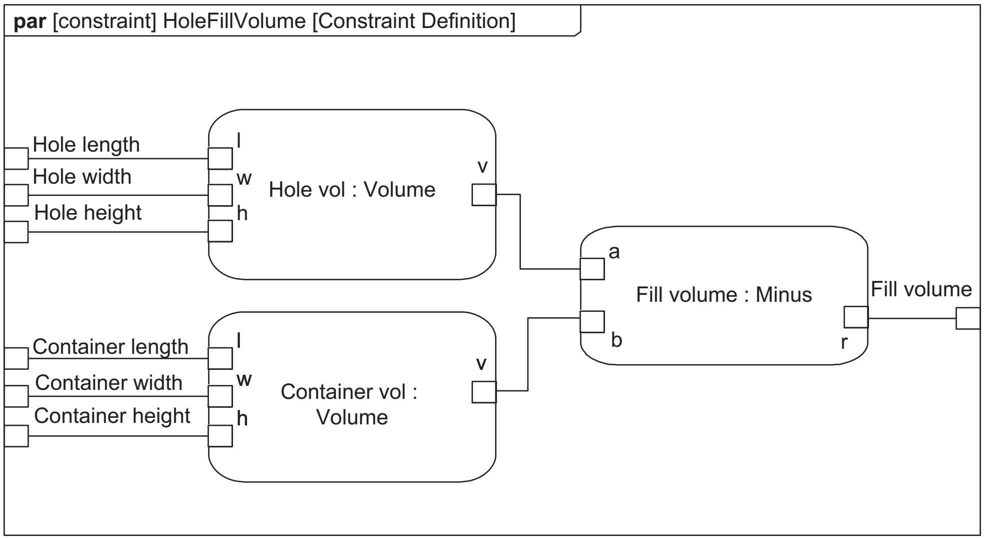

Figure 5.51 Example parametric diagram showing how higher level constraints can be constructed for the Coffin Escape Stunt

Note how, in Figure 5.51, the *parameters* of the high-level *constraint block* are attached to the *diagram frame* with *binding connectors* used to connect these to the *constraint parameters* of the internal *constraint properties*.

Having defined this high-level 'HoleFillVolume' constraint Figure 5.48 can now be redrawn to show how it can be used. This is shown in Figure 5.52.

The same approach could be taken for other groups of *constraints blocks*, resulting in a high-level *parametric diagram* that uses perhaps three or four high-level *constraint blocks*. This is left as an exercise for the reader.

It would be expected that, over time, an organisation would develop a library of constraint definitions, with lower level constraints being grouped into higher level ones for particular application usages.

5.5.4.3 Summary

SysML *parametric diagrams* show how constraints are related to each other and to properties of System Elements. They use *constraint blocks*, defined on *block definition diagrams*, which contain a *constraint expression* that relates together a number of *constraint parameters*. Each *constraint block* can be used multiple times

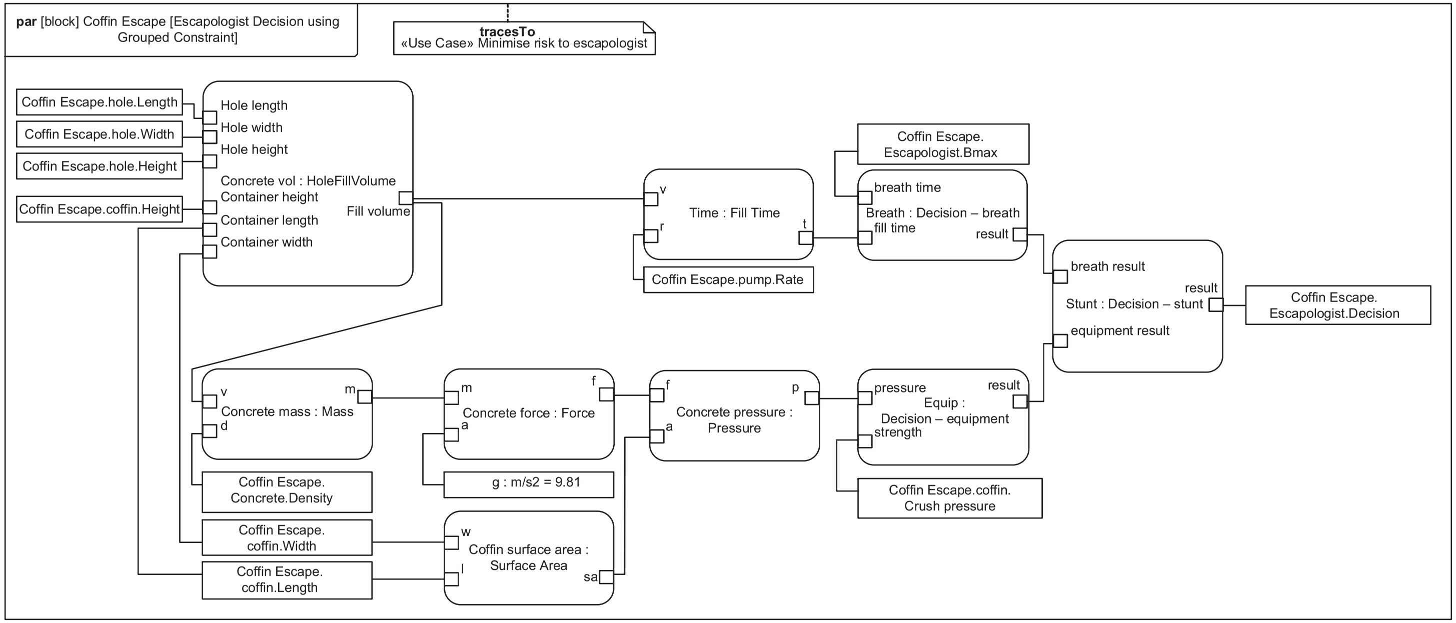

Figure 5.52 Example parametric diagram showing use of a high-level grouped constraint for the Coffin Escape Stunt

on multiple *parametric diagrams*, which relate the defined constraints to each other and to System Elements.

Parametric diagrams allow properties and behaviour of a System to be constrained and can provide an invaluable aid in understanding the often complex relationships between System properties. Modelling such inter-relationships allows analysis and design decisions to be made and also can be used to test whether Requirements have been or indeed can be satisfied. The use of *parametric diagrams* as Scenarios is discussed further in Chapter 9.

5.5.5 Requirement diagrams

The SysML has a dedicated *requirement diagram* that is used to represent Requirements and their relationships. This diagram is, in essence, a tailored *block definition diagram* consisting of a stereotyped *block* with predefined *properties* and a number of stereotyped *dependencies* and fixed-format *notes*. The various relationships provided by the *requirement diagram* also form an essential and central part of the Traceability Views that are a fundamental aspect of a model-based approach to systems engineering.

5.5.5.1 Diagram elements

Requirement diagrams are made up of three basic elements: *requirements*, *relationships* and *test cases*. *Requirements* are used, unsurprisingly, to represent Requirements, which can be related to each other and to other elements via the *relationships*. *Test cases* can be linked to *requirements* to show how the *requirements* are verified.

Figure 5.53 shows the partial meta-model for *requirement diagrams*. From the model it can be seen that a 'Requirement diagram' is made up of one or more 'Requirement', zero or more 'Relationship' and zero or more 'Test Case'. There are six types of 'Relationship': the 'Derive', 'Nesting', 'Satisfy', 'Trace', 'Refine' and 'Verify' relationships.

The notation used in SysML *requirement diagrams* is shown in Figure 5.54. This is followed by a description of the how the notation is used.

Central to the *requirement diagram* is the *requirement*. This is shown in SysML as a rectangle with the *stereotype* «requirement». The rectangle also contains a human-readable name for the *requirement*. In addition, all *requirements* have two *properties* predefined by SysML: the *id#* and *txt properties*. The *id# property* is there to hold a unique identifier for the *requirement*. The *txt property* holds descriptive text for the *requirement*. The display of *id#* and *txt* is optional and Figure 5.54 shows these compartments for 'Requirement1' and omits them for 'Requirement2', 'Requirement3' and 'Requirement4'.

A Requirement may be decomposed into one or more sub-Requirements, for example when the Requirement is not atomic in nature and it is desired to decompose it into a number of related atomic sub-Requirements. In SysML this decomposition is known as *nesting* and is indicated with a *nesting relationship* such as that shown between 'Requirement1' and 'Requirement2'.

When carrying out Requirements analysis it is often necessary to derive additional Requirements. A derived Requirement is one that is not explicitly stated by a

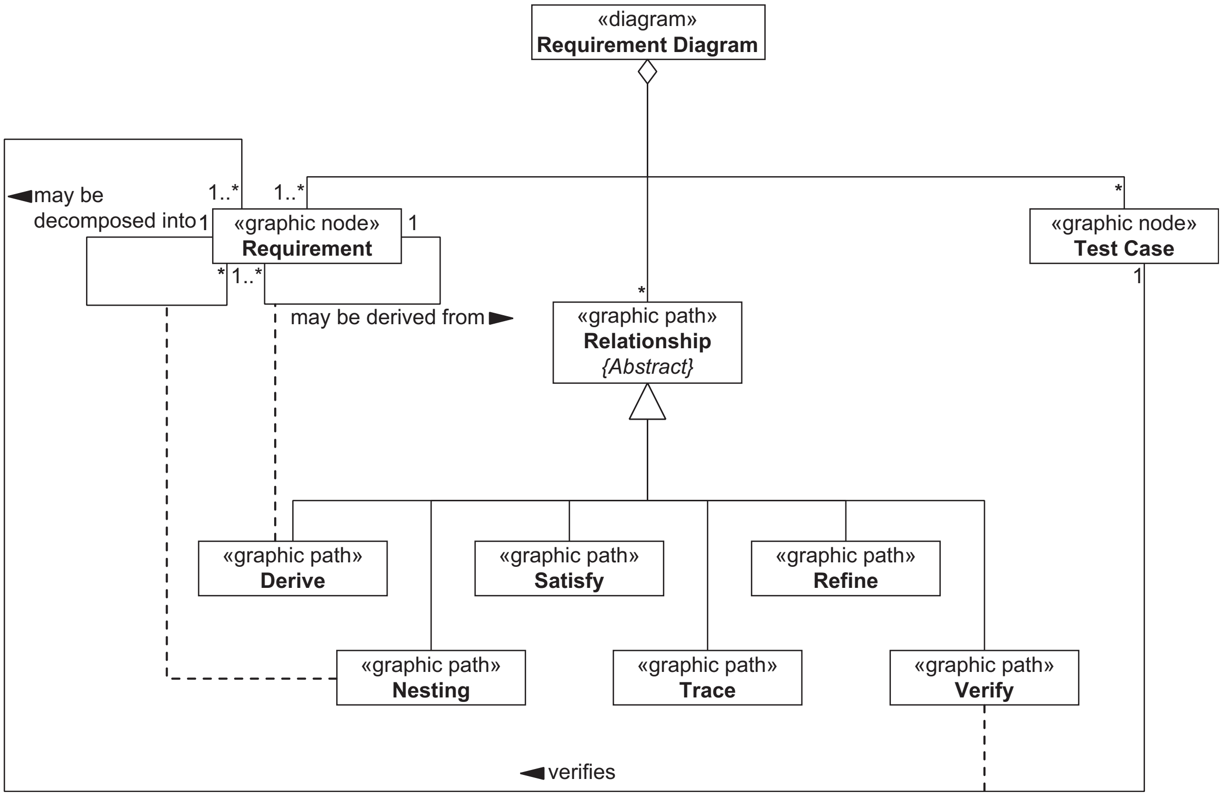

Figure 5.53 Partial meta-model for the requirement diagram

Stakeholder Role but one that has been derived by systems engineers from an explicit, stated Requirement as part of the requirements analysis process. Such derived Requirements can be linked back to their source Requirements in SysML by using a *derive relationship*, an example of which is shown in Figure 5.54 showing that 'Requirement3' is derived from 'Requirement1'.

The SysML *requirement diagram* also supports four other types of *relationships* that are used in the following ways:

- *Satisfy relationship*. This is used to show that a *model element* satisfies a *requirement*. It is used to relate elements of a design or implementation model to the Requirements that those elements are intended to satisfy. Although Figure 5.54 shows a *satisfy relationship* between a *block* and a *requirement*, it can be used between any SysML *model element* and a *requirement*.
- *Trace relationship*. This is used to show that a *model element* can be traced to a *requirement* or vice versa. This provides a general-purpose relationship that allows *model elements* and *requirements* to be related to each other.

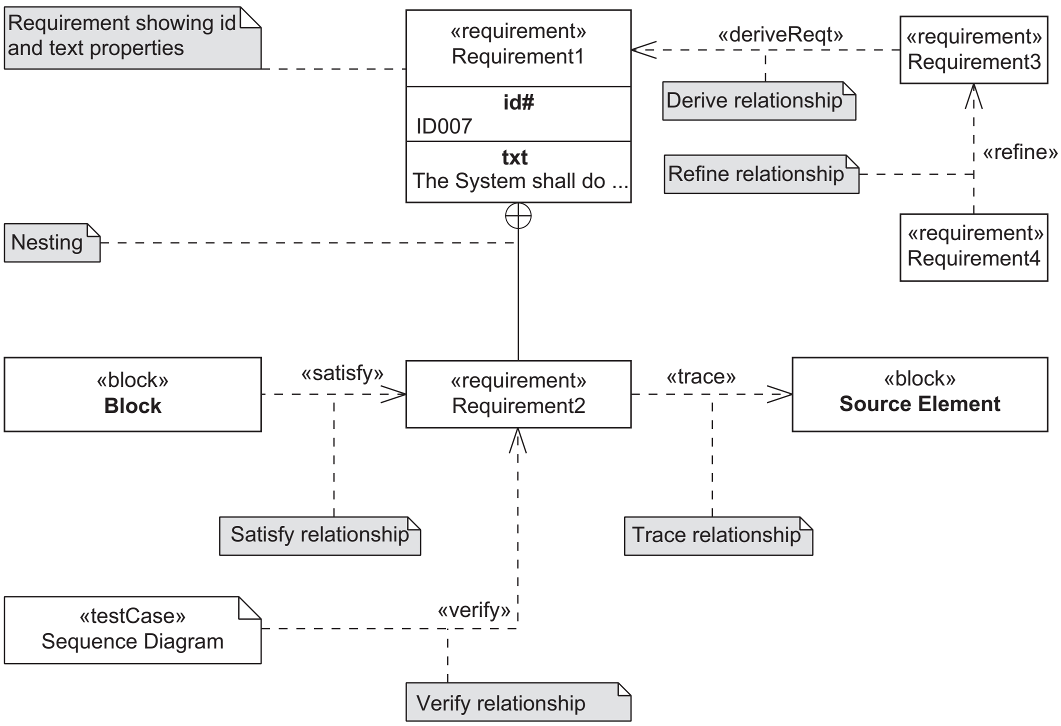

Figure 5.54 Summary of requirement diagram notation

An example of this is shown by the *trace relationship* between 'Requirement2' and 'Source Element' in Figure 5.54.

- *Refine relationship*. This is used to show how *model elements* and *requirements* can be used to further refine other *model elements* or *requirements*. This could be, for example, one *requirement* refining another as shown in Figure 5.54 where 'Requirement4' *refines* 'Requirement3'.
- *Verify relationship*. This is used to show that a particular *test case* verifies a given *requirement* and so can only be used to relate a *test case* and a *requirement*. However, a *test case* is not a specific type of SysML element. Rather it is a *stereotype*, «testCase», which can be applied to any SysML *operation* or *behavioural diagram* to show that the stereotyped element is a *test case* intended to verify a *requirement*. This stereotyped element – the *test case* – can then be related to the *requirement* it is verifying via the *verify relationship*. The *test case* is shown on a *requirement diagram* as a SysML *note* containing the name of the SysML element or diagram that is acting as a *test case* along with the *stereotype* «testCase». This is shown in Figure 5.54 by the *verify relationship* between the *test case* called 'Sequence Diagram' and 'Requirement2'.

Unfortunately, the definition of the «testCase» *stereotype* in the SysML specification [1] prevents the *stereotype* being applied to SysML *parametric diagrams*. This is a missed opportunity since *parametric diagrams*, discussed earlier in this section, are an ideal mechanism by which Formal Scenarios (*test cases*) can be modelled, which is possible using *sequence diagrams*. Readers who are adopting the techniques and approaches described in this book are urged to use the SysML's stereotyping mechanisms to define their own *test case stereotype* that can be applied to *parametric diagrams*. Similarly, a verify *stereotype* could be defined that can take a *use case* as a target given the issues with the verify relationship discussed earlier in this section.

These various types of *relationship* allow the modeller to explicitly relate different parts of a model to the *requirements* as a way of ensuring the consistency of the model. However, where possible one of the specific types of *relationship*, such as *satisfy*, should be used in preference to the more generic *trace relationship*, which has weakly defined semantics since it says nothing about the nature of the *relationship* other than that the two elements can be traced in some general and unspecified manner.

It should also be noted that, although shown in Figure 5.54 using stereotyped *dependencies*, these relationships can also be shown in SysML using special versions of the *note*. These *callout notes* can be useful when relating elements in widely different parts of a model since it avoids the need to produce additional diagrams specifically to show the *relationships*. However, they can lead to inconsistency, particularly when modelling is not being carried out using a tool (or using a tool that does not enforce consistency). Using the stereotyped *dependencies* gives an immediate and direct indication of the *relationship* since the two elements are explicitly connected by the *dependency*. Using *callout notes* hides the immediacy of the *relationship* inside the text of the note and also requires that two *notes* are added to the model: one to the source of the *relationship* and one to the target. If one of these notes is omitted the model will be inconsistent. An example of the use of *callout notes* is given in Section 5.5.5.2.

5.5.5.2 Examples

This section presents some examples of *requirement diagrams* and related diagramming elements. Further examples will be found in the case study in Chapter 13.

Figure 5.55 shows a number of SysML *requirements* for the Coffin Escape Stunt, each of which has its *id#* and *txt property* shown. Some of these *requirements* are broken down further into sub-requirements via *nesting*. At least two of these *requirements*, ES004 and ES005, have descriptive text in their *txt property* that could be considered to be untestable. In the case of ES005, the sub-requirements further describe what is meant by '. . . the risk to the escapologist is minimised'. However, in the case of ES004 further analysis is required. This might result in a number of *derived requirements* being created as shown in Figure 5.56.

The three *requirements* ES004-D001, ES004-D002 and ES004-D003 shown in Figure 5.56 are each derived from ES004 and show how the vague and

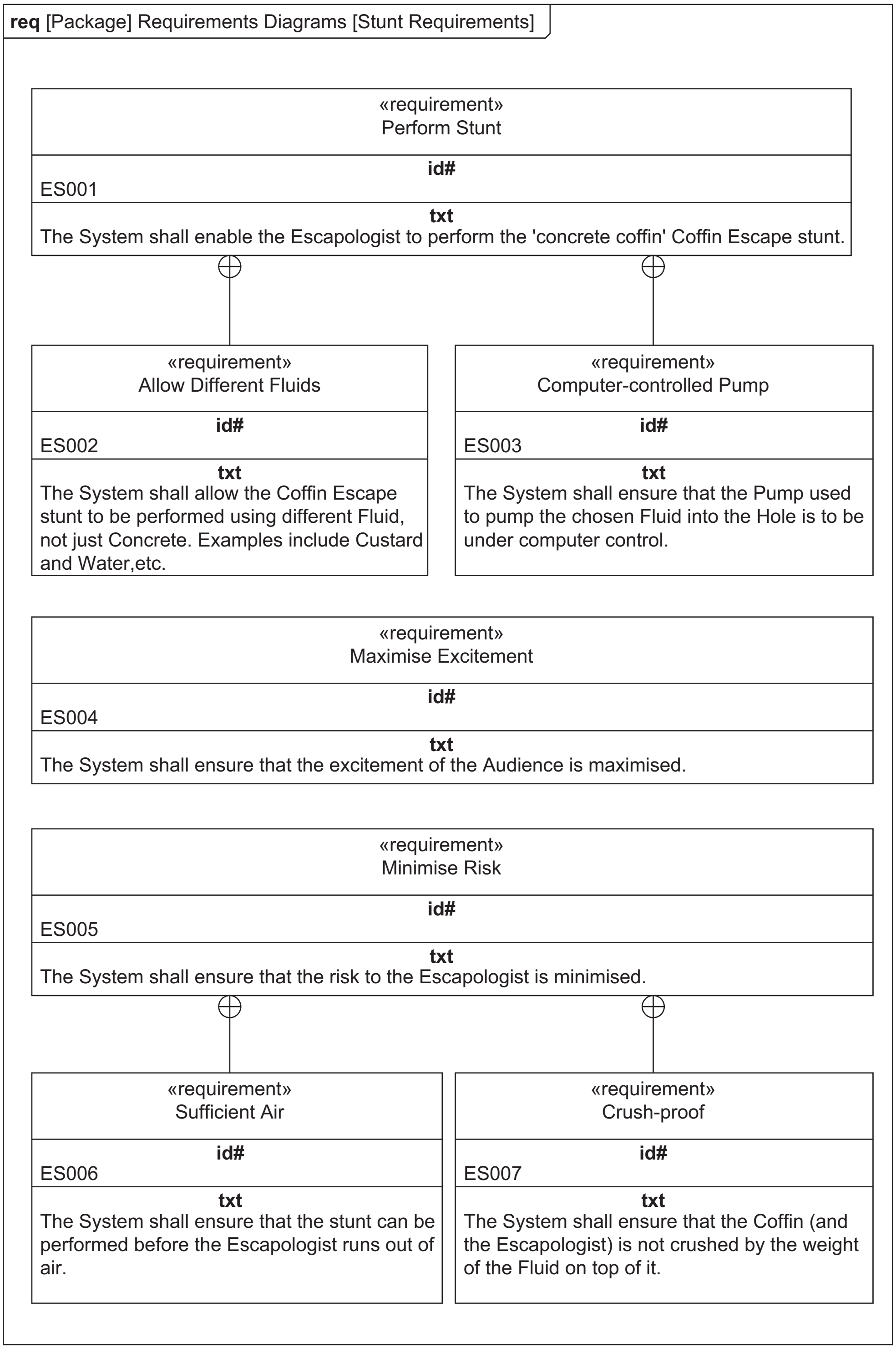

Figure 5.55 Example requirement diagram showing Requirements for the Coffin Escape Stunt

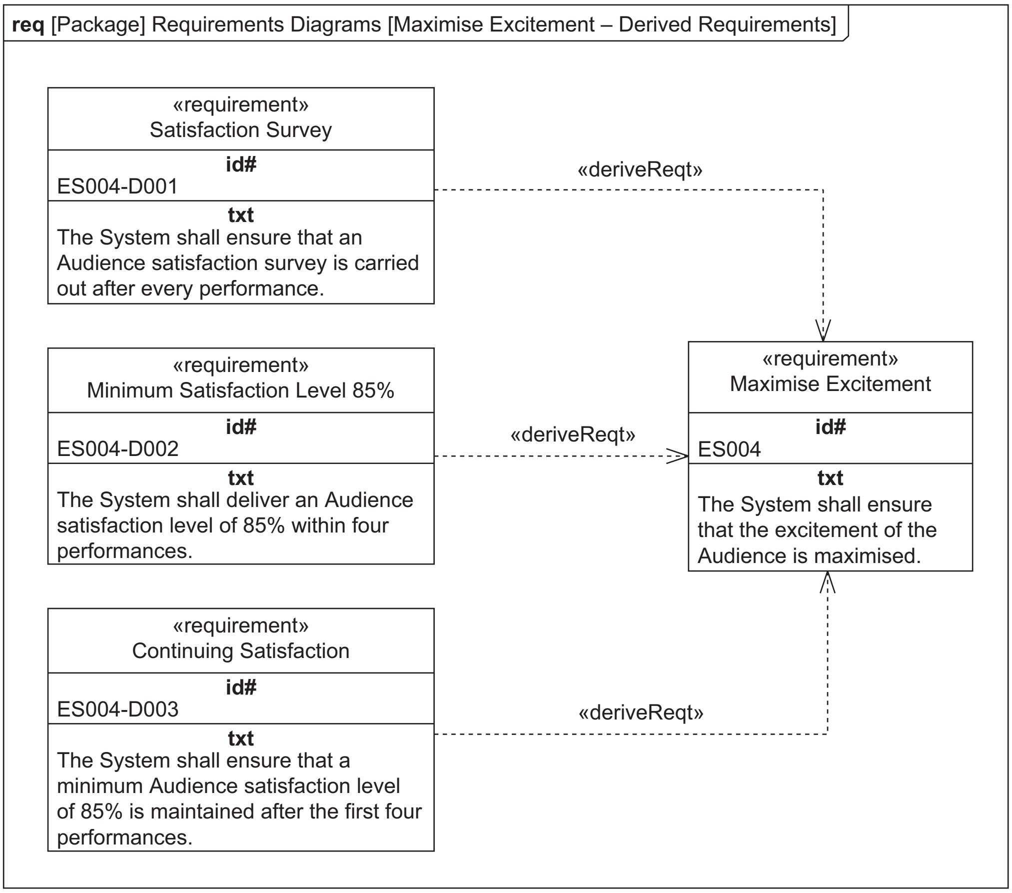

Figure 5.56 Example requirement diagram showing derived Requirements

untestable *requirement* that 'The System shall ensure that the excitement of the audience is maximised' may be further specified in a way that *is* testable.

Sometimes turning off the *id#* and *txt properties* of a *requirement* can make a diagram easier to read, particularly when additional information such as *trace relationships* are shown. This has been done in Figure 5.57, which shows the same *requirements* as are shown in Figure 5.55, but with the *id#* and *txt compartments* hidden and *trace relationships* added linking the *requirements* to *blocks* representing the source of the *requirements*. There is no significance in the sizing of the various *requirement*s, it has been done simply to ease the layout of the diagram.

A similar diagram is given in Figure 5.58, which concentrates on a single *requirement*, showing how it *traces* to source elements and in addition, showing a *use case* that *refines* the *requirement*. A seemingly obvious, but often overlooked,

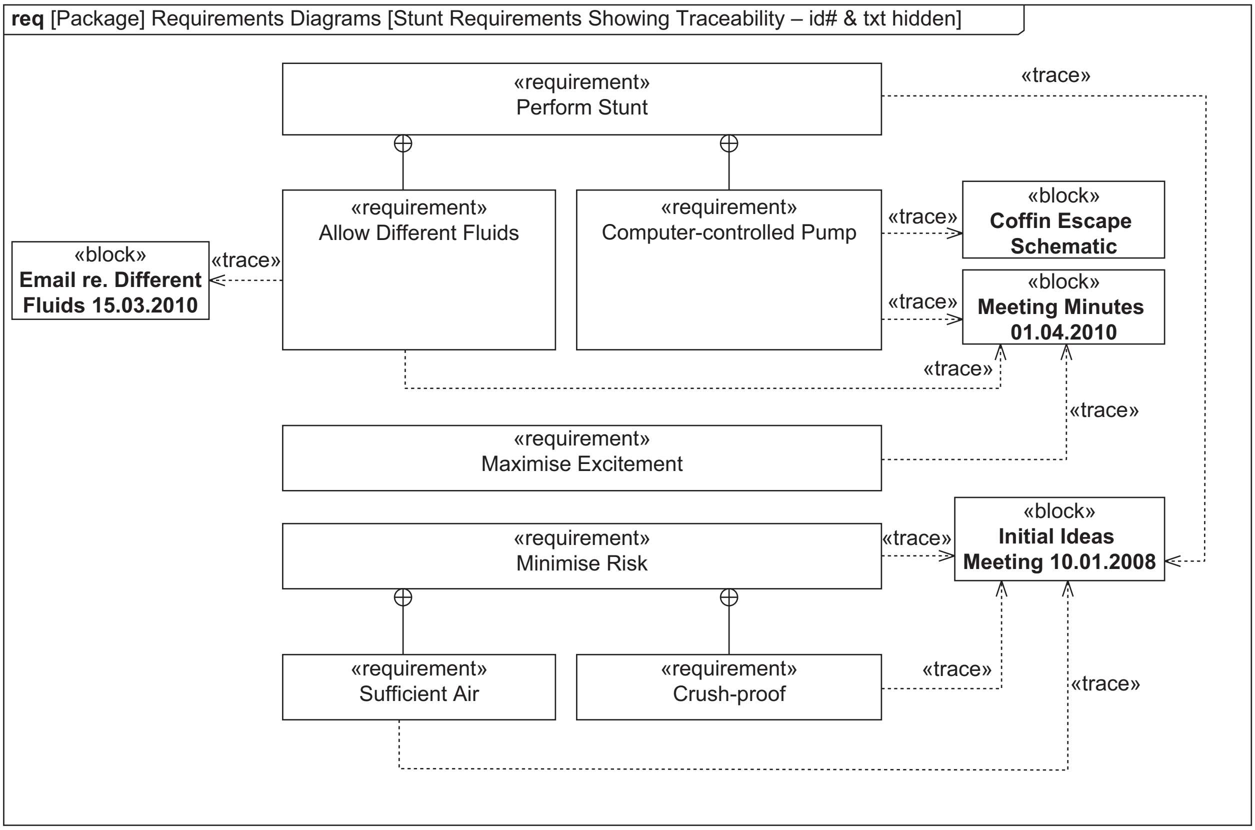

Figure 5.57 Example requirement diagram showing «trace» relationships

aspect of modelling is highlighted in Figure 5.58, namely that of keeping diagrams as simple as possible. There is often a temptation to overload diagrams with too many elements so that they add to the complexity and lack of understanding of the system rather than helping. The information shown on the four example diagrams earlier in this section could have been shown on a single diagram, but this would have made the communication of the understanding of the *requirements* and their *relationships* to other *model element* harder to achieve. Any sensible modelling tool will allow *model elements* to be reused on a number of different diagrams and this is to be encouraged, not only for *requirements diagrams* but for *any* of the SysML diagrams. If you find a diagram is becoming too complex (more than around 9 or 10 elements, as a crude heuristic), break it down into a number of simpler diagrams. Miller's comments on the limits on our capacity to process information are as valid today as when they were first written and apply just as much to SysML models. See Reference 2.

The final example of a *requirement diagram* is shown in Figure 5.59. This diagram shows exactly the same information as that shown in Figure 5.58 but uses

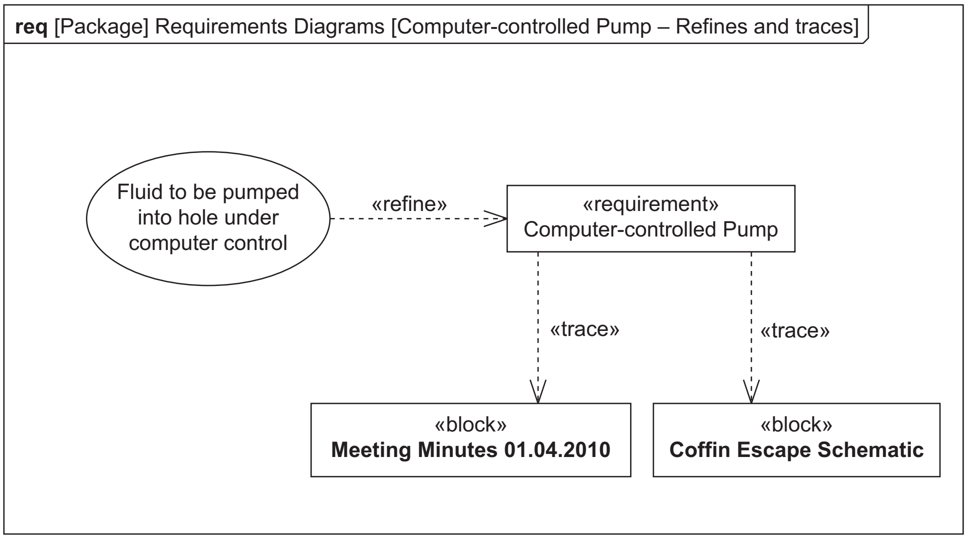

Figure 5.58 Example requirement diagram showing «refine» and «trace» relationships

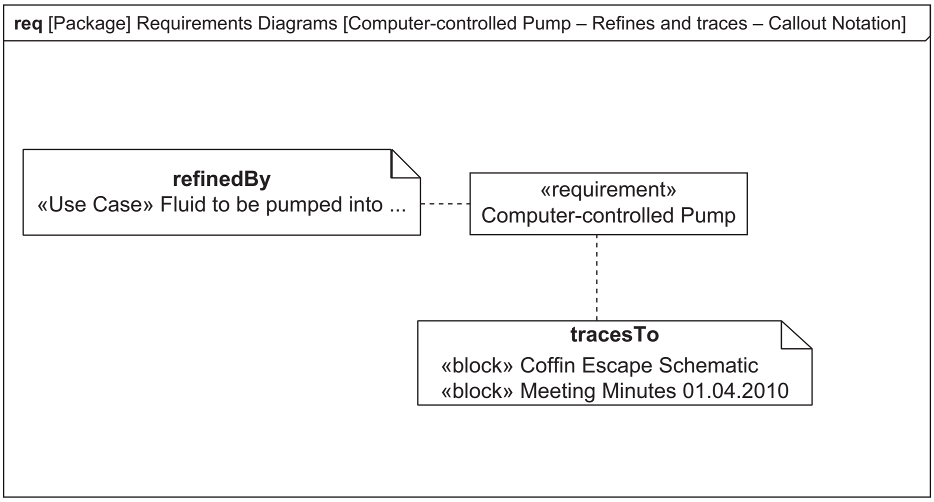

Figure 5.59 Example requirement diagram showing «refine» and «trace» relationships using callout notes

the *callout* notation rather than explicit *refine* and *trace relationships*. Some of the immediacy of information is lost using the *callout* notation since the symbols used do not, in this example, show graphically that the other model elements involved are a *use case* and two *blocks*. One has to read the content of the *callout notes* to understand the types of *model elements* involved. For this reason the authors recommend, where possible and appropriate, the explicit *relationships* as in Figure 5.58.

5.5.5.3 Summary

SysML *requirement diagrams* are used to show *requirements* and their *relationships* to each other and how they trace to, are satisfied by, are refined by and are verified by other *model element*. Wherever possible, use of the more specific types of *relationship* (such as *satisfy*) is preferred over the more generic *trace*. Each *requirement* has a name, unique identifier and a description. Most SysML tools allow the identifier and description to be hidden if desired, in order to simplify diagrams. Additional *properties* such as 'priority' may be defined if needed and examples are given in the SysML specification [1].

It should also be noticed that the scope of the requirement diagram may, and should, be extended to include other types of Need from the MBSE Ontology, rather than being restricted to Requirements only. The MBSE Ontology states that there are four types of Need: Requirement, Capability, Goal and Concern, each of which may be visualised using the SysML *requirement* concept.

5.5.6 State machine diagrams

So far we have been considering the SysML structural diagrams. In this section we now start looking at the SysML behavioural diagrams, beginning with the *state machine diagram*. *State machine diagrams* have been discussed in some detail in Chapter 4 and thus some of this section will serve as a recap. The focus here, however, will be the actual *state machine diagram*, whereas the emphasis previously has been on general behavioural modelling.

State machine diagrams realise a behavioural aspect of the model. They model the order in which things occur and the logical conditions under which they occur for instances of *blocks*, known in SysML as *instance specifications*. They show such behaviour by relating it to meaningful *states* that the System Element, modelled by a *block*, can be in at any particular time, concentrating on the *events* that can cause a change of *state* (known as a *transition*) and the behaviour that occurs during such a *transition* or that occurs inside a *state.*

5.5.6.1 Diagram elements

State machine diagrams are made up of two basic elements: *states* and *transitions*. These *states* and *transitions* describe the behaviour of a *block* over logical time. *States* show what is happening at any particular point in time when an *instance specification* typed by the *block* is active. *States* may show when an *activity* is being carried out or when the *properties* of an *instance specification* are equal to a particular set of values. They may even show that nothing is happening at all – that is to say that the *instance specification* is waiting for something to happen. The elements that make up a *state machine diagram* are shown in Figure 5.60.

Figure 5.60 shows the partial meta-model for *state machine diagrams*. *State machine diagrams* have a very rich syntax and thus the meta-model shown here

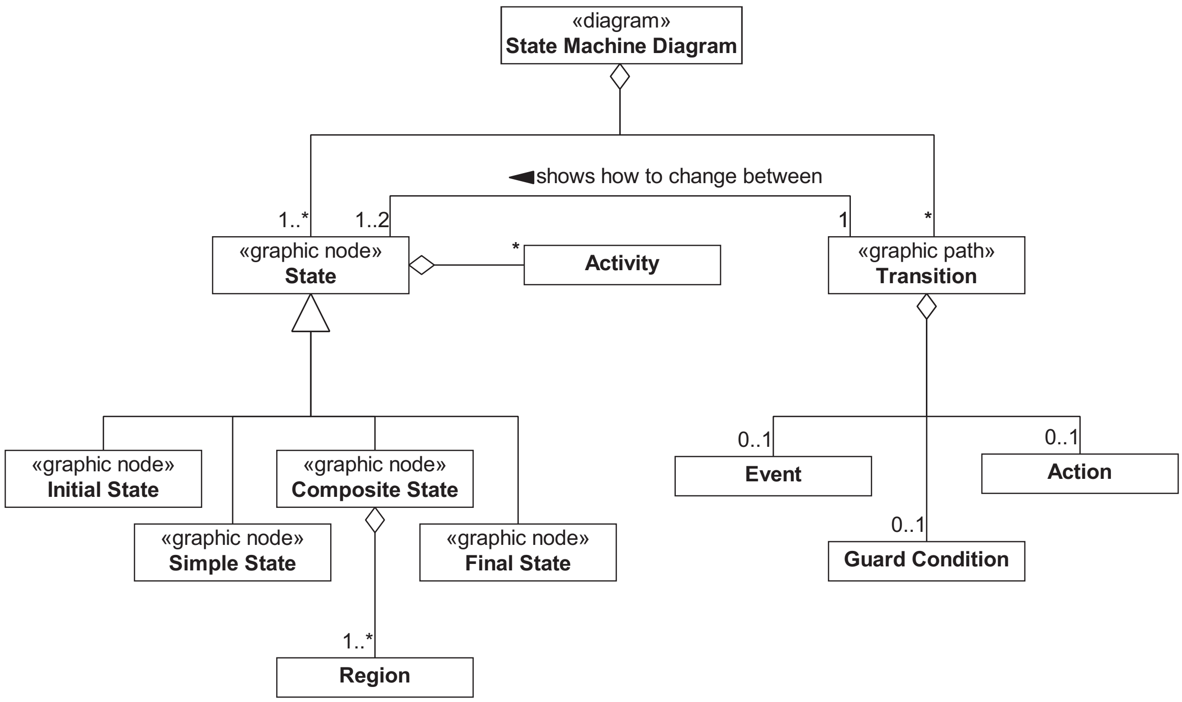

Figure 5.60 Partial meta-model for the state machine diagram

omits some detail – for example, there are different types of *action* that are not shown. See References 1 and 4 for more details.

From the model, it can be seen that a ‘State Machine Diagram’ is made up of one or more ‘State’ and zero or more ‘Transition’. A ‘Transition’ shows how to change between one or two ‘State’. Remember that it is possible for a *transition* to exit a *state* and then enter the same *state*, which makes the *multiplicity* one or two rather than two, as would seem more logical.

There are four types of ‘State’: ‘Initial State’, ‘Simple State’, ‘Composite State’ and ‘Final state’. Each ‘State’ is made up of zero or more ‘Activity’. An ‘Activity’ describes an on-going, non-atomic unit of behaviour and is directly related to the *operations* on a *block*. A ‘Composite State’ is divided into one or ‘Region’. When there are more than one ‘Region’, each ‘Region’ is used to model concurrent (i.e. parallel) behaviour.

Each ‘Transition’ may have zero or one ‘Guard Condition’, a Boolean condition that will usually relate to the value of a *block* property. The ‘Guard Condition’ must evaluate to true for the ‘Transition’ to be valid and hence capable of being crossed.

A ‘Transition’ may also have zero or one ‘Action’. An ‘Action’ is defined as an activity whose behaviour is *atomic*. That is, once started it cannot be interrupted and will always complete. An ‘Activity’, on the other hand, is *non-atomic* and can be interrupted. An ‘Action’ should be used for short-running behaviour.

Finally, a ‘Transition’ may have zero or one ‘Event’ representing an occurrence of something happening that can cause a ‘Transition’ to fire. Such an ‘Event’ can be thought of as the receipt of a *message* by the *state machine*.

If an ‘Event’ models the receipt of a *message*, often sent from one *state machine* to another, then how does one model the sending of such a *message* from a *state machine*? The answer is that there are actually two types of *event*: *receipt events* and *send events*.

The type of *event* described earlier in this section, which corresponds to the receipt of a *message* and which can trigger a *transition*, is actually an example of a *receive event.* A *send event* represents the origin of a *message* being sent from one *state machine* to another. It is generally assumed that a *send event* is broadcast to all elements in the System and thus each of the other elements has the potential to receive and react upon receiving the *event*. Obviously, for each *send event* there must be at least one corresponding *receipt event* in another *state machine*. This is one of the basic consistency checks that may be applied to different *state machine diagrams* to ensure that they are consistent. A *send event* is usually modelled as the *action* on a *transition*.

The notation for the *state machine diagram* is shown Figure 5.61.

The basic modelling elements in a *state machine diagram* are *states*, *transitions* and *events*. *States* describe what is happening within a system at any given point in time, *transitions* show the possible paths between such *states* and *events* govern when a *transition* can occur. These elements were discussed in detail in Chapter 4 and the reader is referred to that chapter. However, there are a number of elements in Figure 5.61 that weren’t discussed in Chapter 5 and which need discussion here, namely:

- *Composite states*
- *Entry activities*
- *Exit activities*

Figure 5.61 shows two *composite states*: ‘Composite State (Concurrent)’ and ‘Composite State (Sequential)’. *Composite states* allow *states* to be modelled that have internal behaviour that is further decomposed into *states*. They can be thought of as *states* that have their own *state machine diagrams* inside.

Let us consider ‘Composite State (Sequential)’ first. This *composite state* has a single *region* (the part of the *state* beneath the box containing the name). Since there is only one *region* the behaviour takes place sequentially within the *state* and hence this is a *sequential composite* state. In this example, ‘Simple State 1’ is entered first. This then leads on to ‘Simple State 2’ and when this *state* is left the *final state* is entered.

Now consider ‘Composite State (Concurrent)’. This has two *regions* separated by a dashed line. Each *region* represents concurrent (i.e. parallel) behaviour and hence this is a *concurrent composite state*. The *transition* to ‘Composite State (Concurrent)’ causes both *regions* to become active and therefore the two small *state machine diagrams* in the *regions* become active. When both have completed, then the *transition* from ‘Composite State (Concurrent)’ to the *final state* can fire.

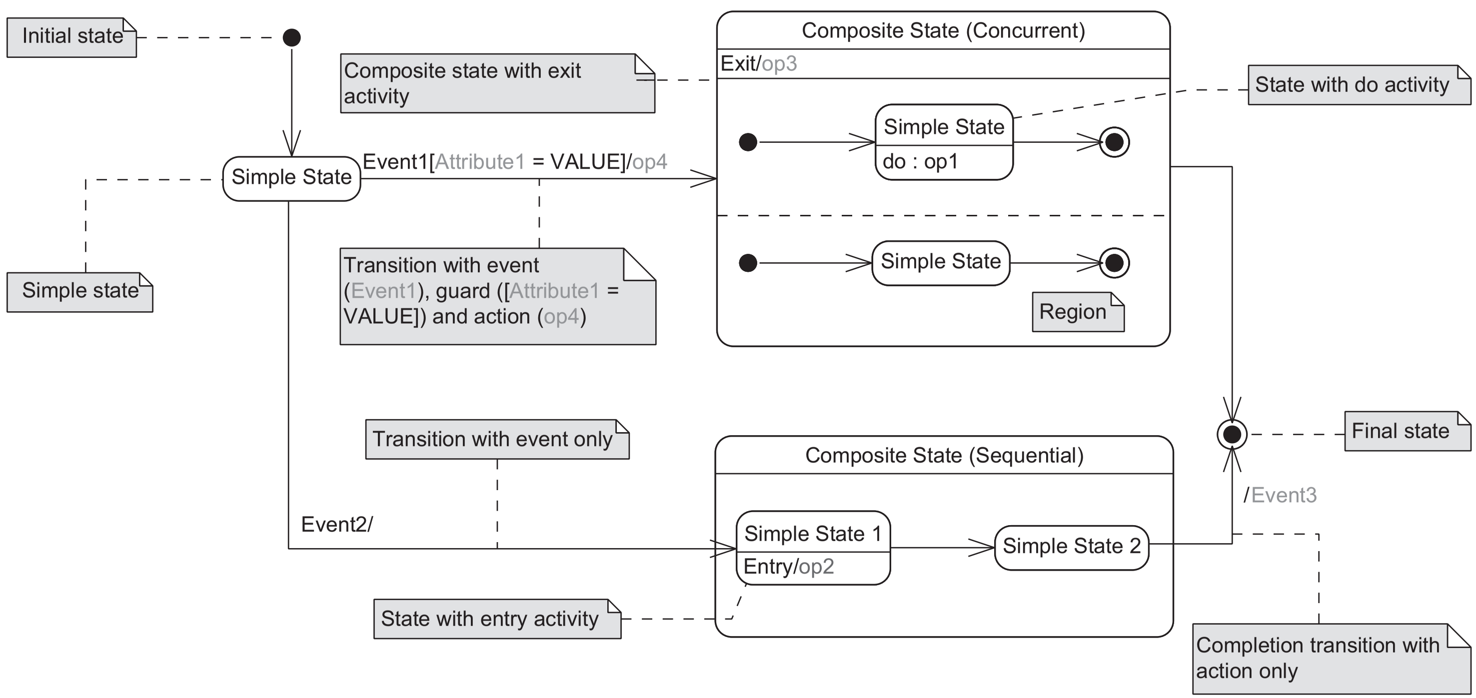

Figure 5.61 Summary of state machine diagram notation

Examples of *composite states*, along with a discussion of when *sequential composite states* are used, can be found in Section 5.5.6.2.

Entry and *exit activities* can be seen in 'Simple State 1', shown as 'Entry/op2' (an *entry activity)*, and in 'Composite State (Concurrent)' shown as 'Exit/op3' (an *exit activity*).

An *entry activity* represents an *activity* that takes place every time a *state* is entered. The notation is the keyword 'Entry/' followed by the behaviour to take place (in the example here, the invocation of an *operation* 'op2').

An *exit activity* represents an *activity* that takes place every time a *state* is exited. The notation is the keyword 'Exit/' followed by the behaviour to take place (in the example here, the invocation of an *operation* 'op3').

Unlike normal *activities* both the *entry activity* and the *exit activity* cannot be interrupted; they behave more like *actions* as they are guaranteed to run to completion. Section 5.5.6.2 gives examples.

Before moving on to consider some examples of *state machine diagrams* it is worth discussing some alternative notation that can be used for *events* (both *receipt events* and *send events*) and for modelling decision points (known as *junction states*).

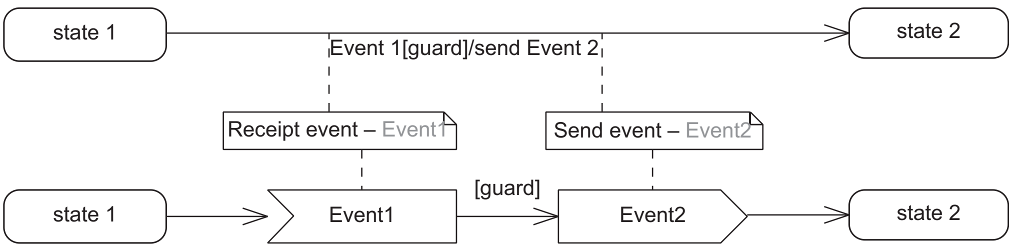

Figure 5.62 Alternative notations for receipt and send events

Figure 5.62 shows the two possible notations for modelling *receipt events* and *send events*. The top part of the diagram shows the textual notation. There is no keyword to indicate "receipt", an *event* preceding a *guard condition* represents a *receipt event.* The widely used notation for representing a *send event* is to place the word "send" in front of the *event name* as part of the *action* on the *transition*. Note, however, that this is a convention and is not specified by the SysML standard. Exactly the same transition is shown at the bottom of the diagram, but this time using graphical symbols that explicitly show which is a *receipt event* and which is a *send event*. This notation is also used on *activity diagrams* discussed in Section 5.5.8 below.

Figure 5.63 shows alternative notations that can be used when there are two or more *transitions* from a *state* that have the same *event* (or indeed no *event*) but different *guard conditions*. In the example, the same *event* 'Event1' will lead either to 'state 2' or 'state 3' depending on the value of the *guard condition.* This can be represented as two separate *transitions* from 'state 1' as in the upper part of the diagram, or as a single *transition* from 'state 1' to a *junction state* (the diamond) followed by two *transitions* from the *junction state*.

As to which notation to use? Well, use whatever you feel is best. Diagramming guidelines might specify (see Chapter 6 for a discussion of diagramming guidelines).

However, if they don't, you are advised to choose a style and use it consistently within a model. At least in that way your *state machine diagrams* will have a consistent look and feel.

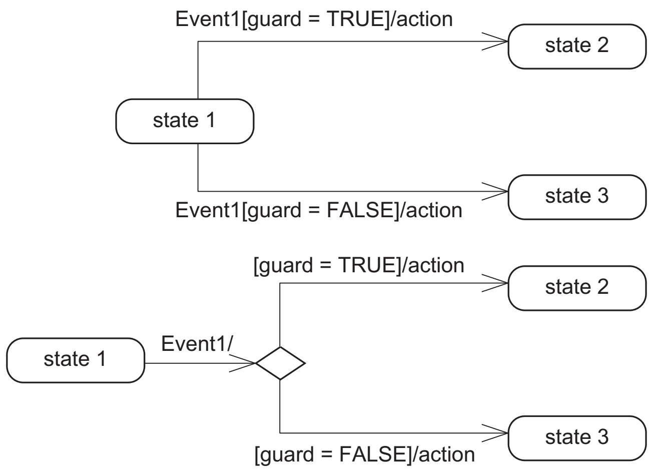

Figure 5.63 Alternative notations for decisions

5.5.6.2 Examples

This section presents some examples of *state machine diagrams* and related diagramming elements. Further examples will be found in the case study in Chapter 13.

The *block definition diagram* in Figure 5.64 shows a single *block* that models the 'Pump' used in the Coffin Escape Stunt. This was seen previously in Section 5.5.1.2

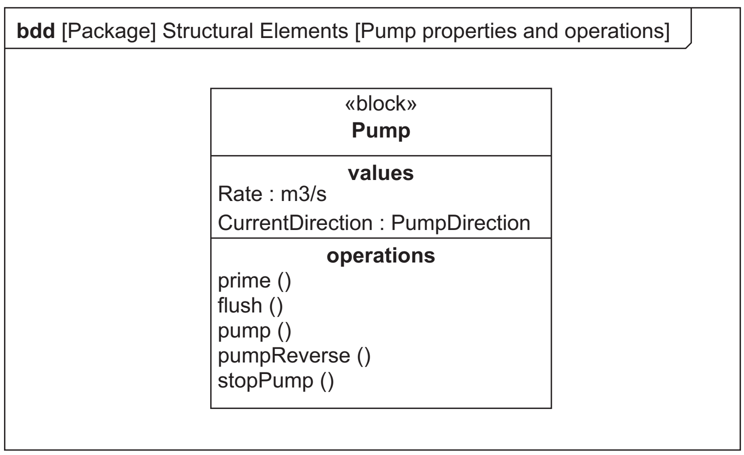

Figure 5.64 Example block definition diagram showing Pump properties and operations

when we looked at example *block definition diagrams*. The *block* has a number of *operations* and the 'Pump' that it models can be in a number of meaningful *states*, such as being powered down and pumping in either direction. It should, therefore, have its behaviour modelled using a *state machine diagram.* This has been done in Figure 5.65.

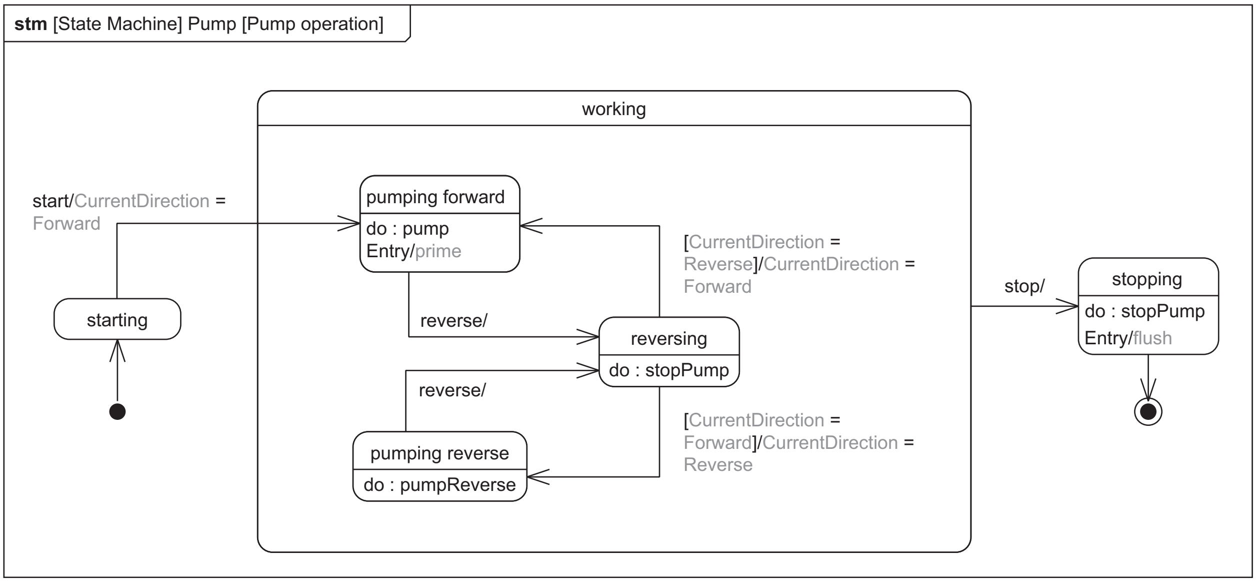

Figure 5.65 Example state machine diagram showing pump behaviour

The *state machine diagram* in Figure 5.65 has three main *states*, 'starting', 'working' and 'stopping', an *initial state* and a *final state*. The *state* 'working' is a *composite state*. It has one *region* and is therefore a *sequential composite state*. It contains three *states*: 'pumping forward', 'pumping reverse' and 'reversing'.

The *state machine* represented by this *state machine diagram* can be considered to come into existence when the 'Pump' is turned on. When this happens the *state machine diagram* begins in the *initial state* and then immediately transitions to the 'starting' *state*. It will stay in this *state* until the 'start' *event* is received. On receipt of this *event* the *transition* will fire and the *state machine* will move into the 'pumping forward' *state*. There are a number of points to discuss here. First, the *transition* has an *action* 'CurrentDirection = Forward'. As is common with many *actions* this is assigning a value to a *property* of the owning *block*. Is 'CurrentDirection' a *property* of the 'Pump' *block*? Yes, as it can be seen from Figure 5.64. So this *action* is consistent with the structural aspects of the model. Second, the *transition* crosses the boundary of the 'working' *composite state* and enters the 'pumping forward' *state* contained within 'working'. This is perfectly okay and is very common when working with *sequential composite states*. This initial *transition* and associated behaviour captures the fact that the 'Pump' in this example always starts pumping in the normal forward direction.

Once running, the ‘Pump’ can be switched to pump in a reverse direction. However, it has to stop pumping normally before it can make this change of direction. Similarly, if pumping in reverse, it can be switched back to pumping normally but, again, it has to stop pumping first. The operator does not have to explicitly tell the ‘Pump’ to stop before switching direction. The ‘Pump’ has to handle this itself. This is what the three *states* inside ‘working’, together with their associated *transitions*, do.

When the ‘pumping forward’ *state* is entered, the ‘Pump’ primes itself. This is achieved with an *entry activity* ‘Entry/prime’. This invokes the ‘prime’ *operation* of the ‘Pump’. This cannot be interrupted; the ‘Pump’ will always complete its ‘prime’ operation before it does anything else. Once the ‘Pump’ has finished priming itself, it then begins pumping via an *activity* ‘do: pump’. This can be interrupted. If not interrupted, then the ‘pump’ *operation* will run to completion. If it is interrupted, then the ‘pump’ *operation* will cease and the ‘pumping’ state will be left along whichever *transition* fired causing the interruption.

So what *transitions* are possible from ‘pumping’ and what will cause them to happen? The most obvious is the *transition* from ‘pumping’ to ‘reversing’. This *transition* has an *event* ‘reverse’ and no *guard condition* or *action*. If ‘reverse’ is received by the *state machine diagram* while in the ‘pumping’ *state* then this *transition* will fire and the ‘reversing’ *state* will be entered. Don’t forget: the ‘do: pump’ *activity* can be interrupted, so this *event* can cause the ‘pump’ *operation* to cease prematurely. Another possibility, perhaps not so obvious, is the *transition* from the ‘working’ *sequential composite state* to the ‘stopping’ *state*. This *transition* is drawn from the boundary of ‘working’. This means that it is a valid *transition* from all of the *states* contained within. Essentially all three *states* have a *transition* triggered by the ‘stop’ *event* to the ‘stopping’ *state*’. This illustrates the common use of *sequential composite states*; they are used to enclose *states* that all have the same *transitions* from them, allowing a cleaner diagram to be produced. Again, this *transition*, should it fire, will end the ‘pump’ *operation* prematurely.

If the *transition* to ‘reversing’ fires, then the *state machine* will move into the ‘reversing’ state where an *activity* will invoke the ‘stopPump’ *operation.* Again, this behaviour can be interrupted by the *transition* triggered by the ‘stop’ *event* from the ‘working’ *state*. However, it cannot be interrupted by either of the two *transitions*, which directly leave the ‘reversing’ *state*. Why? Because neither of the two *transitions* from ‘reversing’ has *events*. They only have *guard conditions* and *actions*. Only *transitions* with *events* can interrupt behaviour in a *state*. Those without *events* will be checked once any behaviour inside the state has completed. Thus, as soon as the ‘stopPump’ *operation* has finished (assuming the ‘stop’ *event* has not caused the *transition* to ‘stopping’ to fire), then the two *guard conditions* on the *transitions* are checked. Whichever is true determines which *transition* takes place. Both of these *guard conditions* check the value of the ‘CurrentDirection’ *property* to establish whether the ‘Pump’ is currently pumping in the normal direction or is pumping in reverse. In this case, the *guard condition* ‘[CurrentDirection = Forward]’ will be true, since this is the direction that was set on entry to the ‘pumping forward’ *state*.

Therefore, the *transition* to the 'pumping reverse' *state* will fire, and the *action* 'CurrentDirection = Reverse' is executed to track that the 'Pump' is now in the 'pumping reverse' *state*.

The behaviour of the 'pumping reverse' *state* is now the opposite of the 'pumping forward' *state*. There is no need for the 'Pump' to prime itself as this was already done and the 'Pump' has just been pumping, so the 'pumpReverse' *operation* is immediately invoked. This will either run to completion or be interrupted in exactly the same way as for 'pump' in the 'pumping' *state*. A 'reverse' *event* will cause the *transition* to 'reversing' to fire or a 'stop' *event* will cause a *transition* to 'stopping' to fire. If the *transition* to 'reversing' happens, then the behaviour is described previously except that the other *guard condition* is now true and the *transition* back to 'pumping forward' will take place.

Thus, the 'start' *event* will start the 'Pump' pumping normally and each receipt of the 'reverse' *event* will cause it to toggle to pumping in reverse and then back to pumping normally, with the 'Pump' stopping automatically before changing direction.

When in any of the 'pumping forward', 'reversing' or 'pumping reverse' then receipt of the 'stop' *event* will cause the *transition* to the 'stopping' *state* to fire. On entry to this *state* the 'Pump' is flushed ('Entry/flush') before the 'stopPump' *operation* is invoked.

If all of the preceding explanation of the behaviour of the *state machine diagram* in Figure 5.65 seems convoluted, perhaps it will help to reinforce the benefits of modelling with a language such as SysML. An experienced modeller would have understood all of the above description simply by looking at the diagram in Figure 5.65.

Finally, an important consideration when constructing *state machine diagrams* is that of determinism. When leaving a *state* it is important that only one of the *transitions* can be followed. This means that the *events* and *guard conditions* on all the *transitions* from a *state* must be mutually exclusive; in this way only one *transition*, at most, will ever occur. If more than one *transition* could occur, then the *state machine diagram* is said to be non-deterministic and the exact behaviour is impossible to determine. There is a place for non-deterministic *state machine diagram* but their discussion is outside the scope of this book.

5.5.6.3 Summary

State machine diagrams realise a behavioural aspect of the model. They model the order in which things occur and the logical conditions under which they occur for instances of *blocks*, known in SysML as *instance specifications*. They show such behaviour by relating it to meaningful *states* that the System Element, modelled by a *block*, can be in at any particular time, concentrating on the *events* that can cause a change of *state* (known as a *transition*) and the behaviour that occurs during such a *transition* or that occurs inside a *state*.

There are a few rules of thumb to apply when creating *state machine diagrams*:

- All *blocks* that exhibit behaviour (have *operations*) must have their behaviour specified. If the System Element modelled by the *block* can be in a number of *states* then this behaviour should be modelled using a *state machine diagram*.

If it does not exhibit such stateful behaviour, then consider using *activity diagrams*. Whichever is chosen, the behavioural aspect of the *block* must be modelled.

- All *operations* in a particular *block* that has its behaviour modelled using a *state machine diagram* must appear on its associated *state machine diagram*. *States* may be empty and have no *activities*, which may represent, for example, an idle *state* where the System is waiting for an *event* to occur. *Messages* are sent to and received from other *state machine diagrams* as *send events* and *receipt events*.

Also, remember that there is a difference between behaviour modelled using *actions* on a *transition* and behaviour modelled using *activities* within a *state*. *Actions* are *atomic*. They are considered to take zero logical time and once started cannot be interrupted. *Activities*, on the other hand, do take time to run and can be interrupted (but remember that *entry activities* and *exit activities* are guaranteed to complete). It is important to differentiate between *activities* and *actions* as they can have a large impact on the way in which the model of a System will evolve and an even bigger impact on how it is implemented.

5.5.7 Sequence diagrams

This section introduces and discusses *sequence diagrams*, which realise a behavioural aspect of the model. The main aim of the *sequence diagram* is to show a particular example of operation of a System, in the same way as movie-makers may draw up a storyboard. A storyboard shows the sequence of events in a film before it is made. Such storyboards in MBSE are known as Scenarios. Scenarios highlight pertinent aspects of a particular situation and ignore all others. Each of these aspects is represented as an element known as a *life line*. A *life line* in SysML represents an individual participant in an interaction and will refer to an element from another aspect of the model, such as a *block*, a *part* or an *actor*. *Sequence diagrams* model interactions between *life lines*, showing the *messages* passed between them with an emphasis on logical time or the sequence of *messages* (hence the name).

5.5.7.1 Diagram elements

Sequence diagrams are made up of two main elements, *life lines* and *messages*, along with additional elements that allow other diagrams to be referenced, *interaction uses*, and constructions such as looping and parallel behaviour to be represented, represented using *combined fragments*. These elements are shown in Figure 5.66.

Figure 5.66 shows the partial meta-model for *sequence diagrams*. From the model it can be seen that a ‘Sequence Diagram’ is made up of one or more ‘Life Line’, one or more ‘Message’, zero or more ‘Interaction Uses’ and zero or more ‘Combined Fragment’, which has types ‘Loop Combined Fragment’, ‘Parallel Combined Fragment’ and ‘Alternative Combined Fragment’. An ‘Interaction Use’ references a ‘Sequence Diagram’ and each ‘Combined Fragment’ spans one or more ‘Life Line’. A ‘Message’ connects two ‘Occurrence Specification’, each of which occurs on a ‘Life Line’. Each ‘Life Line’ is made up of zero or more ‘Execution Specification’.

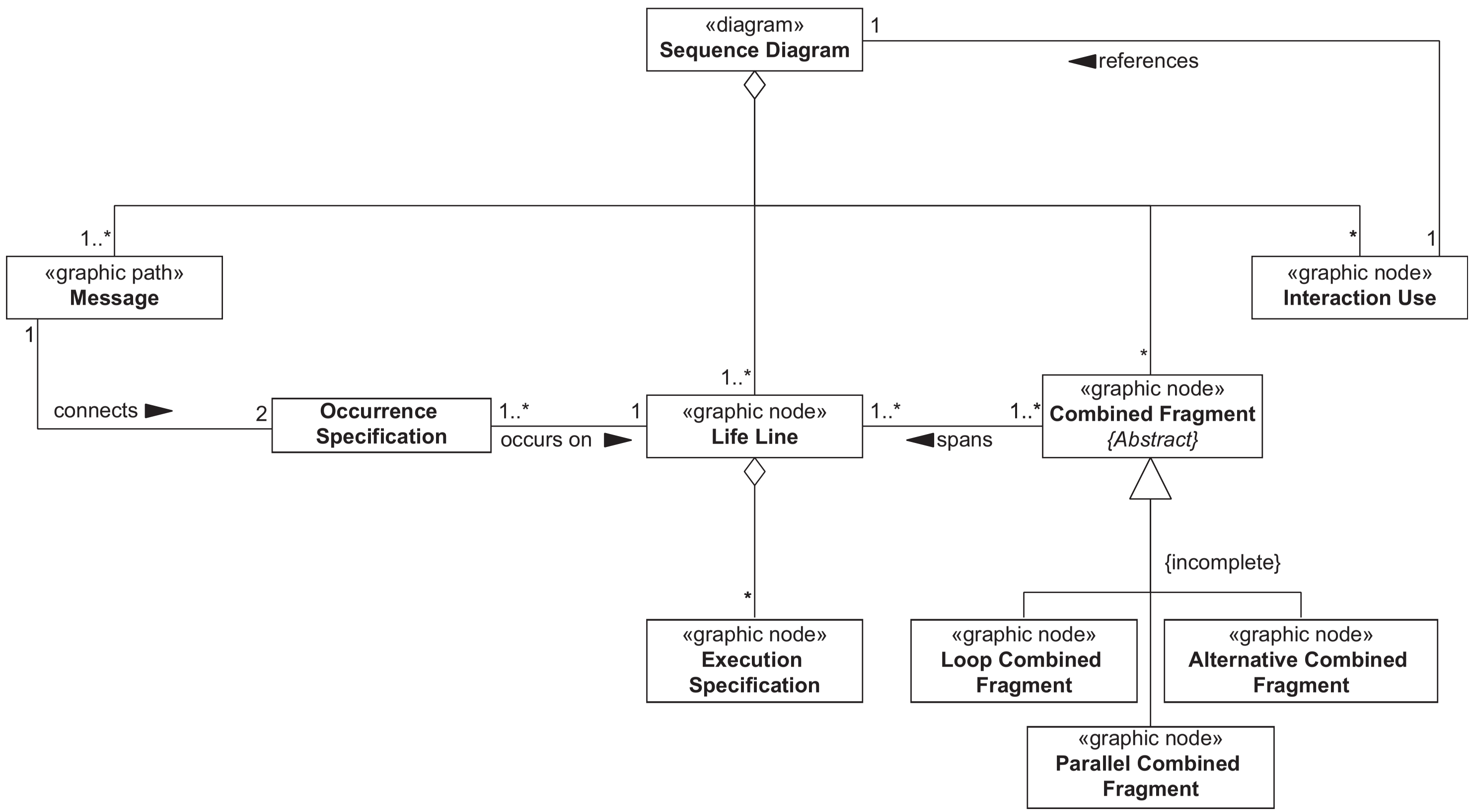

Figure 5.66 Partial meta-model for the sequence diagram

The notation for the *sequence diagram* is shown in Figure 5.67.

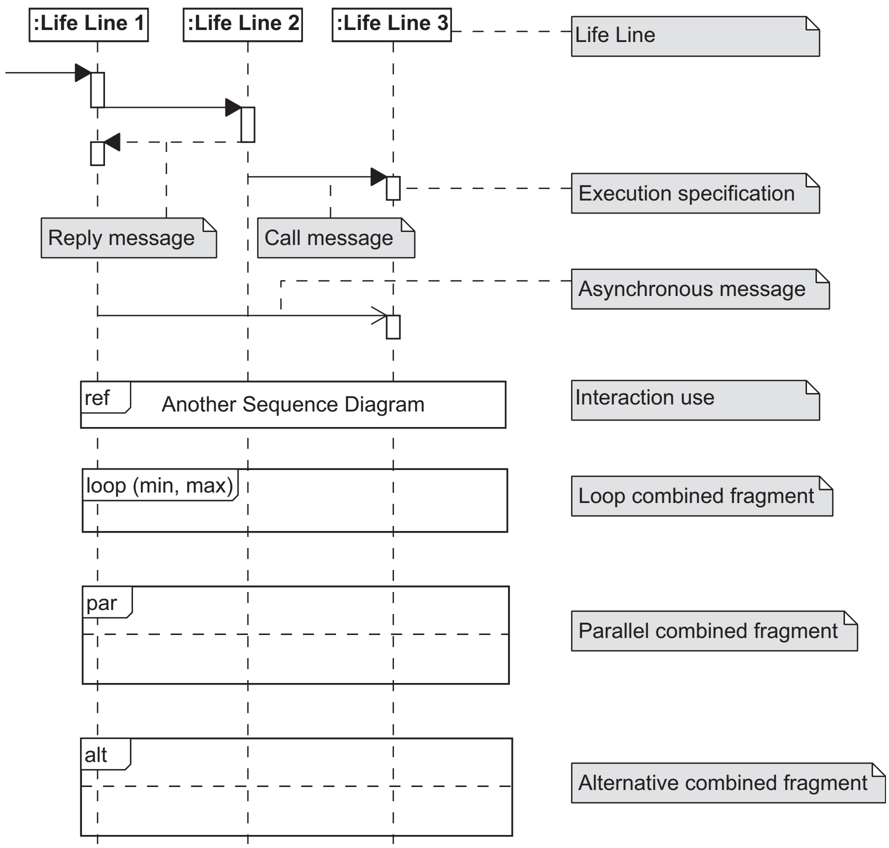

Figure 5.67 Summary of sequence diagram notation

The main element of a *sequence diagram* is the *life line*, representing a participant in a Scenario over a period of time. It is represented by a rectangle with a dashed line hanging below it, as shown in Figure 5.67. The dashed line represents logical time extending down the diagram, with earlier times at the top and later times at the bottom. The *sequence diagram* is the only SysML diagram in which layout is important, as indicated by this time dimension. A *life line* will refer to an element from another aspect of the model, such as a *block* or an *actor*; it can be thought of as an instance of that element that is taking part in the Scenario. This is reflected in the labelling of the *life line*, placed inside the rectangle, which takes the following form:

name : type

The *name* part of the label is optional and is used to give the *life line* a unique identifier in the case where multiple *life lines* of the same *type* are used on the same diagram. The *type* indicates the *block* or *actor* that the *life line* is an instance of and the rectangle can be adorned with the *stereotype* «block» or the stick man symbol to emphasise that the *life line* is typed by a *block* or an *actor* (see, for example, Figure 5.73).

The sequence of interaction between *life lines* is shown by *messages* drawn between the sending and receiving *life lines*. These *messages* can be annotated with text describing the nature of the interaction and show the sequence of interactions through time. The portion of time during which a *life line* is active is shown by the small rectangles on the dashed line, known as *execution specifications*. A *life line* can send a *message* to itself, to show that some internal behaviour is taking place. See, for example, Figure 5.73. The two *occurrence specifications* connected by a *message* are not explicitly shown, but are the points on the *life line* where a *message* leaves and joins a *life line*.

Complex Scenarios can be represented containing looping, parallel and alternative behaviour, shown using various types of *combined fragment*. In addition, a *sequence diagram* can refer to another via the *interaction use* notation, allowing more and more complicated Scenarios to be developed. Examples of the *combined fragment* and *interaction use* notation are shown in Figure 5.67. They are described further in the following subsections. However, it is worth sounding a note of caution here. The various *combined fragment* notations can be nested, allowing very complicated Scenarios to be modelled. In particular, the use of the *alternative combined fragment* notation allows alternative paths through a Scenario to be shown. What this means is that the *sequence diagram* is showing more than one Scenario. From a SysML perspective, there is nothing wrong with doing this. However, from a modelling perspective such an approach can, in all but the simplest of cases, lead to confusing diagrams. Apart from showing very simple alternatives on a single diagram the authors would recommend a one diagram, one scenario approach.

Showing parallel processing

Parallel paths through a Scenario can be shown in *sequence diagrams* using a *parallel combined fragment*. Each parallel path appears in a separate compartment within the *combined fragment frame*. The parallel compartments are divided by a dashed line, and the combined fragment uses the keyword *par*.

Figure 5.68 shows a *sequence diagram* with two *parallel combined fragments*, each of which has two parallel regions. The first *parallel combined fragment* shows the ‘Begin stunt’ *message* being sent from the ‘Set up’ *life line* to the ‘Start’ *life line* at the same time as the ‘Set up’ *life line* sends the ‘Begin stunt’ *message* to the ‘Escape’ *life line*. Similarly, the second *parallel combined fragment* shows the ‘Start escape’ *message* being sent between the ‘Start’ and ‘Escape’ *life lines* at the same time that it is sent between the ‘Escape’ and ‘Monitor’ *life lines*.

Referencing other diagrams

Often, when modelling Scenarios, common behaviour is observed. Rather than having to repeat this behaviour on every *sequence diagram* that needs it, SysML allows other *sequence diagrams* to be referenced to allow reuse of Scenarios.

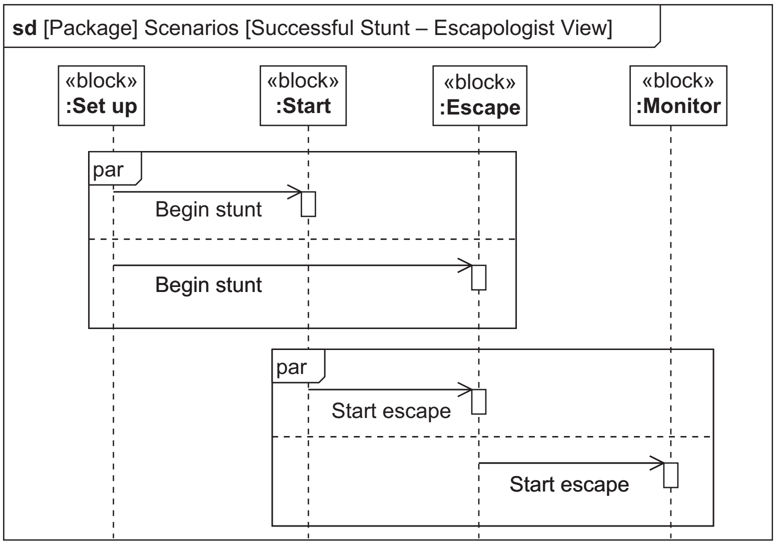

Figure 5.68 Example sequence diagram showing a parallel combined fragment

For example, say that we have some common functionality that we want to show on multiple Scenarios. First, we model this using a *sequence diagram*. An example is shown in Figure 5.69.

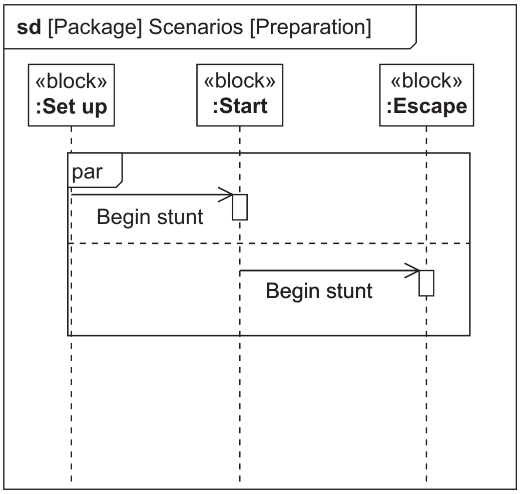

Figure 5.69 Example sequence diagram defining common functionality to be referenced

This functionality can then be reused on another *sequence diagram* using an *interaction use*. Each referenced Scenario appears in a separate frame with the keyword *ref*, as shown in Figure 5.70.

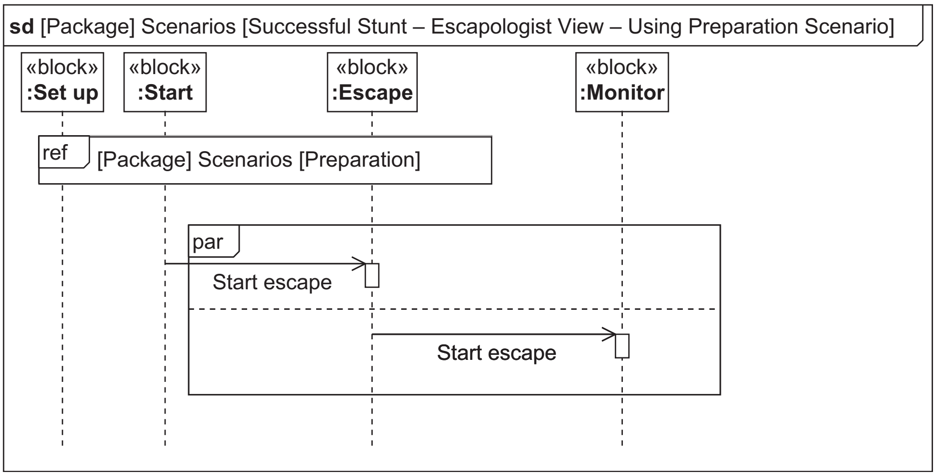

Figure 5.70 Example sequence diagram showing the use of a reference combined fragment

The *life lines* that appear in the *sequence diagram* referenced must appear on the referencing diagram and the *interaction use* must be placed over those *life lines* as in Figure 5.70.

Showing alternatives

Sometimes two or more Scenarios are so similar that showing alternative paths on a single diagram rather than one per diagram is desirable. SysML allows Scenarios to be modelled in this way using *alternative combined fragments*.

This consists of a *frame* with the keyword *alt* that is divided into separate compartments, one for each alternative, by dashed lines. Each compartment should have a *guard condition* that indicates the conditions under which that alternative is executed. The absence of a *guard condition* implies a true condition. The *guard condition else* can be used to indicate a condition that is true if no other *guard conditions* are true. Although there is nothing in SysML to prevent the use of *guard conditions* where more than one can evaluate to true, this leads to a non-deterministic *sequence diagram* and is to be avoided. An example of a *sequence diagram* showing two alternatives is shown in Figure 5.71.

The diagram in Figure 5.71 shows two Scenarios, since the *alternative combined fragment* has two compartments. Both Scenarios begin with the 'Assistant' sending a 'start' *message* to the 'Pump Controller', which itself sends a 'start' *message* to the 'Pump'. The 'Pump' then sends itself two *messages*, 'prime' followed by 'pump'.

In the first Scenario, when the *guard* 'Emergency = FALSE' holds, the first alternative takes place. The 'Assistant' sends a 'stop' *message* to the 'Pump Controller', which itself sends a 'stop' *message* to the 'Pump'. The 'Pump' then sends itself two *messages*, 'flush' followed by 'stopPump'.

In the second Scenario, when the *guard* 'Emergency = TRUE' holds, the second alternative takes place. The 'Assistant' sends a 'reverse' *message* to the 'Pump

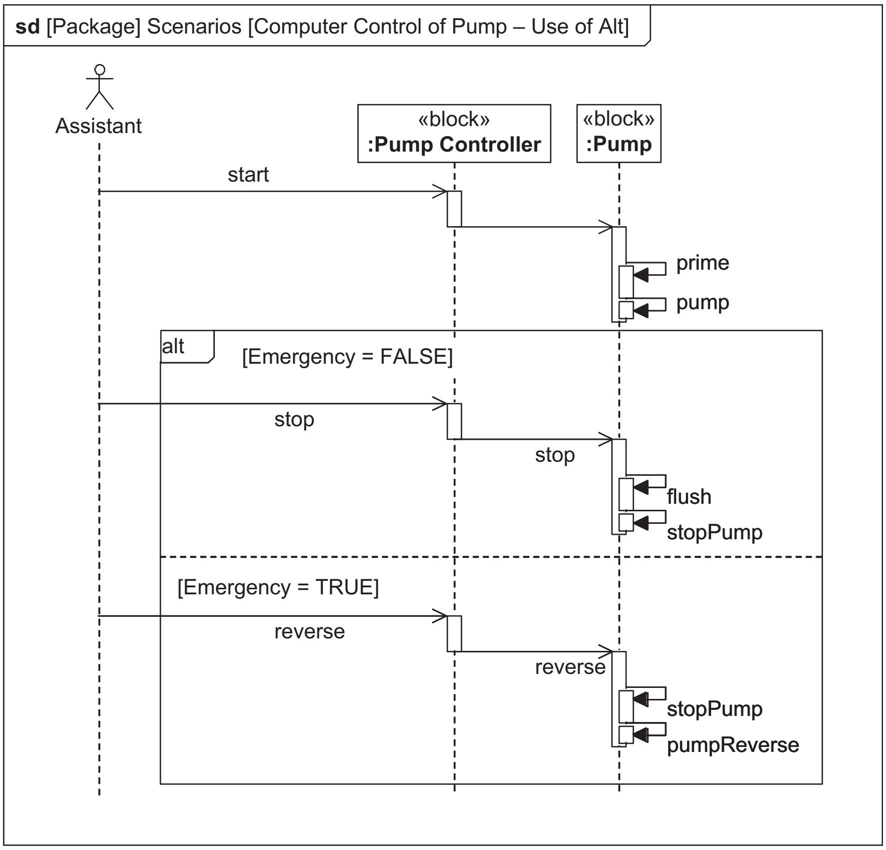

Figure 5.71 Example sequence diagram showing the use of the alternative combined fragment

Controller', which itself sends a 'reverse' *message* to the 'Pump'. The 'Pump' then sends itself two *messages*, 'stopPump' followed by 'pumpReverse'.

Showing loops

The final *combined fragment* to be considered allows looping behaviour to be shown. The *looping combined fragment* is shown using a *frame* with the keyword *loop*. The keyword may be accompanied by a *repetition count* specifying a *minimum* and *maximum* count as well as a *guard condition*. The loop is executed while the *guard condition* is true but *at least* the minimum count, irrespective of the *guard condition* and *never* more than the maximum count.

The syntax for loop counts is

- loop minimum = 0, unlimited maximum
- loop(repeat) minimum = maximum = repeat
- loop(min, max) minimum & maximum specified, min <= max

An example *sequence diagram* showing a loop combined fragment is shown in Figure 5.72.

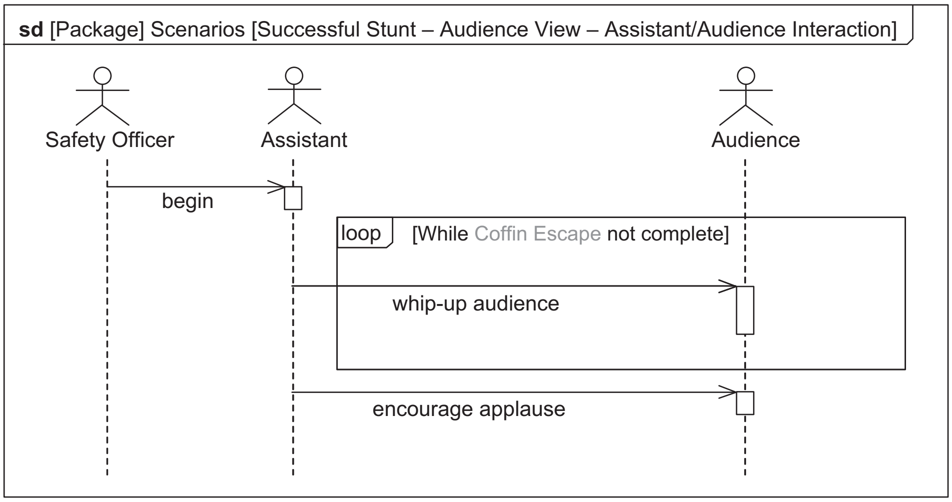

Figure 5.72 Example sequence diagram showing the use of a loop combined fragment

The diagram shows a *loop* with no *repetition count* (which is the same as a loop forever) and a *guard condition* that indicates that the *loop* is to continue while the Coffin Escape Stunt is not complete.

There are many other types of *combined fragment* defined, but the four discussed here are the most often used. For details of the other types of *combined fragment*, such as the *break* or *opt combined fragments*, see Reference 5.

In addition, there is nothing to prevent the nesting of *combined fragments*. For example, a loop may have a *parallel combined fragment* inside it, with *instance uses* and perhaps even *alternative combined fragments* in each parallel region. Remember, though, that one of the key aims of modelling is to improve the communication of complex ideas and such diagrams, while valid SysML should be used with caution as diagrams can rapidly become very difficult to understand and make the communication worse rather than better.

There is much more notation available for use on *sequence diagrams*, including the modelling of *timing constraints* between *messages* and the distinction between *synchronous* and *asynchronous* messages. See References 1, 3, 5 and 6 for further information.

5.5.7.2 Examples

This section presents some examples of *sequence diagrams*. Further examples of sequence diagrams can be found in the case study in Chapter 13.

Figure 5.73 is an example of a *sequence diagram* that treats the System (in this case the 'Coffin Escape') as a black box; that is, it concentrates on the interactions between Stakeholder Roles and the System, modelling the System as a single *life*

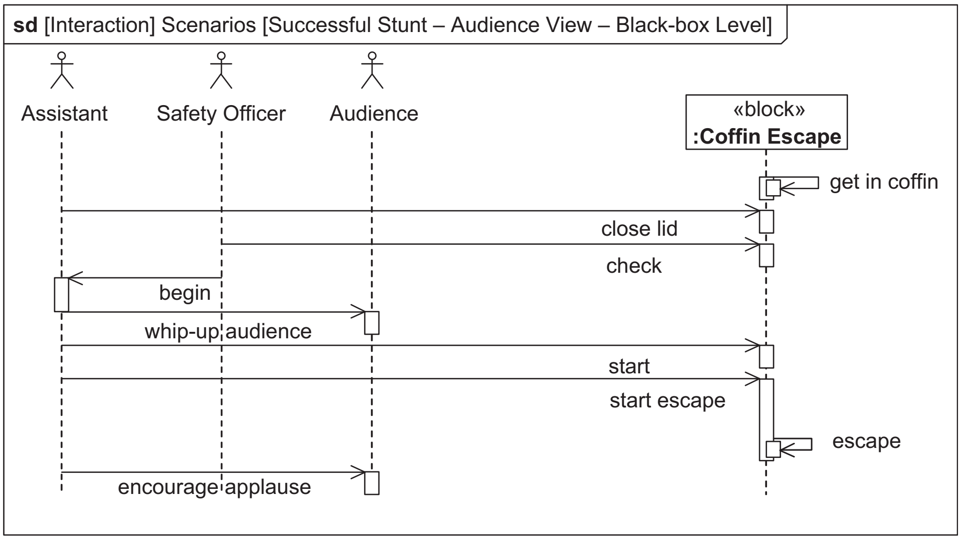

Figure 5.73 Example sequence diagram showing actors as life lines and System as a single block

line. As well as showing these interactions, it also shows some interactions that are internal to the System, namely the 'get in' and 'escape' *messages*.

Three other interactions are also worthy of comment, namely the 'begin', 'whip-up audience' and 'encourage applause' *messages*. These are of interest because they are between Stakeholder Roles rather than between Stakeholder Roles and the System. Some people (and indeed some SysML tools) would consider such interactions as illegal.

Nevertheless, these are essential interactions that are needed to fully describe the Scenario (in this case, that of a successful stunt) as it is impossible to model this Scenario fully without showing them. When considering the System to be the 'Coffin Escape' consisting of equipment, Processes and the Escapologist, then the Stakeholder Roles shown in Figure 5.73 as *actor life lines are* outside the System. But this is a question of context. In the wider context of the stunt being performed that includes all the necessary supporting roles and the audience, then these Stakeholder Roles are part of the System and therefore these interactions become interactions between System Elements.

Figure 5.74 shows a simple Scenario, that of the assistant starting and stopping the pump used in the stunt. However, unlike in Figure 5.73, the System is no longer treated as a black box. In this diagram, the individual elements of the System are shown along with the relevant Stakeholder Role who is shown interacting with one of the System Elements (the 'Pump Controller'). The internal interactions between the 'Pump Controller' and the 'Pump' are also shown, as is the behaviour that takes place inside the 'Pump'. Thus, it can be seen that when the 'Pump' receives a 'start' *message* it primes itself and then begins pumping. Similarly, on receipt of a 'stop' *message* it first flushes itself before stopping. Such white box Scenarios are

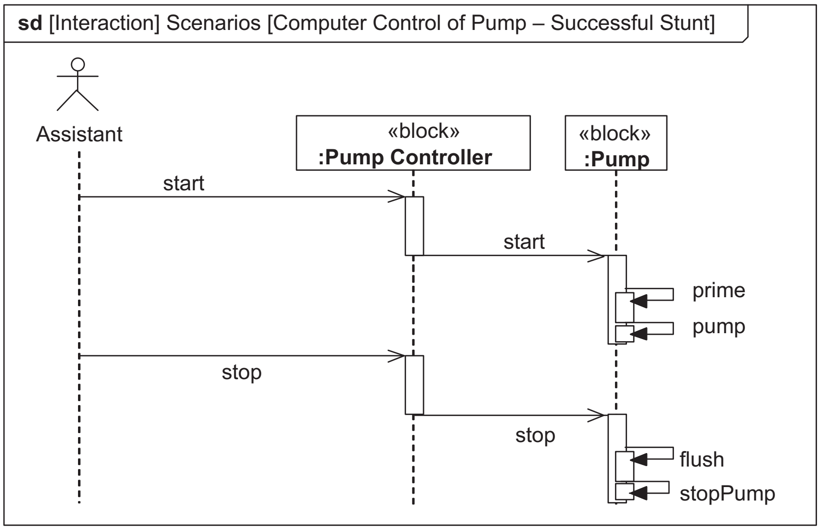

Figure 5.74 Example sequence diagram showing interactions between System Elements

typically developed from black box Scenarios, which may have been developed earlier during the requirements engineering process. An equivalent black box Scenario for Figure 5.74 is shown in Figure 5.75.

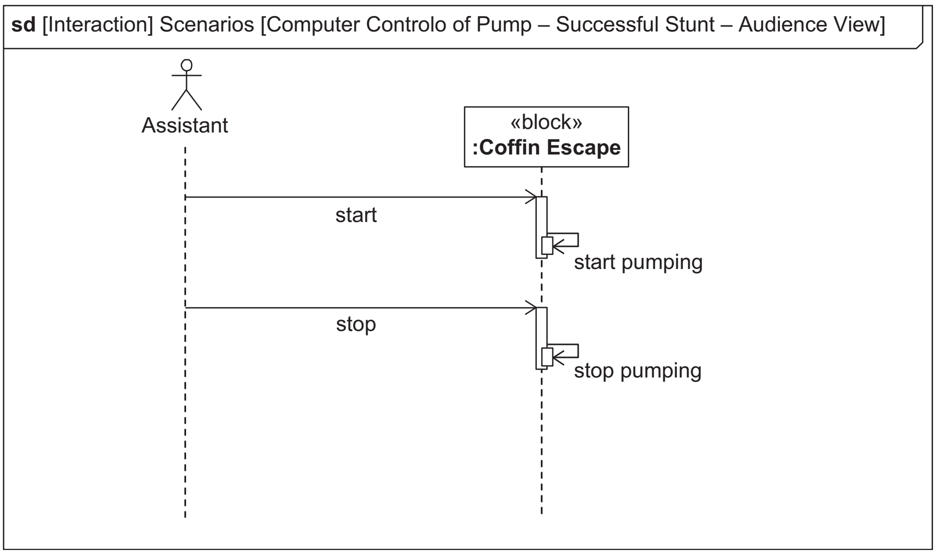

Figure 5.75 Example sequence diagram showing equivalent black box Scenario

As Figure 5.75 is intended to be the black box Scenario from which Figure 5.74 is developed, the diagrams should be consistent. One would expect the interactions between the ‘Assistant’ and the ‘Coffin Escape’ System in Figure 5.75 to be the same as those between the ‘Assistant’ and the relevant System Element (in this case the ‘Pump Controller’) in Figure 5.74, as indeed they are. Similarly the interactions of the System with itself in Figure 5.75 should be consistent with those between System Elements in Figure 5.74. In this case, although the *messages* are not labelled the same, they are consistent with one another. The difference here is due to the differing levels of abstraction shown on the two diagrams. A single *message* at the black box System level is refined into a number of *messages* between and within System Elements when the Scenario is modelled in more detail.

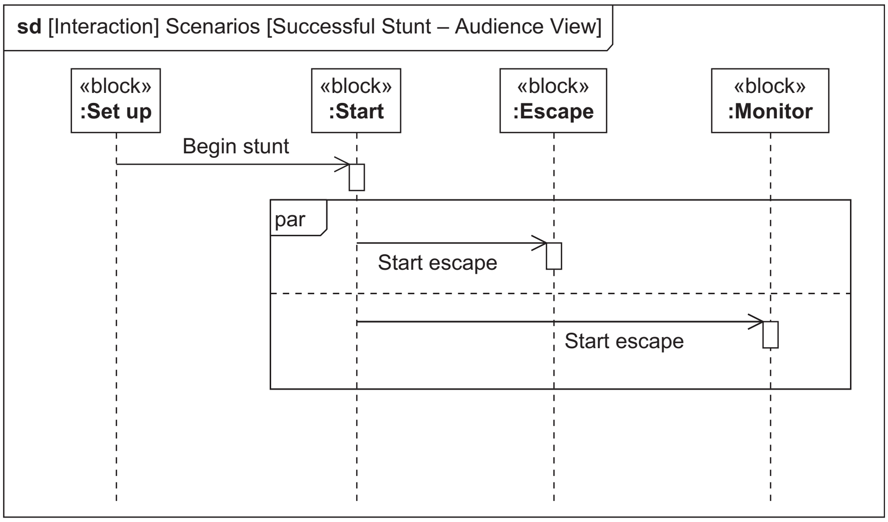

Figure 5.76 Example sequence diagram showing use of parallel combined fragment

The final example in this section, Figure 5.76, shows a Scenario where the System Elements are not pieces of equipment but rather represent Processes that are carried out as part of the System. The *messages* between the Processes show how one Process initiates another, in this case for the Scenario showing the successful execution of the stunt. In this Scenario the ‘Start’ Process, on completion, has to trigger the ‘Escape’ and ‘Monitor’ Processes that have to run in parallel. This is shown by the use of the *parallel combined fragment*, containing two parallel regions, surrounding the two ‘Start Escape’ *messages* sent by the ‘Start’ Process.

5.5.7.3 Summary

Sequence diagrams are used to model Scenarios. They show behaviour through time, through the passage of *messages* between *life lines* that represent the participants in the Scenario. When modelling Scenarios, this can be done as black box Scenarios, modelling the System as a single *life line*, or as white box Scenarios that show System Elements:

- Black box Scenarios are often generated when the Scenario is placing the emphasis on the interactions from the point of view of one or more Stakeholder Roles. An example of such a diagram is the Stakeholder Scenario View in ACRE (see Chapter 9).
- White box Scenarios are often generated when the emphasis is on the interactions between System Elements. An example of such a diagram is the System Scenario View in ACRE (see Chapter 9).

In practice, Stakeholder Roles often have to be shown interacting with System Elements, so the distinction is often blurred.

5.5.8 Activity diagrams

This section looks at another behavioural diagram, the *activity diagram*. *Activity diagrams*, generally, allow very low-level modelling to be performed compared to the behavioural models seen so far. Where *sequence diagrams* show the behaviour between elements and *state machine diagrams* show the behaviour within elements, *activity diagrams* may be used to model the behaviour within an *operation*. The other main use for *activity diagrams* is for modelling Processes. For a detailed discussion of Process modelling with SysML see Chapters 7 and 8.

5.5.8.1 Diagram elements

The main elements that make up *activity diagrams* are shown in Figure 5.77.

Figure 5.77 shows a partial meta-model for *activity diagrams*. It shows that an ‘Activity Diagram’ is made up of three basic elements: one or more ‘Activity Node’, one or more ‘Activity Edge’ and zero or more ‘Region’. There are three main types of ‘Activity Node’, which are the ‘Action’, the ‘Object’ and the ‘Control Node’ all of which will be discussed in more detail later in this section. The ‘Action’ is where the main emphasis lies in these diagrams and represents a unit of behaviour on the ‘Activity Diagram’. There are many different types of ‘Action’ available, the discussion of which is beyond the scope of this book. We will treat them all the same, but for a full discussion see Reference 4. An ‘Action’ can also have zero or more ‘Pin’, which can be used to show an ‘Object Flow’ that carries an ‘Object’. This is discussed further.

An ‘Activity Edge’ connects one or two ‘Activity Node’; it can connect an ‘Activity Node’ to itself, hence the multiplicity of one or two, rather than just two. The ‘Activity Edge’ element has two main types – ‘Control Flow’ and ‘Object Flow’. A ‘Control Flow’ is used to show the main routes through the ‘Activity Diagram’ and connects together one or two ‘Activity Node’. An ‘Object Flow’ is used to show the flow of information between one or more ‘Activity Node’ and does so by carrying the ‘Object’ type of ‘Activity Node’.

The other major element in an *activity diagram* in the ‘Region’ has two main types: ‘Interruptible Region’ and ‘Activity Partition’. An ‘Interruptible Region’ allows a boundary to be put into an *activity diagram* that encloses any *actions* that

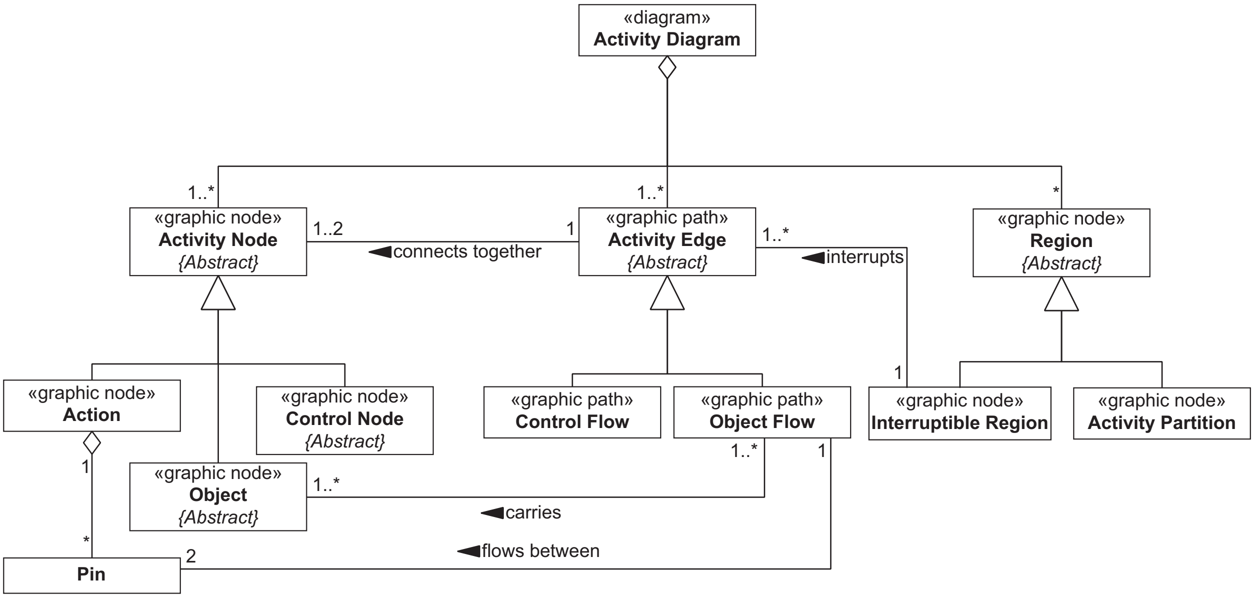

Figure 5.77 Partial meta-model for the activity diagram

may be interrupted. This is particularly powerful for Systems where behaviour may be interrupted by atypical conditions, such as software interrupts and emergency situations. For example, by a direct user interaction or some sort of emergency event. The 'Activity Partition' is the mechanism that is used to visualise swim lanes that allow different *actions* to be grouped together for some reason, usually to show responsibility for the *actions*.

The diagram in Figure 5.78 shows an expanded view of the types of 'Control Node' that exist in SysML. Most of these go together in twos or threes, so will be discussed together.

- The 'Initial Node' shows where the *activity diagram* starts. Conversely, the end of the *activity diagram* is indicated by the 'Activity Final Node'. The 'Flow Final Node' allows a particular *flow* to be terminated without actually finishing the diagram. For example, imagine a situation where there are two parallel control flows in a diagram and one needs to be halted whereas the other continues. In this case, a *final flow node* would be used as it terminates a single *flow* but allows the rest of the diagram to continue.
- The 'Fork Node' and 'Join Node' allow the *flow* in an *activity diagram* to be split into several parallel paths and then re-joined at a later point in the diagram. *Fork nodes* and *join nodes* (or *forks* and *joins* as they are usually known) use a concept of token passing, which basically means that whenever a *flow* is split into parallel *flows* by a *fork*, then imagine that each *flow* has been given a *token*. These *flows* can only be joined back together again when all *tokens* are present on the *join* flow. It is also possible to specify a Boolean condition on the *join* to create more complex rules for re-joining the *flows*.

- The ‘Decision Node’ and ‘Merge Node’ also complement one another. A ‘Decision Node’ allows a *flow* to branch off down a particular route according to a *guard condition*, whereas a ‘Merge Node’ allows several *flows* to be merged back into a single *flow*.

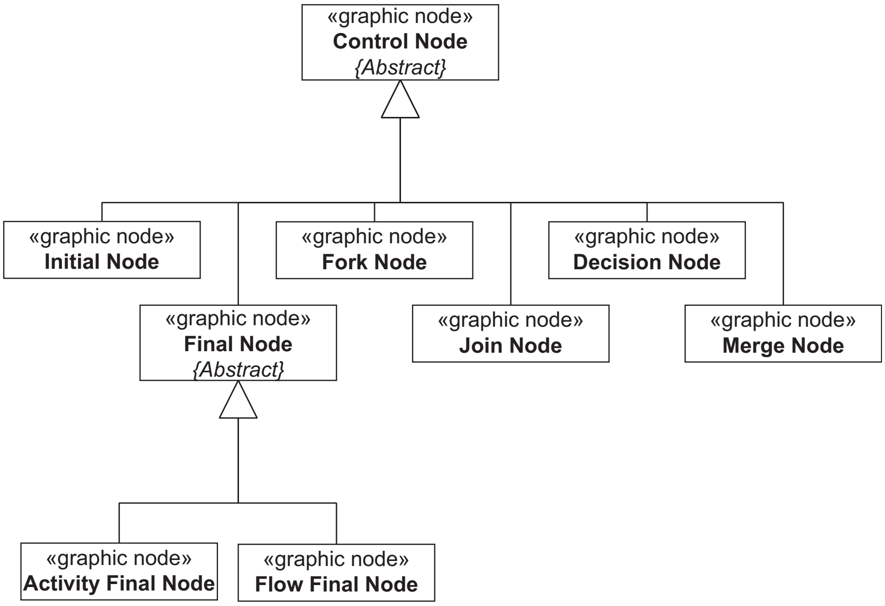

Figure 5.78 Expanded partial meta-model of the activity diagram, focusing on ‘Control Node’

There are three types of symbol that can be used on an *activity diagram* to show the flow of information carried by an ‘Object Flow’: the ‘Object Node’, the ‘Signal’

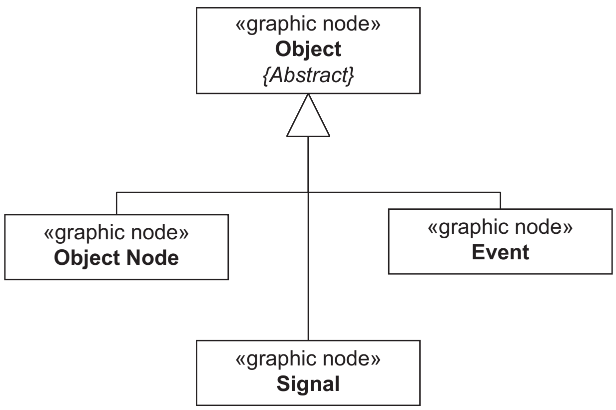

Figure 5.79 Expanded partial meta-model for the activity diagram, focusing on ‘Object Node’

and the 'Event'. See Figure 5.79. The 'Object Node' is used to represent information that has been represented elsewhere in the model by a *block* and which is forming an input to or an output from an *action*. It can be thought of a representing an *instance specification*. The 'Event' symbol is used to show an *event* coming into an *activity diagram*, whereas a 'Signal' is used to show an *event* leaving an *activity diagram.* They correspond to *receipt events* and *send events* of a *state machine diagram*. There is a special type of 'Event', known as a 'Time Event' that allows the visualisation of explicit timing events.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their *stereotypes*, and is illustrated in Figure 5.80.

In addition to the elements mentioned so far, SysML has notation that can be applied to an 'Activity Edge' and an 'Object Node'. This notation makes use of the

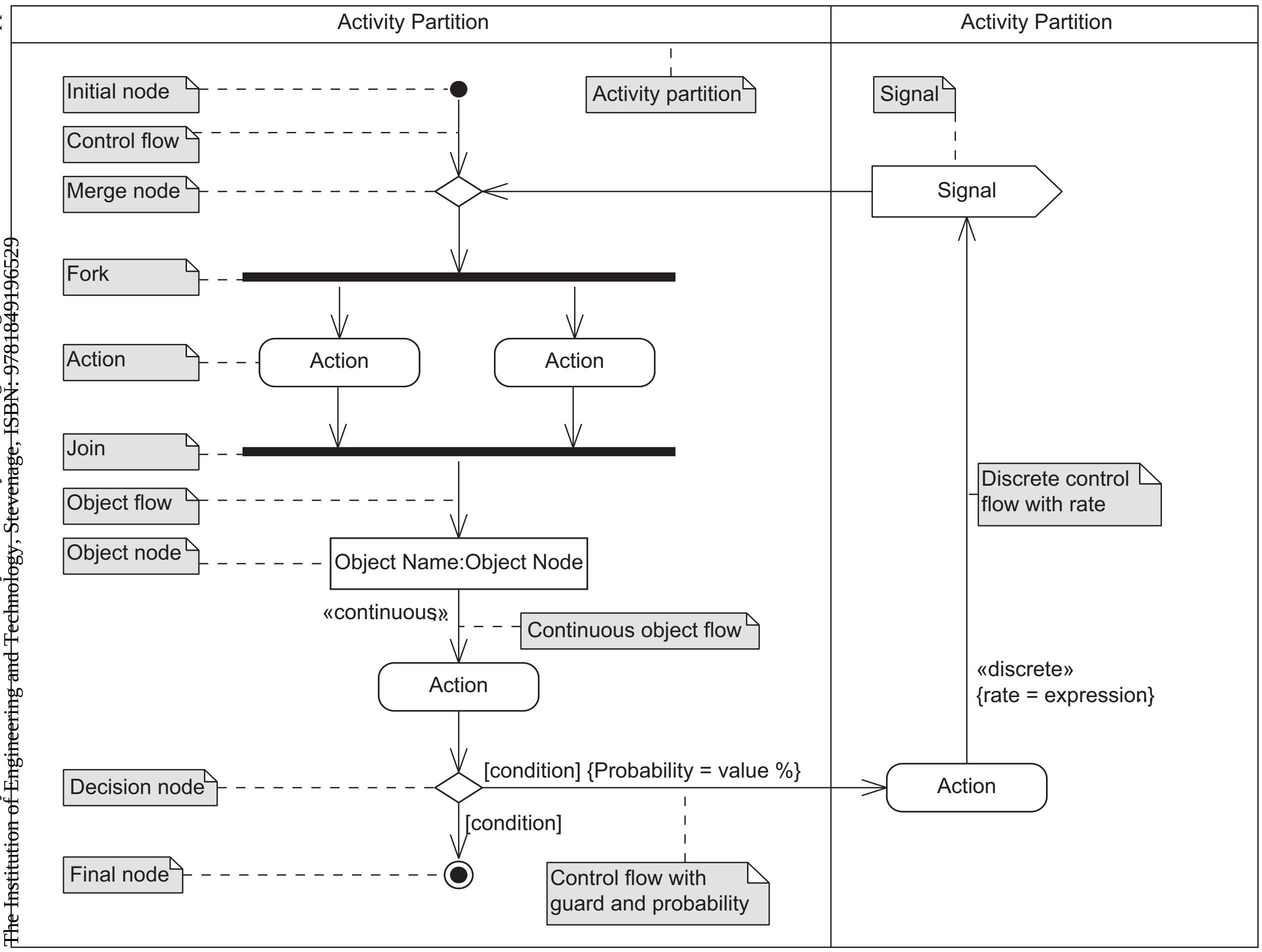

Figure 5.80 Summary of activity diagram notation

existing *constraint* and *stereotype* notation that is already present in SysML and simply defines some standard *constraints* and *stereotypes* for use on *activity diagrams*.

The first of these notations allows a *rate* to be applied to an 'Activity Edge' (and, more specifically, normally to an 'Object Flow') in order to give an indication of how often information flows along the edge. *Flows* can be shown to be *discrete* or *continuous*. This is shown by use of the «discrete» or «continuous» *stereotypes* placed on the *flow*. Alternatively the actual rate can be shown using a *constraint* of the form: {rate = expression}. For example, if data or material passed along a *flow* every minute, then this could be shown by placing the *constraint* {rate = per 1 minute} on the *flow*.

The second notation allows for a probability to be applied to an 'Activity Edge' (typically on 'Control Flow' edges leaving a 'Decision Node') and indicates the probability that the *edge* will be traversed. It can be represented as a number between 0 and 1 or as a percentage. All the probabilities on edges with the same source must add up to 1 (or 100%). It is important to note that the actual *edge* traversed is governed by the *guard conditions* on the 'Decision Node' and not by the probability. The probability is nothing more than an additional piece of information that can be added to the diagram.

The other notation modifies the behaviour of an 'Object Node' and is indicated by the use of the *stereotypes* «nobuffer» and «overwrite». If an *object node* is issued by an *action* and is not immediately consumed by its receiving *action*, then that *object node* can block the operation of the originating *action* until it is consumed by the receiving *action*. «nobuffer» and «overwrite» modify this behaviour:

- «nobuffer» means that the marked *object node* is immediately discarded if the receiving *action* is not ready to receive it. The originating *action* will not be blocked and can continue to generate *object nodes*, which will be discarded if not yet needed.
- «overwrite» means that the marked *object node* is overwritten if the receiving *action* is not ready to receive it. The originating *action* will not be blocked and can continue to generate *object nodes*. The latest generated will overwrite the previous one if not yet needed.

Figure 5.81 shows some additional notation that covers *interruptible regions* and the use of *pins* rather than *object nodes*.

Interruptible regions are shown by a dashed soft box surrounding the region to be interrupted. There must always be a normal flow of control through the *interruptible region*. In this example, the *flow* is into 'Action3' then to 'Action4' and then out of the *region*. There must also be an *event* that causes the interruption: 'Event1' in the example. The *event* is connected by a *control flow* to an *action* outside the *interruptible region*, which acts as an interrupt handler: 'Action7' in the example. The *control flow* is either annotated with a lightning bolt symbol, as here, or may be drawn as such a lightning bolt. In the example above the *interruptible region* shows that while 'Action3' or 'Action4' are taking place, they may be interrupted by 'Event1', which will cause control to transfer to 'Action7'.

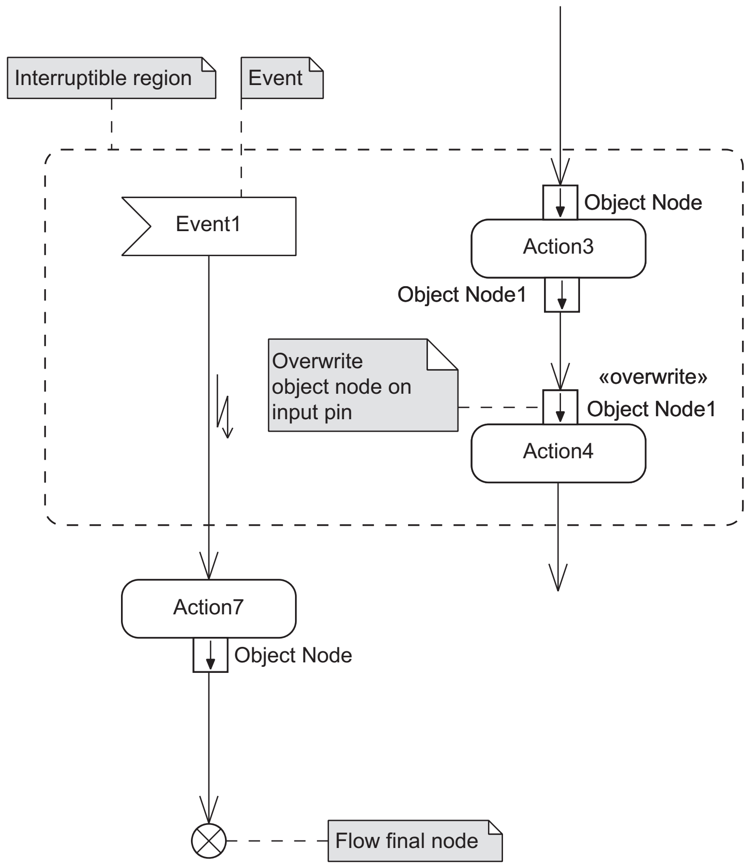

Figure 5.81 Activity diagram notation for showing interruptible regions and use of pins rather than object nodes

The diagram also shows the notation for a *flow final node* and shows how *pins* may be used instead of explicit *object nodes*. The part of the diagram involving 'Action3' and 'Action4' is equivalent to the one shown in Figure 5.82.

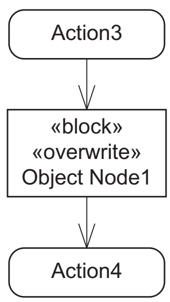

Figure 5.82 Object node notation equivalent to pin notation

Which notation is better, *pins* or *object nodes*, is a matter of personal preference (and perhaps organisational diagramming guidelines and options available in your SysML tool). The authors are firmly in favour of explicit *object nodes* rather than the version using *pins*.

5.5.8.2 Examples

The section will give a number of examples of *activity diagrams*. Additional examples can be found in Chapters 7, 8 and 13.

Figure 5.83 shows an example *activity diagram* containing a single *activity partition*. This is labelled with the model element (in this case 'Assistant') that is responsible for all the behaviour taking place inside that *activity partition.* It is

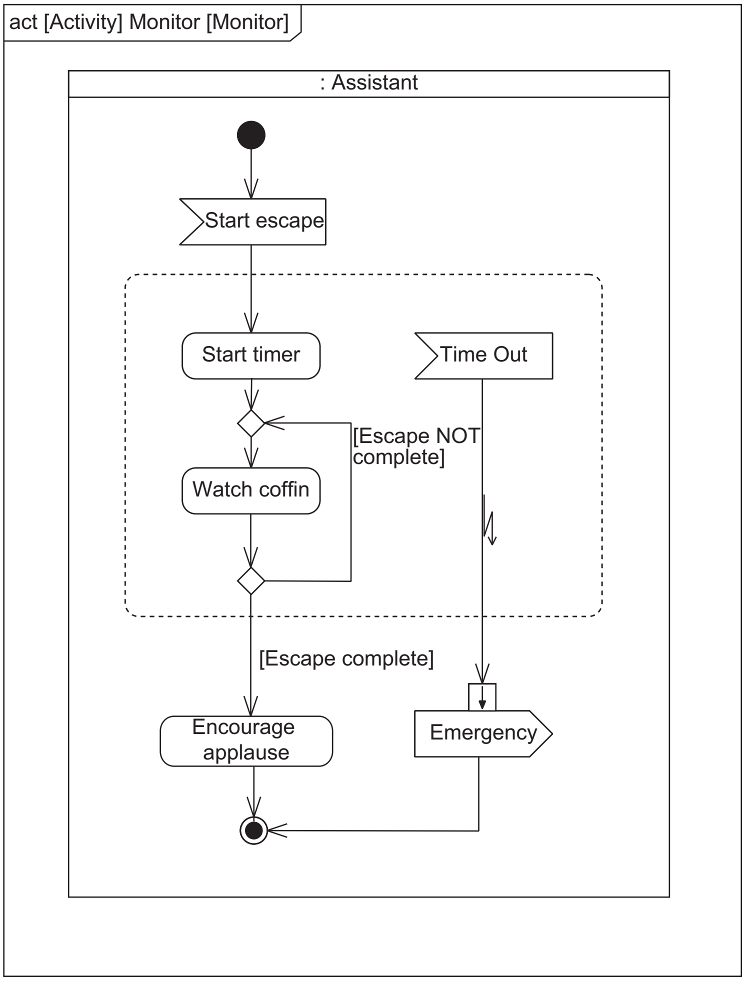

Figure 5.83 Example activity diagram showing decision, merge and interruptible region

possible to have multiple such *activity partitions* and an example is given later in this section. An *activity partition* is usually labelled with the name of a *block* or an *actor* that specifies the type of the model element responsible for the *activity partition*.

The behaviour in this *activity diagram* begins on receipt of a ‘Start escape’ event, after which control passes into an *interruptible region* where the *action* ‘Start timer’ takes place. Once this *action* is completed, control falls through a *merge node* and the ‘Watch coffin’ action takes place. When this *action* is completed a *decision node* is reached. If the *guard condition* ‘[Escape completed]’ is true, then control passes to the ‘Encourage applause’ *action* and once this is finished the *activity final node* is reached and the *activity diagram* terminates. If, instead, the *guard condition* ‘[Escape NOT complete]’ is true, then control passes back up to the *merge node* before re-entering the ‘Watch coffin’ *action*. The *merge node* is simply used to merge alternative *control flows* back into a single *control flow*.

However, the normal behaviour is not the only way in which this *activity diagram* can end. If the ‘Time out’ *event* is received at any time the ‘Start timer’ or ‘Watch coffin’ *actions* are executing, then the *interruptible region* is exited and the ‘Emergency’ *signal* is sent out of this diagram. Note the use of the *pin* on the *signal* in order to connect the *event* to the *signal*.

Another *activity diagram* is shown in Figure 5.84. This time all the behaviour is the responsibility of the ‘Escapologist’ and the *activity diagram* begins on receipt of the ‘Begin stunt’ *event*. When this is received, control enters a *fork node*, which leads to two parallel branches in which the ‘Escapologist’ is undertaking both the ‘Free hands’ *action* and the ‘Count down time’ *action*. Each of these leads into a *join node* and when both are completed, then control passes to the ‘Emerge’ *action.* If either of the two parallel *actions* failed to complete, then the ‘Emerge’ *action* would never be reached. After ‘Emerge’ is finished, the ‘Escapologist’ executes the ‘Take a bow’ *action* and then the *activity diagram* finished via the *activity final node*.

The final example we will consider here is shown in Figure 5.85. In this *activity diagram* there are two *activity partitions* and we can see from the diagram that the ‘Assistant’ is responsible for carrying out the ‘Whip-up audience’ and ‘Start pump’ actions and for issuing the ‘Start escape’ *signal*. The ‘Safety Officer’ is responsible for everything else in the diagram.

On receipt of the ‘Begin stunt’ event, the ‘Safety Officer’ will carry out the ‘Perform final check’ *action.* When this is complete control enters a *decision node* that has two branches leaving it. If the *guard condition* ‘[Problems found]’ is true then the ‘Safety Officer’ carries out the ‘Cancel stunt’ *action* and *activity diagram* terminates via the *activity final node.*

If, however, the *guard condition* ‘[No problems]’ is true, then responsibility passes to the ‘Assistant’ who carries out the ‘Whip-up audience’ and ‘Start pump’ *actions* in sequence and finally issues the ‘Start escape’ *signal*. The *activity diagram* then terminates via the *activity final node.*

However, this is not the end of the *actions* that the ‘Assistant’ has to carry out. How can this be, if there are no further *actions* in Figure 5.85? Look back at

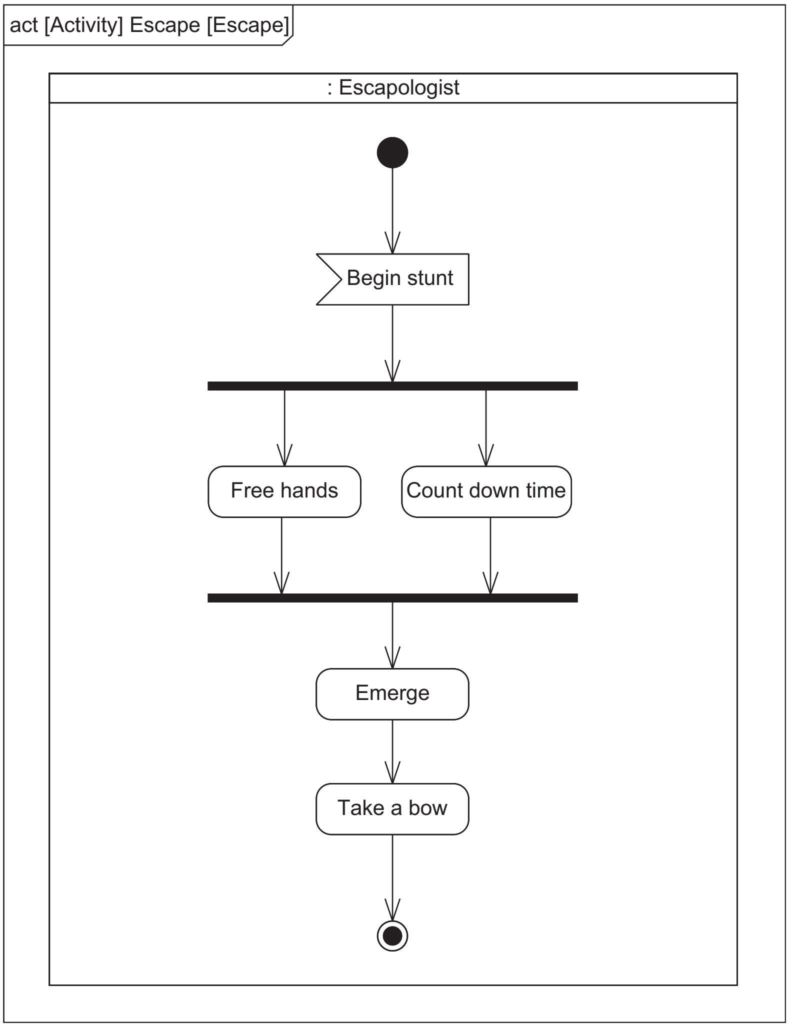

Figure 5.84 Example activity diagram showing fork nodes and join nodes

Figure 5.83. The *activity diagram* there is kicked off on receipt of a 'Start escape' *event*. This is the very event that the 'Assistant' has just issued as the 'Start escape' *signal* in Figure 5.85. The two *activity diagrams* are connected by this *event/signal* pair. This is an excellent example of the kinds of consistency between diagrams that you should be looking for when modelling. An *event* that comes into an *activity diagram* (or into a *state machine diagram* as a *receipt event*) must come from somewhere. There must be a corresponding *signal* on another *activity diagram* (or *send event* on a *state machine diagram*) that is the source of the *event*. This would,

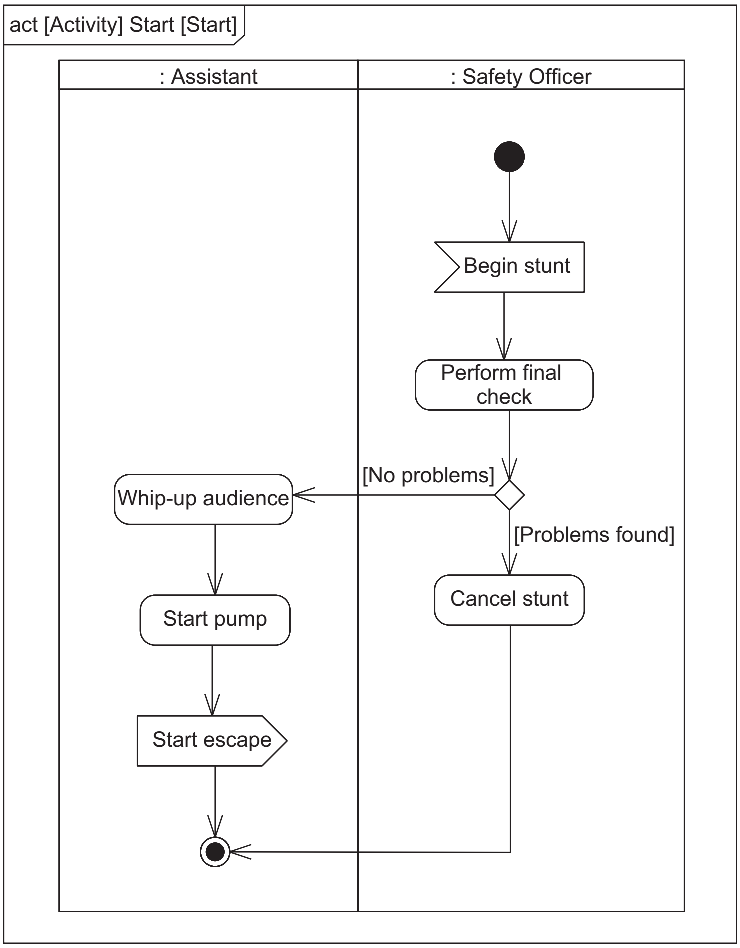

Figure 5.85 Activity diagram showing multiple activity partitions

perhaps, be less confusing if SysML used the same names across *activity diagrams* and *state machine diagrams*, but Table 5.3 may help you to remember.

Table 5.3 Equivalence of event terminology between activity *and* state machine diagrams

Activity diagram		**State machine diagram**
Event	is same as	*Receipt event*
Signal	is same as	*Send event*

Some of the nature of this communication and further consistency can be seen by looking at Figure 5.76. This shows the communication between a number of

System Elements (actually Processes). The internal behaviour of these processes is what has been modelled by the *activity diagrams* earlier in this section. Thus, the ‘Start escape’ *signal* in Figure 5.85 corresponds to the beginning of the ‘Start escape’ *message* in Figure 5.76 as it leaves the ‘:Start’ *life line*. The Start escape’ *event* in Figure 5.83 corresponds to the end of the ‘Start escape’ *message* in Figure 5.76 as it enters the ‘:Monitor’ *life line*.

Thus, *activity diagrams* can communicate with other *activity diagrams* or with *state machine diagram* and vice versa. Furthermore, the *messages* corresponding to these *events* and *signals* can be modelled as *messages* on *sequence diagrams*. Isn’t consistency great?

5.5.8.3 Summary

Activity diagrams are very powerful SysML *behavioural diagrams*, which can be used to show both low-level behaviour, such as *operations*, and high-level behaviour, such as Processes. They are very good for helping to ensure model consistency, relating to *state machine diagrams*, *sequence diagrams* and *block definition diagrams*.

Activity diagrams concentrate on *control* and *object flow*, showing behaviour defined using *actions* that use and produce *object nodes*. That is, they concentrate on behaviour that deals with information flow and transformation, rather than behaviour that concentrates on change of state (as in the *state machine diagram*) or that concentrates on the sequencing of messages (as in the *sequence diagram*). However, all of these diagrams can (and should) be used together to give a complete and consistent model of the interactions between System Elements.

5.5.9 Use case diagrams

The SysML *use case diagram* realises a behavioural aspect of a model, with an emphasis on functionality rather than the control and logical timing of the System. The *use case diagram* represents the highest level of behavioural abstraction that is available in the SysML. However, the *use case diagram* is arguably the easiest diagram to get wrong in the SysML. There are a number of reasons for this:

- The diagrams themselves look very simple, so simple in fact that they are often viewed as being a waste of time.
- It is very easy to go into too much detail on a *use case diagram* and to accidentally start analysis or design, rather than conducting context modelling.
- *Use case diagrams* are very easy to confuse with data flow diagrams as they are often perceived as being similar. This is because the symbols look the same as both *use cases* (in *use case diagrams*) and processes (in a data flow diagram) are represented by ellipses. In addition, both *use cases* and processes can be decomposed into lower level elements.
- *Use case diagrams* make use of perhaps the worst symbol in SysML, the stick person notation used to represent *actors*. This is discussed further in Section 5.5.9.1.

Nevertheless, *use case diagrams* are central to systems engineering, forming the basis of the model-based approach to requirements engineering as embodied by the ACRE approach described in Chapter 9, being used to model the Needs in Context for the System under development.

5.5.9.1 Diagram elements

Use case diagrams are made up of four main elements as shown in Figure 5.86.

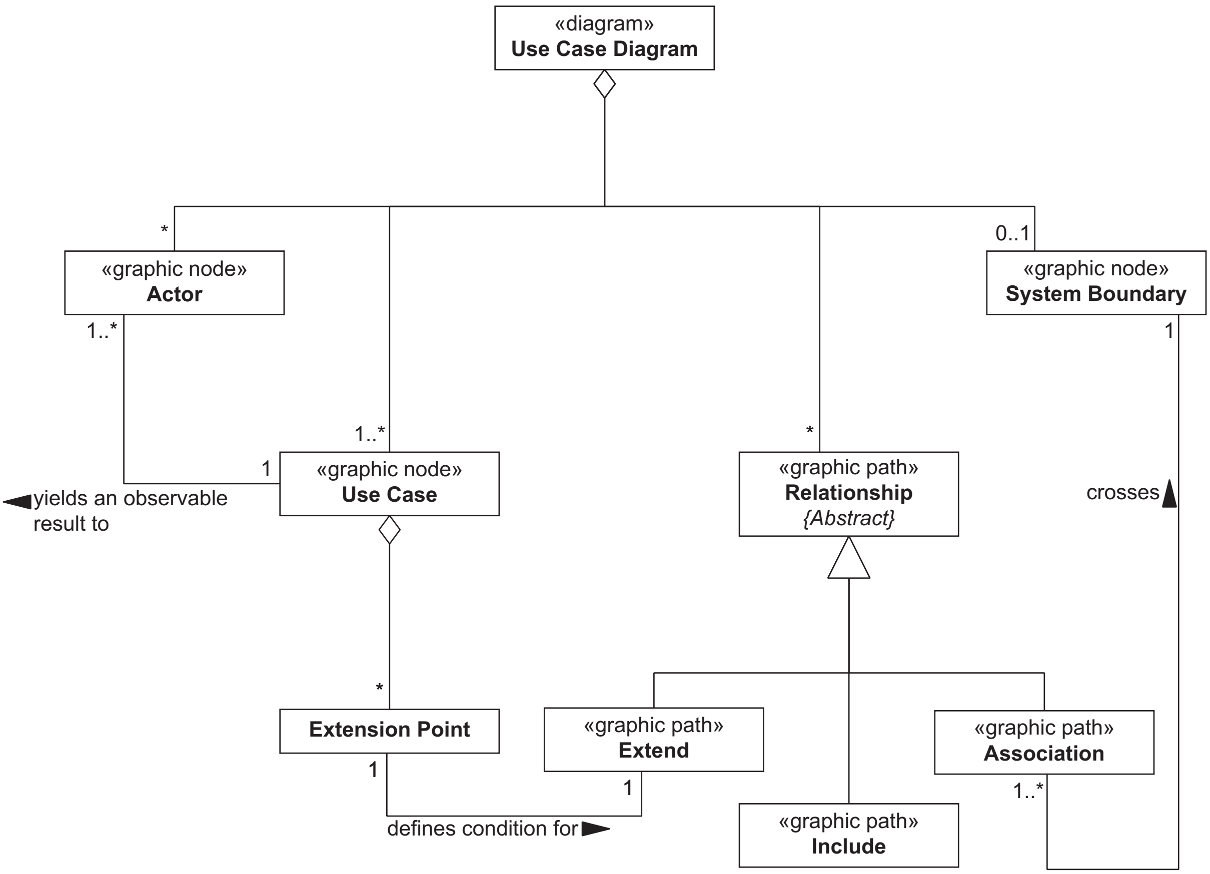

Figure 5.86 Partial meta-model for the use case diagram

Figure 5.86 shows a partial meta-model for *use case diagrams*. It shows that a ‘Use Case Diagram’ is made up of one or more ‘Use Case’, zero or more ‘Actor’, zero or one ‘System Boundary’ and zero or more ‘Relationship’. Each ‘Use Case’ yields an observable result to one or more ‘Actor’. There are three types of ‘Relationship’: the ‘Extend’, ‘Include’ and ‘Association’. A ‘Use Case’ can be made up of zero or more ‘Extension Point’, each of which defines the condition for an ‘Extend’ *relationship*. Each ‘Association’ crosses the ‘System Boundary’.

The notation that is used on *use case diagrams* is shown in Figure 5.87.

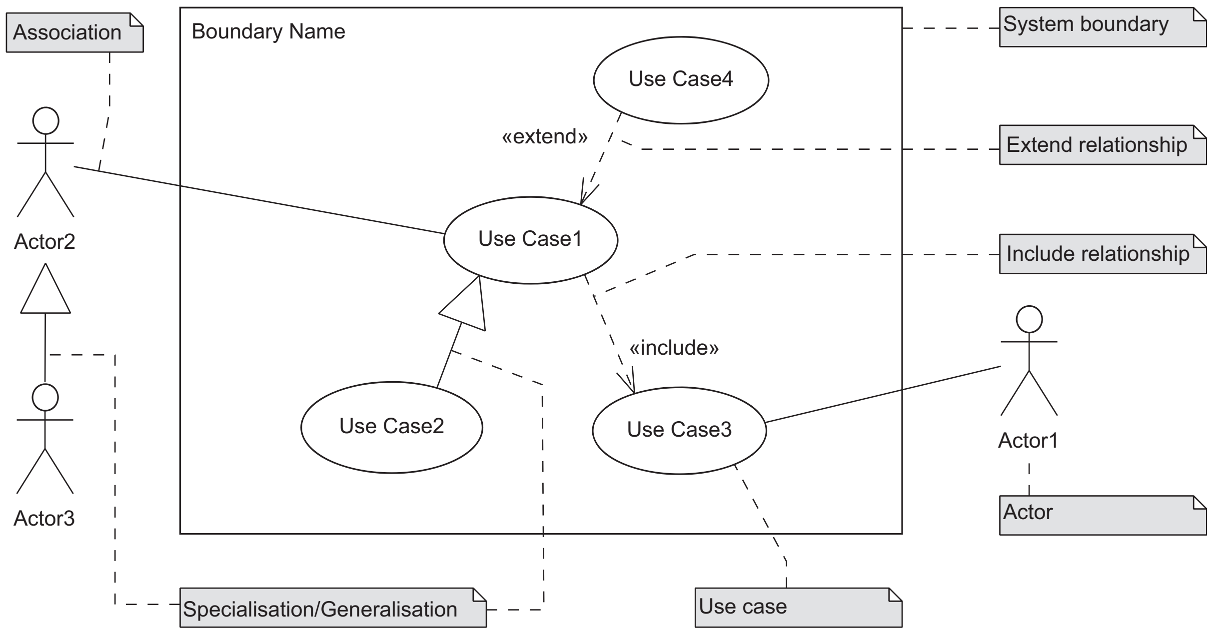

Figure 5.87 Summary of use case diagram notation

Use case diagrams are composed of four basic elements: *use cases*, *actors*, *relationships* and a *system boundary*. As a minimum a *use case diagram* must contain at least one *use case*; all other elements are optional.

Each *use case* describes behaviour of the system that yields an observable result to an *actor*. It is with the *actor* that the SysML notation is at its weakest, in terms of both the symbol and the name. The stick man symbol and the name *actor* suggest that this concept represents that of a person. This is not the case. An *actor* represents the role of a Person, place *or* thing that interacts with, is impacted by or has an interest in the System. So, while an *actor* can, indeed, represent a Person, it can also be used to represent an Organisation, other System or even a piece of legislation or a Standard. Furthermore, it is essential to understand that it is the role that is represented. This means that you should never see the names of People or Organisations or Standards, etc., on a *use case diagram*, but the role that they are playing. An *actor* named 'ISO15288' would be wrong, but one named 'Systems Engineering Standard' would be correct. It is also worth noting that a given role may be taken by more than one person, place or thing and that a given person, place or thing may take on more than one role.

In terms of the MBSE Ontology, the *actor* is directly analogous to the concept of the Stakeholder Role rather than the concept of the Person. The *use case* is directly analogous to the concept of the Use Case that represents a Need that has been put into Context.

Use cases are related to *actors* and to other *use cases* using a number of different types of *relationship*:

- *Association relationship*. This is used to relate *use cases* to *actors* and, unlike when used on a *block definition diagram*, is a simple unadorned line with neither *name* nor *multiplicity* as can been seen in the *association* between 'Actor2' and 'Use Case1' in Figure 5.87.
- *Include relationship*. This is used when a piece of functionality may be split from the main *use case*, for example to be used by another *use case*. A simple way to think about this is to consider the included *use case* as always being part of the parent *use case*. This is used to try to spot common functionality within a *use case*. It is highly possible that one or more of the decomposed *use cases* may be used by another part of the System. It is shown using a dashed line with an open arrow head, the line bearing the *stereotype* «include». The direction of the arrow should make sense when the model is read aloud. In Figure 5.87 'Use Case1' includes 'Use Case3'.
- *Extend relationship*. This is used when the functionality of the base *use case* is being extended in some way. This means that sometimes the functionality of a *use case* may change, depending on what happens when the System is running. A simple way to think about this is to consider the extending *use case* as sometimes being part of the parent *use case*. Extending *use cases* are often used to capture special, usually error-handling, behaviour. The *extend relationship* is also shown using a dashed line with an open arrow head, the line bearing the *stereotype* «extend». It is important to get the direction of the *relationship* correct, as it is different from the '«include»' direction. The direction of the arrow should make sense when the diagram is read aloud. In Figure 5.87 'Use Case4' extends 'Use Case1'. Every *use case* should be described (normally using text). Such a description must define the *extension points* where the behaviour of the *use case* is extended by the extending *use case*. An *extension point* has no specific graphical notation.
- *Specialisation/generalisation relationship*. This is exactly the same *relationship* as found on *block definition diagrams* and is used when one *use case* is a *specialisation* of another. Just like when used with *blocks*, *generalisation* between *use cases* allows for inheritance of behaviour and relationships. For example, consider the *use case diagram* shown in Figure 5.88. The general Use Case 'Allow stunt to be performed using different fluids' is specialised by the two Use Cases 'Perform using concrete' and 'Perform using custard', which inherit the behaviour described in 'Allow stunt to be performed using different fluids' as well as including the Use Case 'Ensure fluid chosen is suitable for venue', which is included by 'Allow stunt to be performed using different fluids'.

In a similar way, *generalisation* can be used between *actors*, as is shown in Figure 5.87, when one *actor* is a specialisation of another.

The final element that can appear on a *use case diagram* is the *system boundary*, used when describing the Context of a System. As its name suggests, the

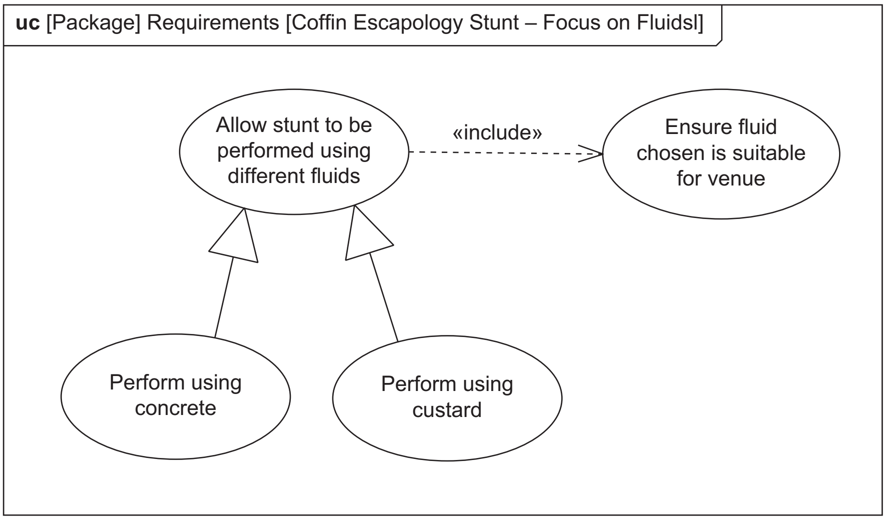

Figure 5.88 Example use case diagram showing generalisation

system boundary defines the boundary of the System from a particular point of view, that is Context. Everything inside the *system boundary* is part of the System, and everything outside the *system boundary* is external to the System. *Actors* are always outside the *system boundary*, and indeed, an *association* between an *actor* and a *use case* that crosses a *system boundary* indicates that there is an Interface between the *actor* and the System (which may be a sophisticated software and hardware Interface but equally could be an Interface in which a Person passes a note on a piece of paper to another Person).

System boundaries are not mandatory on a *use case diagram*. They are used when *use cases* are being shown in a Context. Where a *use case diagram* is being drawn simply to expand on a *use case*, as shown in Figure 5.88, then no *system boundary* is needed.

5.5.9.2 Examples

This section presents some examples of *use case diagrams* and related diagramming elements. Further examples of use case diagrams can be found in Chapter 13 and throughout Chapters 7–11 and 14–16. In addition, this section concludes with some guidance notes on common patterns that are often seen in *use case diagrams* and that can guide the modeller in refinement of the *use case diagrams*.

Figure 5.89 shows a *use case diagram* identifying the high-level Use Cases for the Coffin Escape Stunt. The Context, as indicated by the presence and title of the *system boundary*, is for the stunt System rather than from the point of view of an individual Stakeholder Role. The relevant high-level Stakeholder Roles are shown as *actors*, with associations connecting them to the Use Cases in which they have an interest and the relationships between the Use Cases are shown. There are two

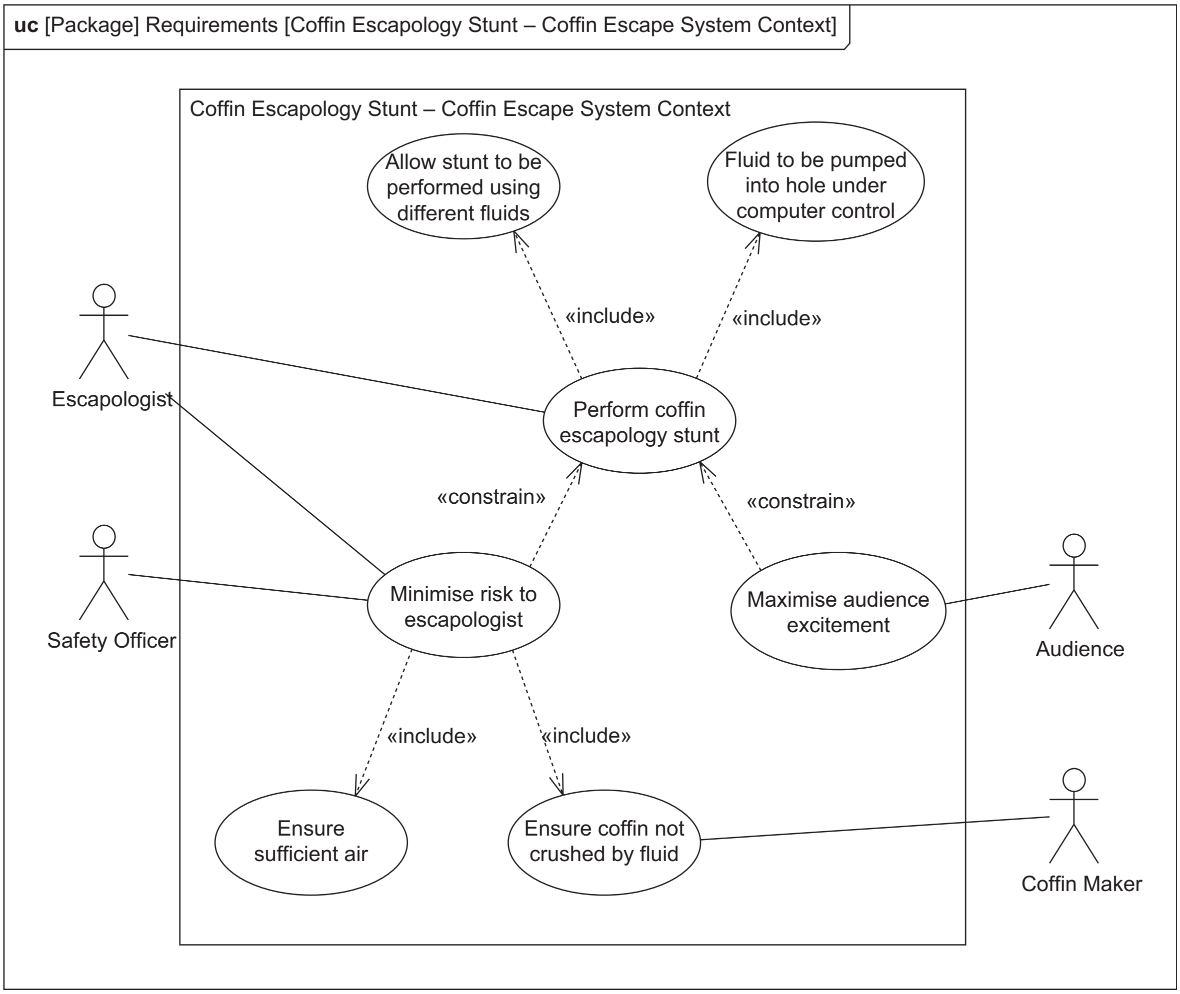

Figure 5.89 Example use case diagram showing System Context

points worth highlighting about this diagram: the number of Use Cases shown and the use of the «constrain» *dependency*.

The diagram shows only seven Use Cases, yet this is the top-level *use case diagram* showing the Use Cases for the whole coffin stunt System. Surely there must be more Use Cases than this? The answer to this is, of course, yes there are. However, this does not mean that all these Use Cases have to be shown on a single diagram. Other *use case diagrams* can be drawn that break these Use Cases down further and put them into the correct Context. Don't forget that these diagrams are produced to aid understanding and communication. A complicated diagram with tens of Use Cases on it may look impressive but is rarely of any practical use (other than for illustrating just how complicated the system is). Consider a System such as an aeroplane. There will be 1000s of Use Cases for the complete System, but how many high-level Use Cases are there? Probably not many more than ‘Take off safely’, Land safely’, ‘Have a fully-laden range of X km’, ‘Have a carrying capacity of X kg’, etc.

The second point to discuss is that of the «constrain» *dependency*, such as the one between ‘Minimise risk to escapologist’ and ‘Perform coffin escapology stunt’. The «constrain» *dependency* is not part of standard SysML, but is an extension used by the authors to show that one *use case* constrains another in some way. It is created using the SysML stereotyping mechanisms built into the language that allows existing language elements to be extended and is discussed in detail in Section 5.3.

Figure 5.90 shows another *use case diagram* showing Needs in Context. However, rather than showing the Use Cases for the entire System, this diagram shows them from point of view of a single Stakeholder Role, namely the

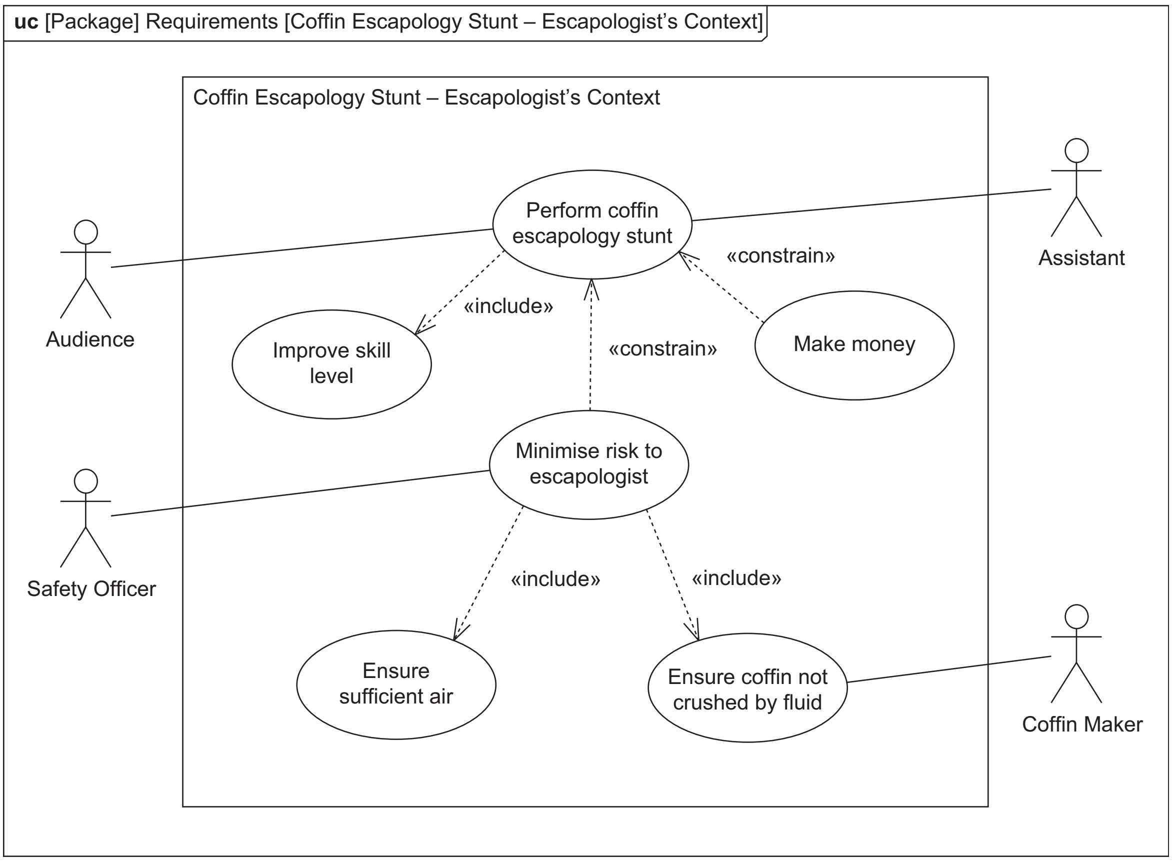

Figure 5.90 Example use case diagram showing a Stakeholder Role’s Context

escapologist. Unsurprisingly some of the Use Cases are also shown in Figure 5.89, since the Escapologist is one of (if not *the*) main Stakeholder Roles in any escapology stunt. However, some of those in Figure 5.89 (such as 'Maximise audience excitement') are not of direct interest to the Escapologist and are therefore not shown in Figure 5.90. Conversely, there are Use Cases that are only relevant to the Escapologist (such as 'Improve skill level'), which are shown in Figure 5.90 but are not relevant from the System Context and are therefore not shown in Figure 5.89. This whole idea of Context is central to the ACRE approach to requirements engineering discussed in much more detail in Chapter 9. Note also the use of the «constrain» *dependency* in Figure 5.90.

As discussed in Section 5.5.9.1 a *use case diagram* does not have to show any *actors* or contain a *system boundary*. An example of such a *use case diagram* is shown in Figure 5.91.

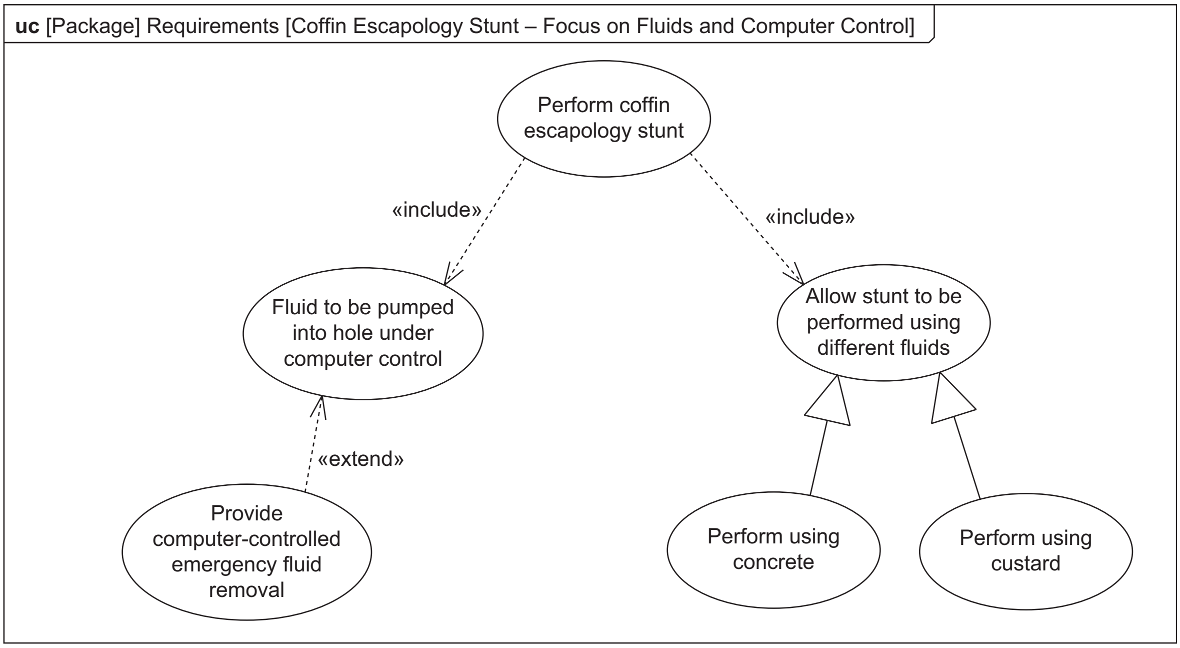

Figure 5.91 Example use case diagram without system boundary or actors

Figure 5.91 is focusing on Use Cases related to the use of different fluids in the stunt and to the computer control of the pump used in the stunt. Two specific types of fluids are identified and are shown via the use of the *generalisation relationship* between 'Allow stunt to be performed using different fluids' and 'Perform stunt using concrete' and 'Perform stunt using custard'. A Use Case representing special case behaviour 'Provide computer-controlled emergency fluid removal' *extends* the standard 'Fluid to be pumped into hole under computer control' Use Case.

When developing *use case diagrams* there are a number of common patterns that should be looked for as an aid towards the production of good *use case* *diagram*. This section concludes with a look at these patterns, which cover the following possible situations:

- *Use case* at too high a level
- *Actor* at too high a level
- Repeated *actors*
- Something missing

Each of these four patterns is discussed in the following sub-sections.

Use case too high-level

One common mistake is to model *use cases* at too high a level. Consider Figure 5.92.

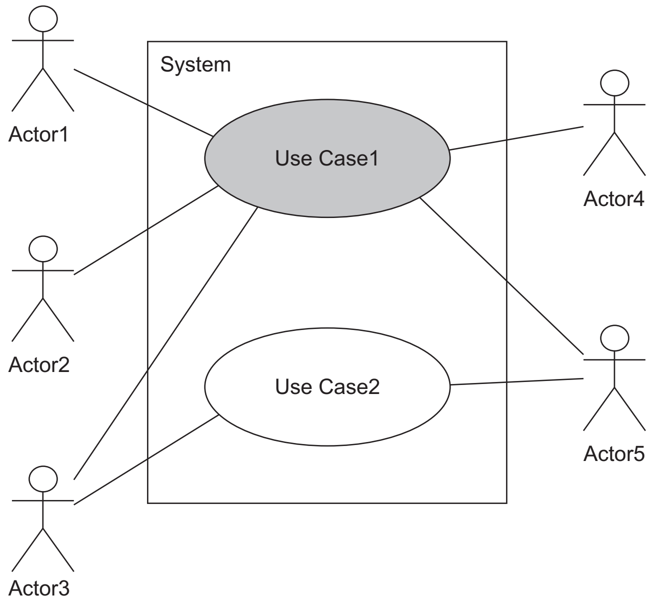

Figure 5.92 Use case too high level

Figure 5.92 shows a *use case*, ‘Use Case1’, that is linked to all *actors*. Such a pattern may indicate that the *use case* is at too high a level and that it should be decomposed further, making use of «include» and «extend» *dependencies* to link it to more detailed *use cases*. The *actors* would then be associated with the more detailed *use cases* rather than all being connected to the top-level *use case*.

Actor too high-level

Another common error is to model *actors* at too high a level. Consider Figure 5.93.

Figure 5.93 shows an *actor*, ‘Actor2’ (drawn with a surrounding box for emphasis), that is connected to every *use case*. Such a pattern may indicate that:

- The *actor* is at too high a level and that it should be decomposed further.
- The diagram has been drawn from the point of view of the Stakeholder Role represented by that *actor*.

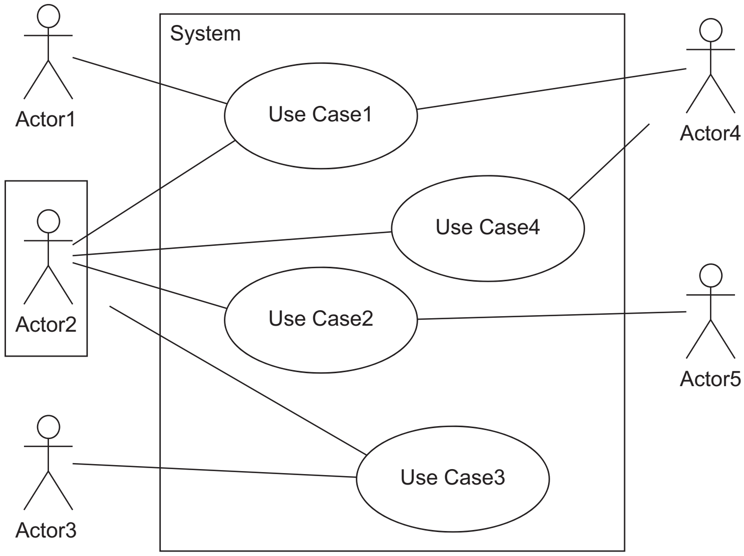

Figure 5.93 Actor too high level

If the *actor* is at too high a level, then it should be decomposed further and replaced on the diagram with the new *actors*. These *actors* will then be associated with the relevant *use cases* rather than being associated with all the *use cases*.

If the diagram has been drawn from the point of view of the Stakeholder Role represented by that *actor*, that is the *use case diagram* is drawn for that Stakeholder Role's Context, then the *actor* should be removed from the diagram. The *system boundary* should indicate that the diagram is drawn for that Stakeholder Role's Context.

Repeated actors

Sometimes a pattern is seen in which two or more *actors* are connected to the same *use cases*. Figure 5.94 shows this.

Here we see two *actors*, 'Actor1' and 'Actor 2' (drawn with a surrounding box for emphasis), that are both connected to the same three *use cases*. This pattern may indicate that the *actors* are representing the same Stakeholder Role. Alternatively, it may indicate that instances of Stakeholder Roles have been used (check for names of specific people, organisations, standards, etc.). Instances should never be used. Remember that a Stakeholder Role represents the role of something that has an interest in the Project, not an actual instance involved. Any duplicate *actors* should be removed from the diagram.

Something missing – use cases without actors and actors without use cases

What does it mean if we have *use cases* or *actors* that are not related to anything? Consider Figure 5.95.

Figure 5.95 has a *use case*, 'Use Case5', and an *actor*, 'Actor5', that are not connected to anything else on the diagram.

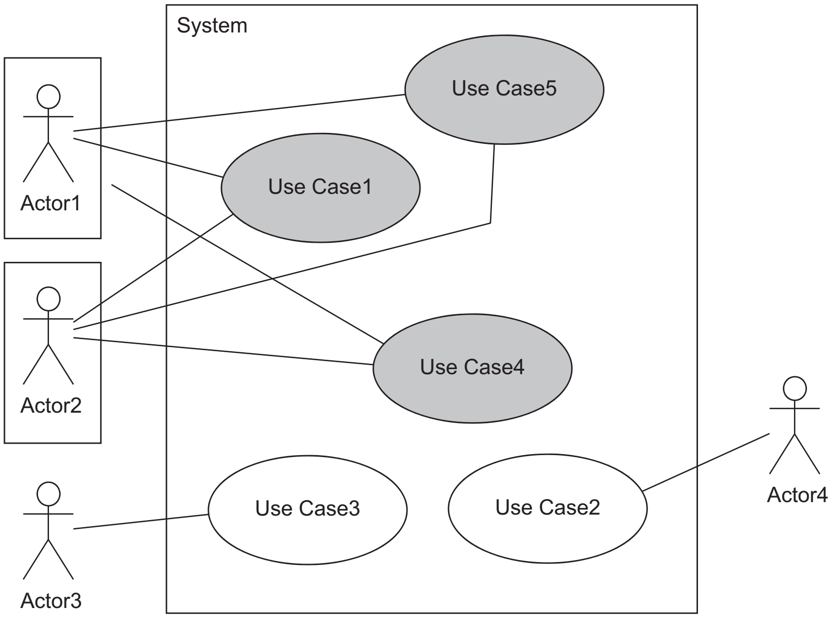

Figure 5.94 Repeated actors

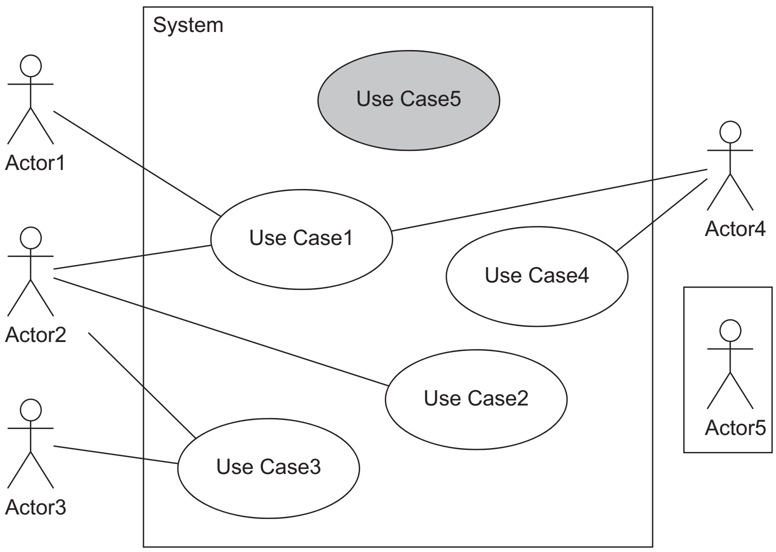

Figure 5.95 Something missing? Basic use case diagram checks

'Use Case5' has no *actors* associated with it. There are four possible reasons for this:

1. The *use case* is not needed and should be removed from the diagram.
2. There is an *actor* (or *actors*) missing that should be added to the diagram and linked to the *use case*.
3. There is an internal relationship missing; the *use case* should be linked to another *use case*.
4. There is an external relationship missing; the *use case* should be linked to an *existing actor*.

'Actor5' has no *use cases* associated with it. There are three possible reasons for this:

1. The *actor* is not needed and should be removed from the diagram.
2. There is a *use case* (or *use cases*) missing that should be added to the diagram and linked to the *actor*.
3. There is a *relationship* missing; the *actor* should be linked to an existing *use case*.

These two errors are very common, particularly when creating initial *use case diagrams*, and should be checked for on all *use case diagrams*.

5.5.9.3 Summary

Use case diagrams show the highest level behaviour of a system and are used to show Needs (Requirements, Concerns, Goals or Capabilities) in Context, along with the Stakeholder Roles involved and the relationships between them. This is the central theme of the ACRE approach described in Chapter 9, realised in its Requirement Context View.

Care is needed when producing *use case diagrams*. They should not be over-decomposed so that they start to look like data flow diagrams and become diagrams detailing the design of the System as they exist to show high-level behaviour as Needs in Context. There are a number of common patterns that should be looked for when producing *use case diagrams*, which can help you to spot when *use cases* or *actors* are at too high a level, where an *actor* has been repeated or where there is something missing from a *use case diagram*.

5.6 Auxiliary constructs

The SysML specification defines a number of *auxiliary constructs*, among which is included the *allocation*. The *allocation* will be described here. Some other examples of *auxiliary constructs* are given in Chapter 13. For full information on the other *auxiliary constructs*, see Reference 1.

An *allocation* is used to show how various model elements are allocated *to* and allocated from other elements. Such *allocations* may be used to show deployment or more generally to relate different parts of a model as the design progresses.

Figure 5.96 shows the partial meta-model for *allocations* and shows that an 'Allocation' can be represented in two ways: as an 'Allocation Compartment'

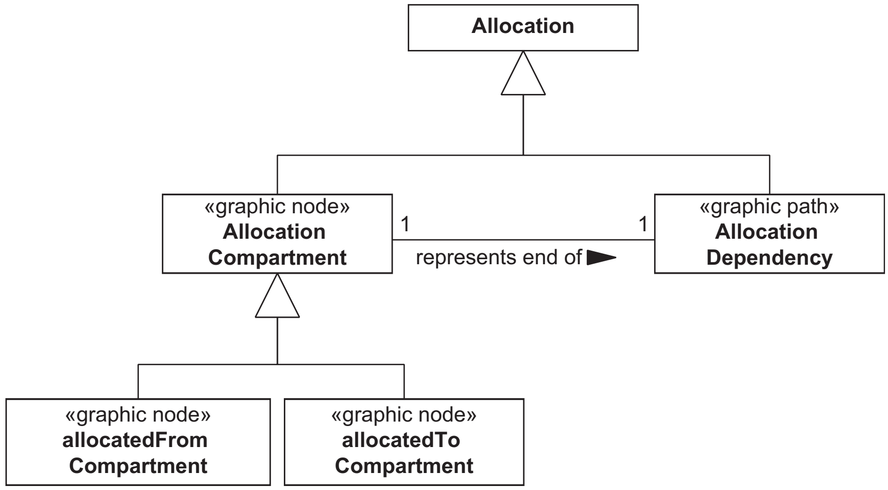

Figure 5.96 Partial meta-model for allocations

(either an 'allocatedFrom Compartment' or an 'allocatedTo Compartment') on an existing graphic node or as an 'Allocation Dependency' between model elements, with each end of such a *dependency* equivalent to one of the two types of 'Allocation Compartment'.

Rather than showing an 'Allocation Compartment' as a *compartment* of the relevant model element, it can also be shown using a *callout note* notation. This can be seen in Figure 5.97, where the notation used for *allocations* is shown.

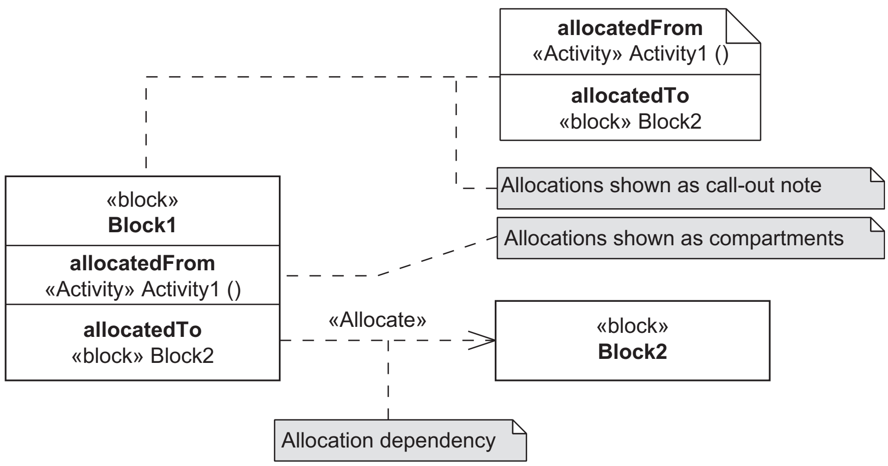

Figure 5.97 Summary of allocation notation on a block definition diagram

Allocations can be shown on diagrams other than the *block definition diagram* but the notation used is essentially the same. The following diagrams show examples of the notation in use.

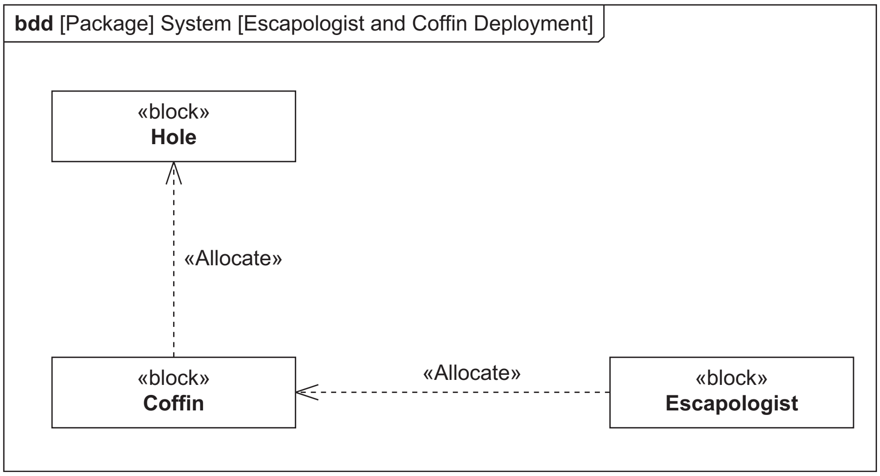

Figure 5.98 Example block definition diagram showing allocation using a dependency

Figure 5.98 shows *allocation* of the 'Escapologist' to the 'Coffin' and the 'Coffin' to the 'Hole' using the *allocation dependency* notation. The *block definition diagram* here is essentially being used a kind of deployment diagram (a diagram type present in UML but rather inexplicably, given the nature of systems engineering, absent from the SysML).

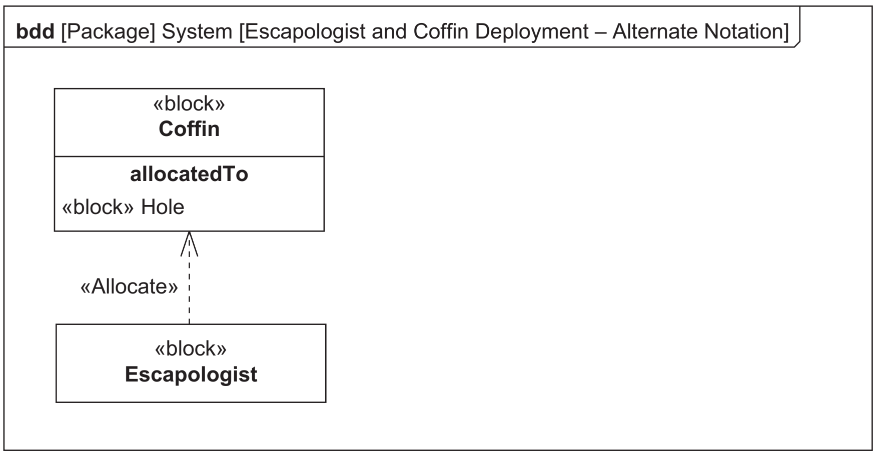

Figure 5.99 Example block definition diagram showing allocation using compartments

Figure 5.99 shows exactly the same information as is shown in Figure 5.98, but makes use of both *allocation compartments* and an *allocation dependency*. Note also that this diagram is lacking the 'Hole' *block* found in Figure 5.98. The *block* and the *relationship* to it can be deduced from the *allocatedTo compartment* in the 'Coffin' *block.*

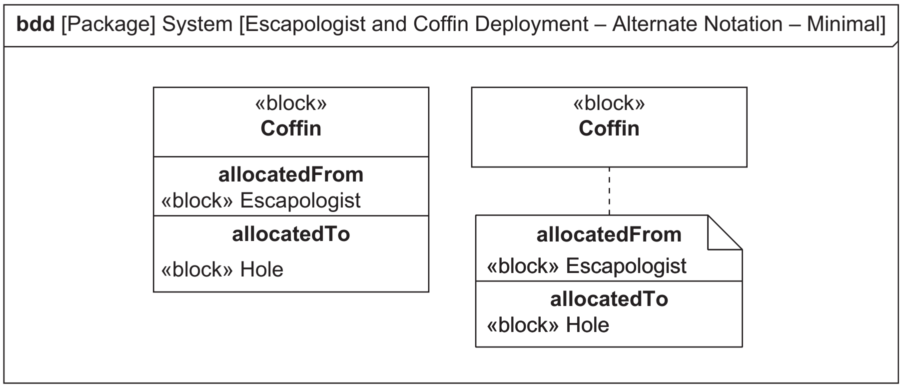

Figure 5.100 Example block definition diagram showing allocation – the minimalist approach

Finally, we can go very minimalist, as in Figure 5.100 where everything is done using *allocation compartments*. The diagram also shows how these *allocation compartments* would be shown using the *callout note* notation. In a "real" model, both notations would not be shown on the same diagram.

5.7 Summary

This chapter has described each of the nine SysML diagrams in turn, along with some of the auxiliary notation, and has provided examples of their use.

In order to conclude this chapter, there are a few pieces of practical advice that should be borne in mind when modelling using the SysML:

- Use whatever diagrams are appropriate. There is nothing to say that all nine diagrams should be used in order to have a fully defined System – just use whatever diagrams are the most appropriate.
- Use whatever syntax is appropriate. The syntax introduced in this book represents only a fraction of the very rich SysML language. It is possible to model most aspects of a system using the syntax introduced here. As you encounter situations that your known syntax cannot cope with, it is time to learn some more. There is a very good chance that there is a mechanism there, somewhere, that will.
- Ensure consistency between models. One of the most powerful aspects of the SysML is the ability to check the consistency between diagrams, which is often glossed over. Certainly, in order to give a good level of confidence in your models, these consistency checks are essential.

- Iterate. Nobody ever gets a model right the first time, so iterate! A model is an evolving entity that will change over time and, as the model becomes more refined, so the connection to reality will draw closer.
- Keep all models. Never throw away a model, even if it is deemed as incorrect, as it will help you to document decisions made as the design has evolved.
- Ensure that the system is modelled in both structural and behavioural aspects. In order to meet most of the above criteria, it is essential that the system is modelled in both aspects, otherwise the model is incomplete.
- Ensure that the system is modelled at several levels of abstraction. This is one of the fundamental aspects of modelling and will help to maintain consistency checks.

Finally, modelling using the SysML should not change the way that you work, but should aid communication and help to avoid ambiguities. Model as many things as possible, as often as possible, because the more you use the SysML, the more benefits you will discover.

References

1. Object Management Group. *SysML Specification* [Online]. 2012. Available from http://www.omgsysml.org [Accessed April 2013]
2. Miller G.A. ‘The magical number seven, plus or minus two: some limits on our capacity for processing information’. *Psychological Review*. 1956;**63**:81–97
3. Holt J. *UML for Systems Engineering – Watching the Wheels*. 2nd edn. IET Publishing: Stevenage, UK; 2004
4. Rumbaugh J., Jacobson I., Booch G. *The Unified Modeling Language Reference Manual*. 2nd edn. Boston, MA: Addison-Wesley; 2005
5. Booch G., Rumbaugh J., Jacobson I. *The Unified Modeling Language User Guide*. 2nd edn. Boston, MA: Addison-Wesley; 2005
6. Holt J., Perry S. *SysML for Systems Engineering*. IET Publishing: Stevenage, UK; 2008