Professional Applications of Computing Series 7

SysML for Systems Engineering

Jon Holt and Simon Perry

PROFESSIONAL APPLICATIONS OF COMPUTING SERIES 7

SysML for Systems Engineering

Other volumes in this series:

Volume 1 **Knowledge discovery and data mining** M.A. Bramer (Editor)
Volume 3 **Troubled IT projects: prevention and turnaround** J.M. Smith
Volume 4 **UML for systems engineering: watching the wheels, 2nd edition** J. Holt
Volume 5 **Intelligent distributed video surveillance systems** S.A. Velastin and P. Remagnino (Editors)
Volume 6 **Trusted computing** C. Mitchell (Editor)

SysML for Systems Engineering

Jon Holt and Simon Perry

The Institution of Engineering and Technology

Published by The Institution of Engineering and Technology, London, United Kingdom

First published 2008

The Institution of Engineering and Technology
Michael Faraday House
Six Hills Way, Stevenage
Herts, SG1 2AY, United Kingdom

www.theiet.org

British Library Cataloguing in Publication Data

Holt, Jon
SysML for systems engineering. - (Professional applications of computing series; 7)
1. SysML (Computer science) 2. Systems engineering
I. Title II. Perry, Simon III. Institution of Engineering and Technology
620'.001171

ISBN 978-0-86341-825-9

Typeset in India by Newgen Imaging Systems (P) Ltd, Chennai
Printed in the UK by Athenaeum Press Ltd, Gateshead, Tyne & Wear

This book is dedicated to the memory of Martha
JDH

To my mother and father, Sue and Keith, thanks for everything
SAP

Contents

Acknowledgments

No piece of work is carried out in isolation – and this book is certainly no exception. As usual, there are way too many people to thank so please don't be offended if you have been missed out.

First and foremost, thanks to all the Brass Bullet Ltd, especially: Ian, Mike, Guy and Sylvia. Special thanks go to Kay who keeps the business running while we are all off writing books. Simon, who was only a brief acknowledged in the last IET book has now moved up to co-author, has been an excellent co-author and has had to put up with a constant stream of my rambling, thoughts and occasionally bizarre ideas.

A few people observe a big thanks including: Duncan (who did the technical proof) and his merry men, Graham Jackson for doing some of the application research for the escapology, Mike and Sue Rodd who I am eternally grateful to and everyone who I have dealt with over the last few years. Special thanks to the INCOSE UK Chapter who gave the brontosaurus an award, the BCS for support with process modelling and the good folks at APMP. Particular thanks to all at the IET who were involved in the production of this book and who did their usual professional job.

As always, the people who matter most are left until last – my beautiful wife Rebecca and children Jude, Eliza and Roo, who provide me with both my motivation and reward in life.

Jon Holt, November 2007

As usual Jon has said (almost) everything that needs to be said, but I would like to thank him for making me an offer I couldn't refuse and giving me the opportunity to work with him at Brass Bullet Ltd. Finally I have the job that I always wanted.

I would like to echo his thanks to all those mentioned above, but my special thanks must go to all at Brass Bullet Ltd. Writing this book would not have been possible without their help and support. Being a sounding-board for our thoughts and ideas can't be easy.

Finally, extra special thanks and all my love go to my wife Sally. Her love and encouragement keep me sane (well, almost).

Simon A. Perry, November 2007

Chapter 1
Introduction to systems engineering

> '. . . it's thin at one end, much, much thicker in the middle, and then thin again at the far end.'
>
> 'The Brontosaurus Theory', Ann Elk, Monty Python sketch, 1974

1.1 Introduction

How do you solve a problem like systems engineering? Systems engineering is one of those disciplines that can be applied to just about anything by just about anyone. Also, people have been engineering systems since the dawn of mankind, so why is it a subject that is very often misunderstood?

Systems engineering is often seen as an emerging discipline, which may be true in terms of formal education, such as university programmes, but without systems engineering we wouldn't have Stonehenge or the pyramids – these are examples of systems that have clearly been systems-engineered. Also, consider political systems, financial systems, belief systems and so on. All of these have not happened by chance and must have been planned and developed in some way, according to an approach.

In the same way, many people will be badged with the term *systems engineer*, even though they are not. Many people will have evolved into a systems engineer from being a specialist in a particular field. For example, many people have evolved from being a software engineer into a systems engineer; many people who were systems analysts or designers are now systems engineers. One of the great misconceptions about systems engineering is that it is all about requirements engineering – sure, requirements are tremendously important, but systems engineering is more than a sum of its parts.

In order to understand how this has occurred, it is first necessary to try to define exactly what we mean by systems engineering.

1.2 Information sources

There is no single definition for the term *systems engineering*. This may be due to the vast scope of systems engineering as it can be applied to any type of application

in any domain and, hence, is very difficult to put a boundary on. It is one of those terms that mean different things to different people. However, as time goes on, there is a natural trend towards standardizing the terms and concepts used within the field. There are many places that one could visit to attempt to define systems engineering and in this book, for the sake of brevity, it is essential that a common source be used for the concepts and terminology used herein. The two main sources that were looked at were international standards and international industrial bodies.

There are many international standards that can be applied to systems engineering – see Chapter 6 for more examples of this. However, an obvious port of call is the International Standards Organization – ISO. The International Standards Organization is the world's largest and most recognized standards body, and produces many thousands of standards. The standard that is of particular interest for the purposes of this book is *IEC/ISO 15288 – systems engineering – systems life cycle processes* [1]. This standard identifies a high-level set of processes that may be used in any organization in order to understand and apply systems-engineering practices within it. Therefore, this standard will be used as one of the two key references throughout this book and, as much as practically possible, all terms and concepts in this standard will be adopted.

The other source of reference that is to be used for this book is from an industrial organization that produces a systems-engineering best-practice handbook – *The INCOSE Systems Engineering Handbook*. The International Council on Systems Engineering is a worldwide organization whose mission is to 'to advance the state of the art and practice of systems engineering in industry, academia, and government by promoting interdisciplinary, scalable approaches to produce technologically appropriate solutions that meet societal needs' [2].

The INCOSE handbook is based on what is perceived as being international best practice and, at the time of writing, this handbook is in version 3.0. Version 3.0 of the handbook, rather conveniently for the purposes of this book, is based directly on the processes that are defined in ISO 15288 [3].

Therefore, the two main references that are used in this book are ISO 15288 and the INCOSE handbook, which, for all intents and purposes, use the same core set of systems-engineering processes. The two sources are not 100 per cent consistent with one another but, where this becomes an issue, a discussion will be raised in the book.

1.3 Defining systems engineering

There are many definitions of systems engineering, all of which tend to differ depending on the context and the point of view or background of the author. This section presents a few of the more widely recognized definitions and then discusses each in turn.

Definition 1 'Systems engineering is a discipline that concentrates on the design and application of the whole (system) as distinct from the parts. It involves looking at a problem in its entirety, taking into account all the facets and all the variables relating the social to the technical aspect' [4].

Notice that in this definition there is an emphasis on looking at the bigger picture and this is brought up several times: 'whole [system]', 'problem in its entirety' and 'all the facets'. This is a key concept in systems engineering, where a system is looked at across its whole life cycle and not just one small part. Also notice here that non-technical facets of the system are mentioned as having an influence on the system.

Definition 2 'Systems engineering is an iterative process of top-down synthesis, development and operation of a real-world system that satisfies, in a near optimal manner, the full range of requirements for the system' [5].

There are a few concepts here that are introduced that are not in the previous definition. The first is the concept of an *iterative process*. Real-life systems are rarely developed in a linear fashion as, even with what may be perceived as a linear life-cycle model, for example, there will be much iteration involved inside each stage. The second interesting point here is that requirements have been mentioned for the first time in any of the definitions. Indeed, this definition talks about satisfying requirements, which must be one of the basic goals of any systems engineer. This is also qualified, however, by stating that this should be in a near-optimal manner. Obviously, in an ideal scenario, all requirements should be met in an optimum fashion but, in the real world, it is often the best that can be achieved given the resources and technology at one's disposal. This definition also includes a rather contentious statement in that the development is said to be 'top-down', which could be interpreted as being rather limited.

Definition 3 'Systems engineering is an inter-disciplinary approach and means to enable the realisation of successful systems' [6,7].

The INCOSE definition of systems engineering is rather more terse than the previous two, yet no less accurate. This statement simply states what must be the highest-level requirement for any systems engineer, which is to realize successful systems by using any appropriate means necessary.

Definition 4 'Systems engineering is the implementation of common sense' [8].

The final definition that is looked at here is definitely from the 'less is more' camp and makes a rather bold statement about systems engineering generally, in that it is mostly good common sense. But, of course, as any schoolchild will tell you, the strange thing about common sense is that it is not all that common!

So, there have been four definitions presented here, each of which is correct, yet each of which is very different. This is, perhaps, symptomatic of a discipline that includes all other disciplines, that cannot be bounded and that can be applied to any system in any domain!

1.4 The need for systems engineering

It has been established that it is difficult to pin down an exact definition for systems engineering. However, it is not so difficult to pin down why we *need* systems engineering. Many systems end in failure or disaster. The term *failure* here refers to a system where the project never reached delivery and where time and money were wasted. The term *disaster* here refers to a system where people were hurt or the environment was damaged as a result of the system failing.

One of the problems with systems is that the inherent complexity of modern-day systems is increasing all the time. This complexity does not just apply in the actual system, or system of interest as it will be referred to in this book, but also in the interactions with other systems.

There are many examples of disasters and failures that can be found in the literature and these are often grouped by their application domain. In all cases the impact of other systems is enormous and is difficult to calculate. In the following tables, just a few of the more famous disasters and failures are listed and a very brief summary of the end result is provided. In order to appreciate the full scale of the impact of such failures and disasters, readers are encouraged to look into some of these in more detail to gain a fuller insight into the horrors of what can happen when things go wrong. Perhaps the most frightening thing about all these examples is not the end result, but how easily things might have turned out differently if a more rigorous systems-engineering approach had been applied throughout the project and product life cycles.

The examples in Table 1.1 identify some of the better-known failures and disasters, and one of the first questions that enter many people's minds is that, surely, these were all examples of 'risky' systems.

Table 1.1 Examples of failures and disasters

Name	**Application domain**	**Location**	**End result**
Bhopal	Chemical	India	Massive loss of human life, damage to the environment
Flixborough	Chemical	UK	Environmental damage
Therac 25	Medical	USA	Loss of life
Challenger	Aerospace	USA	
DC-10 doors	Aerospace		Loss of human life
Chernobyl	Nuclear	USSR	Massive loss of human life, damage to environment
Windscale	Nuclear	UK	Massive environmental damage
Three Mile Island	Nuclear	UK	Massive environmental damage

One of the requirements for good systems engineering is to minimize the risk inherent in a system. This risk may take on a number of different contexts, such as business risk, technical risk, financial risk, environmental risk. The context of the risk is very important here, as it is possible to have risk manifest itself at different levels in different contexts. For example, a system may have a very low financial risk, as it may be worth little money or be a system with adequate resources. However, consider the situation where a project associated with the system is in a new area where no domain knowledge exists. In this scenario, it may be that there is a very low financial risk, but a very high technical risk. In such cases, which occur all too often, it is necessary to be able to trade off between these different contexts.

Risk is not a new concept, however, and as time goes on the nature of risk is changing. Historically, the greatest risks to mankind were always natural, such as earthquakes and floods. However, there are now new – manmade – risks, such as war and pollution. To compound this even further, these two types of risk do not exist in isolation, and will impact one another. Manmade systems influence natural ones, such as deforestation projects causing landslides. Also, natural systems will influence manmade ones, such as a strong wind spreading the effects of radiation. Again, this comes back to the concept of different contexts, where one context will influence or affect another context.

It is essential, therefore, that, when we consider our systems, we also consider other systems that we interact with, whether they are physical systems, people or, indeed, the environment that we are working in. These relationships are very complex and it is important that we can understand them in some way and relate them effectively to our system model.

In an ideal world, it would be nice to eliminate risk altogether. However, in the real world, this is simply not possible. Also, progress demands taking risk whether it is making a first flight, developing new power technologies or crossing a busy road.

One important aspect of risk that needs to be understood and considered is the nature of responsibility. It is possible that, as time moves on, the nature of responsibility changes. For example, if a person took up smoking in the 1940s, it could be argued that the tobacco companies would be responsible for any smoking-related illnesses, as the risks involved with smoking were not understood at the time. However, if someone in the 2000s starts to smoke and as a consequence has a smoking-related illness, then it is that individual's fault, since the health risks relating to smoking are now well understood and are emblazoned across the front of all tobacco products.

Therefore, it is essential that both the risk and the responsibility for the risk be well understood.

1.5 The three evils of engineering

Projects fail and disasters occur for many reasons. However, there are three underlying reasons why things go wrong in life, which are known as the 'three evils' of systems engineering: complexity, a lack of understanding and communication issues.

1.5.1 Complexity

The concept of complexity will be illustrated in two ways – one that emphasizes the importance of relationships, and one that uses a brontosaurus to visualize the nature of the evolution of complexity.

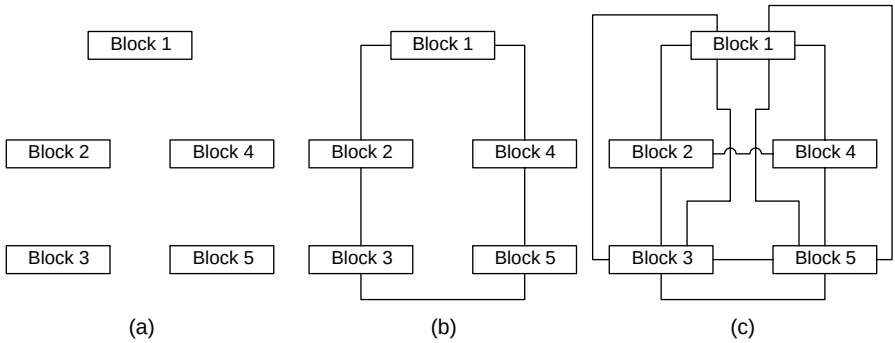

Figure 1.1 Complexity manifesting through relationships

For the first example, consider three boxes that may represent three elements within a system, as shown in Figure 1.1(a). Each element may represent almost anything, ranging from a text sentence that represents a requirement, to an assembly that makes up a system, to users that relate to a system. Each of these elements may very well be understood by whoever is reading the diagram, but this does not necessarily mean that the system itself is understood.

Consider now Figure 1.1(b), and it is quite clear that this diagram is more complex than the previous one, although nothing has changed in the elements themselves, only the relationships between them.

Consider now Figure 1.1(c) where it is, again, obvious that this diagram is more complex than its predecessor and far more complex than the first.

In fact, the more relationships that are added between the system elements, the higher the complexity of the overall system. More and more lines could be drawn onto this diagram and the complexity will increase dramatically, despite the fact that the complexity of each of the three elements has not actually increased as such.

The point that is being communicated here is that, just because someone understands each element within a system, this does not, by any means, mean that the system itself is understood. The complexity of a system manifests itself by relationships between things – in this case the system elements. It should be borne in mind, however, that these elements may exist at any level of abstraction of the systems, depending on what they represent. Therefore, the complexity may manifest itself at any point in the system. The complexity of the whole of the system is certainly higher than the complexity of the sum of its parts.

This may be thought of as being able to see both the woods *and* the trees.

The second way that complexity is illustrated is through the concept of the 'brontosaurus of complexity'. In this slightly bizarre analogy, complexity is visualized as a brontosaurus, in that the complexity of a system at the outset is represented by the dinosaur's head and, as the project life cycle progresses, this complexity increases (travelling down the neck); it increases even further (through the belly) before reducing and finally ending up at the tail of the brontosaurus.

This fits with the shape of the brontosaurus, that is 'thin at one end, much, much thicker in the middle, and then thin again at the far end' [9].

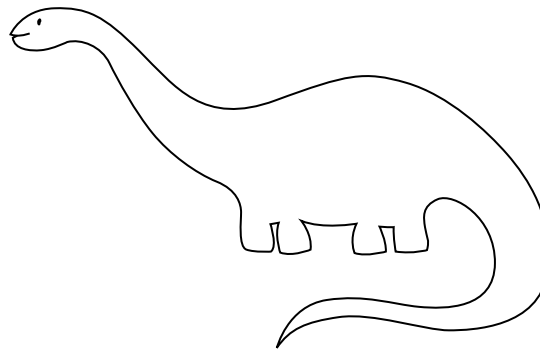

Figure 1.2 A brontosaurus

The perceived complexity of a project is almost always low to begin with, but balloons during the analysis of a project, as the understanding of the full impact of the requirements and the constraints is fully understood. By the time the problem is fully understood, a project is well and truly in the 'belly of the brontosaurus', whereas, when the design begins and becomes optimized, the project should be heading towards the 'tail of the brontosaurus'. By applying the brontosaurus-of-complexity analogy, it is shown that one must go from the head (initial ideas and requirements) to the tail (system), but that it is impossible to do this without going through the belly of the beast.

Consider the situation when a project is at the head of the brontosaurus, then this may be visualized as the illustration in Figure 1.1. As the complexity of the project increases and we move down the neck of the brontosaurus, so the complexity increases, as shown in Figure 1.1b. In fact, the more relationships that are added between the system elements (and, hence, the more interactions between them) the closer to the belly we actually get.

Many projects will fail because the project never leaves the belly or, in some cases, is left even higher up in the neck. If a project stays in the head or neck, there is a large danger of the system being oversimplified and the complexity inherent in the system is never uncovered until it is too late. If the project remains in the belly, however, the complexity has been realized, but it has not been managed effectively. Unfortunately, when a project is in the belly of the brontosaurus, it may seem to the project personnel that the world is at an end and that there is no understanding of the project as a whole.

This can be quite soul-destroying and is a situation that many systems engineers would have found themselves in at some point in their careers. Successful systems engineering is about being able to see the brontosaurus as a whole and that there is life after the belly.

In a final twist to this analogy, there is an ironic major difference between complexity and the brontosaurus. Complexity is difficult to visualize, but definitely exists in any system. A brontosaurus, on the other hand, is easy to visualize (see Figure 1.2) but never actually existed (it was demonstrated in 1974 that the brontosaurus was actually the apatosaurus).

1.5.2 Lack of understanding

A lack of understanding may occur at any stage of the system life cycle and also may occur during any process. Consider the following examples of a lack of understanding affecting a project.

- A lack of understanding may occur during the conception stage of a project, during a requirement-related process. If the requirements are not stated in a concise and unambiguous fashion (or, in reality, as unambiguously as possible) then this lack of understanding will cascade throughout the whole project. It is widely accepted that mistakes made during early stages of the life cycle cost many times more to fix during later stages, so it makes sense to get things right as early as possible [10].
- A lack of understanding may occur during the development stage of a project, during an analysis-related process. For example, there may be a lack of domain knowledge during analysis that may lead someone to state false assumptions, or actually to get something completely wrong due to insufficient knowledge. Again, uncorrected mistakes here will lead to larger problems further down the development life cycle.
- A lack of understanding may occur during the operational stage of a project, during an operation-related process. Incorrect usage of a system may lead to a system failure or disaster. For example, people not following safety procedures, people not using the correct tools, etc.

Of course, these examples are merely a representation of some ways that a lack of understanding can manifest itself in a system – there are many other places in the life cycle where problems may occur.

1.5.3 Communication problems

The third of the three evils is the problem of communication or, rather, *ineffective* communication. The richness and complexity of human communication is what separates humans from other species. One of the earliest recorded examples of project failure is that of the Tower of Babel, as described wonderfully by Fred Brookes [11]. The Tower of Babel started life as a very successful project and, indeed, the first few stages went off without a hitch and the project was running on schedule, within budget and was meeting all the project requirements. However, one of the key stakeholder's requirements was not considered properly, which was to cause the downfall of the

project. When the stakeholder intervened in a divine fashion, the communication between project personnel was effectively destroyed.

Communication problems may occur at any level of the organization or project.

- *Person-to-person level*. If individuals cannot communicate on a personal level, then there is little hope for the project's success. This may be because people have different spoken languages, technical backgrounds or even a clash of personalities.
- *Group-to-group level*. Groups, or organizational units, within an organization must be able to communicate effectively with one another. It has already been pointed out that the world of systems engineering is populated by people with different backgrounds and from different disciplines and this often manifests itself with groups of like-minded people, none of whom can communicate with anyone outside their circle of interest. These groups may be from different technical areas, such as hardware and software, or may span boundaries, such as management and technical, or marketing and engineering.
- *Organization-to-organization level*. Different organizations speak different languages – each will have its own specific terms for different concepts, as well as having an industry-specific terminology. When two organizations are working in a customer–supplier relationship, the onus is often on the supplier to speak the customer's language so that communication can be effective, rather than the customer having to speak the supplier's language. After all, if the supplier won't make an effort to speak the customer's language, it is quite likely that they will not remain customers for very long.
- *System-to-system level*. Even nonhuman systems must be able to communicate with one another. Technical systems must be able to communicate with technical systems, but also with financial systems, accountancy systems, environmental systems, etc.
- *Any combination of the above*. Just to make matters even more confusing, just about any combination of the above communication types is also possible.

These problems, therefore, lead to ambiguities in interpreting any sort of communication, whether it is a spoken language or an application-specific or technical language.

1.5.4 The vicious triangle of evil

Having established that these three evils of engineering exist, matters become even worse. Each of these three evils does not exist in isolation, but they will actually feed into one another. Therefore, unmanaged complexity will lead to a lack of understanding and communication problems. Communication problems will lead to unidentified complexity and a lack of understanding. Finally, a lack of understanding will lead to communication problems and complexity.

The three evils, therefore, form a triangle of evil that it is impossible to eliminate. In fact, the best that one may hope for is to address each of these evils in its entirety and try to minimize it. It is important always to consider these three evils at all stages of a project and when looking at any view of a system.

1.6 Systems-engineering concepts

There are several key systems-engineering concepts and it is important to have these identified and understood before any applications of systems engineering can be discussed. Therefore, based on the information in *The INCOSE Systems Engineering Handbook* and, hence, ISO 15288, the following generic systems-engineering meta-model has been generated. This meta-model is simply a defined set of concepts and terminology along with their relationships to one another. This diagram by no means represents every concept in systems engineering, but it does represent a set of the key concepts when considering systems engineering.

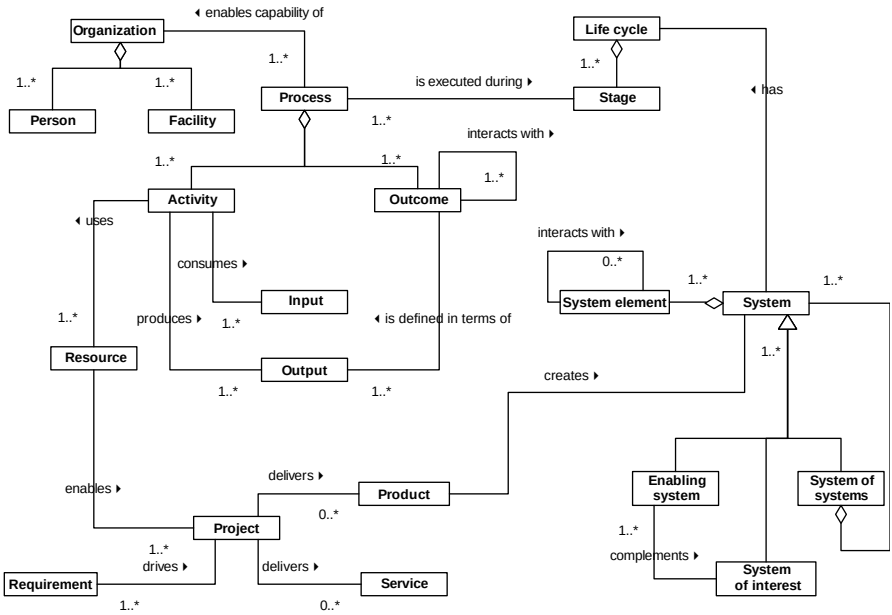

Figure 1.3 Systems engineering generic meta-model

The diagram in Figure 1.3 shows the key concepts that must be defined in order successfully to apply systems-engineering techniques. Rather than simply list each of these concepts and provide a description, it is useful to think of several of these concepts at the same time, as they are naturally related. This diagram may be augmented by grouping several of these concepts together into logical groupings, as shown in Figure 1.4, which is identical to the one in Figure 1.3 except that certain concepts have been grouped into packages.

These groups are processes, life cycles, projects and systems. Each of these groups will be discussed in more detail in subsequent sections, and it is essential that a common high-level understanding be established at the outset of this book.

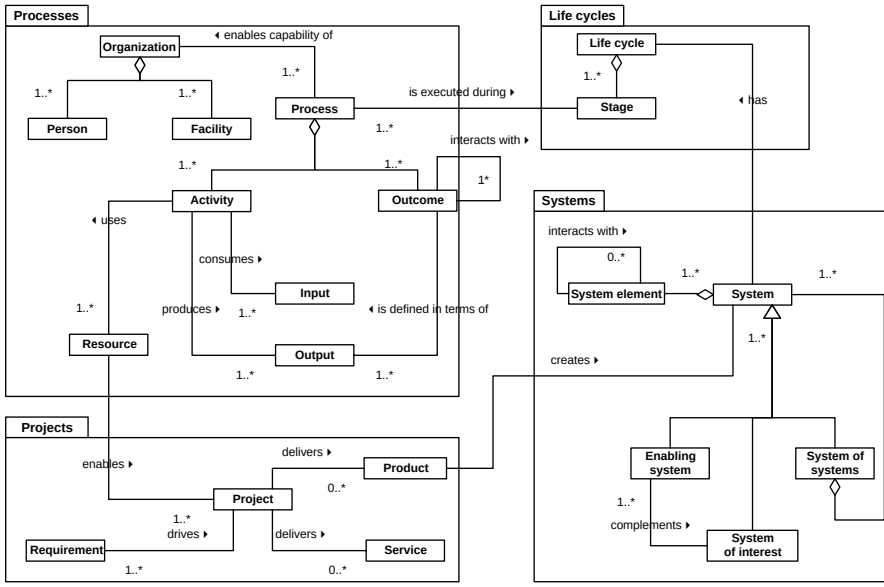

Figure 1.4 Systems engineering meta-model with groups

Each group will be introduced briefly and the main elements of the diagram defined and described. The terms and concepts that are described in this chapter will set the terminology that will be used throughout the rest of this book, so it is important that each concept and term can be understood. Finally, the relevance and importance of the concepts in each group are discussed, with other examples introduced wherever necessary.

1.6.1 Processes

Processes are an integral part of any organization. In real life, everything follows a process, be it a person, an organization or a system – be it technical, natural, financial, social or whatever. The importance of process modelling will de discussed in more detail in Chapter 6 and only the key elements will be introduced here.

It is essential to establish a good understanding of the processes that exist and that are needed to realize the requirements of a system or project effectively.

The diagram in Figure 1.5 shows the systems engineering meta-model, with the area of particular interest – that of processes, shaded in for emphasis. The key concept here is that of the ‘process’. A process is defined as ‘an interrelated set of activities etc.’. A process, however, simply describes an approach to doing something that, in the context of this book, refers to our approach to systems engineering. A number of processes are executed at each stage of the life cycle. It is important here to note that processes are not the same as stages, a misunderstanding that often occurs in the

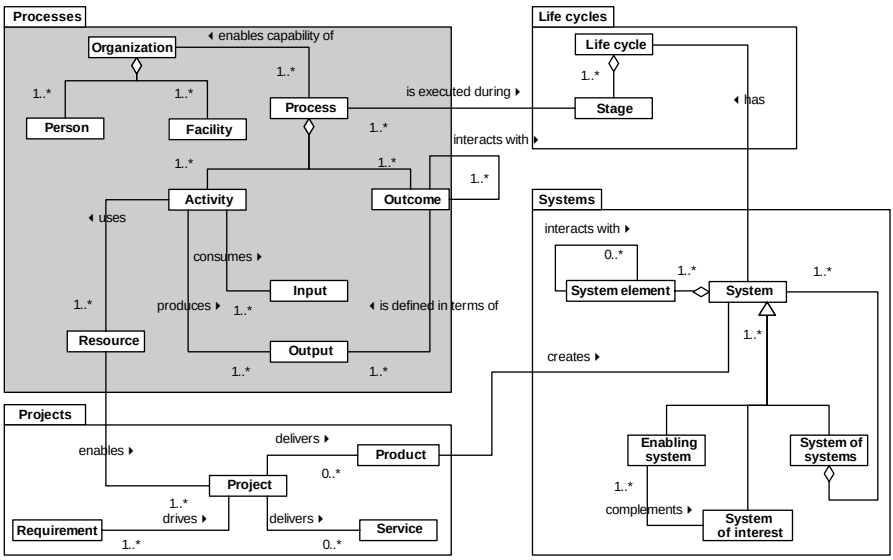

Figure 1.5 Systems-engineering concepts with a focus on 'processes'

world of systems engineering. A process is made up of two main elements:

- *Activity*. An activity may be thought of as a unit of behaviour in a process. These are often referred to by different terms in different standards, such as practice, step, etc. Both ISO 15288 and *The INCOSE Systems Engineering Handbook* use this term in the same way. A process is defined by a set of interrelated activities. The artefacts of a process are defined in terms of one or more 'Input' or 'Output'. An activity also consumes one or more 'Resources', which may be human, systems, money, assets, etc.
- *Outcome*. An outcome describes something that is generated or that occurs as the result of executing a number of activities in a process. This is one of those concepts where the definitions in ISO 15288 and the INCOSE handbook actually differ somewhat. In ISO 15288, an outcome occurs as the result of an activity, but is not the same, conceptually, as an artefact related to a process – not necessarily a produced thing, such as a document, specification or subsystem. In the INCOSE handbook, the artefacts of a process are defined as inputs and outputs, but these are not necessarily the same as the concept of an outcome in ISO 15288. This inconsistency of terms has been addressed by relating the process outputs to the process outcomes and by stating that a process 'Outcome' is defined in terms of one or more 'Output'.

Processes are executed within the bounds of an 'Organization' that is defined as being made up of the following.

- *Person*. This represents the staff at the organization, each of whom will hold a number of roles and responsibilities and have a set of competencies defined.

- *Facility*. This represents the role of any systems or services that are available to the organization.

  Processes are described in more detail in Chapter 6.

1.6.2 Systems

Perhaps the most obvious concept that needs to be looked at is that of the system itself. The INCOSE systems-engineering handbook and ISO 15288 identify a number of key concepts that need to be understood, which are presented in Figure 1.6. However, there are a few more concepts that will be introduced for the purposes of this book that will be expanded upon later.

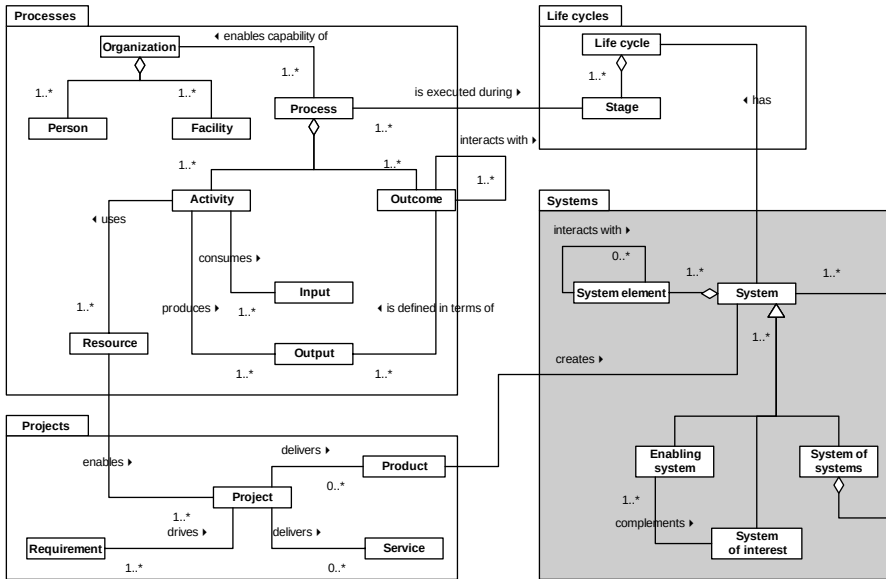

Figure 1.6 Systems engineering concepts with a focus on 'systems'

The key concept when thinking of systems is, unsurprisingly, the system. A system may be represented as some sort of hierarchy, with a system being made up of one or more system elements, each of which interacts with one or more other system elements. The diagram here represents a system at its highest level, with only a single level of decomposition. In reality, it is usual to break a system down into any number of levels, for example: subsystems, assemblies, subassemblies.

There are three main types of system that must be considered.

- *System of interest*. This is the system that is being developed by the project at hand. This is also what many people think of when considering systems engineering – a single system with interfaces to the outside world. However, in order to ensure that

the system of interest is realized successfully and optimally, one must consider the bigger picture.

- *Enabling systems*. These are systems that are outside the boundary of the system of interest and that complement the system of interest. It should also be borne in mind that the concepts of a system of interest and an enabling system are entirely dependent on the context of the system. For example, consider two systems: a robot and a human who controls the robot. From one point of view (context), the human is the system of interest and the robot is the enabling system, whereas from another point of view (context) the robot is the system of interest and the human is the enabling system.
- *System of systems*. Any system may be considered as being part of a larger system of systems. Therefore, in the above example, both systems described may be considered a single system that consists of (in this case) two systems. This may also be abstracted up to any higher level, such as the country, the world, the universe.

It is essential that any systems modelling can cope with being able to represent any of these types of system. Indeed, it will be seen later in this book that there are several ways to model these different types of system.

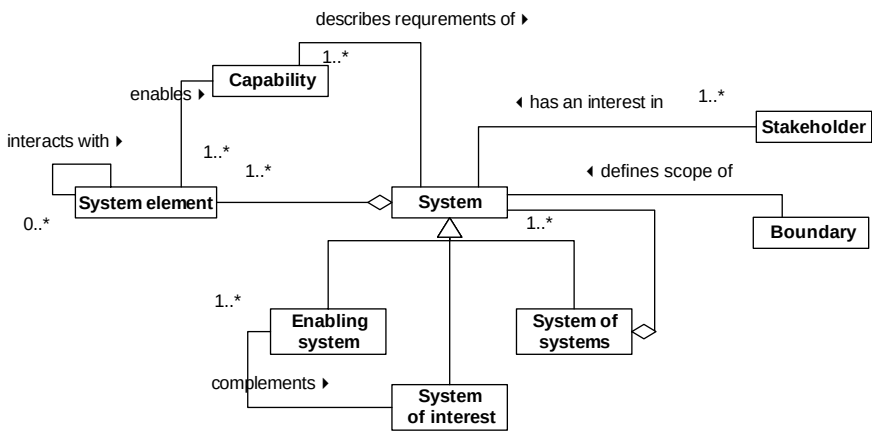

Figure 1.7 Expanded systems-engineering terms and concepts

The diagram in Figure 1.7 shows the same concepts as in the previous diagram, but three more key concepts have been added here.

- *Capability*. Like many terms associated with systems engineering, 'capability' will have different meaning for different people, depending on the application domain, industry or educational background. For the purposes of this book, a capability describes the requirements of a system at a high level. When a system is expressed in terms of its capability, no particular solution is held in mind but the emphasis is on what the system should be able to do, rather than how it should do it, or what technologies it should use.

- *Stakeholder*. A stakeholder, for the purposes of this book, represents the role of a person, place or thing that has an interest in the project. A role is *not* necessarily a person, which is a very common mistake to make, but may be, for example, a system, an organization or even a standard. Also, people often think that there is a one-to-one mapping between the stakeholder and the name of the person, place or thing that is realizing it. There are many instances where a stakeholder may be realized by many different names, or where a single name takes on a number of different roles. The words *stakeholder*, *role*, *actor*, *person* and so on are often used interchangeably, which can lead to a great deal of confusion between people working with different definitions and different organizations.
- *Boundary*. The system boundary draws the line between what is inside and what is outside the system. Effectively, the system boundary defines the scope and context of the system.

These new terms are very important, as they allow the concept of a 'context' to be introduced, which is a very useful tool when it comes to trying to understand the three basic types of system that were introduced in this section: systems of interest, enabling systems and systems of systems.

1.6.3 The 'context'

In order to illustrate the idea of a context, it is first necessary to consider a simple example. Imagine a system that must be developed by a particular project that is to be some sort of robot system. There are few requirements for the sake of this example. The aim of the project is to provide a vehicle that can navigate terrain, both automatically and through manual operation, and that can stay alive. The system must have a self-contained power supply and must be able to avoid obstacles while navigating the terrain.

This may be visualized by drawing up a 'use case diagram'. Use case diagrams will be described in more detail later in the book, but, for the sake of this example, all that one needs to be able to understand is that the ellipses in the diagram represent requirements, the stick people represent stakeholders and the large rectangle represents the system boundary.

The diagram in Figure 1.8 shows the context of the system – what the system is intended to do, where the boundary lies and what lies outside the boundary of the system. In fact, the last sentence as written here is not true. The diagram does not show *the* context of the system, but shows *a* context of the system. This point is critical to understanding systems. A context simply represents a system *from a particular stakeholder's point of view*. This diagram shows the context from the point of view of the robot product.

It is now useful to tie this diagram back to the concepts that were introduced previously.

Everything inside the system boundary represents the requirements of the 'system of interest'. The stakeholders (represented by stick people) outside the boundary of the system may be considered to be the 'enabling systems'.

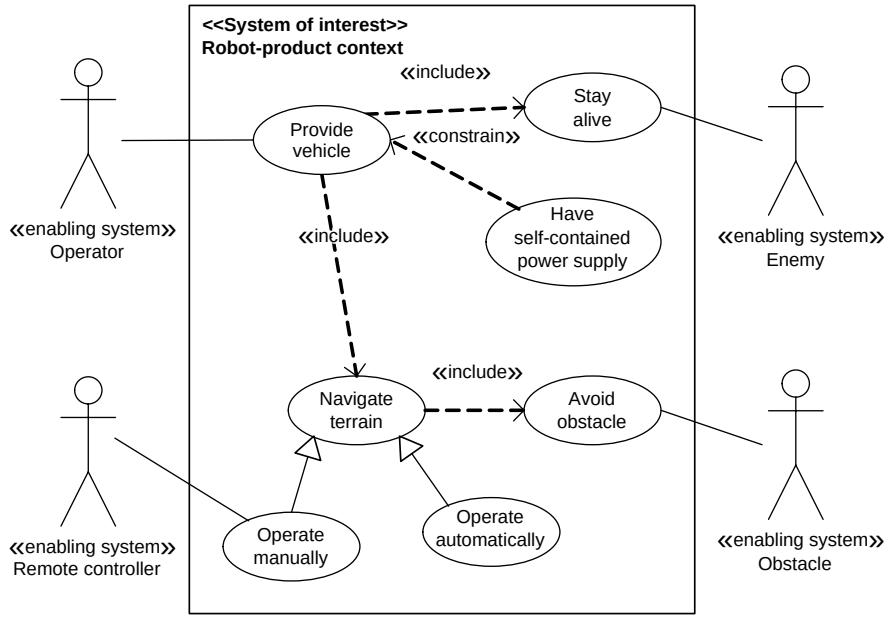

Figure 1.8 Example context

Consider now, however, the same system – the robot – but this time from the point of view of the enemy. This is the same system, but from a different point of view – hence it represents a different context.

Consider the diagram in Figure 1.9, where the same system is looked at from several different points of view, based on the stakeholders from the previous diagram.

The diagram in Figure 1.9 shows the same system from six different stakeholders' points of view, all of which are quite different. Note that, as stated previously, one stakeholder's system of interest is another stakeholder's enabling system, and vice versa. Also, when viewed in its entirety, this may be considered a system of systems. This same system of systems may be represented at a higher level of abstraction, by a single context. When considering a system of systems, we look at the system from differing points of view, but this time they may be based on levels of abstraction, rather than from specific stakeholders' points of view.

The diagram in Figure 1.10 shows a higher-level context for the same system of systems, but this time from the country's (home nation's) point of view. In this case, a single context is shown, compared with the many interacting systems that were shown in Figure 1.9. Each of these two approaches is valid and, again because of the concept of the context, the more appropriate one will depend on the point of view being adopted.

The main point here is to demonstrate that there are several ways that the same concepts – systems of systems – can be visualized, depending on why we need to visualize and from which point of view we are visualizing.

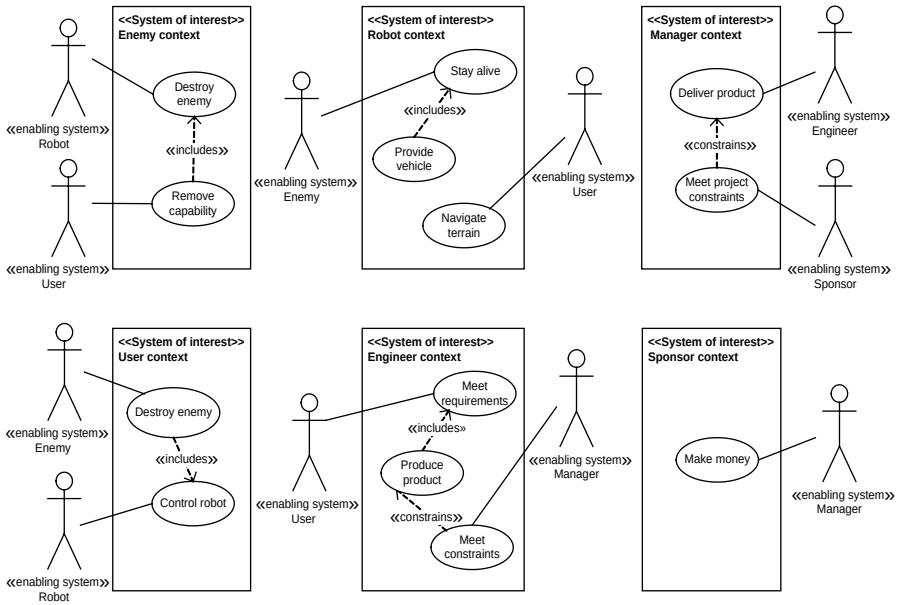

Figure 1.9 Several contexts

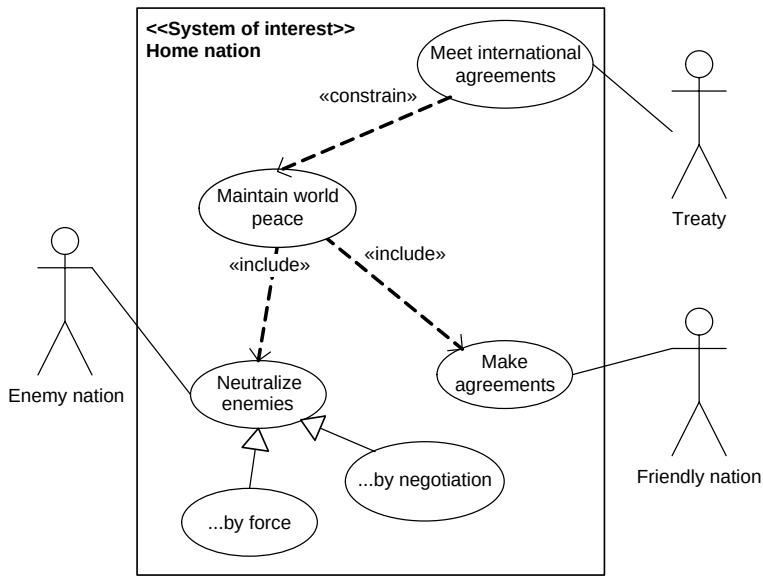

Figure 1.10 Single, high-level context

In summary, therefore, it is essential to look at things from different points of view in order to get a full understanding of the system as a whole.

1.6.4 Life cycles

Perhaps one of the most widely misunderstood, and most important, aspects of systems engineering is that of the life cycle.

Everything in life has a life cycle, whether it is a person, a product or a project. Just to confuse matters, there are very often a number of life cycles within life cycles. For example, consider the life cycle of a passenger train (a product) and then consider the number projects that will be run in order to cover the whole life cycle of the train. Each of the projects has its own life cycle.

For the purposes of this book, the main emphasis will be on project life cycles.

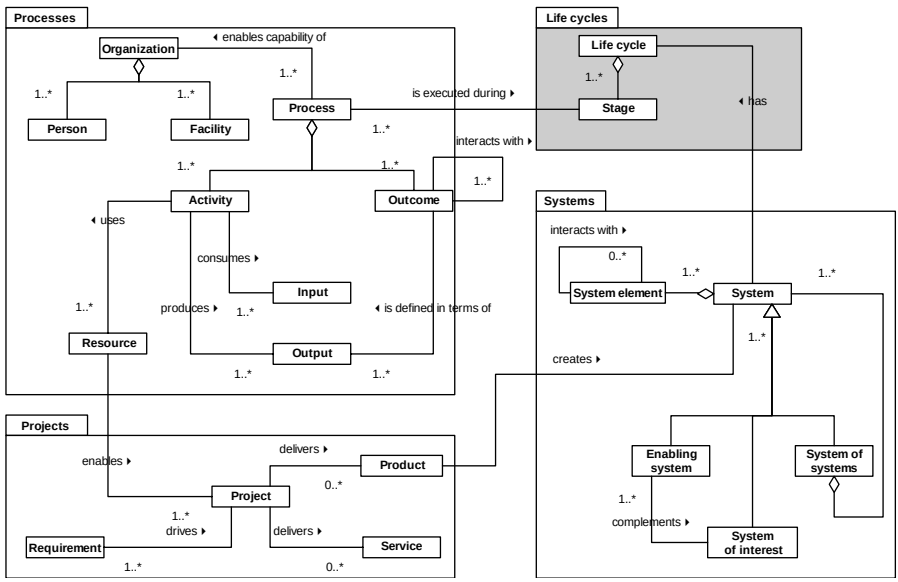

Figure 1.11 Systems-engineering concepts with a focus on 'life cycles'

The diagram in Figure 1.11 shows that a life cycle is made up of a number of stages, the names of which will vary according to whatever standard or best-practice model is being used. A life cycle starts from when the idea is first thought of, and ends when the project or product is decommissioned.

People often use informal phrases for life cycles, such as 'cradle to grave', 'lust to dust' and 'sperm to worm'. Interestingly enough, all these analogies relate to the life of a human, but each can be viewed as being incorrect, as they all assume that the life cycle starts at conception. Although this is very often the case, there are many cases where people may be 'trying' for a baby for many years involving all sorts of complex procedures, in which case there is a long time spent in the life cycle before any conception takes place.

During each stage, a number of processes will be executed. A very common misconception is to confuse the terms *process* and *stage* which is a hang-up from historical life-cycle models, such as the waterfall model, where there was a simple one-to-one ratio between each stage and the processes executed within it. For example, many textbooks will still talk about a 'requirements' stage, where a single set of activities is carried out. In more up-to-date life-cycle models, such as an iterative-style approach, there will be many processes executed within a single stage.

Another common misconception is between the terms *life cycle* and *life-cycle model*. A life cycle is a structural concept and simply identifies a number of stages – in SysML this would be realized using a block diagram. A life-cycle model, on the other hand, is a behavioural concept that describes what actually is supposed to happen or has happened. This would be realized in SysML using an interaction diagram, such as a sequence diagram.

Life cycles and life-cycle models will be discussed in more detail in Chapter 6.

1.6.5 Projects

Systems would not come into existence without projects. Projects are the things that deliver products and services and every project will have a life cycle.

A project will deliver a number of products or services. In the case of a product, this is usually something tangible that many people will think of as a system. On the other hand, although a service may be viewed as being far less tangible, it is nonetheless a system that is being delivered.

Figure 1.12 shows that any project requires a number of resources to enable it, whether they are human or physical.

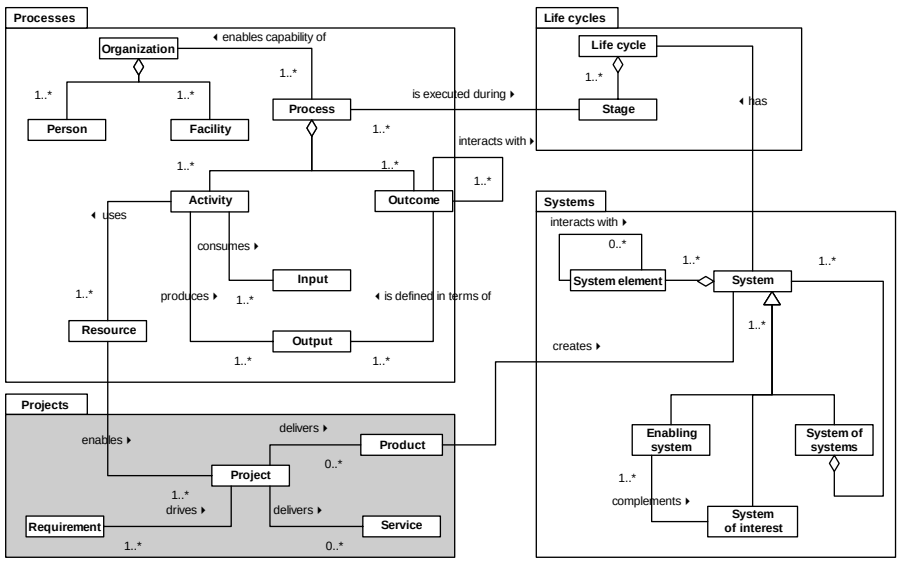

Figure 1.12 Systems engineering concepts with a focus on 'projects'

All projects are driven by requirements. This is a keystone to the whole of systems engineering and, indeed, is a massive field of work and research in itself. Requirements will not only drive the project, but are also a benchmark for quality, as all acceptance tests are based directly on requirements. There are many definitions for the term *quality* but two of the more widely used ones are: 'fitness for purpose' and 'conformance to requirements' [12]. Therefore, if the requirements are not fully understood, then the system cannot be validated properly and, hence, quality cannot be demonstrated.

1.7 Modelling

The three evils of systems engineering were introduced earlier in this chapter and one of the main thrusts of this book is the promotion of modelling. Indeed, modelling is one way that can help us address these evils, since modelling allows us to identify complexity, aid understanding and improve communication.

1.7.1 Defining modelling

In order to understand modelling, it is important to define it and (for a change) this is relatively straightforward. We define a model as a *simplification of reality* [13]. It is important to simplify reality in order to understand the system. This is because, as humans beings, we cannot comprehend complexity.

If a model is a simplification of reality, there are many things then that may be thought of as a model.

- *Mathematical* models, such as equations that represent different aspects of a system, that may be used as part of a formal analysis or proof.
- *Physical* models, such as mock-ups, that may be used to provide a picture of what the final system will look like or may be used as part of a physical simulation or analysis.
- *Visual* models, such as drawings and plans, that may be used as a template for creation or the basis of analysis.
- *Text* models, such as written specifications, that are perhaps the most widely used of the tools at our disposal. Regarding text as a model can be initially quite surprising but, the second we start to describe something in words, we are simplifying it in order to understand it.

This is by no means an exhaustive list, but it conveys the general message.

It is important to model so that we can identify complexity, increase our understanding and communicate in an unambiguous (or as unambiguous as possible) manner.

In order to model effectively, it is essential to have a common language that may be used to carry out the modelling. There are many modelling approaches, including graphical, mathematical and textual, but, regardless of the approach taken, there are a number of requirements for any modelling language.

1.7.2 The choice of model

The choice of model refers to the fact that there are many ways to solve the same problem. Some of these ways will be totally incorrect, but there are always more correct ways than one to solve the problem at hand. Although all these approaches may be correct, some will be more *appropriate* and, hence, more correct for the application. For example, if one wanted to know the answer to a mathematical equation, there are several approaches open: you may simply ask someone else what the answer is; you may guess the answer; you may apply formal mathematical analysis and formulae; or you may enter the equation into a mathematical software application. All may yield a correct answer, but the most appropriate approach will be application-dependent. If you were merely curious as to the answer to the equation, then guessing or asking someone else may be entirely appropriate. If, on the other hand, the equation was an integral part of the control algorithm for an aeroplane, then something more formal would be more appropriate.

It is important that we have a number of different tools available in order to choose the most appropriate solution to a problem, rather than just rely on the same approach every time. Therefore, one requirement for any modelling language is that it must be flexible enough to allow different representations of the same information to allow the optimum solution to be chosen.

1.7.3 The level of abstraction

Any system may be considered at many different levels of abstraction. For example, an office block may viewed as a single entity from an outside point of view. This is what will be referred to as a *high level of abstraction*. It is also possible to view a tiny part of the office block, for example the circuit diagram associated with a dimmer switch in one of the offices. This is what is known as a *low level of abstraction*. As well as the high and low levels of abstraction, it is also necessary to look at many intermediate levels, such as each floor layout on each level, each room layout, the lifts (or elevators), the staircases and so on. Only by looking at something at high, low and in-between levels of abstraction is it possible to gain a full understanding of a system.

Therefore, the second requirement for any modelling language is that any system must be able to be represented at different levels of abstraction.

1.7.4 Connection to reality

It has already been stated that, by the very nature of modelling, we are simplifying reality and there is a very real danger that we may oversimplify to such a degree that the model loses all connection to reality and, hence, all relevant meaning.

One type of modelling where it is very easy to lose the connection to reality is that of mathematical modelling. Mathematical modelling is an essential part of any engineering endeavour, but it can often be seen as some sort of dark art, because very few people possess a sufficient knowledge to make it usable and, indeed, many people are petrified of maths! Consider the example of the mathematical operation of differentiation, which is used to solve differential equations. As every schoolchild knows, differentiation can be applied in a quite straightforward manner to achieve a

result. What this means in real life, however, is another matter for discussion entirely. Differentiation allows us to find the slope of a line that, when taken at face value, and particularly when at school, can be viewed as being utterly meaningless. To take this example a little further, we are then told that integration is the *opposite* of differentiation (what is the opposite of finding the slope of a line?), which turns out to be measuring the area underneath a line. Again, when first encountered, this can be viewed as being meaningless. In fact, it is not until later in the educational process, when one is studying subjects such as physics or electronics, that one realizes that finding the slope of a line can be useful for calculating rate of change, velocity, acceleration, etc. It is this application, in this example, that provides the connection to reality and hence helps communicate 'why'.

The third requirement for any modelling language, therefore, is that it must be able to have a strong connection to reality and, hence, be meaningful to observers who, in many cases, should require no specialist knowledge, other than an explanation, to understand the meaning of any model.

1.7.5 Independent views of the same system

Different people require different pieces of information, depending on who they are and what their role is in the system. It is essential that the right people get the right information at the right time. Also, for the purpose of analysing a system, it is important to be able to observe a system from many different points of view. For example, consider the office block again, where there would be different types of people requiring different information: the electricians require wiring diagrams, not colour charts or plumbing data; the decorators require colour charts, not wiring diagrams; and so on.

There is a potentially very large problem when considering things from different points of view, and this is consistency. Consistency is the key to creating a correct and consistent model and, without any consistency, it is not possible to have or demonstrate any confidence in the system.

The fourth requirement, therefore, for any modelling language is that it must allow any system to be looked at from different points of view and that these views *must* be consistent.

1.8 SysML – the system modelling language

It has been established, therefore, that there is a clear need for a common language that can be used as an enabler for our systems-engineering activities. The most widely used language, to date, to achieve this has been the Unified Modelling Language, or UML, as it is widely known. However, there were many arguments why UML is suitable or not for systems engineering, but the irrefutable facts of the matter are these.

- The UML has been used very successfully for systems-engineering activities in many application domains for many years.
- There are some areas of the UML that can be improved upon.

If these two statements can be accepted as being true, then the SysML may very well be the language for you!

The SysML provides an excellent set of extension mechanisms to the UML that can be used to augment and enhance the basic capabilities of UML. There are some aspects of the SysML that are contentious and there are many strong arguments against, for example, leaving out major parts of the original UML modelling language.

It must also be pointed out that everything that can be modelled in SysML is an extension of what already exists in UML. In fact, it is possible to represent all SysML constructs in existing UML constructs, as is shown in Appendix B.

This book intends to look at all aspects of the SysML, both the good and bad, and allow the reader to decide which parts of the SysML will provide them with the most benefits.

1.9 Using this book

When using this book, it is important to remember that it is not, in itself, intended to be any sort of ultimate reference for systems engineering. It cannot possibly hope to cover the modelling of all the areas of systems engineering that are necessary to an organization. Nor can it cover every modelling concept within the Systems Modelling Language.

However, when used in conjunction with some of the information sources mentioned earlier, this book should form part of the greater systems knowledge.

In using this book as part of the greater systems knowledge, there are also a number of points to consider concerning the other information sources used.

- Both the handbook and ISO 15288 are evolving entities and, therefore, are subject to change. However, that said, it is unlikely there will be any major conceptual changes in either (one would certainly hope not), but it may be that much of the terminology and actual techniques used may change as time goes on.
- Both *The INCOSE Systems Engineering Handbook* and ISO 15288 are intended to be used as guides only and should not be taken as being carved in stone. Therefore, do not simply accept everything in them, but look for the parts that are relevant and suitable for the systems that you work on, rather than taking everything verbatim from them.
- Both represent best practice and, therefore, must be tailored to suit your requirements. Too many people want complete off-the-shelf solutions to all life's problems but, unfortunately, life is more complex than that. Indeed, if it were possible simply to pick up a best-practice manual that applied to everyone and solved all life's problems, there would be no need for systems engineering.

1.9.1 Competency

The world of competencies and competency frameworks is a varied and complex one. It is yet another term where there are many different definitions, but one thing that

is agreed by all sources is that, in order to carry out systems engineering effectively, competent people are needed to carry out the roles in a project.

At the highest level, competency may be seen as being able to demonstrate three things:

- *knowledge*, concerning application, and domain knowledge that is essential to understand our systems;
- *skills*, such as techniques and tools that are essential to be able to realize our project successfully; and
- *attitude*, such as behavioural and professional attitudes that allow engineers to work in an efficient manner and to understand the value in what we do.

It is the intention of this book to address all three at some level, so that people may become, in time, competent in applying the SysML to systems-engineering projects. There is only so much that a book can achieve, and an essential part of evolving one's competence from beginner to expert is to practise these techniques.

For those interested in competency frameworks, the one that is used as a basis for this book is the INCOSE competency framework.

1.10 Conclusions

This chapter has introduced, at a high level, the world of systems engineering. There were a number of key concepts that were introduced that were classified into four main groupings: projects, processes, systems and life cycles. It has shown how there are three evils that affect us all, which are complexity, a lack of understanding and communication issues. One way to address these three issues is to apply modelling. In order to model effectively, it is essential to choose a common modelling language that can be used by all relevant people. In order to choose an appropriate modelling language there are certain requirements that must be satisfied in that the modelling language must present a number of modelling options, be able to represent information at different levels of abstraction, have strong connection to reality and be able to represent the system from different points of view.

The modelling language that will be described in this book is the systems-engineering modelling language, or SysML, which is a visual modelling language, based on the UML that satisfies all the above-mentioned requirements.

The remainder of this book looks in detail at the language itself and then explores some common uses and applications for modelling.

1.11 References

1 International Standards Organization. 'ISO 15288, Lifecycle management – system life cycle processes'. ISO Publishing; 2003
2 International Council on Systems Engineering (INCOSE). 'INCOSE systems engineering handbook – a guide for system life cycle processes and activities'. Version 3.0; June 2006

3 International Standards Organization, op. cit.
4 Federal Aviation Agency (USA FAA). *Systems Engineering Manual* (definition contributed by Simon Ramo); 2006
5 Eisner, Howard. *Essentials of Project and Systems Engineering Management*. New York: Wiley; 2002
6 International Council on Systems Engineering, op. cit.
7 International Council on Systems Engineering (INCOSE). 'INCOSE competencies framework'. Issue 2.0; November 2006
8 Holt J. *UML for Systems Engineering: Watching the Wheels*. 2nd edn. London: IEE Publishing; 2004 (reprinted, IET Publishing; 2007)
9 Elk A. 'The brontosaurus sketch', *Monty Python's Flying Circus*. BBC TV; 1974
10 Pressman R. *Software Engineering: A Practitioner's Approach: European Adaptation*. Maidenhead: McGraw-Hill Publications; 2000
11 Brookes F.P. *The Mythical Man-Month*. Boston, MA: Addison-Wesley; 1995
12 International Standards Organization. 'ISO 9001, Model for quality assurance in design, development, production, installation and servicing'. ISO Publishing; 1994
13 Booch G., Rumbaugh J. and Jacobson I. *The Unified Modelling Language User Guide*. Boston, MA: Addison-Wesley; 1999

Chapter 2
An introduction to SysML

'You can't make a silk purse from a sow's ear.'

Traditional

2.1 Introduction

This chapter provides a short overview of the Systems Modelling Language (the SysML), setting out its intended purpose, its relation to the Unified Modelling Language (the UML) on which it is based and providing a brief chronology of the SysML to date with a high-level comparison of the various forms that SysML has taken since its inception.

2.2 What is SysML?

SysML is defined on the website of the OMG (the Object Management Group – the US-based industry-standards body who manage and configure the SysML) as 'a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities'.

The primary audience for SysML is systems engineers. Currently, systems engineers use a wide range of modelling languages, tools and techniques. SysML is intended to unify the diverse modelling languages currently used by systems engineers in similar manner to how UML unified the modelling languages used in the software industry, and originates from an initiative between the OMG and the International Council on Systems Engineering (INCOSE) in 2003 to adapt UML for systems-engineering applications (we will look in more detail at the history of SysML below).

SysML allows engineers to model system requirements, system behaviour and system structure. The focus of the new constructs in SysML is geared towards some fundamental systems-engineering concepts, such as requirements engineering and system behaviour. Although requirements have traditionally been realized using use case diagrams, which consider requirements from a behavioural point of view, the introduction of the requirements diagram in SysML allows the structural relationships between requirements to be modelled.

2.2.1 SysML's relation to UML

The SysML is often presented as being an entirely new modelling language that is aimed at the systems-engineering world, but there have been several languages that have been used in the past. Perhaps the most relevant of these languages is the UML. The SysML is actually based on the original UML and, as such, cannot be deemed a completely new language – more a set of useful additions to the existing core UML modelling concepts and diagrams.

The UML is a general-purpose graphical modelling language aimed at *software* engineers which, on its appearance in 1997, represented a major unification of the large number of such languages that sprang up in the late 1980s and early 1990s.

The UML defines 13 types of diagram that allow the requirements, behaviour and structure of a software system to be defined. Since its appearance the UML has been increasingly adopted for use outside the software field, and is now widely used for such things as systems engineering and process modelling.

The six structural and seven behavioural diagrams of the UML are shown in Figures 2.1 and 2.2. Each diagram is also briefly described.

The six structural diagrams of UML are as follows.

1 *Class diagram* – used to realize the structural aspects of a system, showing what things exist and the relationships between them.
2 *Package diagram* – used to identify and relate packages. Note that packages may also be shown on other diagrams and are used to group elements of the model.
3 *Composite structure diagram* – has two distinct uses: used to simplify the representation of complex class structures that use aggregation and composition by allowing the elements of the composition to be shown within the aggregating part; also used to show collaborations, that is some sort of communication, that exist between structural elements.
4 *Object diagram* – heavily related to the class diagram, used to represent real-life examples of the classes found in class diagrams.
5 *Component diagram* – with a strong relationship to both class and object diagrams, used to show the components of a system, that is the 'packaged' classes that form the delivered elements of a system. Used to show 'what's in the box'.
6 *Deployment diagram* – used to show how system components are deployed, whether onto other systems, into organizations or into physical locations.

The seven behavioural diagrams of UML are as follows.

1 *State machine diagram* – used to model the behaviour of an object; that is they model the behaviour during the lifetime of a class.
2 *Activity diagram* – used, generally, to model low-level behaviour such as that within an operation and also to model workflows.
3 *Use case diagram* – represents the highest level of abstraction of a system and is used to model the requirements and contexts of a system.
4 *Sequence diagram* – used to model interactions between life lines in a system, with an emphasis on the logical timing of interactions between the life lines. A

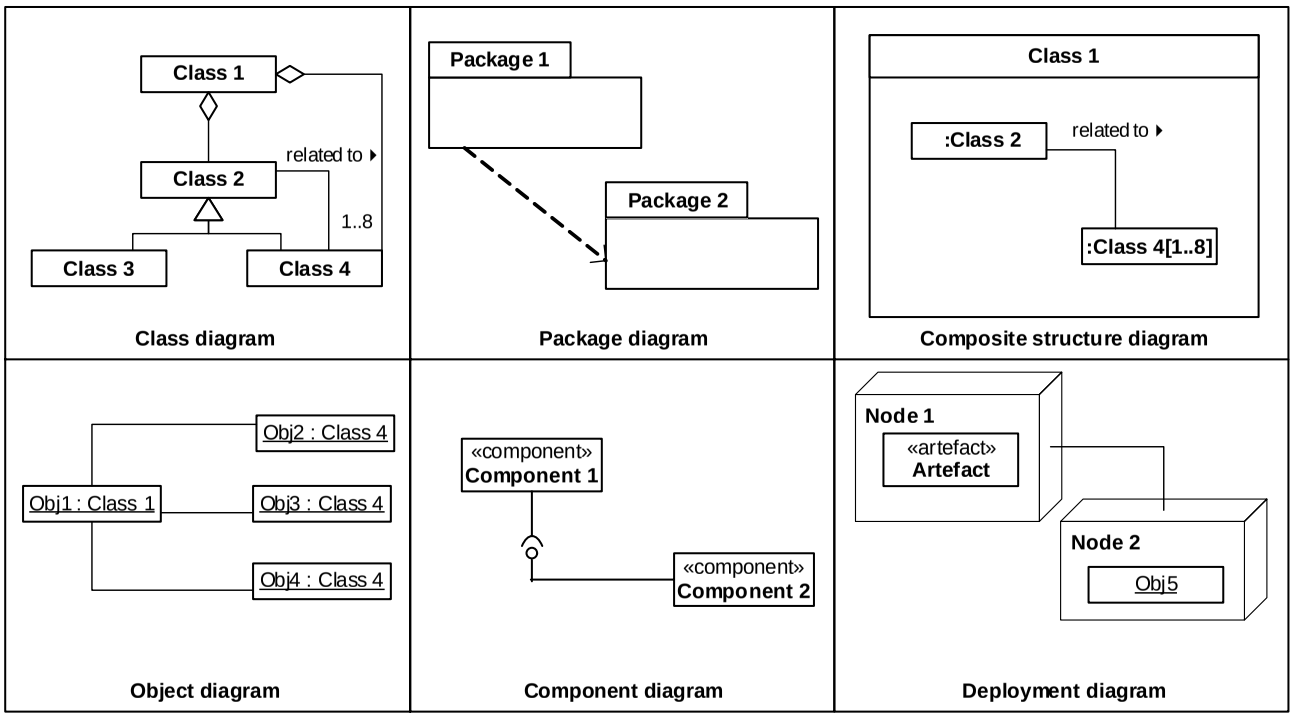

Figure 2.1 UML structural diagrams

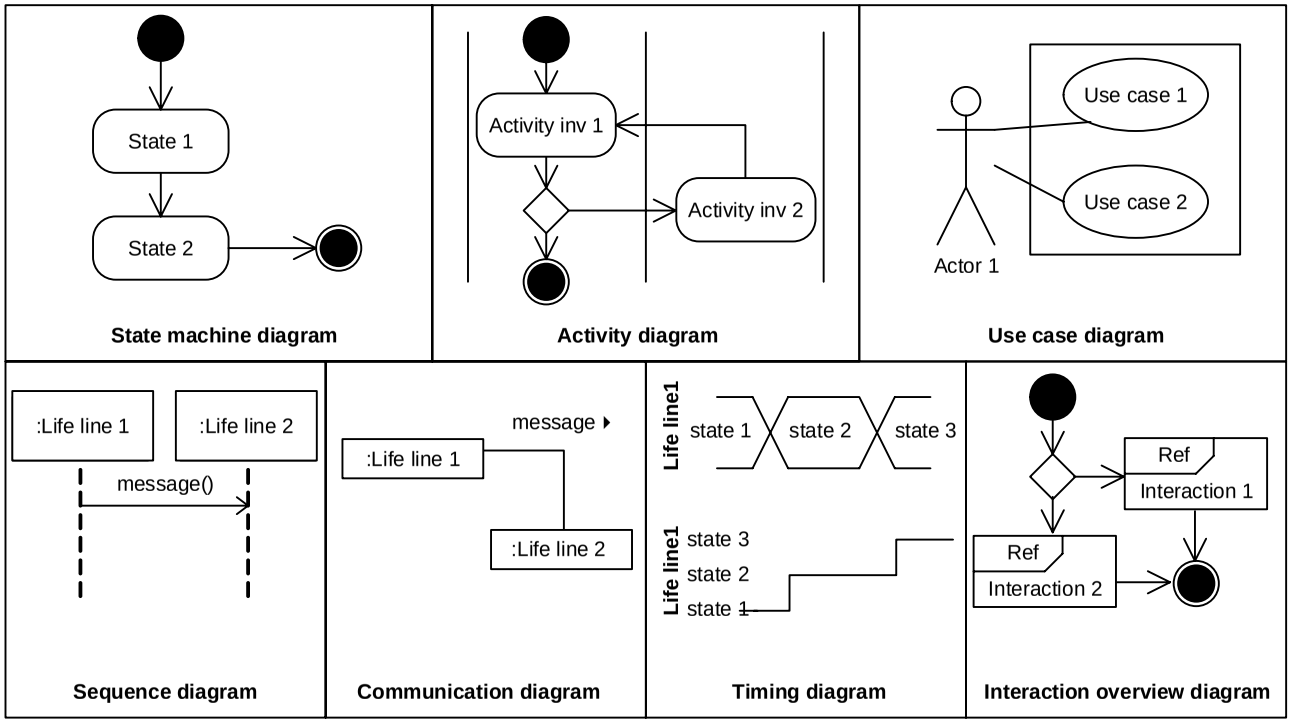

Figure 2.2 UML behavioural diagrams

life line is an individual participant in an interaction and refers to an element from another aspect of the model, such as an instance of a class.

5 *Communication diagram* – used to model interactions between life lines in a system, with an emphasis on the organizational layout of the elements.
6 *Timing diagram* – used to realize timing aspects of interactions.
7 *Interaction overview diagram* – used to assemble complex behaviours from simpler scenarios, with a syntax related to that of the activity diagram.

Despite this growing use of UML for systems engineering, there was still a perceived need for a tailored version of UML that was aimed specifically at systems engineers, with some of the elements and diagrams of UML considered to be aimed more at software systems removed. The result of this perceived need is the SysML, which in its latest version has nine diagrams compared with the thirteen of UML. The history of SysML is considered in more detail in Section 2.3, with a comparison of the various versions given in Section 2.4.

So what is the relationship between SysML and UML? Figure 2.3 illustrates the overlap between the two languages.

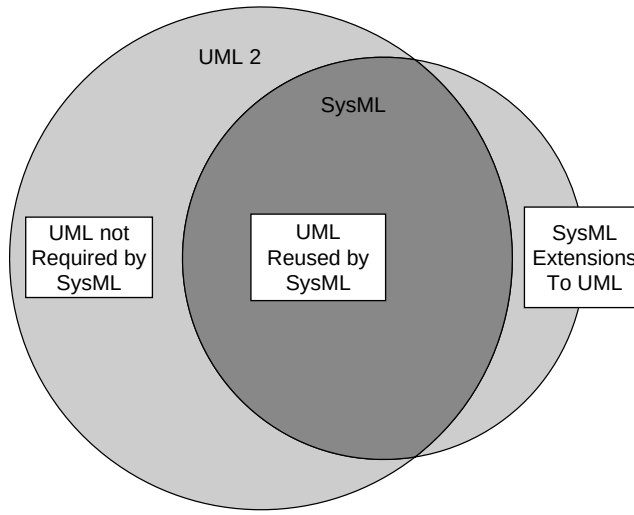

Figure 2.3 The relationship between SysML and UML

As can be seen in Figure 2.3, SysML makes use of much of the UML. However, some parts of the UML are not required by SysML; in particular the following diagrams are not used: *object diagram*, *deployment diagram*, *component diagram*, *communication diagram*, *timing diagram* and *interaction overview diagram*. In addition, SysML adds some new diagrams and constructs not found in UML: *the parametric diagram*, *the requirement diagram and flow ports*, *flow specifications and item flows*. Those parts of the UML that are reused by UML are also subject

to some changes to make the notation more suitable for systems engineering, e.g. replacing the concept of the class with that of the *block*.

Section 2.4.3 gives an overview of the contents of the latest version of the SysML. Chapter 4 describes the nine SysML diagrams in detail.

2.3 The history of SysML

SysML has undergone a long and tedious evolution, with a history extending back to 2003, when the OMG issued the 'UML for Systems Engineering Request for Proposal' (RFP), following a decision by INCOSE to customize UML for systems-engineering applications. In response to this RFP, there have been several major drafts of SysML, which have differed in terms of the content and concepts. At one point, the SysML team split into two groups, each led by a different CASE (Computer-Aided Systems/Software Engineering) tool vendor, who both produced separate specifications for the standard, one of which was then chosen as the basis for the official SysML by the OMG. The tangled relationships between the various versions are shown in the SysML block definition diagram in Figure 2.4.

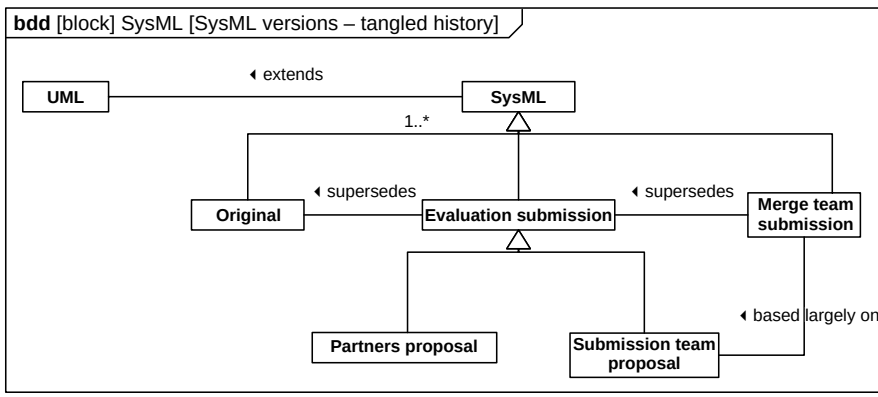

Figure 2.4 The tangled history of SysML

Figure 2.4 shows that one or more versions of 'SysML' extends 'UML' and that there have been three main types of 'SysML': the 'Merge team submission' supersedes the 'Evaluation submission' which supersedes the 'Original' version. The 'Evaluation submission' itself consisted of two different types, the 'Partners proposal' and the 'Submission team proposal'. The 'Merge team submission' is based largely on the 'Submission team proposal'.

The development of the SysML has involved a number of organizations from industry, government and academia, as shown in Table 2.1.

What is of interest in the list of contributors is the large number of CASE tool vendors involved in the definition of the language. This is particularly apparent when

Table 2.1 Organizations involved in the development of SysML

Industry and government
American Systems
BAE Systems
The Boeing Company
Deere & Company
EADS Astrium GmbH
Eurostep Group AB
Lockheed Martin Corporation
Motorola, Inc.
National Institute of Standards and Technology (NIST)
Northrop Grumman Corporation
oose.de Dienstleistungen für innovative Informatik GmbH
Raytheon Company
Thales
Tool vendors
ARTiSAN Software Tools
EmbeddedPlus Engineering
Gentleware AG
IBM
I-Logix, Inc.
Mentor Graphics
PivotPoint Technology Corporation
Sparx Systems
Telelogic AB
Vitech Corp
Academia
Georgia Institute of Technology
Liaison organizations
INCOSE
ISO 10303-233 (AP233) Working Group

the two evaluation teams are considered: the SysML Partners team was led by Telelogic, while the Submission Team was led by ARTiSAN, both prominent CASE tool vendors.

2.3.1 A brief chronology

An attempt to untangle the major events in the development of SysML, to date, is shown as a brief chronology in Table 2.2. Even though the ‘great SysML battle’ has, apparently, been won, given SysML’s complicated past the reader may expect further issues with its evolution into the future.

Table 2.2 A brief chronology of SysML

2003	
28 March	UML for Systems Engineering Request for Proposal issued
2004	
04 February	Initial submission presented to the OMG
03 August	SysML Specification v0.8 issued
2005	
10 January	SysML Specification v0.9 issued
30 August	Team splits in two
14 November	SysML Partners (led by Telelogic) issues SysML Specification v1.0a and on the same date SysML Submission Team (led by ARTiSAN) issues SysML Specification v0.98
2006	
13 February	Two teams merge to submit a single proposal: SysML Merge Team issues SysML Specification v0.99
03 April	SysML Specification v1.0 draft issued
26 April	Vote to request adoption of the OMG SysML specification passed
04 May	Final adopted specification issued
2007	
April	An OMG Finalization Task Force plans to recommended a final version OMG SysML

Indeed, as of the time of writing, it appears that not all those involved in the battle are happy to accept the outcome. The merged version that was adopted in April 2006 is now known as OMG SysML (throughout this book we will, wherever possible, use the far simpler, and shorter, SysML to mean OMG SysML). However, the original SysML website maintained by the 'SysML Partners' states that OMG SysML 'is derived from open source SysML'. What is not clear from this is whether the SysML Partners intend to develop their SysML Specification v1.0a further as a rival open-source version of SysML competing with the 'official' OMG SysML.

Section 2.4 compares the main versions of SysML in more detail.

2.4 A comparison of versions

To systems engineers attempting to use and understand SysML since its announcement in 2004, the ever-changing nature of the different, and sometimes competing, versions has been a source of some confusion. In this section the four main versions of SysML (as shown in Figure 2.4) are considered. For each version, two diagrams give a high-level visual comparison with UML and the major differences noted. In addition, a comparison is made between the two evaluation submissions that were both presented to the OMG in November 2005 following the split of the SysML team in August 2005.

2.4.1 The original version

For the purpose of considering the various versions of SysML, the first major version of SysML worth considering was Specification v0.9, issued in January 2005. The structural and behavioural diagrams in v0.9 are shown in Figure 2.5. The main features of v0.9, with a comparison with UML, are summarized here.

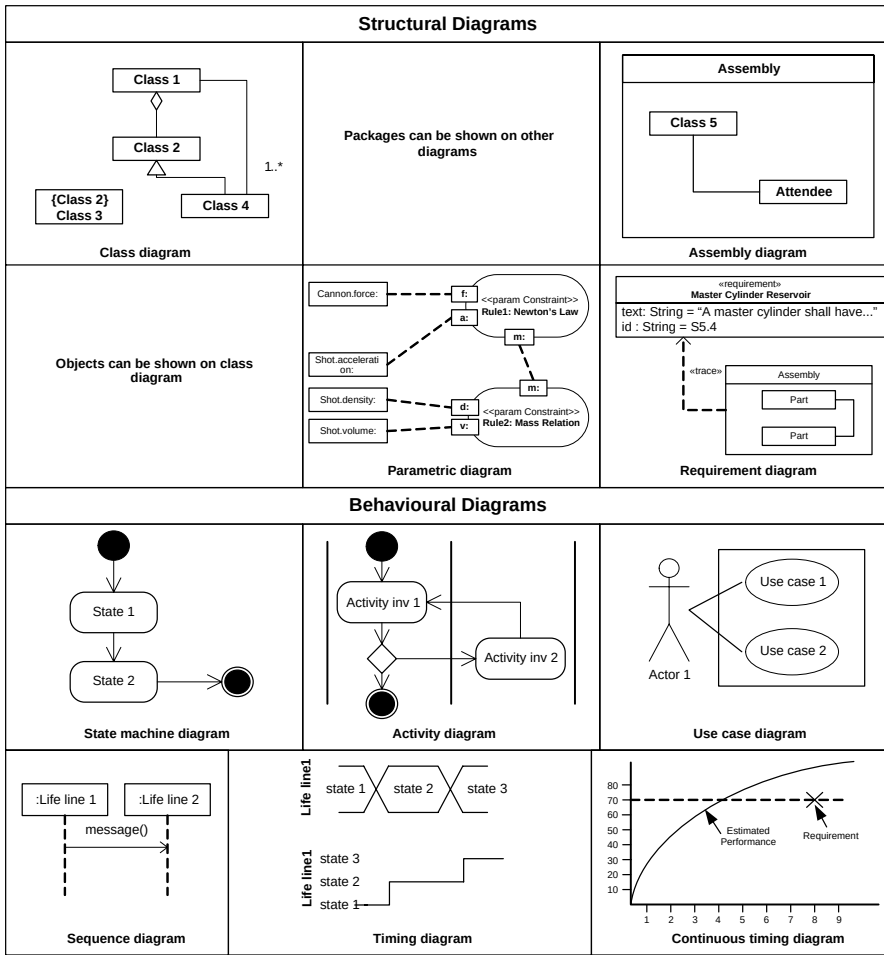

Figure 2.5 SysML structural and behavioural diagrams – original version

UML not required by SysML
Specification v0.9 did not include the following UML diagrams:

- object diagram – although objects could be shown on class diagrams;
- package diagram – although packages could be shown on other diagrams;

- deployment diagram;
- component diagram;
- communication diagram; and
- interaction overview diagram.

Also, while Specification v0.9 supported ports, there was no support for the concepts of interfaces.

UML reused by SysML
Specification v0.9 reused seven UML diagrams; some of which were renamed or had minor changes to their syntax. The seven diagrams, with the changes briefly noted, are:

- class diagram – with some minor additions to its syntax;
- composite structure diagram – renamed the *assembly diagram* in SysML and with some minor additions to its syntax;
- state machine diagram;
- activity diagram – with some minor additions to its syntax;
- use case diagram;
- sequence diagram; and
- timing diagram.

SysML extensions to UML
Specification v0.9 added three new diagrams and a number of new constructions:

- parametric diagram – this new diagram allowed the definition of sets of inter-related parametric equations which act as constraints on the system;
- requirement diagram – this special case of the class diagram allowed the modelling of detailed system requirements;
- continuous timing diagram – a new timing diagram that allowed continuous timing information to be shown; and
- allocations – a set of notations that allow different aspects of a SysML model to be related to each other.

2.4.2 The two evaluation versions

In August 2005 the SysML team split into two, forming the SysML Partners team and the SysML Submission team. Each team put forward a separate, and somewhat different, proposal to the OMG in November 2005. However, both of these submissions introduced similar name changes and new constructions to the original v0.9 submission. These two so-called evaluation versions are considered here.

2.4.2.1 SysML Partners' version

The SysML Partners' submission was Specification v1.0a. The structural and behavioural diagrams in v1.0a are shown in Figure 2.6. The main features of v1.0a, with a comparison to UML, are summarized here.

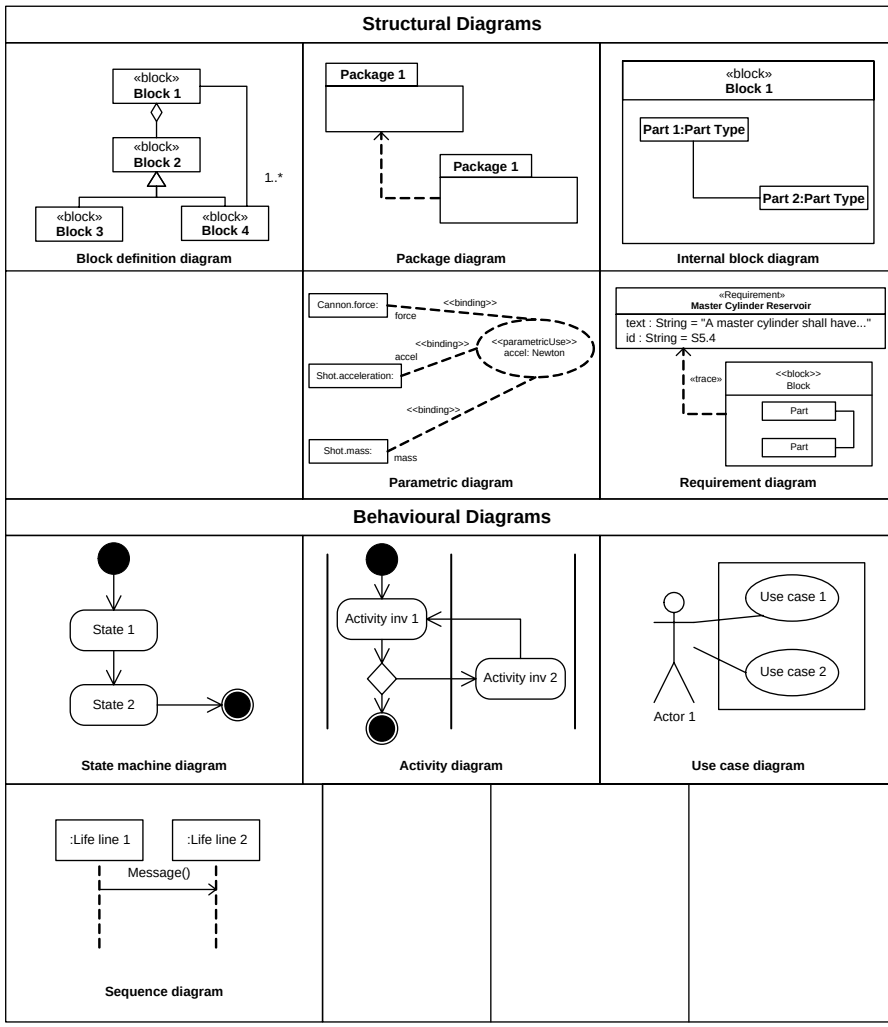

Figure 2.6 SysML structural and behavioural diagrams – SysML Partners' version

UML not required by SysML
Specification v1.0a did not include the following UML diagrams:

- object diagram;
- deployment diagram;
- component diagram;
- communication diagram;
- timing diagram – the continuous timing diagram introduced in the original v0.9 submission was also missing from this version; and
- interaction overview diagram.

UML reused by SysML
Specification v1.0a reused seven UML diagrams, some of which were renamed or had minor changes to their syntax. The seven diagrams, with the changes briefly noted, are:

- class diagram – renamed the *block definition diagram*, this was essentially the same as the UML class diagram;
- package diagram;
- composite structure diagram – renamed the *internal block diagram*, this was essentially the same as the UML composite structure diagram;
- state machine diagram;
- activity diagram – with some minor additions to its syntax;
- use case diagram; and
- sequence diagram.

Service ports and interfaces were reintroduced in Specification v1.0a; these were the same as UML ports and interfaces.

SysML extensions to UML
Specification v1.0a added two new diagrams and a number of new constructions, as follows.

- *Parametric diagram* – this new diagram allowed the definition of sets of interrelated parametric equations, which act as constraints on the system. The notation used was changed from that in the original v0.9 submission.
- *Requirement diagram* – this special case of the class diagram allowed the modelling of detailed system requirements. The notation used was changed from that in the original v0.9 submission.
- *Flow ports and flow specifications* – these related constructs allowed ports to be specified and connected that transfer and receive material, data, energy, etc. This was a major addition to the concept of ports in UML, which were aimed at defining interaction points for service-based interaction. In Specification v1.0a such UML-style ports were renamed as *service ports*.
- *Allocations* – a set of notations that allow different aspects of a SysML model to be related to each other. The notation used was changed from that in the original v0.9 submission.

2.4.2.2 SysML Submission Team version

The SysML Submission Team's submission was Specification v0.98. The structural and behavioural diagrams in v0.98 are shown in Figure 2.7. The main features of v0.98, with a comparison to UML, are summarized here.

UML not required by SysML
Specification v0.98 did not include the following UML diagrams:

- deployment diagram;
- component diagram;

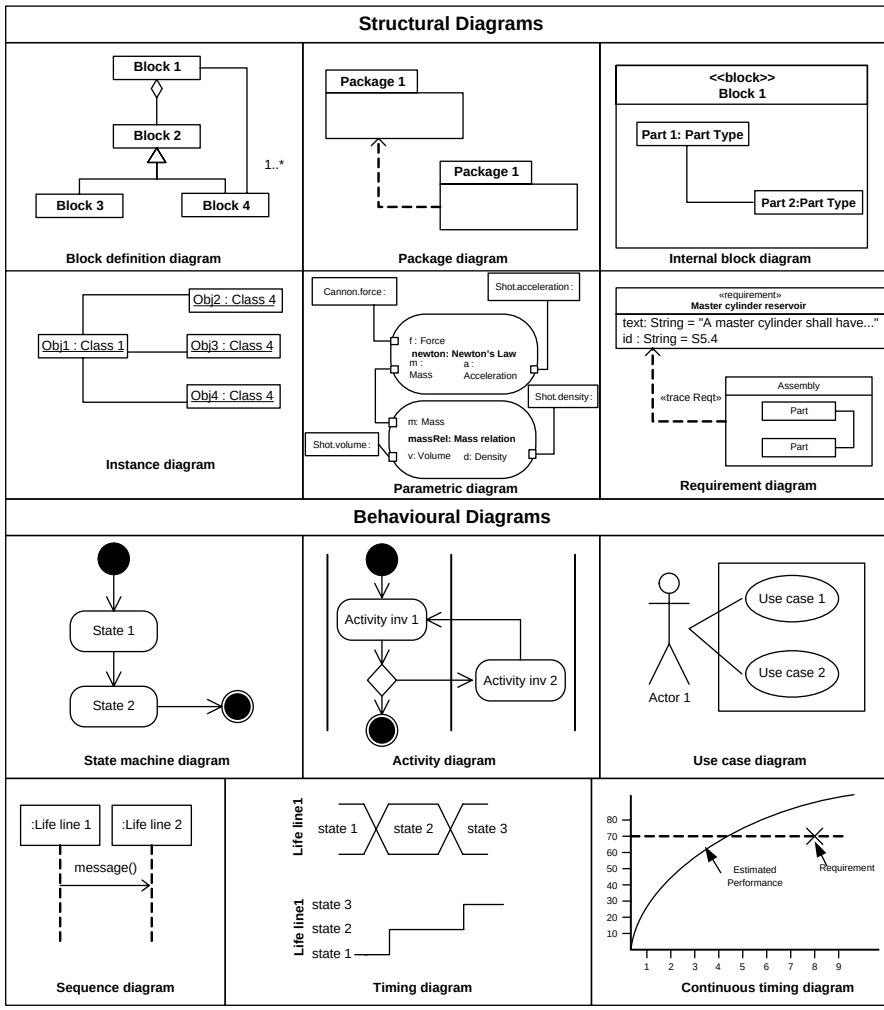

Figure 2.7 SysML structural and behavioural diagrams – SysML Submission Team version

- communication diagram; and
- interaction overview diagram.

UML reused by SysML

Specification v0.98 reused nine UML diagrams, some of which were renamed or had minor changes to their syntax. The nine diagrams, with the changes briefly noted, are:

- class diagram – renamed the *block definition diagram*, this was essentially the same as the UML class diagram;

- package diagram;
- composite structure diagram – renamed the *internal block diagram*, this was essentially the same as the UML composite structure diagram;
- object diagram – renamed the *instance diagram*;
- state machine diagram;
- activity diagram – with some minor additions to its syntax;
- use case diagram;
- sequence diagram; and
- timing diagram.

Client server ports and interfaces were reintroduced in Specification v0.98; these were the same as UML ports and interfaces.

SysML extensions to UML
Specification v0.98 added two new diagrams and a number of new constructions, as follows.

- *Parametric diagram* – this new diagram allowed the definition of sets of interrelated parametric equations, which act as constraints on the system. The notation used was changed from that in the original v0.9 submission.
- *Requirement diagram* – this special case of the class diagram allowed the modelling of detailed system requirements. The notation used was changed from that in the original v0.9 submission.
- *Flow ports, flow specifications and item flows* – these related constructs allow ports to be specified and connected that transfer and receive material, data, energy, etc. via item flows. This was a major addition to the concept of ports in UML, which were aimed at defining interaction points for service-based interaction. In Specification v0.98 such UML-style ports were renamed as *client server ports*.
- *Allocations* – a set of notations that allow different aspects of a SysML model to be related to each other. The notation used was changed slightly from that in the original v0.9 submission.

2.4.2.3 A comparison between the two versions

Both the SysML Partners' submission and that of the SysML Submission Team had a number of things in common, and both differed from the original v0.9 submission but in different ways. The main change in these versions, which they both shared, was the concept of classes and assemblies now being replaced with the single concept of the *block* and the introduction of *flow ports* and *flow specifications*.

While both specifications contained the concept of parametric diagrams, the SysML Partners' submission radically changed the notation used, in fact reusing some little-used UML notation. In addition, the SysML Partners' submission removed the concept of timing diagrams completely. The remaining differences between the two versions and the original v0.9 submission were of terminology and notation rather than any major differences in concepts.

2.4.3 SysML Merge Team version and the Final Adopted Specification

The Merge Team version, Specification v0.99, was submitted to the OMG in February 2006. This version was largely based on the Submission Team specification described in 2.4.2.2 above but with some changes. Table 2.3 shows these changes.

Table 2.3 The main differences between the Submission Team and Merge Team Specifications

Submission Team Specification (Specification v0.98)	**Merge Team Specification (Specification v0.99)**
Instance diagram	Omitted
Timing diagram	Omitted
Continuous timing diagram	Omitted
Client server ports	Renamed service ports

The Merge Team version then formed the basis of the Final Adopted Specification, which was issued in May 2006. Only minor errors and formatting changes were made to the Merge Team version in order to issue it as the Final Adopted Specification.

The structural and behavioural diagrams in the Final Adopted Specification are shown in Figure 2.8. The main features of the Final Adopted Specification, with a comparison with UML, are summarized here.

UML not required by SysML
SysML does not include the following UML diagrams:

- object diagram
- deployment diagram
- component diagram
- communication diagram
- timing diagram
- interaction overview diagram.

UML reused by SysML
SysML reuses seven UML diagrams, some of which are renamed or have minor changes to their syntax. The seven diagrams, with the changes briefly noted, are:

- class diagram – renamed the *block definition diagram* in SysML, this is essentially the same as the UML class diagram;
- package diagram;
- composite structure diagram – renamed the *internal block diagram* in SysML, this is essentially the same as the UML composite structure diagram;
- state machine diagram;

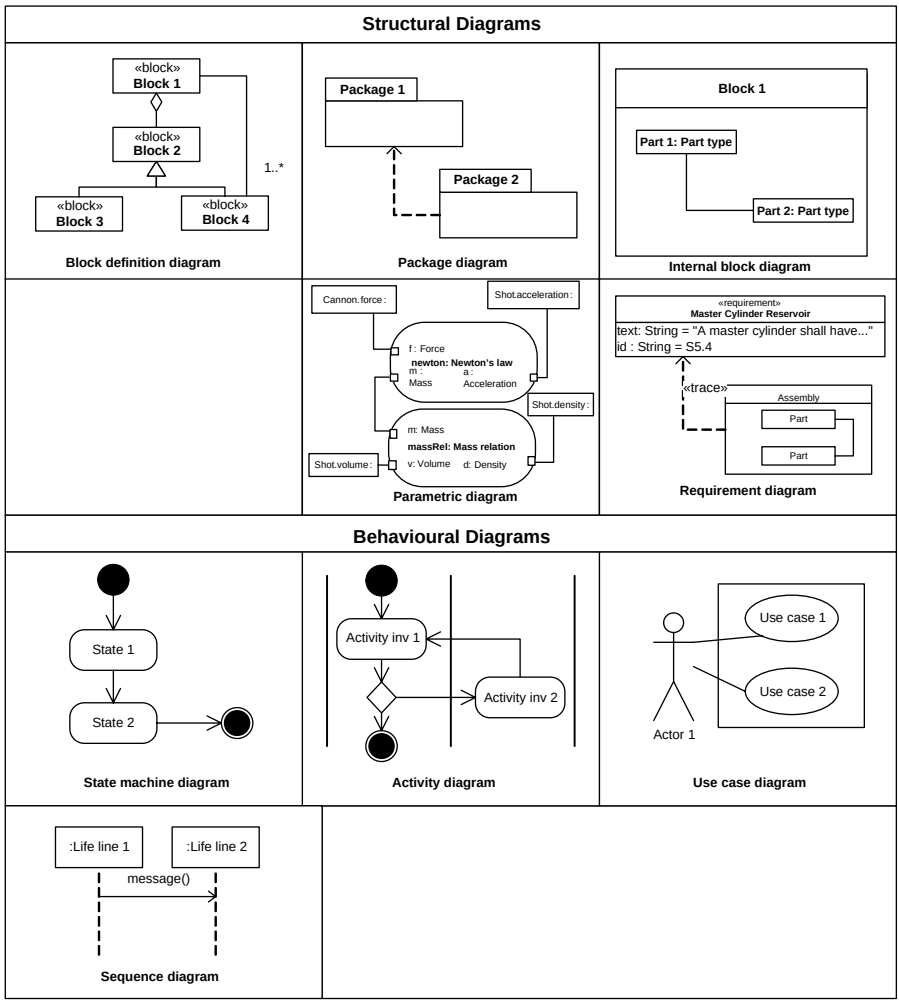

Figure 2.8 SysML structural and behavioural diagrams – Final Adopted Specification

- activity diagram – this has some minor additions to its syntax;
- use case diagram; and
- sequence diagram.

SysML extensions to UML
SysML adds two new diagrams and a number of new constructions.

- *Parametric diagram* – this new SysML diagram allows the definition of sets of interrelated parametric equations, which act as constraints on the system.

- *Requirement diagram* – this is a special case of the block definition diagram and is used to model detailed system requirements.
- *Flow ports, flow specifications and item flows* – these related constructs allow ports to be specified and connected that transfer and receive material, data, energy, etc. via item flows. This is a major addition to the concept of ports in UML, which were aimed at defining interaction points for service-based interaction. In SysML such ports are renamed as *service ports*.
- *Allocations* – a set of notations that allow different aspects of a SysML model to be related to each other.

2.5 Potential concerns with SysML

Those readers familiar with the UML will see from the brief comparison described above that SysML is *not*, contrary to widespread belief, a new modelling language. It is essentially little more than half of the UML with some additions. While many of these additions are useful, they could be achieved using the UML and its existing extension mechanisms – this approach is described further in Appendix B.

What is of concern is the omission of UML diagrams from the SysML. These suggested omissions seem to have come about due to the gross misunderstanding that the diagrams are not useful for systems engineering, only for software engineering. This is extremely dangerous and counterintuitive, particularly since the stated description of SysML, as quoted in Section 2.2 above, says that SysML is applicable to '. . . complex systems that may include hardware, *software*, information, personnel, procedures, and facilities' (authors' italics). The omission of these diagrams from SysML must make its use in such cases more difficult than would be the case had these diagrams remained.

For example, consider the UML *deployment diagram*. In software systems this diagram is used to show the physical deployment of software components to *nodes*. While in UML nodes are defined as something that can host software, either a piece of hardware or another piece of software such as an operating system, the concept behind a node can be extended in general systems engineering applications to cover such things as geographical locations, organizations, other systems and physical locations. This allows the deployment of a system to be modelled in an intuitive and succinct way.

Unfortunately, SysML does not include deployment diagrams, which is surprising given that systems have to be deployed somewhere. Rather, deployment has to be shown in SysML using the new generic *allocation dependencies*, a far less intuitive mechanism.

As another example, consider the UML *object diagram*, which is very useful for showing real-world examples of system elements. One of the earlier versions of the SysML specification (the Submission Team version) included the object diagram, renamed the *instance diagram*. However, the Final Adopted Specification does not include this diagram, or even the concept of an instance, although it does have the

less flexible notation for showing *property-specific types*, which can be used, albeit in a somewhat awkward fashion, to represent instances.

The Final Adopted Specification states,

> Since SysML uses UML 2.1 as its foundation, systems engineers modeling with SysML and software engineers modeling with UML 2.1 will be able to collaborate on models of software-intensive systems. This will improve communication among the various stakeholders who participate in the systems development process and promote interoperability among modeling tools.

This is a very worthwhile aim, but, with SysML omitting so many of the UML diagrams, its likely success has to be questioned.

2.6 Conclusions

The SysML is a new modelling language aimed at systems engineers. It is based on and provides a set of extensions to UML but is not an entirely new language. The extensions are useful and make sense; however, the deletions from the original UML are questionable.

SysML has undergone a long and tangled evolution, but a Final Adopted Specification was issued in May 2006, which the authors hope will have been ratified by the time you are reading this book.

2.7 Further reading

Booch G., Rumbaugh J. and Jacobson I. *The Unified Modeling Language User Guide*. 2nd edn. Boston, MA: Addison-Wesley; 2005

Holt J. *UML for Systems Engineering: Watching the Wheels*. London: IEE Publishing; 2004 (reprinted, IET Publishing; 2007)
Holt J. *A Pragmatic Guide to Business Process Modelling*. Swindon: British Computer Society; 2005

Object Management Group. SysML website [online]. Available from http://www.omgsysml.org [Accessed August 2007]

Object Management Group. UML website [online]. Available from http://www.uml.org [Accessed August 2007]

Rumbaugh J., Jacobson I. and Booch, G. *The Unified Modeling Language Reference Manual*. 2nd edn. Boston, MA: Addison-Wesley; 2005

SysML Partners. SysML website [online]. Available from http://www.sysml.org [Accessed August 2007]

The various versions of the SysML specification can be found at the locations shown in Table 2.4.

Table 2.4 SysML specification locations

Specification	Location
SysML Specification v0.9	http://www.sysml.org/specs.htm http://www.sysml.org/docs/specs/SysML-v0.9-PDF-050110.zip
SysML Specification v1.0a (the SysML Partners submission)	http://www.sysml.org/specs.htm http://www.omg.org/cgi-bin/doc?ad/05-11-05
SysML Specification v0.98 (the SysML Submission Team submission)	http://www.omg.org/docs/ad/05-11-01.pdf
SysML Final Adopted Specification	http://www.omg.org/cgi-bin/doc?ptc/06-05-04

Chapter 3
Modelling

'Always two, there are.'

Yoda

3.1 Introduction

Modelling is fundamental to everything that is presented in this book. So far, a brief introduction to modelling has been discussed, along with a set of requirements for any modelling language. In order to produce any sort of real model it is essential that the model be looked at in terms of two aspects – the structure and the behaviour.

This chapter looks at both structural and behavioural modelling in turn and not only introduces the SysML syntax but, more importantly, introduces the main concepts that must be understood in order to model effectively. In fact, only the briefest of syntax will be looked at in this chapter – for a fuller description of the SysML syntax, see Chapter 4.

3.2 Structural modelling

Previously we have discussed the point that each system must be modelled in two different aspects. This section examines one of these aspects: the structural aspect of the model.

Figure 3.1 shows the five types of diagram that can be used to realize the structural aspect of the model: block definition diagrams, package diagrams, internal block definition diagrams, parametric diagrams and requirement diagrams.

In order to illustrate the concepts behind structural modelling, one of the five structural diagrams will be used to show some simple examples. The diagram chosen is the block definition diagram, as this forms the backbone of the Systems Modelling Language (SysML). In addition, the structural modelling principles that will be shown in this section with reference to block definition diagrams can be applied when using any type of structural model. For more in-depth discussion concerning block definition diagrams and structural modelling, see Reference 1.

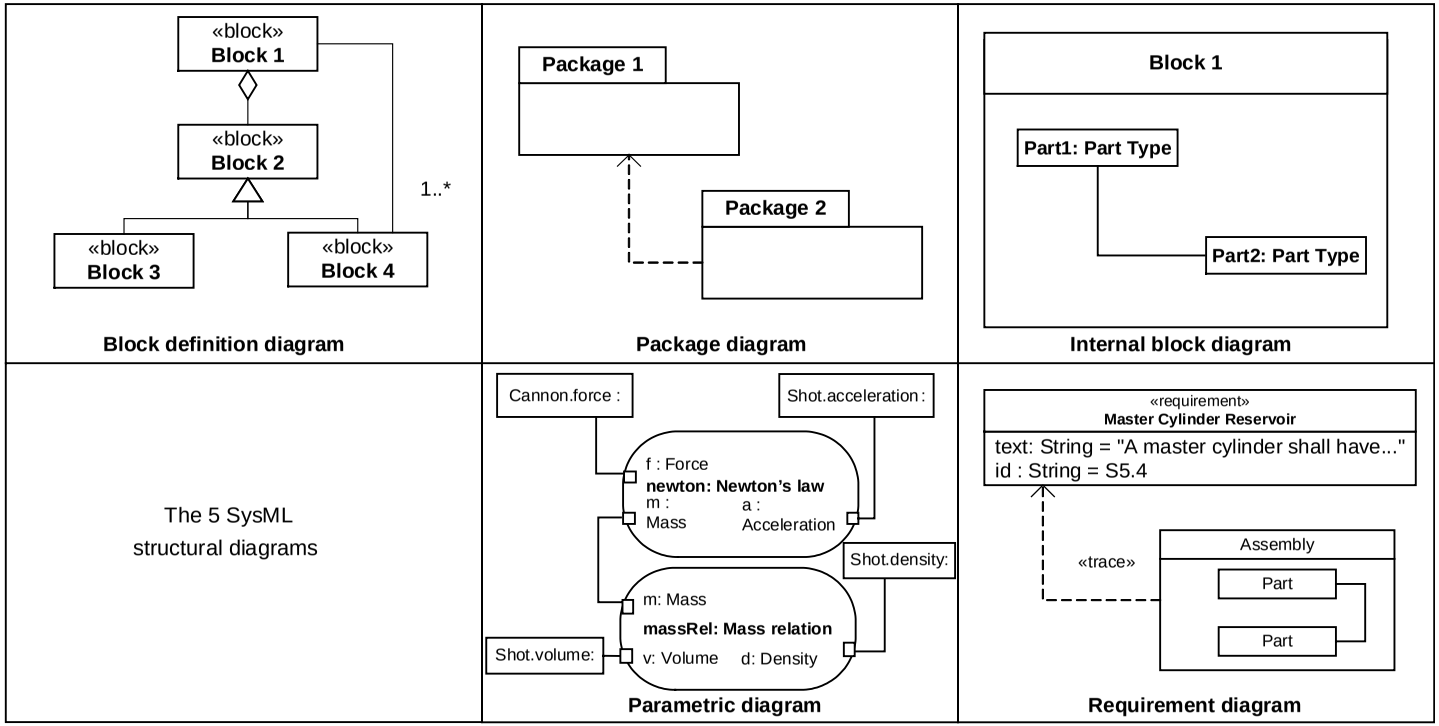

Figure 3.1 Types of structural diagram

3.3 Structural modelling using block definition diagrams

3.3.1 Modelling blocks and relationships

The block definition diagram is, arguably, the most widely used diagram in the SysML. Block definition diagrams have a long history and are present in some shape or form in all of the other methodologies that were mentioned previously in this part of the book. The block definition diagram is also the richest diagram in terms of the amount of syntax available to the modeller. As with all SysML diagrams, it is not necessary to use every piece of syntax, as experience has shown that 80 per cent of any modelling task can be achieved by using approximately 20 per cent of block definition diagram syntax. The simplest (and wisest) approach is to learn the bare minimum syntax and then to learn more as and when circumstances dictate.

3.3.2 Basic modelling

There are two basic elements that make up a block definition diagram, which are the 'block' and the 'relationship' – and, at a very simple level, that is it! Clearly, there are many ways to expand on these basic elements, but, provided they are understood clearly and simply, the rest of the syntax follows on naturally and is very intuitive.

A 'block' represents a type of 'thing' that exists in the real world and, hence, should have a very close connection to reality. Blocks are almost always given names that are nouns, since nouns are 'things' and so are blocks. This may seem a trivial point, but it can form a very powerful heuristic when assessing and analysing systems, as it can be an indicator of whether something may appear as a block on a model.

The second element in a block definition diagram is a relationship that relates together one or more blocks. Relationships should have names that form sentences when read together with their associated blocks. Remember that the SysML is a language and should thus be able to be 'read' as one would read any language. If a diagram is difficult to read, it is a fairly safe bet that it is not a very clear diagram and should perhaps be 'rewritten' so that it can be read more clearly. Reading a good SysML diagram should not involve effort or trying, in the way that any *sentence* should not be difficult to read.

Now that the basics have been covered, it is time to look at example block definition diagrams. It should be pointed out that, from this point, all diagrams that are shown will be SysML diagrams. In addition, although some of the diagrams may seem trivial, they are all legitimate and correct diagrams and convey some meaning – which is the whole point of the SysML!

Figure 3.2 shows two very simple blocks. Blocks are represented graphically by rectangles in the SysML and each must have a name, which is written inside the rectangle. In order to understand the diagram, it is important to read the symbols. The diagram here shows that two blocks exist: 'Block 1' and 'Block 2'. This is one of the

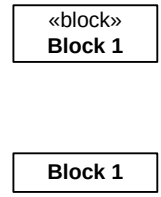

Figure 3.2 Representing blocks

first SysML diagrams in the book and, if you can read this, then you are well on the way to understanding the SysML. The upper block has the word 'block' in chevrons ('≪block≫') in it, which signifies that this block is stereotyped. As every block in a block definition diagram contains this same adornment, it is usually left off the diagram, as shown in the lower block. For the purposes of this book, the ≪block≫ stereotype will be omitted from most diagrams.

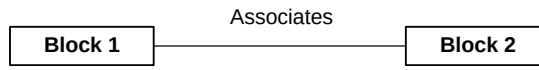

Figure 3.3 Representing a relationship

Figure 3.3 shows how to represent a relationship between two blocks. This particular relationship is known as an *association* and is simply a general type of relationship that relates one or more blocks. The association is represented by a line that joins two blocks, with the association name written somewhere on the line. This diagram reads: two blocks exist: 'Block 1' and 'Block 2' and 'Block 1' associates 'Block 2'.

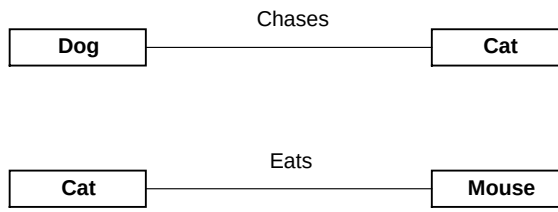

Figure 3.4 Examples of blocks and associations

Figure 3.4 shows two more examples that are based on real life. The top part of the diagram reads: there are two blocks: 'Dog' and 'Cat' where 'Dog' chases 'Cat'. Likewise, the lower part of the diagram reads: there are two blocks: 'Cat' and 'Mouse' where 'Cat' eats 'Mouse'.

This illustrates a very important point concerning blocks, as blocks are conceptual and do not actually exist in the real world. There is no such thing as 'Cat', but there do exist many examples of 'Cat'. A block represents a grouping of things that look and behave in the same way, as, at one level, all examples of 'Cat' will have a common set of features and behaviours that may be represented by the block 'Cat'. What this block is really representing is the blueprint of 'Cat', or the essence of 'Cat'.

Another important point illustrated here is that every diagram, no matter how simple, has the potential to contain a degree of ambiguity. Figure 3.4 is actually ambiguous, as it could be read in one of two ways, depending on the direction in which the association is read. Take, for example, the top part of the diagram: who is to say that the diagram is to be read 'Dog' chases 'Cat' rather than 'Cat' chases 'Dog', as it is possible for both cases to be true. It should be pointed out that for this particular example the world that is being modelled is a stereotypical world where dogs always chase cats, and not the other way around. Therefore, there is some ambiguity as the diagram must be read in only one direction for it to be true; thus, a mechanism is required to indicate direction.

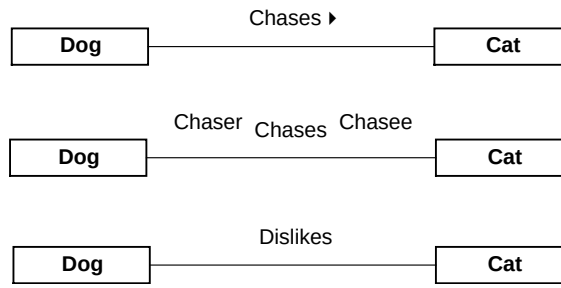

Figure 3.5 Showing direction

The simplest way to show direction is to place a direction marker on the association that will dictate which way the line should be read, as shown in the top part of Figure 3.5. The diagram now reads 'Dog' chases 'Cat' and definitely not 'Cat' chases 'Dog' and is thus less ambiguous than Figure 3.4.

The second way to show direction is to define a 'role' on each end of the association, as shown in the middle part of Figure 3.5. In this case, the two roles that have been defined are 'chaser' and 'chasee', which again eliminates the ambiguity that existed in Figure 3.4.

The lower part of Figure 3.5 introduces a new association called 'dislikes'. This time, however, the lack of direction is intentional, as both statements of 'Dog' dislikes 'Cat' and 'Cat' dislikes 'Dog' are equally true. Therefore, when no direction is indicated, it is assumed that the association can be read in both directions or, to put it another way, the association is said to be *bidirectional*.

Another reason why the diagram may be misunderstood is that there is no concept of the number of cats and dogs involved in the chasing of the previous diagrams. Expressing this numbering is known as *multiplicity*, which is illustrated in Figure 3.6.

Figure 3.6 Showing numbers using multiplicity

The top part of Figure 3.6 shows that each 'Dog' chases one or more 'Cat'. If no number is indicated, as in the case of the 'Dog' end of the association, it is assumed that the number is 'one'. Although the number is one, it does not necessarily indicate that there is only one dog, but rather that the association applies for each dog. The multiplicity at the other end of the 'chases' association states '1..*', which means 'one or more' or somewhere between one and many. Therefore, the association shows that each 'Dog' chases one or more 'Cat', and that each 'Cat' is chased by only one 'Dog'.

The lower part of the diagram shows a case where the multiplicity has been changed, which changes the entire meaning of the model. In this case, the diagram is read as: one or more 'Dog' chases one or more 'Cat'. This could mean that a single dog chases a single cat, a single dog chases any number or a herd of cats, or that an entire pack of dogs is chasing a herd of cats.

Which of the two examples of multiplicity is correct? The answer will depend on the application that is being modelled and it is up to the modeller to decide which is the more accurate of the two.

It is important to note that, if no multiplicity has been indicated, then the association end carries a default value of '1'. However, this can lead to some problems, as in the case when the number is omitted by accident, rather than design, which leaves the multiplicity unspecified. It is suggested, therefore, in the SysML specification that the number '1' should be included in all diagrams to show that the lack of number is not an omission. The only real argument against this is that it can render some diagrams far less readable, hence the decision on whether or not to include the number 1s lies with the modeller.

3.3.3 Adding more detail to blocks

So far, blocks and relationships have been introduced, but the amount of detailed information for each block is very low. After all, it was said that the block 'Cat' represented all cats that looked and behaved in the same way, but it is not defined anywhere how a cat looks or behaves. This section examines how to add this information to a block by using 'properties' and 'operations'.

Consider again the block of 'Cat', but now think about what general properties the cat possesses. It is very important to limit the number of general properties that

are identified to only those that are relevant, as it is very easy to get carried away and overdefine the amount of detail for a block.

For this example, suppose that we wish to represent the features ‘age’, ‘weight’, ‘colour’ and ‘favourite food’ on the block ‘Cat’. These features are represented on the block as ‘properties’ – one for each feature (Figure 3.7).

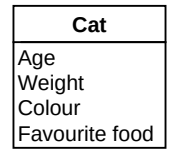

Figure 3.7 Properties of the block ‘Cat’

Properties are written in a box below the block name box. When modelling, it is possible to add more detail at this point, such as the visibility of the property, type, default values and so forth. As properties represent features of a block, they are usually represented by nouns and they must also be able to take on different values. For example, ‘colour’ is a valid property, whereas ‘red’ would not be in most cases, as ‘red’ would represent an actual value of a property rather than a property itself. It is possible for ‘red’ to be a property, but this would mean that the property would have a Boolean type (true or false) to describe a situation where we would be interested only in red cats and not any other type.

Properties thus describe how to represent features of a block, or to show what it looks like, but they do not describe what the block does, which is represented using *operations*. Operations show what a block does, rather than what it looks like, and are thus usually represented by verbs. Verbs, as any schoolchild will be able to tell you, represent ‘doing’ words, and operations describe what a block ‘does’. In the case of the block ‘Cat’ we have identified three things that the cat does, which are ‘eat’, ‘sleep’ and ‘run’ (Figure 3.8).

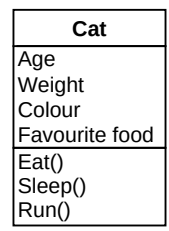

Figure 3.8 Operations of the block ‘Cat’

Operations are represented in the SysML by adding another rectangle below the property rectangle and writing the operation names within it. When modelling software, operations may be defined further by adding extra detail – for example, arguments, return values, visibility.

The block 'Cat' is now fully defined for our purposes and the same exercise may be carried out on any other blocks in the diagram in order to populate the model fully. It should also be pointed out that the blocks may be left at a high level with no properties or operations. As with everything in the SysML, use only as much detail as is necessary, rather than as much as is possible.

3.3.4 Adding more detail to relationships

The previous subsection showed how to add more detail to blocks, while this one shows how to add more detail to relationships, by defining some special types that are commonly encountered in modelling. There are four types of relationship that will be discussed here: 'association', 'aggregation', 'specialization' and 'dependency'. Many types of relationship exist, but these three represent the majority of the most common uses of relationships.

Associations have already been introduced and shown to be a very general type of relationship that relate together one or more block. Therefore, they will not be discussed in any further detail, so the next three subsections cover each of the other special types of relationship.

3.3.4.1 Aggregation and composition

The second type of relationship is a special type of association that allows assemblies and structures to be modelled and is known as *aggregation*.

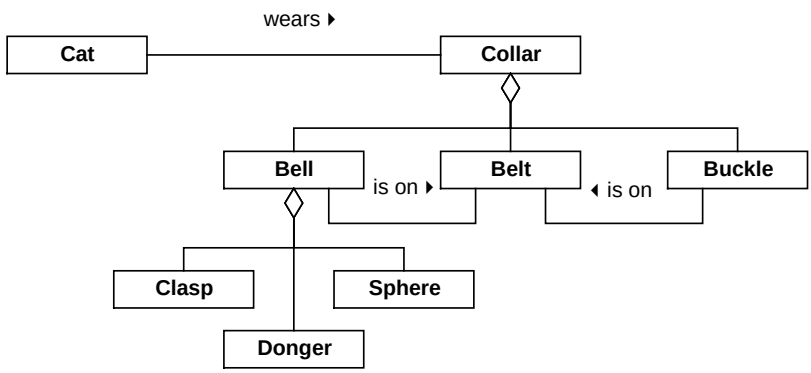

Figure 3.9 Example of aggregation

Figure 3.9 provides an example of aggregation. Aggregation is shown graphically in the SysML by a diamond or rhombus shape and, when reading the diagram, is read by saying 'is made up of'. Starting from the top of the diagram, the model is read as: 'Cat' wears 'Collar'. The direction is indicated with the small arrow and there is a one-on-one relationship between the two blocks. The multiplicity here is implied to be one to one as there is no indication. This makes sense as it is very unlikely that one cat would wear more than a single collar (this would be showing off) and almost

impossible to imagine that more than one cat would share a single collar (this would be cruel)!

The ‘Collar’ is made up of (the aggregation symbol) a ‘Bell’, a ‘Belt’ and a ‘Buckle’. The ‘Bell’ is on the ‘Belt’ and the ‘Buckle’ is on the ‘Belt’.

The ‘Bell’ is made up of (the aggregation symbol) a ‘Clasp’, a ‘Donger’ and a ‘Sphere’.

This is the basic structure of the bell and allows levels of abstraction of detail to be shown on a model.

There is also a second special type of association that shows an aggregation-style relationship, known as *composition*. The difference between composition and aggregation is subtle but very important and can convey much meaning. The simplest way to show this difference is to consider an example, as shown in Figure 3.10.

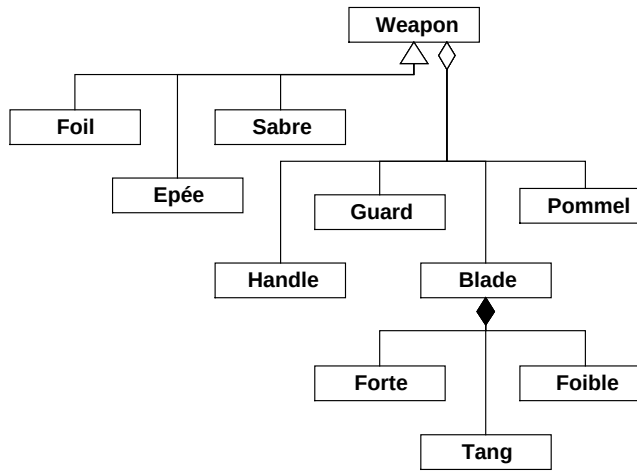

Figure 3.10 Example of the difference between composition and aggregation

The model in Figure 3.10 represents the structure of the types of weapon that may be used in the sport of fencing (as opposed to putting up fences or selling stolen goods). From the model, there are three types of ‘Weapon’: ‘Foil’, ‘Epée’ and ‘Sabre’ (this is a ‘type of’ or ‘specialization’ relationship, which will be discussed in a subsequent section). Each weapon is made up of a ‘Handle’, a ‘Pommel’, a ‘Blade’ and a ‘Guard’. The ‘Blade’ is made up of a ‘Forte’, a ‘Foible’ and a ‘Tang’. There is a clear visual difference here, as the aggregation symbol under ‘Blade’ is filled in, rather than empty, as in the case of the aggregation under ‘Weapon’. The semantic difference, however, is that an aggregation is made up of component parts that may exist in their own right. It is possible to buy or make any of the components under the block ‘Weapon’, as they are assembled into the completed block ‘Weapon’. The block ‘Blade’, however, has three components that cannot exist independently of the block ‘Blade’. This is because a fencing blade is a single piece of steel that is composed of three distinct sections. For example, there is no such thing as a ‘Foible’, since it is

an inherent part of the 'Blade' rather than being an independent part in its own right. Of course, any of these blocks may have their own properties and/or operations, even when used as part of the composition relationship.

3.3.4.2 Specialization and generalization

The third special type of relationship is known as *specialization* and *generalization*, depending on which way the relationship is read. 'Specialization' refers to the case when a block is being made more special or is being refined in some way. Specialization may be read as 'has types' whenever its symbol, a small triangle, is encountered on a model. If the relationship is read the other way around, then the triangle symbol is read as 'is a type of', which is a generalization. Therefore, read one way the block becomes more special (specialization) and read the other way, the block becomes more general (generalization).

Specialization is used to show 'child' blocks, sometimes referred to as 'sub-blocks', of a 'parent' block. Child blocks inherit their appearance and behaviour from their parent blocks, but will be different in some way in order to make them special. In SysML terms, this means that a child block will inherit any properties and operations that its parent block has, but may have additional properties or operations that make the child block special.

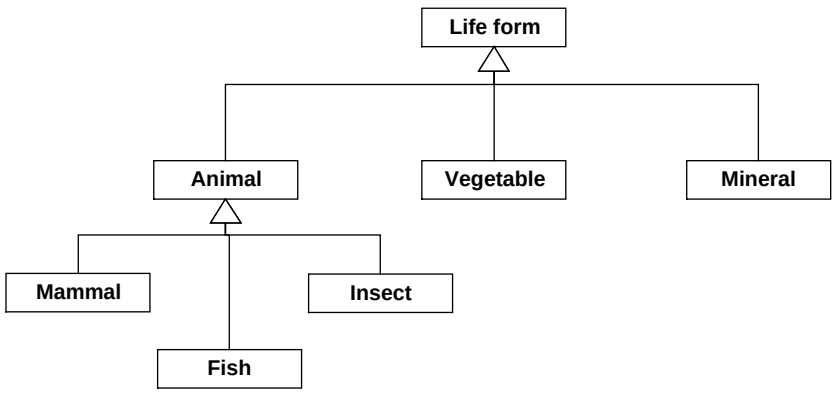

Figure 3.11 Life-form hierarchy

As an example of this, consider Figure 3.11, which shows different types of life known to man. The top block is called 'Life form' and has three child blocks: 'Animal', 'Vegetable' and 'Mineral', which makes 'Life form' the parent block. Going down one level, it can be seen that 'Animal' has three child blocks: 'Mammal', 'Fish' and 'Insect'. Notice now how 'Animal' is the parent block to its three child blocks, while still being a child block of 'Life form'.

The diagram may be read in two ways:

- from the bottom up: 'Mammal', 'Fish' and 'Insect' are types of 'Animal' – 'Animal', 'Vegetable' and 'Mineral' are all types of 'Life form';

- from the top down: ‘Life form’ has three types: ‘Animal’, ‘Vegetable’ and ‘Mineral’ – the block ‘Animal’ has three types: ‘Mammal’, ‘Fish’ and ‘Insect’.

This has explained how to read the diagrams, but has not covered one of the most important concepts associated with generalization and specialization, which is the concept of inheritance, best illustrated by considering Figure 3.12.

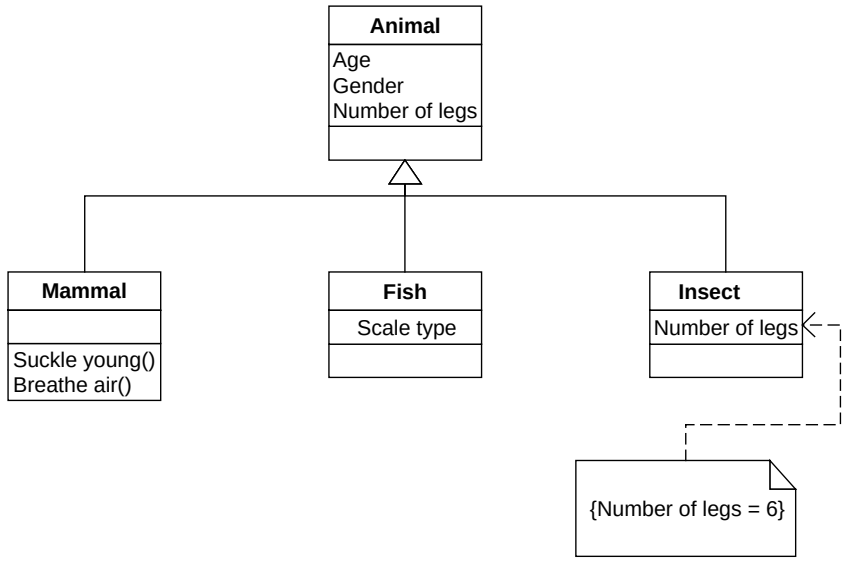

Figure 3.12 Example of inheritance

Figure 3.12 shows an expanded version of Figure 3.11 by adding some properties and operations to the blocks. It can be seen that the block ‘Animal’ has three identifiable properties: ‘age’, ‘gender’ and ‘number of legs’. These properties will apply to all types of animal and will therefore be inherited by their child blocks. That is to say that any child blocks will automatically have the same three properties. This inheritance also applies to the behaviour of the parent block. Therefore, if it were the case that some operations had been defined for the block ‘Animal’, these would also be inherited by each of the child blocks. What makes the child block different or special and therefore an independent block in its own right is the addition of extra properties and operations or constraints on existing properties. Let us consider an example of each one of these cases.

The block ‘Mammal’ has inherited the three properties from its parent block. Inherited properties (properties and operations) are not usually shown on child blocks. In addition, it has had an extra two operations identified, which the parent block does not possess: ‘breathe air’ and ‘suckle young’. This is behaviour that all *mammals* possess, whereas not all *animals* will, or, to put it another way, it is why mammals are special compared with animals generally.

The block 'Fish' has inherited the three properties from its parent block (which are not shown), but has had an extra property added that its parent block will not possess: 'scale type'. This makes the child block more specialized than the parent block.

The block 'Insect' has no extra properties or operations but has a constraint on one of its property values. The property 'number of legs' is always equal to six, as this is in the nature of insects. The same could be applied to 'mammal' to some extent as 'number of legs' would always be between zero (in the case of whales and dolphins) and four, while 'number of legs' for 'fish' would always be zero. Such a limitation is known in the SysML as a 'constraint', which is one of the standard extension mechanisms for the SysML.

From a modelling point of view, it may be argued that the property 'number of legs' should not be present in the block 'Animal', since it is not applicable to fish. This is fine and there is nothing inherently wrong with either model except to say that it is important to pick the most suitable model for the application at hand. Remember that there are many correct solutions to any problem, and thus people's interpretation of information may differ. By the same token, it would also be possible to define two child blocks of 'animal' called 'male' and 'female', which would do away with the need for the property 'gender' as shown in Figure 3.13.

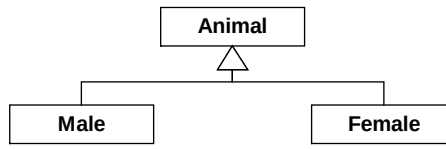

Figure 3.13 Another way to model gender

The model in Figure 3.13 shows another approach to modelling the gender of an animal. Which approach is the better of the two, the one shown in 3.12 or the one shown in 3.13? Again, it is necessary to pick the most appropriate visual representation of the information and one that you, as the modeller, are comfortable with.

As a final example, let us revisit our old friend the cat who is undeniably a life form, an animal and also a mammal. This gives three layers of inheritance, so that the block 'Cat' may now be represented as in Figure 3.14.

The model in Figure 3.14 shows an example of the cat that is now applied to the life-form hierarchy. Here, it can be seen that 'Cat' is a type of 'Mammal' and therefore inherits all its features and behaviour from its parent block. The block 'Cat' shown on the right-hand side of the diagram shows all the inherited properties and operations from its parent blocks. Note the inclusion of the three properties that were inherited from 'Animal', which, although the block is not included on this particular diagram, still influences the child blocks, no matter how far removed.

3.3.4.3 Dependencies

The final type of relationship that will be discussed is that of the 'dependency'. A dependency is used to show that one block is dependent on another. This means that

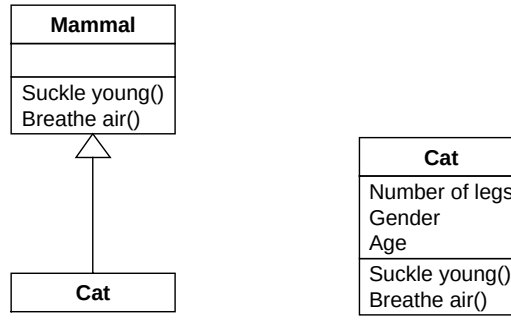

Figure 3.14 A cat's inheritance

a change in one block may result in a change in its dependent block. A dependency is represented graphically by a dashed line with an arrow *on the line end*. For example, consider the simple diagram of Figure 3.15.

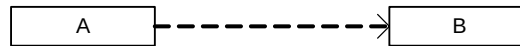

Figure 3.15 A simple dependency

In this example, any change to block 'B' will result in a change to block 'A'.

By their very nature, dependencies are quite weak relationships and really need to be further adorned using stereotypes to add any real meaning. Many diagrams use the dependency in conjunction with stereotypes, such as the use case diagram and the requirements diagram.

Perhaps the most widely used application of dependencies in the UML was to show the concepts of an 'instance' of a class.

3.3.5 A note on instances

As is stated on numerous occasions throughout this book, one baffling decision that was made when SysML was specified was to omit the concept of instances from the SysML notation. An 'instance' is a real-life example of a block and allows actual, real-life entities to be represented. For example, using instances, it is possible to represent a real-life cat (in this case, one of the author's). When instantiating the block 'Cat', the instance is given an identifier and its property values are filled in. In this case, it would be a rather bad-tempered old creature named 'Scumit' with properties of: 'number of legs = 4', 'gender = female' and 'age = 14 (+/–2)' years. Using the SysML notation, there is no mechanism for representing instances, which is a serious shortcoming of the language. Ironically, the SysML does use 'objects' in some diagrams that are defined as instances of classes in UML, which makes the whole matter of no instances even more confusing.

This is discussed further in Chapter 4.

3.3.6 Other structural diagrams

This chapter is intended to introduce structural modelling as a concept, but has so far considered only one of the five structural diagrams: the block definition diagram. However, the concepts introduced here, that of things and the relationships between them, applies to each of the structural diagrams. Therefore, if you can understand the concepts used here, it is simply a matter of learning some syntax and some example applications for the other types of diagram. Indeed, this is achieved in Chapter 4, where each of the nine SysML diagrams is described in terms of its syntax and its typical uses. The important thing to realize about all five types of structural diagram is that they all represent the 'what' of a system. These other four diagrams are the package diagram, internal block definition diagram, parametric diagram and requirement diagram.

3.3.7 Conclusion

In order to conclude this section and in order to see if structural modelling really works, a summary of this section is presented in Figure 3.16.

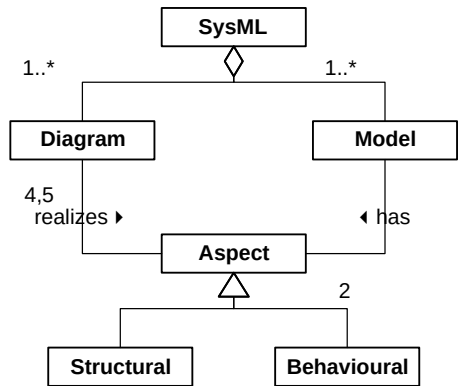

Figure 3.16 Summary of SysML models and their associated diagrams

The diagram in Figure 3.16 shows a summary of the SysML models and how they may be realized using the various types of diagram. The 'SysML' is made up of one or more 'Diagram' and one or more 'Model'. Each 'Model' has two 'Aspect', whereas four or five 'Diagram' realize each 'Aspect'. There are two types of 'Aspect' – 'Structural' and 'Behavioural'.

The actual types of diagram are not shown here, but are shown in Figure 3.17, which introduces a package that groups together the various types of structural model.

The diagram in Figure 3.17 shows the various types of structural diagram. It can be seen from the model that the package 'Structural' (a simple grouping) contains 'Block definition diagram', 'Internal block definition diagram', 'Package diagram', 'Requirements diagram' and 'Parametric diagram'.

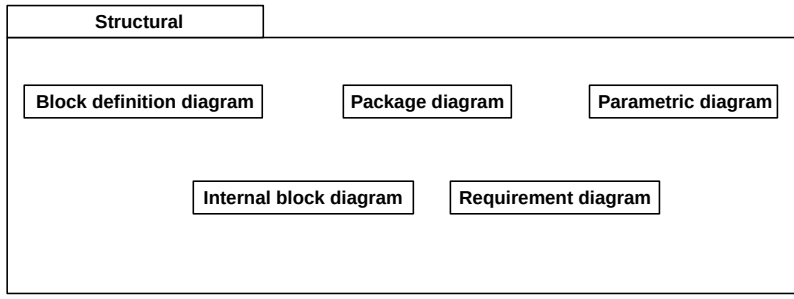

Figure 3.17 Types of structural diagram

The actual elements that made up the block definition diagram can also be modelled using a block definition diagram. This is a concept known as the *meta-model*, which will be used extensively in Chapter 4.

Remember that the SysML is a language and, therefore, should be able to be read and understood just like any other language. Therefore, the model in Figure 3.18 may be read by reading block name, then any association that is joined to it, and then another block name.

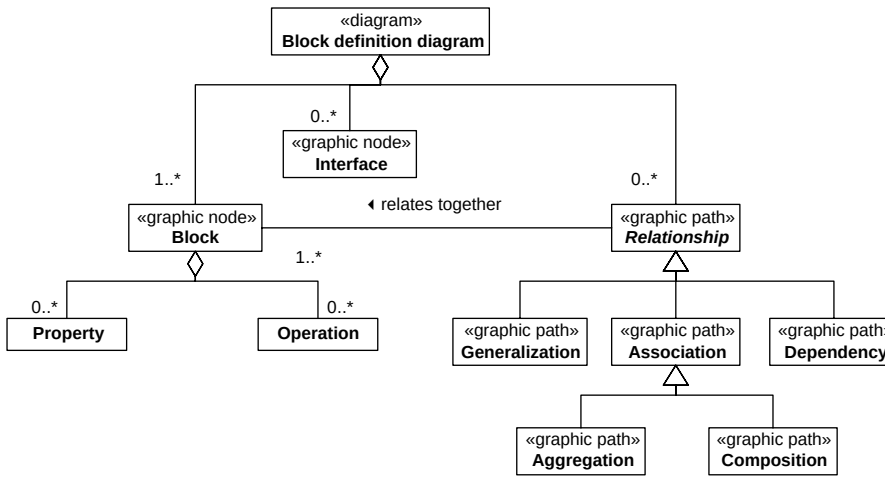

Figure 3.18 Meta-model for a block definition diagram

Therefore, reading the diagram: a ‘Block definition diagram’ is made up of one or more ‘Block’, zero or more ‘Interface’ and zero or more ‘Relationship’. It is possible for a block definition diagram to be made up of just a single block with no relationships at all; however, it is not possible to have a block definition diagram that is made up of just a relationship with no blocks. Therefore, the multiplicity on ‘Block’ is one or more, whereas the multiplicity on ‘Relationship’ is zero or more.

Each ‘Relationship’ relates together one or more ‘Block’. Notice that the word *each* is used here, which means that for every *single* ‘Relationship’ there are one or more ‘Block’. It is also interesting to note that the multiplicity on the ‘Block’ side of the association is one or more, as it is possible for a ‘Relationship’ to relate together one ‘Block’ – that is to say that a ‘Block’ may be related to itself.

Each ‘Block’ is made up of zero or more ‘Property’ and zero or more ‘Operation’. This shows how blocks may be further described using properties and operations, or that they may be left with neither.

There are three main types of ‘Relationship’.

- ‘Association’, which defines a simple association between one or more ‘Block’. There are also two specializations of ‘Association’: a special type of ‘Association’ known as ‘Aggregation’ and one known as ‘Composition’, which show ‘is made up of’ and ‘is composed of’ associations respectively.
- ‘Generalization’, which shows a ‘has types’ relationship that is used to show parent and child blocks.
- ‘Dependency’, which shows that one block is dependent on another. To put this in simple terms, any block that is dependent on another may have to change if the other block changes.

This meta-model concept may be extended to include all the other types of structural diagram and, indeed, this is exactly the approach adopted in Chapter 4 to introduce all the other SysML diagram types.

3.4 Behavioural modelling

The previous section introduced structural modelling, one of the two basic types of modelling using the SysML, by choosing one type of structural diagram and using it to explain basic principles that may then be applied to any sort of structural modelling. This chapter takes the same approach, but with behavioural modelling.

Behavioural models may be realized using four types of SysML diagram, as shown in Figure 3.19. This shows the four types of behavioural SysML diagram, which are: use case diagrams, state machine diagrams, activity diagrams and sequence diagrams.

It was stated in the previous section that structural modelling defined the ‘what’ of a system: what it looks like, what it does (but not *how*) and what the relationships are. If structural modelling tells us ‘what’, then behavioural modelling tells us ‘how’. This ‘how’ is described by modelling interactions within a system. These interactions may be modelled at many levels of abstraction; different types of behavioural diagram allow the system to be modelled at different levels of abstraction. These levels of abstraction may be categorized as follows.

Interactions may be modelled between objects or between subsystems. In UML, such models are realized using the four types of interaction diagram: communication, timing, interaction overview and sequence diagram. In the SysML, however, three of these four diagrams were deemed as unnecessary and only the sequence diagram

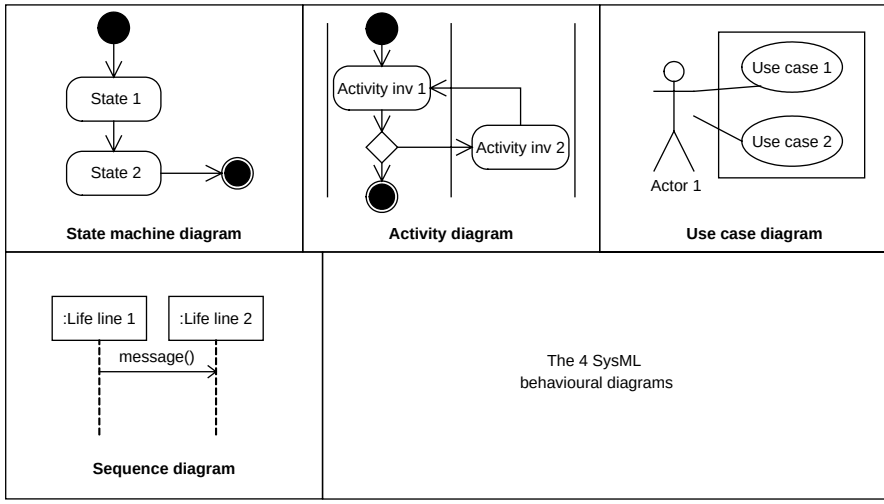

Figure 3.19 The four diagrams for realizing the behavioural aspect of the model

remains. This may be viewed as cause for concern or utter insanity, depending on the view of the reader.

Interactions may also be modelled at a slightly lower level of abstraction by considering the interactions within a block or its associated objects. Modelling the behaviour over the lifetime of a block, or an object, is achieved by using state machine diagrams. State machine diagrams are concerned with one, and only one, block and its objects, although it is possible to create more than one state machine diagram per block. (Please note the dilemma with instances and objects here!)

The lowest level of abstraction for modelling interactions is concerned with the interactions within an operation or, in computer terms, at the algorithmic level. Interactions at the algorithmic level are modelled using activity diagrams. Activity diagrams may also be used to model interactions at a higher level of abstraction, which will be discussed in greater detail later in this book.

The remaining level of abstraction is at the highest possible level, which is the context level of the project, where high-level functionality of the system and its interaction with the outside world is modelled. These high-level interactions are realized using use case diagrams.

The diagram chosen to illustrate behavioural modelling is the state machine diagram. There are a number of reasons for choosing this behavioural diagram over the other four.

State machine diagrams are one of the most widely used diagrams in previous modelling languages and, as such, people tend to have seen something similar to state machine diagrams, if not an actual state machine diagram. This makes the process of learning the syntax easier.

State machine diagrams have a very strong relationship with block definition diagrams, which were discussed earlier in this chapter.

It will be seen later in this chapter that behavioural modelling using state machine diagrams will lead directly into other types of behavioural modelling diagrams. For more in-depth discussion concerning state machine diagrams and behavioural modelling, see Reference 1.

3.5 Behavioural modelling using state machine diagrams

3.5.1 Introduction

State machine diagrams are used to model behaviour over the lifetime of a block, or, to put it another way, they describe the behaviour of objects. Both of these statements are true and are used interchangeably in many books. Remember that objects don't actually exist in SysML, depending which part of the specification you read, but, in order for state machine diagrams to make any sense whatsoever, it is essential that they do exist! All objects are instances of blocks and thus use the block as a template for the creation of its instantiations, which includes any behaviour.

The most obvious question to put forward at this point is, 'Does every block need to have an associated state machine diagram?' The simple answer is no, since only blocks that exhibit some form of behaviour can possibly have their behaviour defined. Some blocks will not exhibit any sort of behaviour, such as data structures and database structures. The simple way to spot whether a block exhibits any behaviour is to see whether it has any operations: if a block has operations, it does something; if it does not have any operations, it does nothing. If a block does nothing, it is impossible to model the behaviour of it. Therefore, a simple rule of thumb is that any block that has one or more operations must have its behaviour defined using, for example, a state machine diagram.

3.5.2 Basic modelling

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 how to change between such states and events dictate which messages are passed on these transitions. Each of these elements will now be looked at in more detail, starting with the state, an example of which is shown in Figure 3.20.

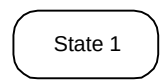

Figure 3.20 A SysML representation of a state

Figure 3.20 shows a very simple state, which is shown in the SysML by a box with rounded corners. This particular state has the name 'State 1' and this diagram should be read as: 'there is a single state, called "State 1" '. This shows what a state

looks like, but what exactly is a state? The following three points discuss the basics of a state.

- A state may describe situations in which the system satisfies a particular condition, in terms of its property values or events that have occurred. This may, for example, be 'loaded' or 'saved', so that it gives an indication as to something that has already happened. States that satisfy a particular condition tend to be used when an action-based approach is taken to creating state machine diagrams. This will be discussed in more detail in due course.
- A state may describe a situation in which the system performs a particular activity or set of actions, or, to put it another way, is actually doing something. States are assumed to take a finite amount of time, whereas transitions are assumed to take no time. There are two things that can be happening during such a state: one or more activities and/or one or more actions. Activities are non-atomic and, as such, can be interrupted, hence, they take up a certain amount of logical time. Actions, on the other hand, are atomic and, as such, cannot be interrupted, and, hence, take up zero logical time. Therefore, activities can appear only *inside* a state, whereas an action can exist either within a state or on a transition. Activities can be differentiated from actions inside states by the presence of the keyword 'do/', whereas action will have other keywords, including: 'entry/' and 'exit/'.
- A state may also describe a situation in which a system does nothing or is waiting for an event to occur. This is often the case with event-driven systems, such as Windows-style software, where, in fact, most of the time the system is sitting idle and is waiting for an event to occur.

There are different types of state in the SysML; however, the states used in state machine diagrams are known as *normal states*. This implies that complexity may exist within a state, such as: a number of things that may happen, certain actions that may take place on the entry or exit of the state, and so forth.

In order for the object to move from one state to another, a transition must be crossed. In order to cross a transition, some sort of event must occur. Figure 3.21 shows a simple example of how states and transitions are represented using the SysML.

From the model in Figure 3.21, it can be seen that two states exist: 'State 1' and 'State 2', represented by rounded boxes. There is a single transition that goes from

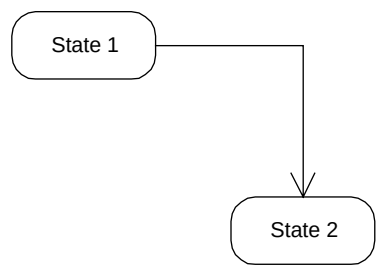

Figure 3.21 States and transitions

'State 1' to 'State 2', which is represented by a directed line that shows the direction of the transition. These transitions are unidirectional and, in the event of another transition being required going in the other direction, an entirely new transition is required – the original transition cannot be made bidirectional.

In order to cross a transition, which will make the object exit one state and enter another, an event must occur. This event may be something simple, such as the termination of an activity in a state (the state has finished what it is doing); then it leaves its present state, or may be more complex and involve receiving messages from another object in another part of the system. Event names are written on the transition lines inside special symbols, which will be introduced later in this section.

State machine diagrams will now be taken one step further by considering a simple example.

3.5.3 Behavioural modelling – a simple example

3.5.3.1 Introduction

In order to illustrate the use of state machine diagrams, a simple example will be used that will not only show all basic concepts associated with state machine diagrams, but will also provide a basis for further modelling and provide a consistent example that will be used throughout this section.

The simple example chosen is that of a game of chess, as it is a game with which most people are at least vaguely familiar, and thus most people will have some understanding of what the game is all about. In addition, chess is, on one level, very simple, which means that it can be modelled very simply, yet, on the other hand, it possesses a great deal of hidden complexity that should emerge as the example is taken further.

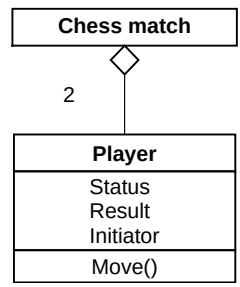

Figure 3.22 A simple block definition diagram for a game of chess

Figure 3.22 shows a block definition diagram that represents, at a very simple level, a game of chess. From the model, a 'Chess match' is made up of two 'Player'. Each 'Player' has properties – 'result', 'status' and 'initiator' – and a single operation 'move'. The property 'result' reflects the result of the game and may have values: 'this player win', other player win', 'draw' or 'undecided'. The property 'status' reflects the current status of the game and may take the values 'checkmate', 'stalemate' and

'game in progress'. Finally, the property 'initiator' represents the player who goes first.

3.5.3.2 Simple behaviour

It was stated previously that any block that exhibits behaviour must have an associated state machine diagram. Applying this rule reveals that, of the two blocks present in the block definition diagram, only the block 'Player' needs to have a state machine diagram. A very simple state machine diagram for a game of chess is shown in Figure 3.23, by defining the behaviour of the block 'Player'.

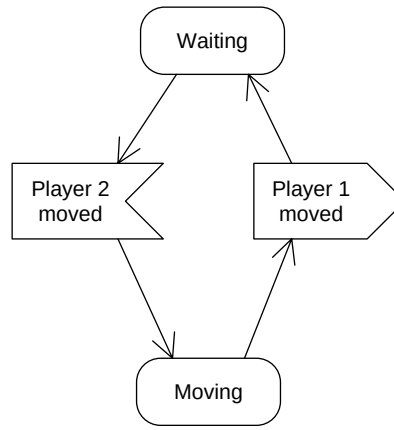

Figure 3.23 A simple state machine diagram for a game of chess

From the model, the object may be in one of two states: 'waiting' or 'moving'. In order to cross from 'waiting' to 'moving', the event 'player 2 moved' must have occurred. In order to cross from 'moving' to 'waiting', the event 'player 1 moved' must have occurred.

The two pentagons, one convex and one concave, represent 'send events' and 'receive events' respectively. A send event is an event that is broadcast outside the boundary of the state machine diagram or, to put it another way, is broadcast to other objects. A receive event is a complement of the send event in that it is accepted from outside the boundary of the state machine diagram.

At a very simple level, this is 'how' a game of chess is played, by modelling the behaviour of each player. However, the model is by no means complete as the chess game described here has no beginning or end and will thus go on for ever. Despite the fact that chess games may *seem* to go on for ever, there is an actual distinct start and end point for the game. This is modelled in the SysML by introducing 'start states' and 'end states'.

The next step, therefore, is to add more detail to this state machine diagram. It is interesting to note that this is actually how a state machine diagram (or any other SysML diagram, for that matter) is created. The diagram almost always starts off as a

simple collection of states and then evolves over time and, as more detail is added, so the model starts to get closer and closer to the reality that it is intended to represent.

3.5.3.3 Adding more detail

The next step is to add a beginning and an end for the state machine diagram, using start and end states. A start state describes what has happened before the object is created, and is shown visually by a filled-in circle. An end state, by comparison, shows the state of the object once the object has been destroyed, and is represented visually by a bull's-eye symbol.

Start and end states are treated just like other states in that they require transitions to take the object into another state, which will require appropriate events. The events would typically be creation- and destruction-type events that are responsible for the birth and subsequent demise of the object.

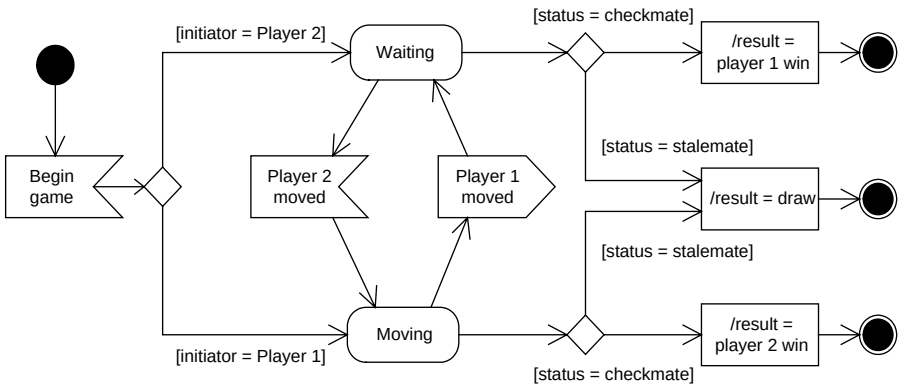

Figure 3.24 Expanded state machine diagram showing start and end states

Figure 3.24 shows the expanded state machine diagram that has start and end states along with appropriate events.

It can be seen from the model that there is a single start state and then a receive event is shown (the concave pentagon). A decision then occurs and, depending on which player goes first, the control of the state machine diagram will go to either the 'waiting state' or the 'moving' state. There are three different end states, depending on which player, if either, wins, or whether the game is a draw. The model is now becoming more realistic; its connection to reality is getting closer, but there is still room for ambiguity.

Notice also in this diagram that some conditions have been introduced that are defined on the outputs of the decision symbol (the diamond). These conditions should relate directly back to properties from its parent block, hence providing some basic consistency between a block and its associated state machine diagram.

We know from the block definition diagram that the block 'Player' does one thing, 'move', but we do not know in which state of the state machine diagram that this

operation is executed. As it happens, it is fairly obvious which state the operation, occurs in: 'moving'. Operations on a state machine diagram may appear as either 'activities' or 'actions', which will be discussed in more detail in due course. This may now be shown by adding the activity to its appropriate state by writing 'do/' in the state box and then adding the activity (operation from the block definition diagram) name, which is shown in Figure 3.25.

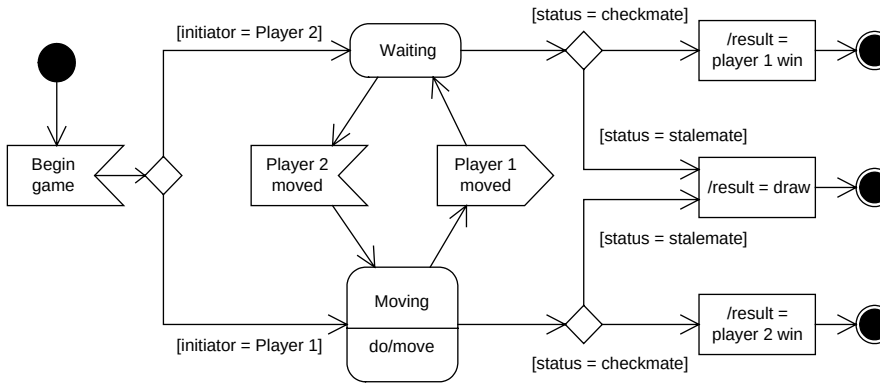

Figure 3.25 Expanded state machine diagram showing activity

The model is now getting even closer to reality; in fact, the model is evolving, which they always do! It is almost impossible to get a model right the first time, since they are living entities and will continue to evolve for as long as they exist. However, there is yet another problem with the state machine diagram, as, although it seems to work well for any situation in a chess game, it is impossible for the game to run! To illustrate this, consider what happens when we begin a game of chess.

3.5.4 Ensuring consistency

The first thing that will happen is that two instances of the block 'Player' need to be created so that we have the correct number of players. The behaviour of each player is described by the state machine diagram for the block 'Player'. For argument's sake, we shall name the two players 'Player 1' and 'Player 2' and see if the state machine diagram will hold up to the full game of chess. Figure 3.26 shows two identical state machine diagrams, one for each object, that are positioned side by side to make comparisons easier.

In order to begin a game of chess, an instance of the block 'Chess match' would be created, which would in turn create two instances of the block 'Player'. In this example, the object names 'Player 1' and 'Player 2' have been chosen. Let us now imagine that a game of chess has been started and that 'Player 1' is to begin. The event that will occur is 'player 1 start', which is present on both state machine diagrams. However, this will put both players straight into the 'moving' state, which will make

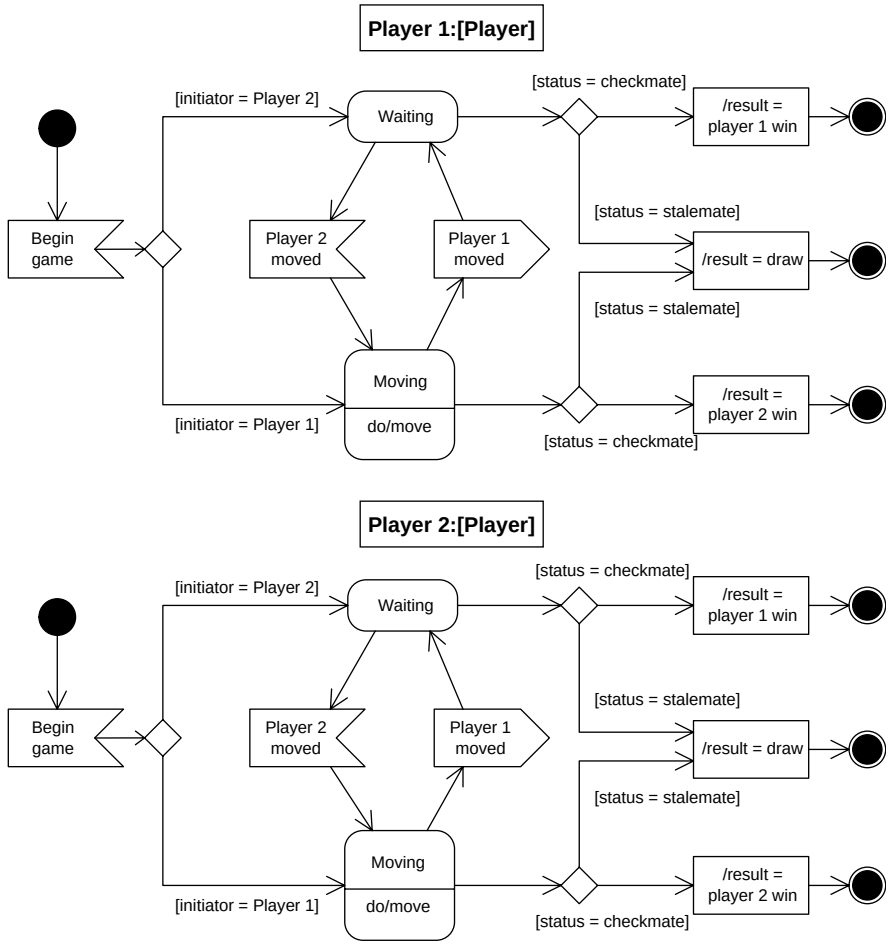

Figure 3.26 Side-by-side comparison of two state machine diagrams

the game of chess impossible to play, despite its being slightly more entertaining. This is because the events were named with reference to one player, rather than being generic so that they are applicable to any player. In order to make the game work, it is necessary to rename the events so that they are player-independent.

It is important to run through (by simulation or animation, if a suitable tool is available) a state machine diagram to check for consistency. In this case, the error was a simple, almost trivial, misnaming of an event. However, this trivial mistake will lead to the system actually failing!

This serves to illustrate a very important point that relates back to the section about modelling: the different levels of abstraction of the same model. The chess game was modelled only at the object level in terms of its behaviour, which, it is entirely possible, would have resulted in the block being implemented and even

tested successfully if treated in isolation and under test conditions. It would only be at the final system test level that this error would have come to light. It would be useful, therefore, if it were possible to model behaviour at a higher level, where the interactions between objects could be shown, as in Figure 3.27.

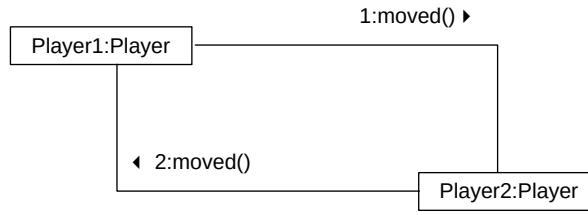

Figure 3.27 Wouldn't-it-be-nice model

Figure 3.27 shows the two objects from the original block definition diagram, but this time the events from the state machine diagrams have been shown. It is clear from this diagram that there would be a problem with the event names at a higher level of abstraction, which could lead to the problem being sorted out far earlier than the previous case. It would have been nice if we had drawn this diagram as part of our behavioural modelling of the game of chess. Luckily, such a diagram does exist in the UML and is known as a *communication diagram* – unfortunately, this diagram has been omitted from the SysML, as it is 'not needed' (although, ironically, the SysML specification *does* contain some communications diagrams).

3.5.5 Solving the inconsistency

There are many ways to solve the inconsistency problems that were highlighted in the previous section, two of which are presented here. The first solution is to make the generic state machine diagram correct, while the second is to change the block definition diagram to make the state machine diagrams correct.

3.5.5.1 Changing the state machine diagram

The first solution to the inconsistency problem that is presented here is to make the state machine diagram more generic, so that the events now match up. The state machine diagram for this solution is shown in Figure 3.28.

Figure 3.28 represents a correct solution to the chess model. Note that in this model the event names have been changed to make them more generic. The first events that have been changed are the events on the transitions from the two start states. Note that this time the names have been changed so that they are no longer specific to either player one or player two. The names are now relative to each instance, rather than specific, so the event names have changed from 'Player 1 begins' to 'this' (player begins) and from 'player 2 begins' to 'other' (player begins).

The other event names that have been changed are on the transitions between the two normal states. Previously, the names were object-specific, being, as they were,

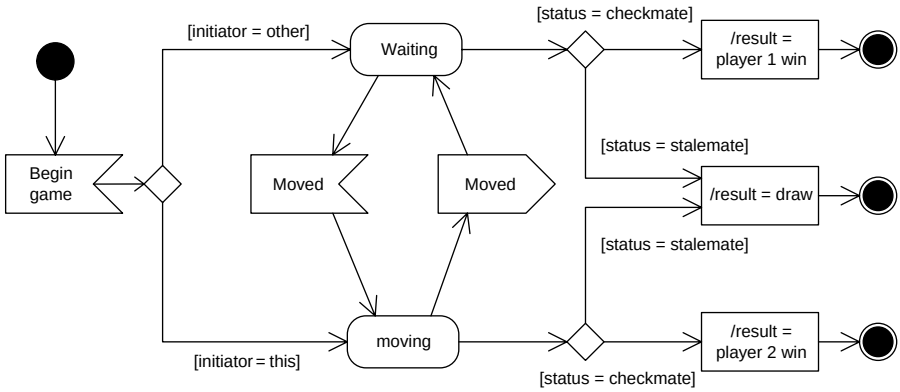

Figure 3.28 New state machine diagram with correct event names

'Player 1 moved' and 'Player 2 moved'. These names have now changed to a more generic 'moved' event, which will apply equally to both objects.

This is by no means the only solution to the problem and another possible solution is presented in the next section, where the block definition diagram is changed to make the state machine diagram correct, rather than changing the state machine diagram.

3.5.5.2 Changing the block definition diagram

The second solution to the consistency problem is to change the block definition diagram rather than the state machine diagram, as shown in Figure 3.29.

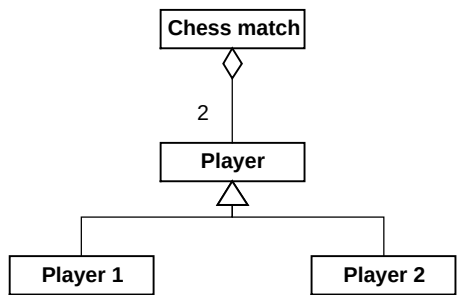

Figure 3.29 A modified block definition diagram

Figure 3.29 shows a modified block definition diagram in which two new sub-blocks of 'Player' have been added. This would mean that, rather than the block 'Player' being instantiated, one instance of each block 'Player 1' and 'Player 2' would be created. This has implications on the state machine diagrams, as the block definition diagram shown here would require a state machine diagram for both 'Player 1' and 'Player 2', rather than a single state machine diagram for 'Player'. This would also mean that the initial state machine diagram shown in Figure 3.25 would now be

correct for ‘Player 1’, but that a new state machine diagram would have to be created for the block ‘Player 2’.

Taking this idea a step further, it is also possible to make the two sub-blocks more specific as, in the game of chess, one player always controls white pieces and the other player controls only black pieces. This would have an impact on the block definition diagram again, as each subclass could now be named according to its colour. This is shown in Figure 3.30.

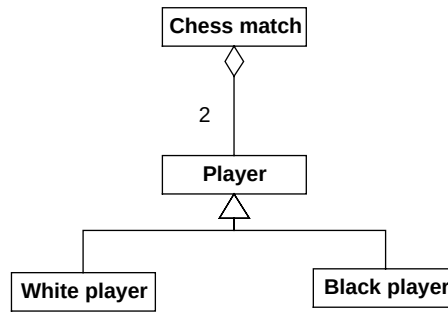

Figure 3.30 Further modification of the chess block definition diagram

Figure 3.30 shows a block definition diagram where the blocks have been named according to colour, rather than simply ‘Player 1’ and ‘Player 2’. This has even more effect on the state machine diagram, as, in the standard rules of chess, the white player always moves first.

3.5.6 Alternative state machine modelling

3.5.6.1 Actions and activities

The state machine diagrams presented so far have had an emphasis on the activities in the system where the main state in the state machine diagram had an explicit activity associated with it. This is slightly different from another approach that is often used in many texts, which is referred to here as an *action-based approach*. In order to appreciate the difference between the action-based and activity-based approaches, it is important to distinguish between actions and activities.

An activity describes an ongoing, non-atomic execution within a system. In simple terms, this means that an activity takes time and can be interrupted. An activity is also directly related to the operations on a block.

An action is an atomic execution within the system that results in either a change in state or the return of a value. In simple terms, this means that an action takes no time and cannot be interrupted.

Activities may be differentiated from actions as they will always appear inside a state (as what is known as an ‘internal transition’) and will use the keyword ‘do/’ as a prefix. Any other keyword used as a prefix (for example, ‘entry/’, ‘exit’) is assumed to be an action.

These two types of execution may change the way that a state machine diagram is created and the type of names that are chosen to define its state machine diagram.

3.5.6.2 Activity-based state machine diagrams

Activity-based state machine diagrams are created with an emphasis on the activities within states. Activities must be directly related back to the block that the state machine diagram is describing and provide the basis for a very simple, yet effective, consistency check between the block definition diagram and state machine diagram. As they take time to execute, activities may be present only on a state machine diagram within a state, as states are the only element on the state machine diagram that takes time – transitions are always assumed to take zero time.

The state names, when using the activity-based approach, tend to be verbs that describe what is going on during the state and thus tend to end with 'ing', such as the two states seen so far in this chapter: 'moving' and 'waiting'.

3.5.6.3 Action-based state machine diagrams

Action-based state machine diagrams are created with an emphasis on actions rather than activities. Actions are often not related directly back to blocks, although they should be, in order to ensure that consistency checks are performed between the two types of diagram. Actions are assumed to take zero time to execute and thus may be present either inside states or on transitions – transitions also take zero time. With an action-based approach, the actions tend to be written on transitions rather than inside states, which leads to a number of empty states. These states have names that reflect the values of the system at that point in time.

The simplest way to demonstrate the difference is to create an action-based state machine diagram for the chess example and then to compare the two different styles of state machine diagram.

Figure 3.31 shows an action-based state machine diagram where the emphasis is on the action '/activate' and '/deactivate' rather than on activities. Notice that the state names here reflect what has happened in the system, implying that the system is doing nothing while waiting for events.

3.5.6.4 Comparing the approaches

Both of the approaches shown here are valid and useful; however, there are a number of observations that can be made concerning the use of both approaches.

- The activity-based approach may lead to a more rigorous approach to modelling as the consistency checks with its associated block are far more obvious.
- The activity-based approach makes use of executions that take time and may be interrupted for whatever reason.
- The action-based approach is more applicable to event-based systems such as Windows-based software, where for much of the time the system is idle while waiting for things to happen.

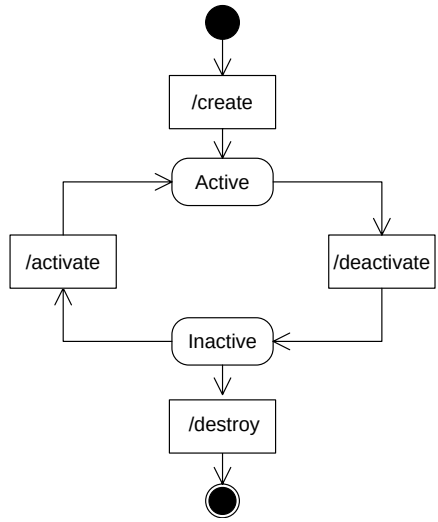

Figure 3.31 Action-based state machine diagram

- In terms of real-time systems, the action-based approach is less suitable then the activity-based approach, as the basic assumption of executions taking zero time goes against much real-time thinking.
- Action-based diagrams tend to be less complex than activity-based, as they tend to have fewer states.

So which approach is the better of the two? The answer is whichever is more appropriate for the application at hand.

3.5.7 Other behavioural diagrams

This chapter has concentrated mainly on state machine diagrams, but the principles that apply to state machine diagrams apply to the other types of behavioural model.

If you can have a good grasp of the models shown here and, just as importantly, understand how they must be checked to ensure consistency between different diagrams (and models), the other diagrams are relatively straightforward.

State machine diagrams are used to describe the behaviour of a block and, hence, its associated objects. They represent a medium level of abstraction.

Activity diagrams emphasize the actions within a system and are generally used at a very low level, or algorithmic level, of abstraction. Activity diagrams may also be used at a medium level of abstraction, but with an emphasis on actions within a system, rather than states. The main difference between an activity diagram and a state machine diagram is that a state machine diagram contains normal states (that may be complex), whereas an activity diagram will only contain activation instances. This is elaborated upon in Chapter 4.

Sequence diagrams model interactions at a high level of abstraction and can be used, among other purposes, to ensure that events passed between state machine diagrams are consistent. It should be pointed out that UML has three other diagrams that operate at the same level of abstraction as the sequence diagram, collectively known as 'interaction diagrams'. These diagrams are: the communication diagram, the interaction overview diagram and the timing diagram. As has been stated previously, these diagrams have been omitted from the SysML.

Use case diagrams represent the highest level of abstraction and show the interactions between the system and its external actors. These are strongly related to sequence diagrams, which may be used to model scenarios as instances of use cases.

All of the points raised above are examined in Chapter 4, where each diagram is looked at in turn; however, one point that should emerge here is that there is a relationship between types of behavioural diagram and they each possess a number of common features.

3.6 Identifying complexity through levels of abstraction

3.6.1 Introduction

One issue that has come up on a number of occasions in this book is that of complexity. Complexity is guilty of being one of the three evils of life and is one of the fundamental reasons why we need to model in the first place. In this section, two systems will be compared in terms of their complexity, which will be identified through looking at the system from different levels of abstraction. A number of modelling issues will arise from this example.

3.6.2 The systems

Imagine the situation where you are put in charge of two projects, neither of which you have any domain knowledge for, and you have to rely on descriptions of each project. This seems to be a prime application for modelling – there is a lack of understanding about the two systems, there is no indication of how complex each is and this information will need to be communicated to the project teams.

The two examples that have been chosen are chess, as seen in the current example, and also Monopoly. Before the modelling begins, ask yourself a simple question – which is the more complex, chess or Monopoly? Take a moment to consider this before reading on.

3.6.3 Structural view

An initial modelling exercise was carried out and two block definition diagrams were produced – one for chess and one for Monopoly.

The diagram in Figure 3.32 shows a simple block-definition-diagram representation of each of the two systems. It is now time to do a simple complexity comparison of the two diagrams – which of the two is more complex? There is really not much to choose between the two, but, if forced to choose one, then Monopoly could be considered slightly more complex than chess, since it has a higher multiplicity on the

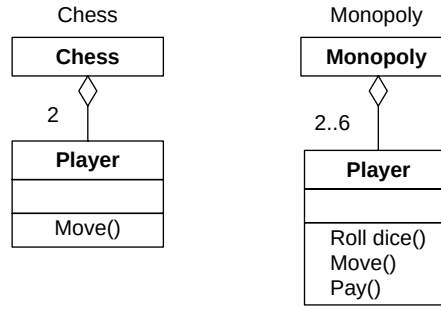

Figure 3.32 Comparing complexity – block definition diagrams

block 'Player' (hence more players in the game) and it has three operations compared with one in the chess example.

As has been pointed out a number of times in this book, it is essential to look at both structural and behavioural aspects of the model, so the next step is to model some behaviour. As each of the blocks has at least one operation, a state machine diagram would seem to be the logical choice for the next diagram.

3.6.4 Behavioural views

Ultimately, there will be several behavioural views generated for this example, but the start point is the state machine diagram.

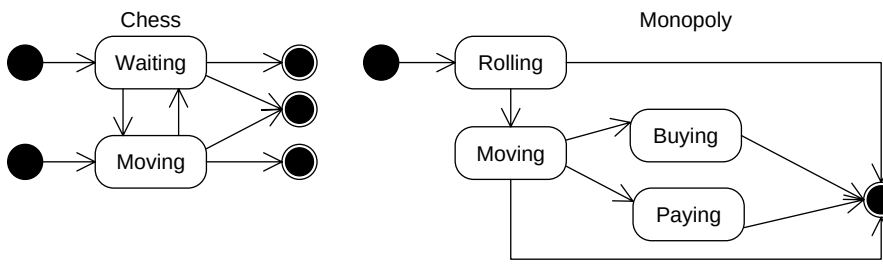

Figure 3.33 Comparing complexity – state machine diagrams

The diagram in Figure 3.33 shows a simple state machine diagram for the block 'Player' for each of the two systems. Again, ask yourself which of the two is the more complex. In this case, there is not much to choose between the two diagrams again. The state machine diagram for chess has fewer states than the Monopoly one, but it also has more interactions between the states. Most people would choose the case of the chess example to be more complex than the Monopoly, which leaves us with a dilemma. When comparing the block definition diagrams of the two examples, it was decided that Monopoly was slightly more complex than chess, but now, when looking at the state machine diagrams, it was decided that the chess example is slightly more complex than Monopoly. This prompts the obvious question: which is, overall, more

complex? Clearly, this is not an easy decision to make, so let us move on and look at another behavioural view but, this time, at a higher level of abstraction than the state machine diagram.

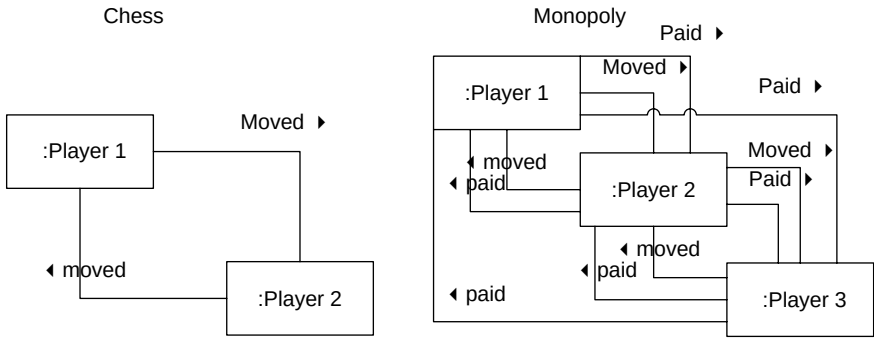

Figure 3.34 Comparing complexity – communication diagrams

The diagram in Figure 3.34 shows a higher level of behaviour of both the chess and Monopoly examples. The diagram used here is the communication diagram, which is one of the diagrams that are deemed no longer useful and have been omitted from SysML. The reason that this diagram is being used for this example is that, although the same level of abstraction could be modelled using a sequence diagram, the communication diagram is far better than a sequence diagram for showing generic interactions and location of life lines and is far clearer.

Meanwhile, SysML omissions aside for now, it is again time to compare the complexity of the two diagrams. In the scenario shown here, the chess example is still very simple – there is only one interaction going each way between the two life lines, resulting in a total of two interactions. In the case of the Monopoly example, there are more interactions on the diagram. Each player passes the control of the game on to the next player, which results in a single interaction between subsequent lifelines. Also, any player can pay money to any other player, which results in an extra six interactions for this scenario with three players, and an extra thirty interactions in the scenario with six players! When comparing these two views, it is abundantly clear that the Monopoly diagram is by far and away the more complex of the two.

As we have looked at both examples from a higher level of abstraction, it is now time to look at both from a lower level point of view.

The diagram in Figure 3.35 shows a lower level of abstraction for both examples using activity diagrams. In the case of the chess example, a single activity diagram is used to describe the behaviour of the single activity from the state machine diagram (and, hence, operation from the block definition diagram), that of ‘move’. Bearing in mind that the ‘move’ activity describes all the planning, strategies and movement of the chess pieces, this would result in a very complex diagram indeed – represented in the diagram by a tangled mess. In the case of the Monopoly example, there are more

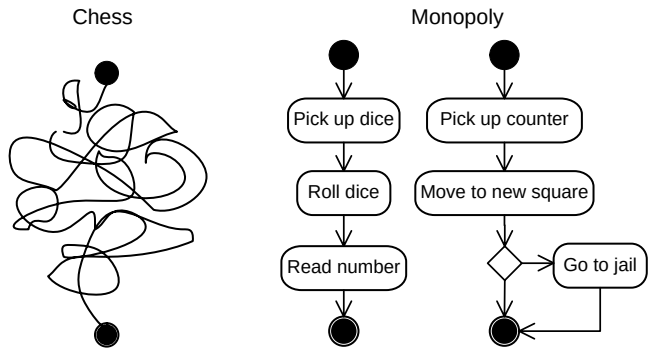

Figure 3.35 Comparing complexity – activity diagrams

activity diagrams (one for each activity – only two are shown here) but each activity diagram is so simple as to be almost trivial.

The diagram in Figure 3.36 shows each of the three behavioural views for each example alongside one another so that they can be compared directly.

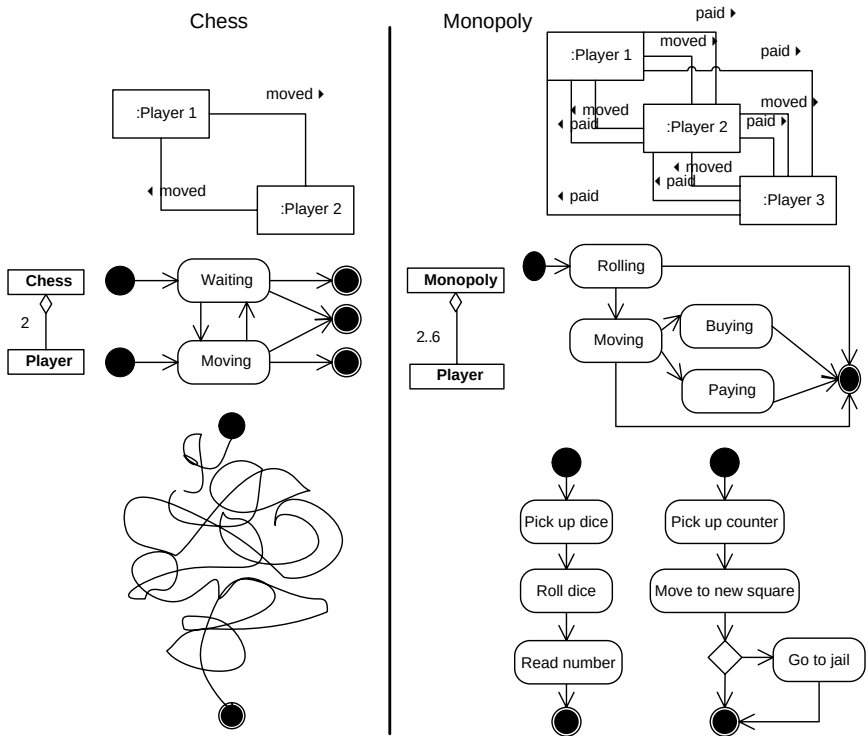

Figure 3.36 Summary of views to assess complexity

The diagram in Figure 3.36 shows all the diagrams that have been produced so far for this example. In summary, the following conclusions were drawn.

- When comparing the original block definition diagrams, there was not much to choose between the two, but the Monopoly example was slightly more complex than chess.
- When comparing the state machine diagrams, again there was not much to choose between the two, but the chess state machine diagram was deemed to be more complex than the Monopoly one.

At this stage, there is nothing to really distinguish between the two examples, as, in each view, one example was deemed slightly more complex than the other. It is interesting to see how things change quite dramatically when the systems are looked at from higher and lower levels of abstraction.

- When looking at the higher level of abstraction, using the outlawed communication diagram, it was very clear that the Monopoly system was far more complex than chess.
- When looking at the lower level of abstraction using the activity diagram, it was very clear that the chess system was far more complex than Monopoly.

There are a few conclusions that may be drawn at this point concerning, not just complexity, but also general modelling.

- The complexity of the systems manifested itself at different levels of abstraction. In the case of chess, the complexity manifested itself at the lower levels of abstraction, whereas in the Monopoly system complexity abounded at the higher levels. This actually makes a nonsense of the question of which is the more complex system. Neither is 'more' complex as such, but in each system the complexity manifested itself at different levels of abstraction.
- If any of these views was taken individually and compared, there is no way whatsoever that any realistic comparison could be made between the two systems. For example, just comparing block definition diagrams gives no real insight into each system. This may seem obvious, but many people will construct a block definition diagram and then state that this is the model of their system. To understand any system and to create a useful model, both the structural and behavioural aspects of the model must be looked at.
- Even when both the structural and behavioural aspects of the model are realized, it is essential to look at the model at different levels of abstraction for the system to be understood.
- It is also possible that the complexity of the system changes depending on the point of view of the stakeholder. For example, imagine a passenger-train system (again), and imagine it now from two different stakeholders' points of view – the engineers involved in train development, and the signalling engineers. The train-development engineers may view the system in a similar way to the chess system, in that the complexity occurs at a low level of abstraction, as individual trains are very complex machines but, at a higher level, they simply drive along

a set of rails. The signalling engineers may view the same system in a similar way to the Monopoly system, in that each train is viewed as a simple system, as a train is a machine that goes backwards or forwards along the rails. However, stopping these simple machines from colliding and making them run on time is a very complex undertaking.

It is essential when modelling, therefore, to look at both the structural and behavioural aspects of the system and to look at the system at different levels of abstraction. In this way, areas of complexity may be identified and hence managed. It is also important to look at the system in different ways and from different stakeholders' points of view, which helps to keep connection to reality for all stakeholders.

3.7 Conclusions

This chapter has looked at the two aspects of modelling that are necessary in SysML – structural and behavioural. Block definition diagrams were used to discuss structural diagrams and state machine diagrams were used to introduce behavioural modelling.

It is when considering behavioural modelling that the missing concept of 'instances' really comes to the fore, which is a serious problem with the SysML notation.

It was seen that, in order to have a complete model, it is necessary to look at the system at different levels of abstraction. Only when a number of these views exist can one start to have a concise, correct and coherent model.

Key to all of this, of course, is consistency, which leads to a good, correct, concise and consistent model, which leads directly to confidence in the system. Confidence means that the system is understood and can be communicated to other people.

Remember:

$$\text{SysML} + \text{consistency} = \text{model}$$

$$\text{SysML} - \text{consistency} = \text{pictures}$$

When modelling, try to remember and understand these concepts, and then try to remember the syntax, not the other way around.

3.8 Reference

1 Holt J. *UML for Systems Engineering: Watching the Wheels*. 2nd edn. London: IEE Publishing; 2004 (reprinted, IET Publishing; 2007)

Chapter 4
The SysML diagrams

'You must unlearn what you have learned.'

Yoda

4.1 Introduction

This chapter describes the nine SysML diagrams. Following an overview, the terminology used throughout the chapter is explained and the structure of SysML diagrams discussed. This is followed by a discussion on the SysML meta-model which forms the basis of this chapter. Following this, each of the nine diagrams is described in turn.

4.1.1 Overview

This chapter introduces the nine types of diagram that may be used in the Systems Modelling Language (SysML). The information in this chapter is kept, very deliberately, at a high level. There are several reasons for this.

- The main focus of this book is to provide practical guidelines and examples of how to use the SysML efficiently and successfully. It is, therefore, a deliberate move to show only a small subset of the SysML language in this chapter. In reality, it is possible to model a large percentage of any problem with a small percentage of the SysML language.
- Experience has shown that the most efficient way to learn, and hence master, the SysML is to learn just enough to get going and then to start modelling with it. This chapter aims to give just enough information to allow readers to start using the SysML. As readers progress and find that they want to express something beyond their present understanding, they are referred to the SysML specification.

Each of the SysML diagrams has a section devoted to it in this chapter. For each diagram type there is an overview, discussion of the diagram elements, examples of how to use the diagram and a discussion of some of the issues that may arise when using the diagram.

4.1.2 Terminology

Figure 4.1 summarizes the basic terminology used for SysML.

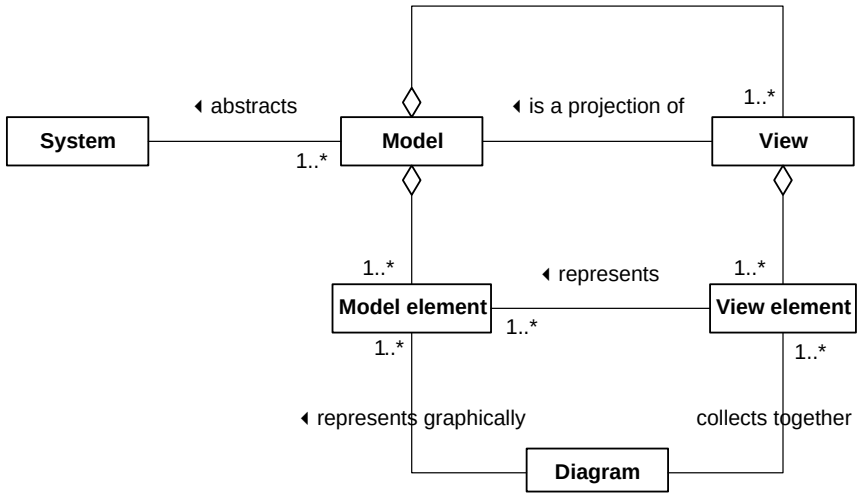

Figure 4.1 SysML terminology

It can be seen from the diagram in Figure 4.1 that one or more ‘Model’ abstracts a ‘System’. Remember that a model is defined as a simplification of reality; therefore, there is a level of abstraction removed between the model and the system. The system itself can represent any sort of system, be it a technical, social, political, financial or any other sort of system.

Each ‘Model’ is made up of one or more ‘View’, each of which is a projection of a ‘Model’. A view is simply the model itself looked at from a particular context or from a particular stakeholder’s point of view. Examples of views include, but are not limited to, logical views, implementation views and contextual views. Therefore, it is possible (and usually very likely) that a model will have a number of views that will differ depending on the stage of the development life cycle or the reader’s viewpoint of the system. At the very least a model should have views representing its structural and behavioural aspects.

Each ‘Model’ is made up of one or more ‘Model element’ and, in turn, each ‘View’ is made up of one or more ‘View element’. There is a direct comparison between these two types of element, as a ‘View element’ represents one or more ‘Model element’. Model elements, in SysML, represent the abstract concepts of each element, whereas the view elements represent whatever is drawn on the page and can be physically ‘seen’ by the reader.

The basic ‘unit’ of any model is the ‘Diagram’ that represents graphically a number of ‘Model element’ and that collects together a number of ‘View element’.

It is the diagram that is created and presented as the real-world manifestation of the model.

4.2 The structure of SysML diagrams

Each diagram in the SysML has the same 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 4.2.

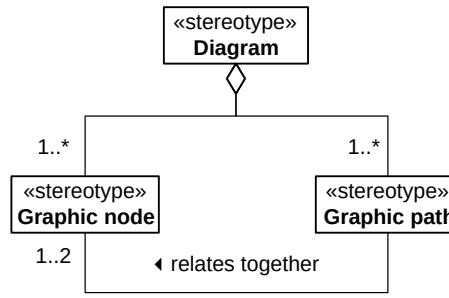

Figure 4.2 Structure of each SysML diagram

The diagram in Figure 4.2 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 . . . well, of a stereotype. Stereotypes are a mechanism by which the SysML can be extended. Indeed, the SysML itself is defined using stereotypes on the underlying UML. Stereotypes are discussed briefly in Section 4.3 below.

4.2.1 Frames

Any SysML diagram must have a graphic node known as a 'Frame' that encapsulates the diagram and that has a unique identifier, in order to make identification of, and navigation between, diagrams simpler. An example of a frame is shown in Figure 4.3.

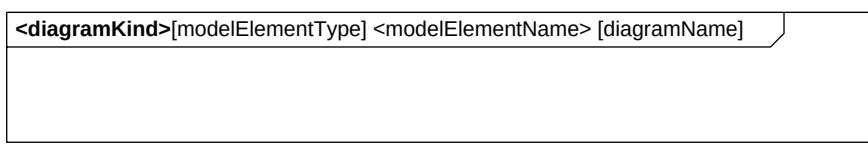

Figure 4.3 Example of a frame in SysML

The diagram in Figure 4.3 shows the symbol for a frame, which is, quite simply, a large rectangle that will contain all the view elements in a particular diagram. The pentagon in the upper left-hand corner contains the name of the diagram in the following format:

< **diagramKind** > [modelElementType] < modelElementName > [diagramName]

Each part is separated from another by a space and **diagramKind** is emboldened. The modelElementType and diagramName parts of the name are in brackets. The diagramKind and modelElementName are mandatory.

diagramKind should have the following names or (abbreviations) as part of the heading:

- activity diagram (act)
- block definition diagram (bdd)
- internal block diagram (ibd)
- package diagram (pkg)
- parametric diagram (par)
- requirement diagram (req)
- sequence diagram (sd)
- state machine diagram (stm)
- use case diagram (uc)

The **diagramKind** is used to indicate the type of the diagram.

The following are the **modelElementType** associated with the different diagram kinds:

- activity diagram – activity
- block definition diagram – block, package, or constraint block
- internal block diagram – block or constraint block
- package diagram – package or model
- parametric diagram – block or constraint block
- requirement diagram – package or requirement
- sequence diagram – interaction
- state machine diagram – state machine
- use case diagram – package

The **modelElementType** indicates the types of SysML elements that are shown on the diagram. For many of the diagrams this information is redundant. However, some of the diagrams can show different types of element. For example, a block definition diagram can show blocks or can show the definitions of constraints (described later in this chapter) using constraint blocks. The modelElementType makes this clear.

The **modelElementName** identifies which model element the diagram is describing. For example, for a state machine diagram that models the behaviour of a block, the modelElementName would be the name of the block. For internal block diagrams, the modelElementName would identify the name of the SysML block for which the internal block diagram is being drawn. When one is modelling processes or producing

meta-models (discussed below), the name of the views produced could be used as the modelElementName.

The **diagramName** is used to give the diagram a unique name. This is particularly important when different diagrams of the same type are drawn for the same model element. The diagramName would differentiate between these diagrams, since they would have the same diagramKind, modelElementType and modelElementName.

4.3 Stereotypes

Stereotypes are, by far, the most widely used of all the SysML extension mechanisms and represent a powerful way to define new SysML elements by tailoring the SysML meta-model, which is described further in Section 4.4.

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. The name of the stereotype is displayed either inside the symbol (in the case of the stereotype applying to a graphical node) or above the symbol (in the case of the stereotype applying to a graphical path).

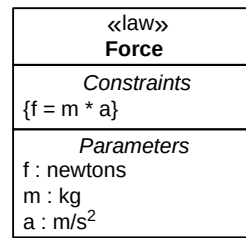

Figure 4.4 Stereotyped constraint block

Figure 4.4 shows a SysML constraint block (these are described in detail in Section 4.8 below). However, in this case the constraint block shown here is *not* a regular SysML one, but it is a stereotyped constraint block. This fact can be established because there is a word, *law*, inside the chevrons above the constraint block name. The English phrase that should be used when reading stereotypes is 'that happens to be a'. Therefore, to read the diagram in Figure 4.4, one would say, 'There is a constraint block called force that happens to be a law.' Of course, we are assuming here that the person reading the diagram actually understands what the term *law* means in the context of this model.

A stereotype can be defined as a tailored version of any type of SysML element that exists within the SysML meta-model. A stereotype *must* be based on an existing SysML element and it is *not possible* to define a brand-new element from scratch. Some stereotypes are already defined within the SysML, such as ≪include≫ and ≪extend≫, which are defined as special types of dependency and indeed ≪constraint≫ used to indicate a constraint block that is defined as a special type of block.

Stereotypes are defined by tailoring the SysML meta-model. Elements from the SysML meta-model are taken and then special types of them are declared that represent the stereotype, as seen in Figure 4.5.

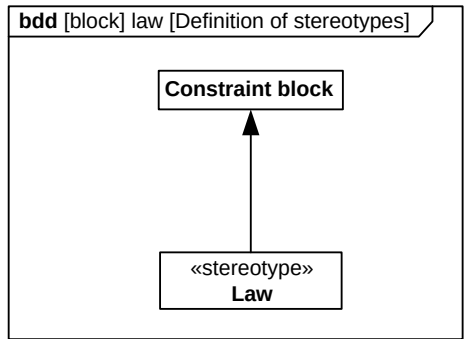

Figure 4.5 Defining a stereotype

The diagram in Figure 4.5 shows the definition of a stereotype. The diagram shows two blocks, 'Constraint block' and 'Law', which are related by a special type of specialization 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 generalization symbol.

The element that is being stereotyped, in this case, is 'Constraint block'. The most important point to realize at this point is that 'Constraint block' is taken *from the meta-model*. The element 'Constraint block' here represents the abstract notion of a constraint block that is found in the meta-model, or, to put it another way, it is the representation of constraint block from the SysML language itself.

The new SysML element to be defined, in this case 'law', is shown in a similar way to a child block on a specialization but using the extension relationship. However, this time it happens to be a stereotype, as indicated by ≪stereotype≫. In addition to the graphical definition, it is often useful to provide a textual description of the stereotype that describes its intended use.

Now, whenever we see a SysML constraint block that happens to be a law, as shown in Figure 4.4, we know that it is not a regular SysML constraint block, but a special type, as indicated by the ≪Law≫ stereotype. Practically, we now have a shorthand way to show that a particular constraint block is actually a law, rather than a regular constraint block. (Different types of constraint blocks are discussed further in Chapter 5.)

4.4 The SysML meta-model

This chapter also introduces the concept of the SysML meta-model.

The SysML specification defines SysML in terms of the underlying UML on which SysML is based, and does so using UML.

The SysML meta-model is a model of the SysML. In keeping with the use of UML in the SysML specification, UML class diagrams have been used to model the SysML throughout this chapter. In most cases 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 diagrams make use of a notation called the *association class* (this notation exists in SysML as, unsurprisingly, the association *block*). An association class is a mechanism for adding properties and operations to an association. It is represented by a class (SysML block) connected to an association via a dashed line. Consider the fragment of the requirement diagram meta-model shown in Figure 4.6. (The full diagram is given in Section 4.9.)

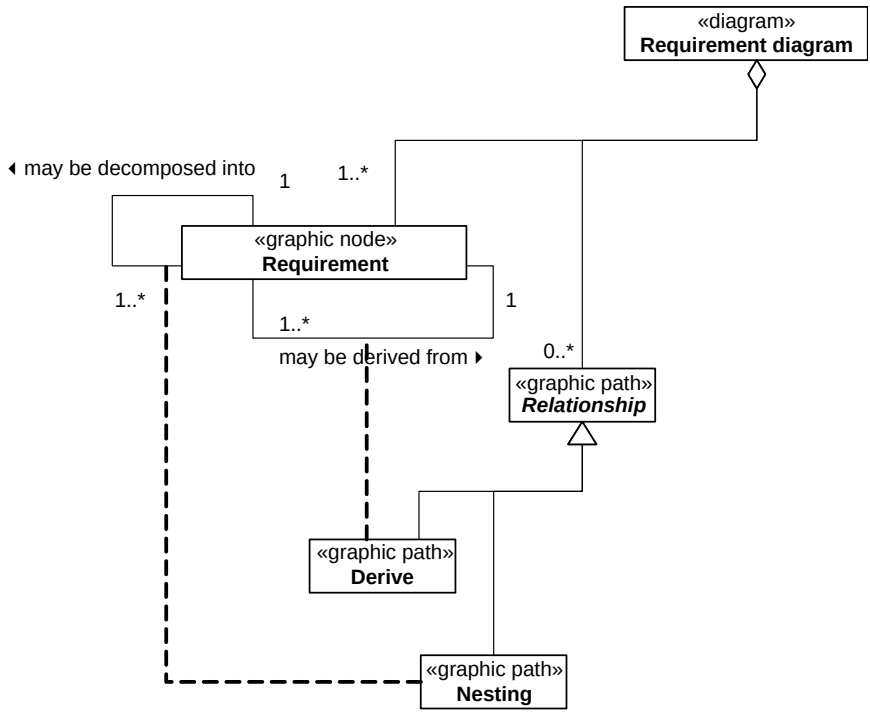

Figure 4.6 A fragment of the requirement diagram meta-model illustrating the use of association classes

The classes (blocks) ‘Derive’ and ‘Nesting’, which happen to be graphic paths, indicate some of the types of ‘Relationship’ that can appear on the requirement diagram. Now consider the ‘Requirement’ class (block) and its associations. Each association is connected by a dashed line to the ‘Derive’ or ‘Nesting’ classes (blocks). This means that, as well as acting as classes in their own right, they are also acting as association classes, adding some extra information to the associations to which they

are attached. Reading the diagram gives: a 'Requirement' may be decomposed into one or more 'Requirement' *via* 'Nesting', and one or more 'Requirement' may be derived from a 'Requirement' *via* 'Derive'.

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 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.

4.4.1 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; 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, 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 authors' point of view, a logical order. Therefore, the various parts of this chapter may be read in any order.

4.4.2 The worked example

This chapter also uses a worked example, which is a robot challenge game that involves building a software-controlled robot from a widely available building system. There are several reasons for choosing this as the subject of the example.

- The robot is an excellent example of a (relatively) simple system. It contains a number of cooperating subsystems such as sensors, power and transmission, and control.
- The robot is an example of a system that consists of both hardware and software, which is typical of many real-life systems.
- There are many aspects that can be modelled, from the structure of the robot, the parts used to build it, to the behaviour that the robot has to display.
- The various robot aspects lend themselves to the use of all the SysML diagrams.
- Now that the scene has been set, it is time to plunge straight in and look at the SysML diagrams themselves.

4.5 Block definition diagrams (structural)

4.5.1 Overview

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 3 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 robot challenge game that runs throughout this chapter.

Block definition diagrams realize 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.

4.5.2 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 4.7 shows a high-level meta-model of block definition diagrams. Recall from Section 4.4 that the SysML meta-models shown in this chapter are actually modelled using UML *class* diagrams, but that with the exception of the occasional use of association classes, these may be taken to be SysML block definition diagrams.

Remember that the SysML is a language and, therefore, should be able to be read and understood just like any other language. Therefore, the model in Figure 4.7 may be read by reading a block name, then any association that is joined to it, and then another block name.

Therefore, reading the diagram: a 'Block definition diagram' is made up of one or more 'Block', zero or more 'Relationship', zero or more 'Interface', zero or more 'Port' and zero or more 'Item flow'. It is possible for a block definition diagram to be made up of just a single block with no relationships at all; however, it is not possible to have a block definition diagram that is made up of just a relationship with no blocks. Therefore, the multiplicity on 'Block' is one or more, whereas the multiplicity on 'Relationship' is zero or more.

Each 'Relationship' relates together one or two 'Block'. Notice that the word 'each' is used here, which means that for every *single* 'Relationship' there are one or two 'Block'. It is also interesting to 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.

Each 'Block' is made up of zero or more 'Property', zero or more 'Operation' and zero or more 'Constraint'. This shows how blocks may be further described using properties, operations and constraints, or that they may be left with neither. There are three types of 'Property'.

- 'Part' properties, which are owned by the block – that is, properties that are intrinsic to the block but which may have their own identity.

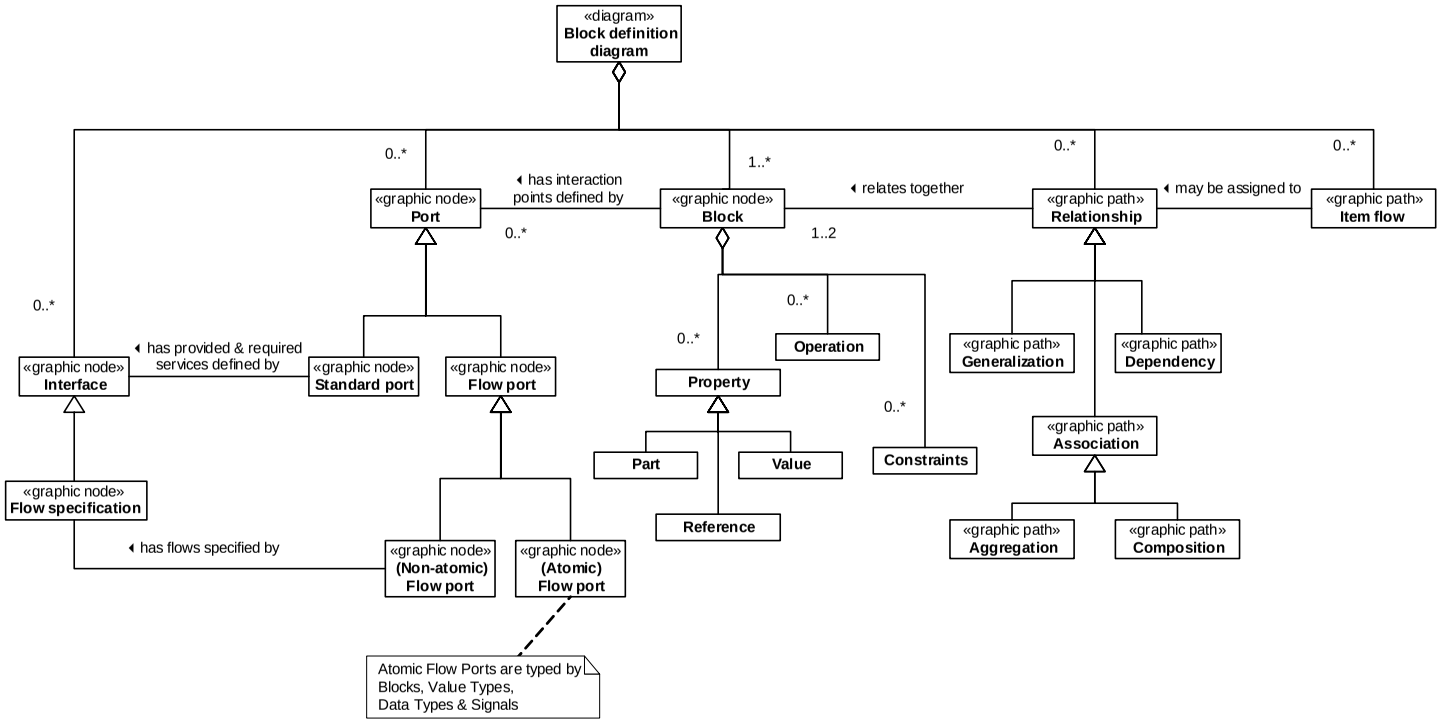

Figure 4.7 Partial meta-model for block definition diagrams

- 'Reference' properties, which are referenced by the block, but *not* owned by it.
- 'Value' properties, which represent properties that cannot be identified except by the value itself, for example numbers or colours.

These various types of property are somewhat confusing. An example will help and is illustrated in Figure 4.8.

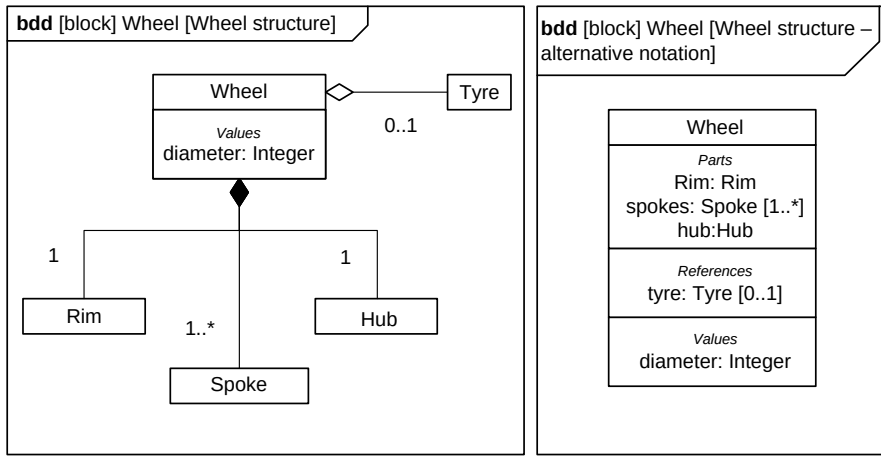

Figure 4.8 Types of property – alternative representations

The diagram on the left in Figure 4.8 models the structure of a wheel and the reader is directed to Figure 4.9 for a description of the notation. The diagram shows that a 'Wheel' is composed of a 'Rim', a 'Hub' and one or more 'Spoke'. It has a 'diameter' which will be represented as an 'Integer'. The 'Wheel' is also made up of zero or one 'Tyre'.

Now consider the diagram on the right. This shows exactly the same information but in a different format, which uses named property compartments rather than via graphical paths and nodes. Since the 'Rim', 'Hub' and one or more 'Spoke' are all intrinsic parts of the 'Wheel' – that is, they can be thought of as having their own identity but form indivisible elements of the 'Wheel' – they are modelled as *part* properties. The 'Tyre' is not owned by the 'Wheel'. It can be removed and attached to another 'Wheel'. For this reason it is modelled as a *reference* property. Finally, the 'diameter' is simply a number – it does not have any individual identity – and is therefore modelled (on both diagrams) as a *value* property.

The diagram on the right is more compact than that on the left, although perhaps not as clear – useful, perhaps, when producing summary diagrams. The authors, generally, prefer the more visual approach.

There are three main types of 'Relationship'.

- 'Association', which defines a simple relationship between one or more 'Block'. There are also two specializations of 'Association': a special type of 'Association'

known as 'Aggregation' and one known as 'Composition', which show 'is made up of' and 'is composed of' associations respectively.
- 'Generalization', 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 affect 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 'Block' has interaction points defined by zero or more 'Port' which have two main types.

- 'Standard port' is used to represent an interaction point that has provided and required *services* defined by an 'Interface'. A service is a piece of functionality, represented by an operation defined on the 'Interface', that the block either provides to other blocks or requires from other blocks to be able to function properly itself. Typically blocks with standard ports and interfaces are used to model software architectures.
- 'Flow port' is used to represent an interaction point through which material, data or energy can enter or leave a block. If only a single type of data, material or energy passes through the flow port, then the port is considered to be *atomic*. If multiple types can pass through then the flow port is considered to be *non-atomic* and the flow is specified by a 'Flow specification' which is a special type of 'Interface'.

Used in conjunction with the 'Flow port' is the 'Item flow', which may be assigned to a 'Relationship' to show the flow of data, material or energy between blocks. The relationship thus annotated will usually connect blocks via their flow ports.

The 'Standard port' and 'Interface' are taken directly from UML (where the port was called simply that, a 'Port'). The 'Flow port' and 'Flow specification' are new and very powerful concepts introduced by SysML. Examples of the use of standard ports, interfaces, flow ports, flow specifications and item flows are given below.

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes, as illustrated in Figure 4.9.

The diagram in Figure 4.9 shows the graphical symbols used to represent elements in a block definition diagram. The basic symbol is the block that 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 may also contain the stereotype ≪block≫, although this may be omitted if it is the only stereotype on the block.

When properties, operations and constraints are present, blocks are easy to identify, as they have additional rectangles, drawn underneath the block name, with the properties, operations and constraints contained within. Each of these additional

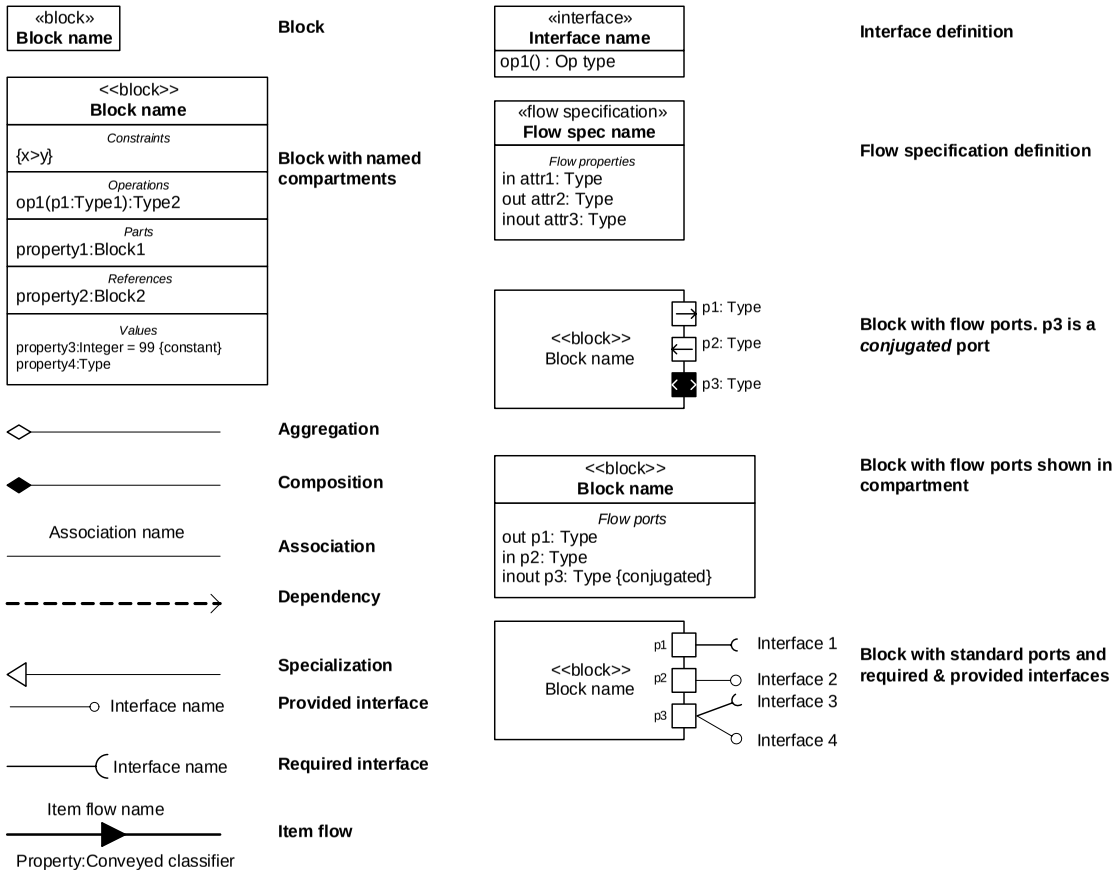

Figure 4.9 Graphical symbols for elements in a block definition diagram

compartments may also be labelled to show what it contains, and the property compartments may be further subdivided to show part, reference and value properties.

Relationships are shown by a simple line, with slight variations depending on the type of relationship. For example, a specialization has a distinctive triangle symbol show next to the parent block, whereas a simple association has no such additional symbol.

Ports are shown as small squares (or rectangles) straddling the edge of the block.

Standard ports are shown using empty squares and are identified with a label. The required and provided interfaces are attached to the standard port using the symbols shown in Figure 4.9 and are labelled with the name of the defining interface.

Flow ports contain a small arrow showing the direction of the flow (whether into the block, out of the block or both) and are labelled with the name of the port and the type of information flowing through the port. For atomic flow ports this will be the name of another block; for non-atomic flow ports this will be the name of a defining flow specification. Flow ports can also be shown as a solid square with a white arrow. Such a flow port is a *conjugated* flow port and its use is described below. Flow ports may also be shown in a separate compartment attached to the block as shown in Figure 4.9.

The interfaces associated with standard ports are defined using special blocks that are stereotyped ≪interface≫ and that may have only operations, but no properties. The operations represent the services provided by a block that has that interface as a provided interface, or the services required by a block that has it as a required interface.

Flow ports that transfer more than one kind of thing are typed using flow specifications that can be thought of as a special kind of interface, but one that has no operations and only properties. They are defined using special blocks that are stereotyped ≪flowSpecification≫ and that have a compartment labelled *flowProperties*. This compartment contains properties defining the items that can flow across a flow port typed by the flow specification. The words ‘in’, ‘out’ and ‘inout’ are used to show the direction that the different items flow.

Item flows are represented by a labelled triangle attached to the middle of an association. The item flow can have a name by which it can be identified and can also be labelled with the property that is transferred. This latter may appear at first to be redundant, as item flows connect flow 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 flow 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 flow port by a specialization relationship. An example of this is given in the following subsection.

4.5.3 Example diagrams and modelling – block definition diagrams

Block definition diagrams may be used to model just about anything, including the following.

- *Modelling physical systems*. The robot challenge game in this chapter is an example of such a physical system.

- *Software*. The SysML can also be used for modelling software and, again, the robot challenge game gives an example of this. Readers may, however, find the extra diagrams present in the UML to be more suitable to software modelling.
- *Process modelling*. Understanding processes, whether they be in standards procedures or someone's head, is very important in the world of systems engineering. The SysML is a very powerful notation for modelling such processes and is the topic of Chapter 6.
- *Teaching and training*. The SysML is a very good tool for teaching and training. The fact that you are reading this book and (hopefully) understanding the concept associated with block definition diagrams means that the SysML is a good and effective training tool. Remember, more than anything, that the SysML is a means to communicate effectively, which is what training and teaching are all about.

Figure 4.10 shows the high-level structure of the robot challenge game.

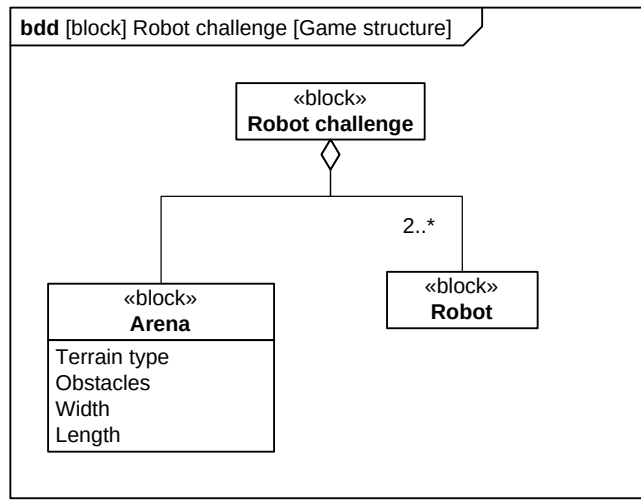

Figure 4.10 High-level block definition diagram for the robot challenge game

First, let us consider the diagram frame. The term 'bdd' indicates that the diagram is a block definition diagram with '[block]' showing that it contains, unsurprisingly, blocks. 'Robot challenge' shows that the diagram is modelling an aspect of the 'Robot challenge' element. Finally, '[Game structure]' is the name of the diagram.

Now consider the contents of the diagram. The 'Robot challenge' is made up of an 'Arena' and two or more 'Robot'. The 'Arena' has a number of properties that define its size, the terrain type and the obstacles contained. All the properties have been grouped into one compartment rather than being divided into parts, references and values. This is allowed – SysML does not require the use of the named compartments but they can be used to add extra information to the model if desired. Also note that the *types* of the properties have been omitted in this diagram (and in most subsequent ones). This has been done to reduce the clutter on the diagrams. In the complete

model *all* properties should have types associated with them. Also, when producing initial models the exact types of the properties will not be known, only that a block has certain properties.

We can now turn to the 'Robot' block and model it in greater detail. This is shown in Figure 4.11.

The frame of this diagram again shows that it is a block definition diagram containing blocks, that it is describing aspects of the 'Robot' element and that the diagram is named 'Robot structure and Interfaces'.

The 'Robot' is made up of a 'Controller' which is attached to a 'Chassis', a piece of 'ControllerSoftware' that lives on the 'Controller', between zero and three 'Sensor', between zero and three 'Motor' and two or four 'Limb'. The 'Chassis' is made up of two 'Strut' and two or four 'Axle'. Each 'Strut' supports one or two of the 'Axle' and each 'Axle' is connected to a 'Limb'. Different types of 'Limb' can be fitted: 'Caterpillar track', 'Spider leg assembly' or 'Wheel'. A note containing a *constraint* is used to show that the robot has two caterpillar tracks, two spider leg assemblies or four wheels fitted. Constraints, which are represented as text describing the constraint enclosed in braces {thus}, can be added to any model element to constrain the meaning of a model.

The 'Controller' is further composed of four 'Push button', having one each of a 'View', 'On/Off', 'Run' and 'Select' button. The small number '1' in the top right corner of each of the button blocks shows that only one of each kind is fitted. Note the use of composition here rather than aggregation. Each of the blocks that make up the robot can be separated from the robot. However the buttons form an integral part of the controller and cannot be removed. For this reason composition better reflects the structure of the controller.

The diagram also shows that the robot's constituent blocks have a number of ports, both standard ports and flow ports. The 'Sensor' has a standard port that requires use of the 'SensorIF' interface to function, whereas the 'Drive' provides services through a standard port that uses the 'Drive' interface and also supplies torque through the 'torque' flow port. Torque also flows into and out of each 'Axle' and into each 'Limb', again through 'torque' flow ports. The 'Controller' has three standard ports that each require the use of the 'Drive' interface and three that each provide services via the 'SensorIF' interface, as well as a standard port, 'Display window', which requires the use of the 'Visual' interface. Each of the controller's 'Push Button' provides services via the 'PushButton' interface. The 'Controller' also has two flow ports: the 'dcin' port accepts DC voltage and the 'ir' port sends and receives infrared data.

Note should also be taken of the use of *packages* to show the conceptual grouping of parts of the robot. Packages are described further in Section 4.7.

Figure 4.11 shows that the 'Controller' provides the 'SensorIF' interface that is required by the 'Sensor' blocks, and that it requires the 'Drive' interface that is provided by the 'Motor' blocks. But how are these blocks connected via these interfaces and just what does it mean to say that an interface is provided or required? Figure 4.12 helps to explain this.

Figure 4.12 shows how the 'Controller' and the 'Sensor' and 'Motor' blocks can be connected via their interfaces. It also defines the 'SensorIF' and 'Drive' interfaces.

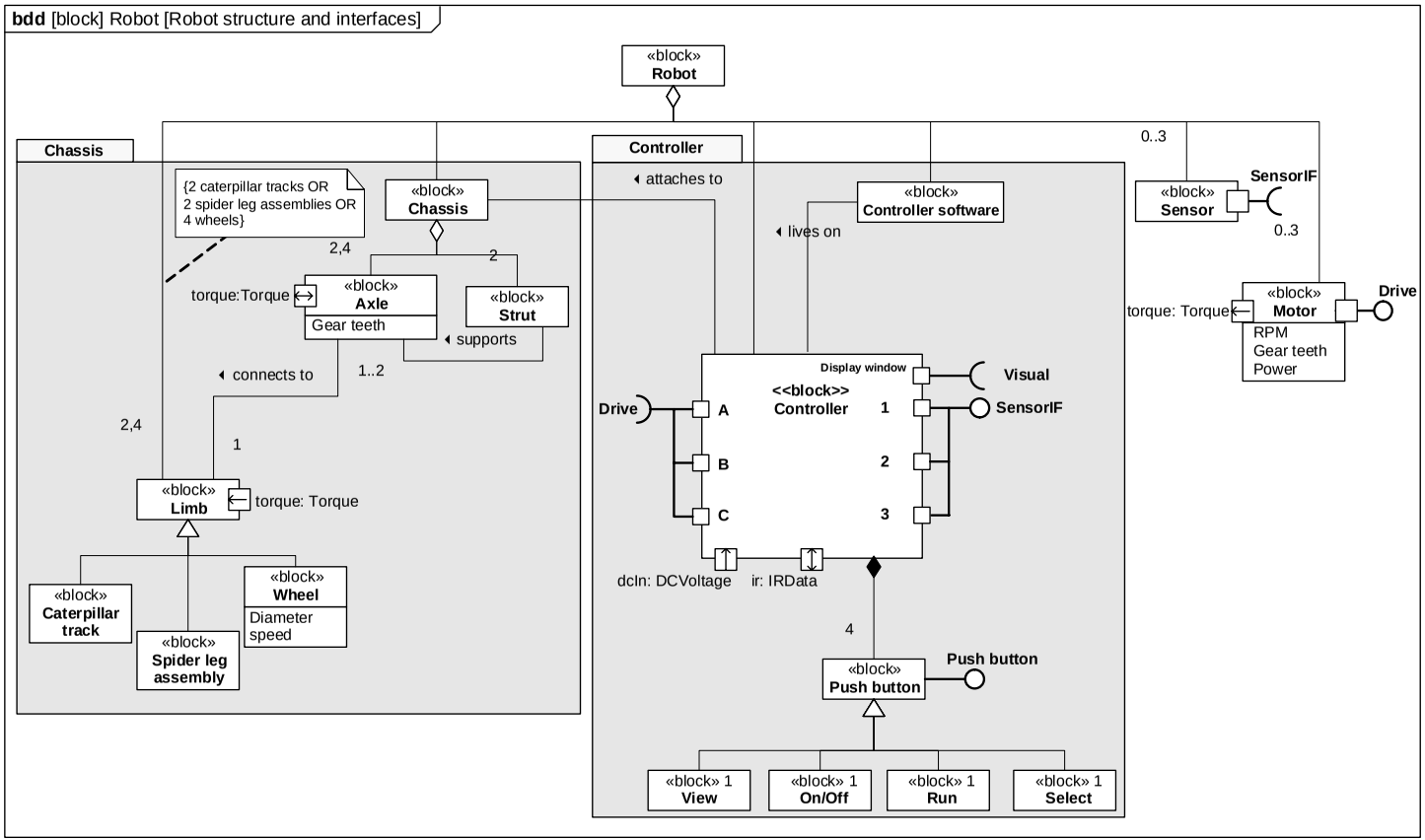

Figure 4.11 Detailed block definition diagram showing structure of robot

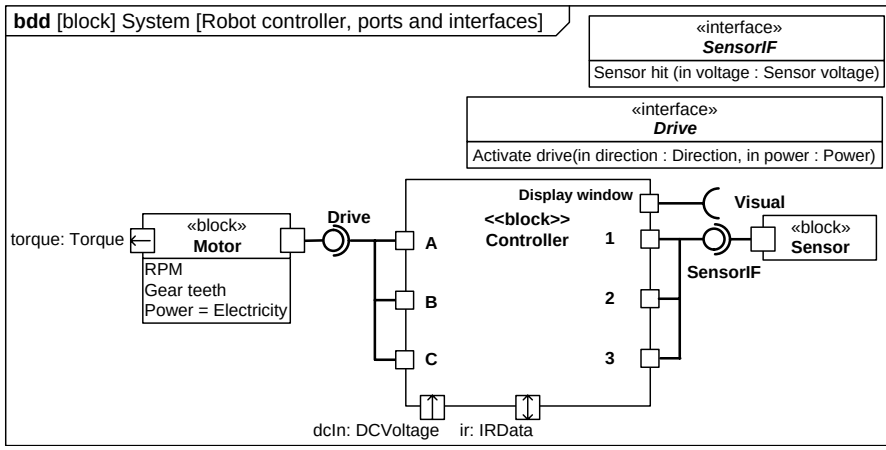

Figure 4.12 Block definition diagram showing robot controller connections and interface definitions

Required and provided interfaces can be connected provided they are of the same type. In fact, SysML allows slightly more flexibility than this. A required interface can be connected to a provided interface that is of the same type *or of a specialization of that type*. This is because specialization can only add new operations to those defined in the parent type, never remove any.

For example, Figure 4.13 defines some additional interfaces and also arranges the interfaces in an interface *type hierarchy*.

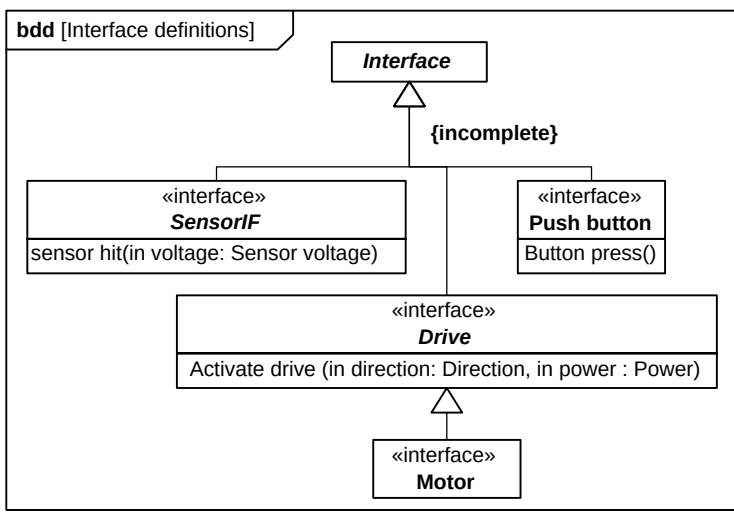

Figure 4.13 Block definition diagram showing interface definitions

The 'Controller' requires the use of the 'Drive' interface, but could also connect to any blocks that provide the 'Motor' interface since this is a type of 'Drive' interface. If Figures 4.11 and 4.12 were modified to show that the 'Motor' provided the 'Motor' interface rather than the 'Drive' interface, the 'Controller' could still be connected to the 'Motor'.

The use of the terms *provided* and *required* interfaces may seem counterintuitive and is a holdover from the UML underlying SysML, stemming from UML's initial use in modelling software systems. The following definitions should help.

- If a block accepts *input* on an interface and *acts* on that input, then a *provided interface* should be used. That is, provided interfaces *accept* inputs and act on them.
- If a component generates *outputs* that have to be acted on by something else, then a *required* interface should be used. That is, required interfaces *generate* outputs that are acted on by another part of the system.

The interface definitions make use of the types 'Direction', 'Power' and 'SensorVoltage'. A block definition diagram can be used to define these types as shown in Figure 4.14.

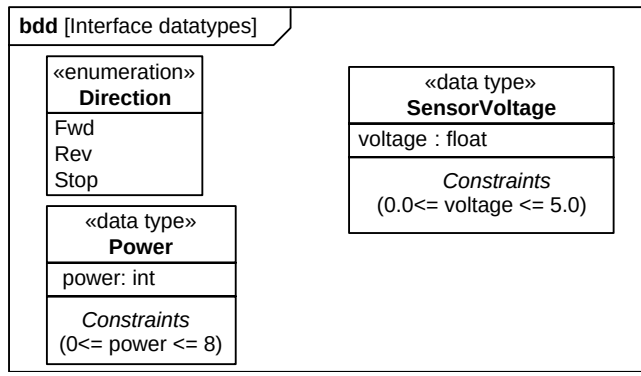

Figure 4.14 Block definition diagram showing data types used on interfaces

The stereotypes ≪dataType≫ and ≪enumeration≫ are used in Figure 4.14 to highlight blocks that represent data types. These are blocks that can be used to type a property or the parameters of an operation, but cannot have their own identity. For example, numbers (such as integers and real numbers) are data types and are themselves used in the definitions above with 'int' indicating integers and 'float' indicating real numbers. Some data types are simply lists of possible values and are known as *enumerations*. For example, the 'Direction' data type can have values 'Fwd', 'Rev' and 'Stop' only. The convention when modelling enumerations is to represent each possible value as an untyped property in a block with the ≪enumeration≫ stereotype.

Having concentrated on the physical aspects of the robot, we now turn to its software elements as shown in Figure 4.15.

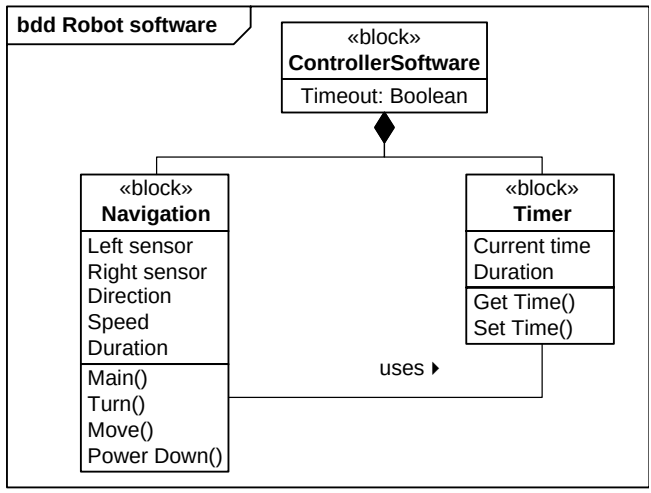

Figure 4.15 Detailed block definition diagram showing structure of ControllerSoftware

The diagram in Figure 4.15 shows that the 'ControllerSoftware' is composed of a 'Navigation' and 'Timer' element and that the 'Navigation' element uses the 'Timer' element. Properties and operations are defined for these three software elements. This block definition diagram defines the *structure*, at a high level, of the robot's 'ControllerSoftware'. In order to complete the definition of the software its *behavioural* aspects must also be modelled. This is done in 4.10.3, 4.11.3 and 4.12.3 below.

As well as modelling the hardware and software of the robot, block definition diagrams can also be used to model the abstract hierarchy of the system and to specify the library of parts used to construct the robot. This is shown in Figure 4.16.

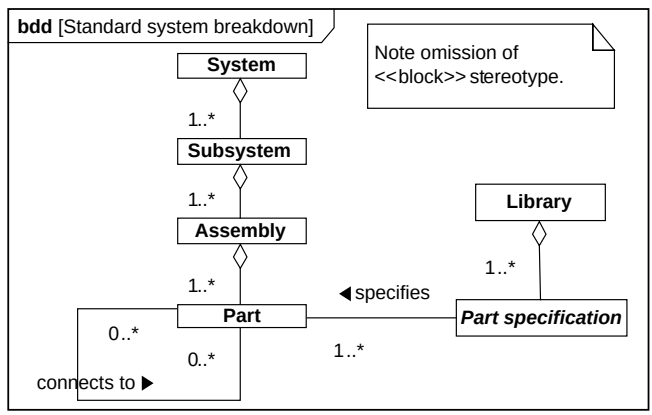

Figure 4.16 Block definition diagram defining a typical system hierarchy and associated library of parts

Figure 4.16 shows that a 'System' is made up of one or more 'Subsystem' made up of one or more 'Assembly'. Each 'Assembly' is made up of one or more 'Part' that may be connected to zero or more other 'Part'. This is a fairly typical system hierarchy that can be applied to a range of application domains. The diagram also models the concept of a 'Library' of one or more 'Part specification', each of which specifies one or more 'Part'. Note also the permitted omission of the ≪block≫ stereotype on this diagram.

The definition of a library of basic parts can then be made and is shown in Figure 4.17.

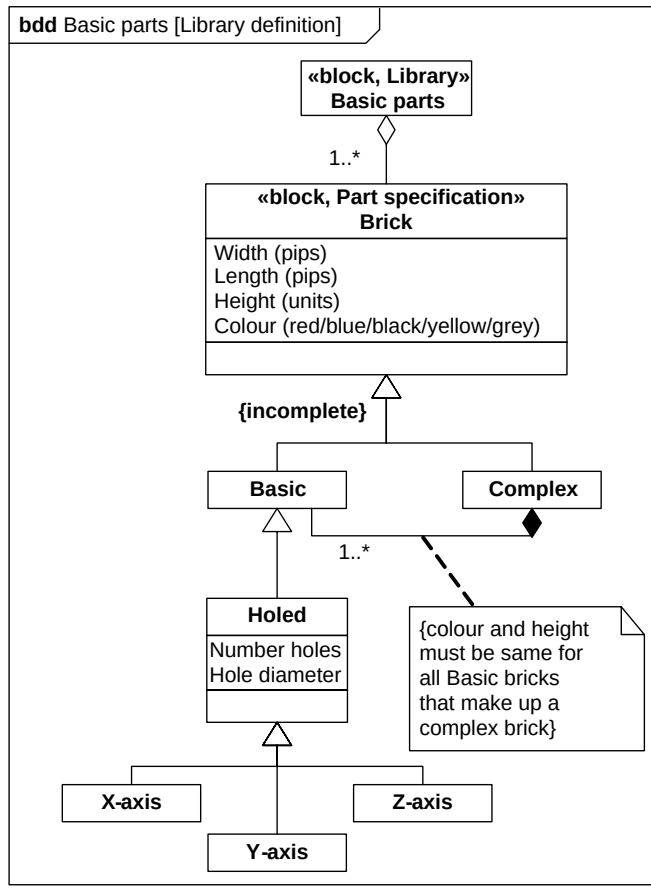

Figure 4.17 Block definition diagram defining the structure of the parts library

In Figure 4.17, a library of 'Basic parts' is defined that is made up of one or more 'Brick'. Note that the blocks representing these elements each have two stereotypes. Both have the ≪block≫ stereotype, with 'Basic parts' also having the ≪Library≫ stereotype and 'Brick' having the ≪Part specification≫ stereotype.

These stereotypes correspond to the blocks shown on Figure 4.16. The ≪block≫ stereotype must be shown here, as SysML only allows it to be omitted when it is the only stereotype on a block. When a block has more than one stereotype they are separated by commas within a single set of guillemets (or chevrons), thus ≪block, Library≫.

The diagram also shows that there are at least two types of 'Brick', namely the 'Basic' brick and the 'Complex' brick, which is composed of one or more 'Basic' bricks. A constraint is used to show how the 'Complex' bricks may be composed. An example of how to define a particular 'Complex' brick is given in 4.6.3 below.

So far we have seen example of flow ports that all transmit only a single type of material, data or energy, that is all examples of *atomic* flow ports. We conclude this section by looking at how to use *non-atomic* flow ports. For this we depart from the ongoing robot example and turn to modelling a pump system used in underwater archaeology. This system is shown in Figure 4.18.

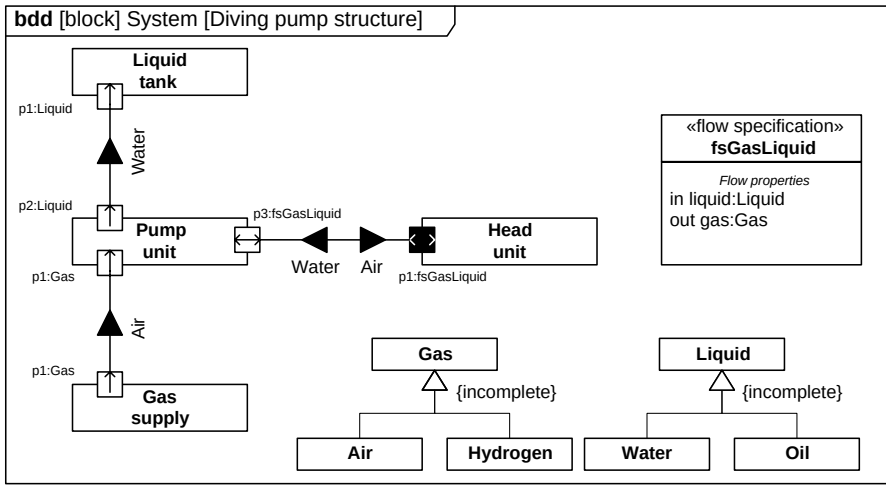

Figure 4.18 Block definition diagram showing the use of flow ports, flow specifications and item flows

The system consists of a 'Gas supply' that feeds air into a 'Pump unit'. This unit sends the air down to a 'Head unit', where the air is used to clear away debris. The 'Head unit' also takes in displaced water, feeding it back to the 'Pump unit', which then sends it to a 'Liquid tank', where it is filtered in order to extract any artefacts carried up with the water.

The 'Gas Supply' has an atomic flow port that sends out 'Gas', which is connected to another atomic flow port on the 'Pump unit' that takes in 'Gas'. The 'Pump unit' also has an atomic flow port that sends out 'Liquid', which is connected to an atomic flow port on the 'Liquid tank' that takes in 'Liquid'. These flow ports are connected by item flows carrying 'Air' and 'Water' respectively. This is OK, since 'Air' is a type of 'Gas' and 'Water' a type of 'Liquid', as also shown on the diagram.

Now the 'Pump unit' needs to send gas down to the 'Head unit' and receive liquid back. In this system this is done via a single dual-hose port, and so we need to represent this using a non-atomic flow port that is typed not by a block but by a flow specification. The term 'fsGasLiquid' defines this flow specification and shows that it represents gas flowing out and liquid flowing in. The bidirectional flow port on the 'Pump unit' is typed by this flow specification. The item flow connecting the 'Pump unit' to the 'Head unit' shows that what actually flows between them is 'Air' and 'Water'.

However, when we come to the 'Head unit' we have a problem: the flows defined in the flow specification are going in the wrong direction for the 'Head unit'. We resolve this by marking its bidirectional flow port as a *conjugate* flow port, indicated by the black background with white arrow. A conjugate flow port is still typed by a flow specification but indicates that the directions of all the flows within the flow specification are reversed. So for the 'Head unit' this means that liquid flows out and gas flows in.

4.5.3.1 Differing views

The question that most people ask at this point is, 'Which is the best diagram?' The question that should be asked, however, is, 'Which is the most appropriate diagram for the problem at hand?' Depending on the criteria for defining what 'best' means, the most appropriate diagram will change. Imagine if the problem called for the simplest (least complex) of models, the most appropriate would be Figure 4.10. If the criteria were more concerned with the physical structure of the robot, then perhaps Figure 4.11 would be more appropriate. If we were concerned with modelling interfaces then Figure 4.13 would be best.

The main point that has to be made here comes back, once more, to communication. It does not matter how many or few diagrams are generated, because, if they cannot be communicated effectively to other people, how is it possible to make an informed and intelligent decision as to which one is the most appropriate for the problem at hand? The SysML provides a method of communication that can iron out many of the ambiguities in the form of a group decision, rather than its being made by a single person.

4.5.4 Using block definition diagrams

Block definition diagrams are 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 as a sentence and they should make sense. A diagram that is difficult to read may simply indicate that there is too much on the diagram and that it needs to be broken down into a number of 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 again here is that block definition diagrams are not used in isolation. They will form the main structural model of a system but must be used in conjunction with the other eight SysML diagrams to give a complete structural and behavioural model of a system. These diagrams are described in the rest of this chapter.

4.6 Internal block diagrams (structural)

4.6.1 Overview

The SysML internal block diagram is based on the UML composite structure and realizes a structural aspect of the model. It is related to both the block definition diagram (described above) and the parametric diagram (described in Section 4.8). Internal block diagrams are used to model the internal structure of a block (hence the name) and can also be used to show how system elements are deployed. By using an internal block diagram, in which compositions and aggregations are implicitly represented by the containment of *parts* within other parts or within the diagram frame representing a block, 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.

4.6.2 Diagram elements

The basic element within an internal block diagram is the *part* that describes a collection of SysML blocks. An internal block diagram identifies parts and their internal structures, showing how they are connected together.

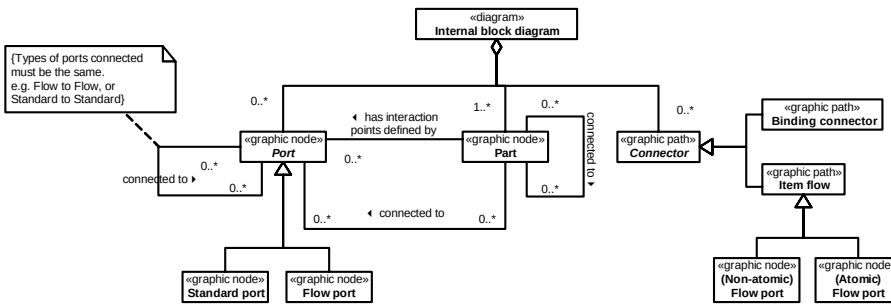

Figure 4.19 Partial meta-model of the internal block diagram

The diagram in Figure 4.19 shows the partial meta-model for the internal block diagram. It can be seen that an ‘Internal block diagram’ is made up of one or more

'Part', zero or more 'Port' and zero or more 'Connector'. A 'Port' defines an interaction point for a 'Part', just as they do for blocks (see 4.5.2) and again come in two types: 'Standard port' and 'Flow port'. A 'Part' 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'. If such a connection is between two 'Flow port', then the connection may be via an 'Item flow'.

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 4.5.2 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.

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes, as illustrated in Figure 4.20.

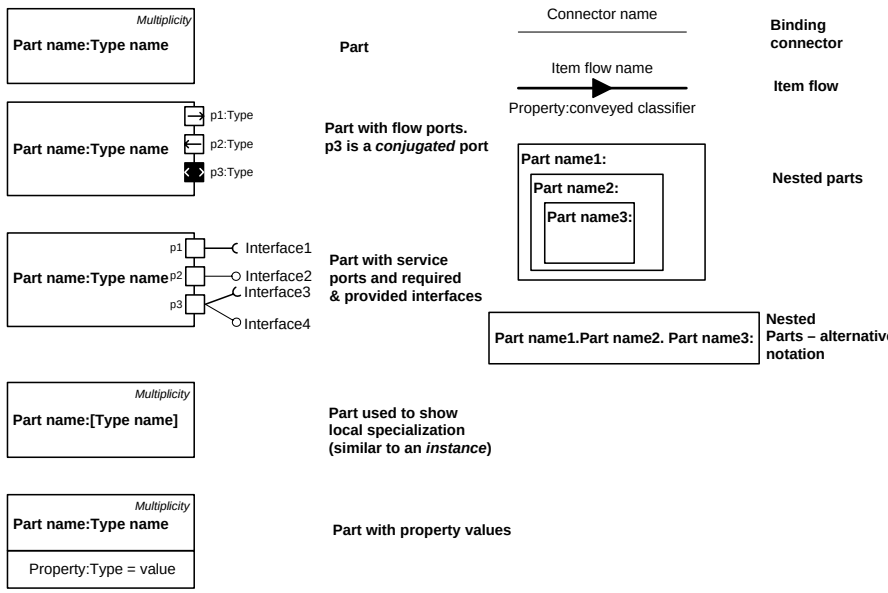

Figure 4.20 Graphical symbols for the internal block diagram elements

The diagram in Figure 4.20 shows the symbols for each of the elements in an internal block diagram. If a part has a multiplicity greater than one, this is shown in the top right corner of the part. Parts can also be nested and this can be shown in two ways: either by drawing parts inside each other, or by using a 'dot' notation. This is shown in Figure 4.20, with 'PartName3' nested inside 'PartName2', which is itself nested inside 'PartName1'. Drawing the parts inside each other makes this nesting visually explicit. The more compact 'dot' notation uses a single rectangle labelled

'PartName1.PartName2.PartName3'; reading from right to left indicates how the parts are contained in one another.

Ports are shown on parts in the same way as for blocks (see 4.5.2 above). Parts and ports are connected using binding connectors or item flows using the graphical paths shown. The values of the properties of a part (which are the same properties as are present in the block that types the part) can be shown in a separate compartment in order to specify default values or when modelling *local specializations.*

In a somewhat strange omission, SysML does not include the UML concept of an *instance*. In UML an instance represents a real-world example of a class and has an associated *object diagram*. While earlier versions of the SysML specification included this (but renamed as the *instance diagram*), the latest version does not. SysML does, however, have the concept of a local specialization, which goes some way to addressing this significant omission. A local specialization is shown by enclosing the type of a part in square brackets. A local specialization can be considered as lying between the concepts of a block and an instance, something often referred to as a *prototypical instance*. Consider the diagrams in Figures 4.21 and 4.22 (and note that the types of the properties have been omitted for clarity).

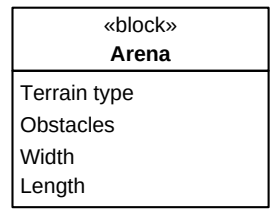

Figure 4.21 Block definition diagram (fragment) showing definition of the block 'Arena'

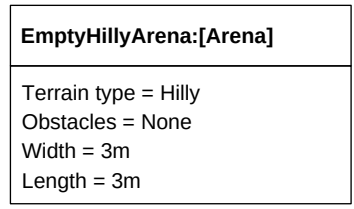

Figure 4.22 Internal block diagram (fragment) showing local specialization of a part of type 'Arena'

Figure 4.21 defines a block, 'Arena', with four properties. In Figure 4.22 a part, 'EmptyHillyArena', is defined and is shown as being of type 'Arena'. The type name is enclosed in square brackets showing that it is a local specialization and all the properties have values assigned to them. This can now be considered as a prototypical instance of the block 'Arena', that is, as one possible way of constructing an 'Arena'. Different local specializations could be modelled that assigned different values to the properties. Each of these local specializations would represent a different

way of constructing an ‘Arena’. However, none of these would represent an actual ‘Arena’; in SysML they are *not* considered to be instances. Nevertheless, this is the nearest that one can get to modelling instances using ‘standard’ SysML notation. One possible way of clarifying whether a local specialization is intended to represent an instance would be, perhaps, to mark the part with the stereotype ≪instance≫. This is, however, not part of the SysML specification but one possible way of addressing SysML’s inexplicable omission of the concept of the instance.

4.6.3 Examples and modelling – internal block diagrams

A common use of the internal block diagram is to show the internal structure of a block with the connections between the parts that make up that block. Revisiting the example block definition diagram showing the structure of a robot (see Figure 4.23) allows us to do this.

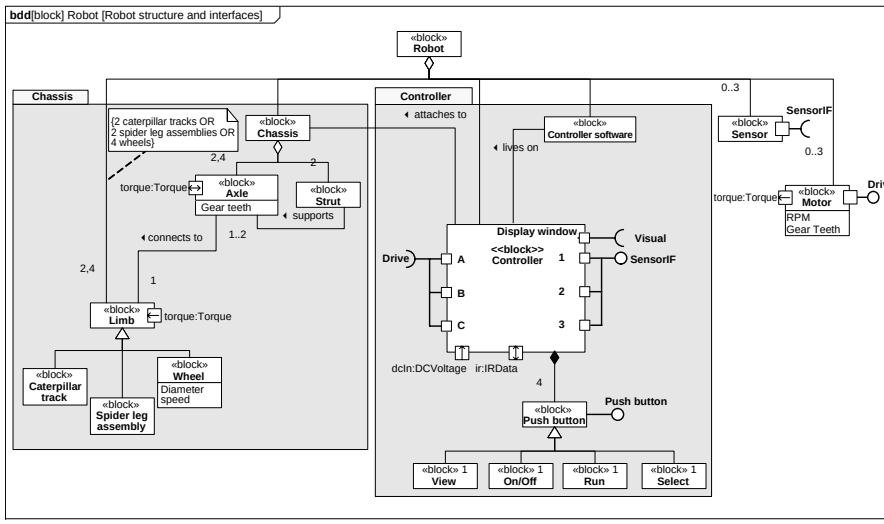

Figure 4.23 Block definition diagram showing the structure of a Robot

Figure 4.23 shows the structure of a robot, with packages bounding those elements that make up the chassis and the controller. Many of the blocks have ports attached showing the services they provide or require and the energy, data or material they transfer between each other. Also note that the robot can be fitted with three different types of ‘Limb’. What if we want to model a robot with four ‘Wheel’, and we want to show the transfer of ‘Torque’ between the various blocks that form part of the ‘Chassis’? We can do this with an internal block diagram, as in Figure 4.24.

Figure 4.24 shows how the wheels are connected to axles, which are themselves connected to struts. It also shows how the two motors transfer torque to the axles and hence to the wheels. The transfer of torque is shown using *item flows*, which are represented by the labelled triangles on each of the associations. The item transferred

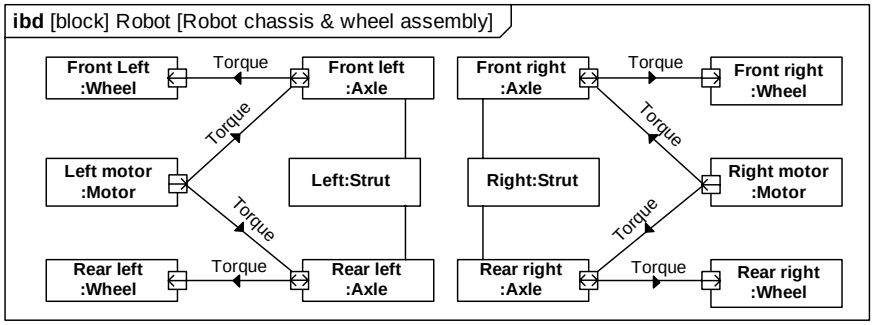

Figure 4.24 Internal block diagram showing the robot chassis and wheel assembly

must be the same type as, or a subtype of, the type used in the definition of each of the flow ports, and the flow direction must correspond to that indicated by the flow port.

Showing some of this information on the block definition diagram would have been possible, but would have resulted in a very cluttered diagram and would likely have obscured the structural connections between the blocks. Using an internal block diagram avoids these problems and emphasizes the relationships between the component parts.

This diagram shows only one way of building a robot as described by the block definition diagram in Figure 4.23, since that diagram shows that a robot could be assembled with spider-leg assemblies rather than wheels. For this reason it could be argued that this internal block diagram should show local specializations of all of the parts, enclosing each type in square brackets. However, this is not quite a prototypical instance, as we are not specifying the values of any of the properties of the parts. For this reason the marking of each part as a local specialization has not been done. Lack of a clear explanation of this notation within the SysML specification does raise questions of its validity and intended meaning.

As an example that illustrates the setting of property values, consider Figure 4.25.

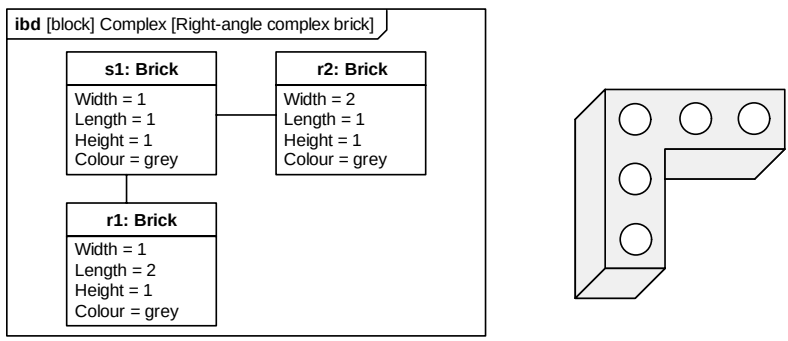

Figure 4.25 Internal block diagram defining the structure of a complex brick (shown to the right)

The use of block definition diagrams to define the structure of a library of parts (in the robot, not the SysML, sense) was discussed in 4.5.2. The diagram in Figure 4.25 shows how the brick on the right may be modelled using an internal block diagram. Here, the various properties of the parts have been assigned values in order to define the exact structure of the complex brick. Again, it could be argued that these parts should be modelled as local specializations, even more so in this case, as we *are* setting property values.

Another omission from SysML is the UML *deployment diagram* that shows how various elements of a system are deployed. The SysML specification defines an *allocation* notation using stereotypes and versions of the comment to show how various elements are allocated to and from other elements.

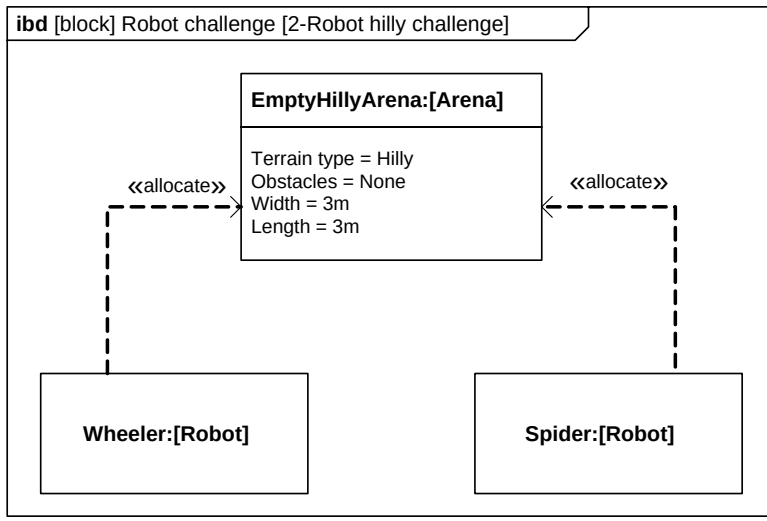

Figure 4.26 Internal block diagram showing local specialization and allocation

Figure 4.26 shows how two different versions of a robot can be deployed to an arena to represent an instance of a robot challenge game. This is done using the ≪allocate≫ dependency, with the robots and the arena being modelled as local specializations. The intention here is that this diagram shows an actual instance of such a game involving two different instances of a robot. The earlier comments on the local specialization notation are particularly relevant here. It is impossible to tell from the diagram whether the parts represent 'real' instances (the intention) or just prototypical instances – possible examples of one such game. This is yet another example of the nonsensical omission of instances from the SysML.

Another example of allocation is shown in Figure 4.27.

The 'ControllerSoftware' that handles the robot's behaviour can be realized in a variety of programming languages, such as Java and C++. The 'ControllerSoftware' is deployed onto hardware modelled as the 'Controller'. Figure 4.27 shows two internal

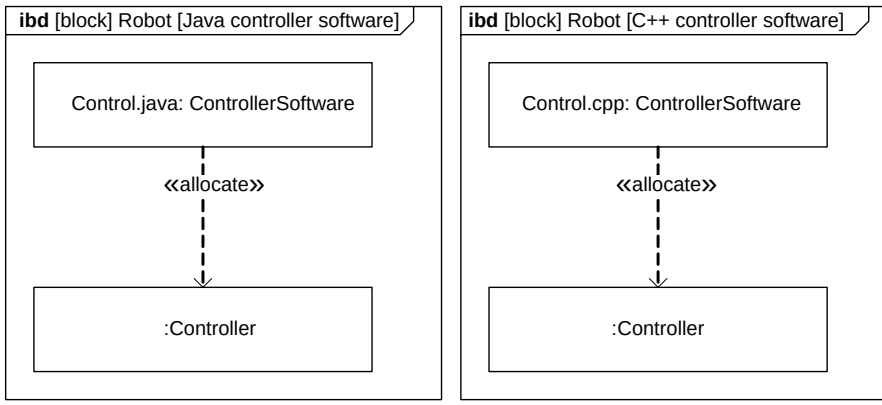

Figure 4.27 Internal block diagrams showing allocation of software to hardware

block diagrams that model the deployment (allocation) of these Java and C++ versions onto the hardware, shown by the two different diagrams.

4.6.4 Using internal block diagrams

The internal block diagram is very strongly related to the block definition diagram, using parts to show the structure of a complex block by, effectively, representing the whole block as its own diagram. 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 (such as aggregation and composition). This is particularly useful for showing elements within an architecture of a system.

Internal block diagrams are also the best, but far from ideal, way in SysML of representing instances and deployment. Both of these concepts had their own diagrams in UML but have been omitted from SysML. From the authors' point of view, the lack of instances and deployment are significant omissions from SysML and omissions that they hope will be addressed in future revisions of the language, although this prompts the question of *why* to bother with SysML in the first place if it needs changing to be more like UML in order to be used successfully.

4.7 Package diagrams (structural)

4.7.1 Overview

A package diagram, as the name implies, simply 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.

4.7.2 Diagram elements

The syntax for the package diagram is very simple and can be seen in Figure 4.28.

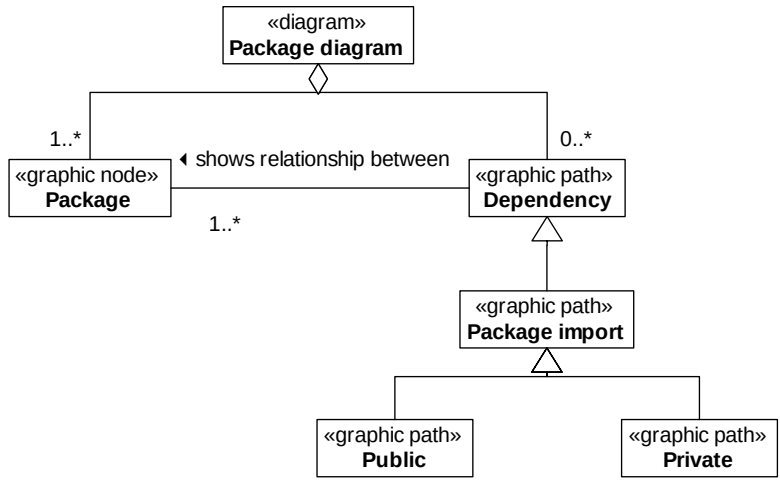

Figure 4.28 Partial meta-model for package diagrams

The diagram in this figure shows the partial meta-model for the package diagram. It can be seen that there are two main elements in the diagram – the 'Package', which happens to be a graphical node, and the 'Dependency', which happens to be a graphical path. There is one type of 'Dependency' defined – the 'Package import'. The 'Package import' has two types, which are 'Public' and 'Private'.

The graphical notation for the package diagram is shown in Figure 4.29.

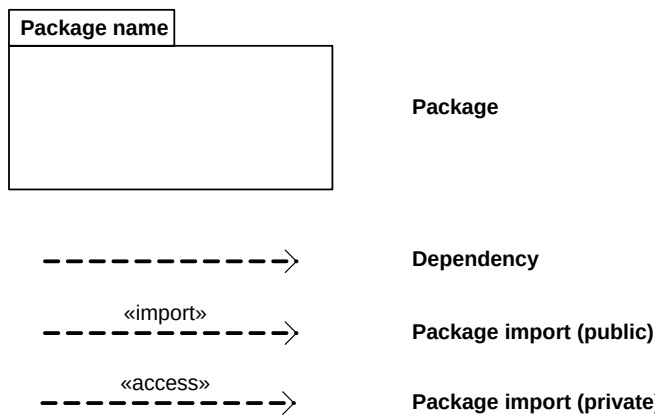

Figure 4.29 Graphical symbols for elements in a package diagram

The diagram in Figure 4.29 shows that there are really only two symbols on the diagram: the graphical node representing a package and the graphical path representing a dependency.

The graphical node representing the package is 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 main graphical path, the dependency, has two stereotypes defined – ≪import≫ and ≪access≫. An ≪import≫ or ≪access≫ means that the package being pointed to (target) is imported into the other package (source), with the target package remaining its own package as part of the source package. Any element-name clashes are resolved with the source package taking precedence over the target package.

≪import≫ and ≪access≫ differ in the *visibility* of the information that is imported. ≪import≫ indicates that a *public* import is being performed. ≪access≫ indicates that a *private* import is being performed. What does this mean? Consider the two examples in Figure 4.30.

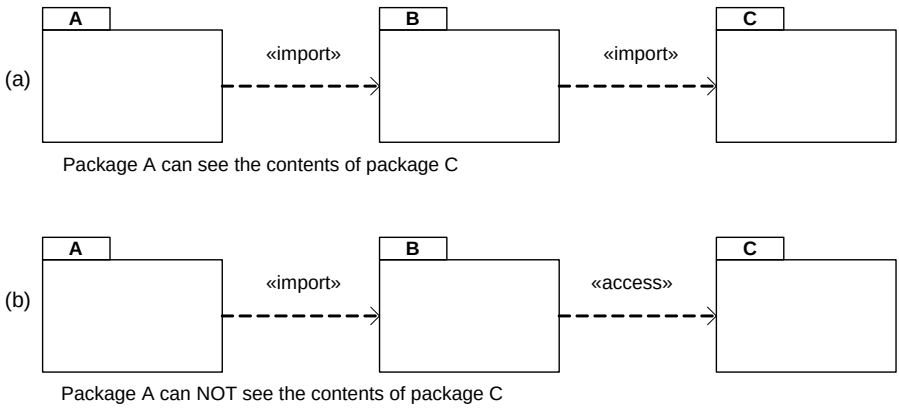

Figure 4.30 Importing packages using ≪import≫ and ≪access≫

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

In example b) package B imports the contents of package C using the private ≪access≫ dependency. Package A then imports the contents of package B using the ≪import≫ dependency. Since A has imported B and B has *privately* imported C, package A *cannot* see the contents on package C.

The diagram in Figure 4.31 shows that a 'Package' is made up of a number of 'Packageable element'. In the SysML, almost any element can be enclosed within a

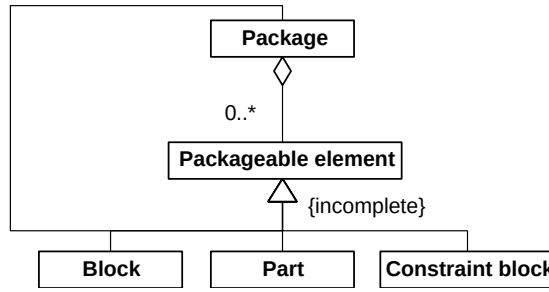

Figure 4.31 Relationships between package diagram elements and the rest of the SysML

package, so only a few examples are shown here (note that this is indicated by the {incomplete} constraint). Note that a 'Package' is itself a 'Packageable element' and thus a package can contain other packages.

4.7.3 Examples and modelling – package diagrams

Package diagrams are typically used to show relationships within a model at a very high level. For example, it is possible to define a particular pattern in a model and then to reuse this either in the same model or in other models. Packages are also useful to show libraries of parts that are used in the production of products.

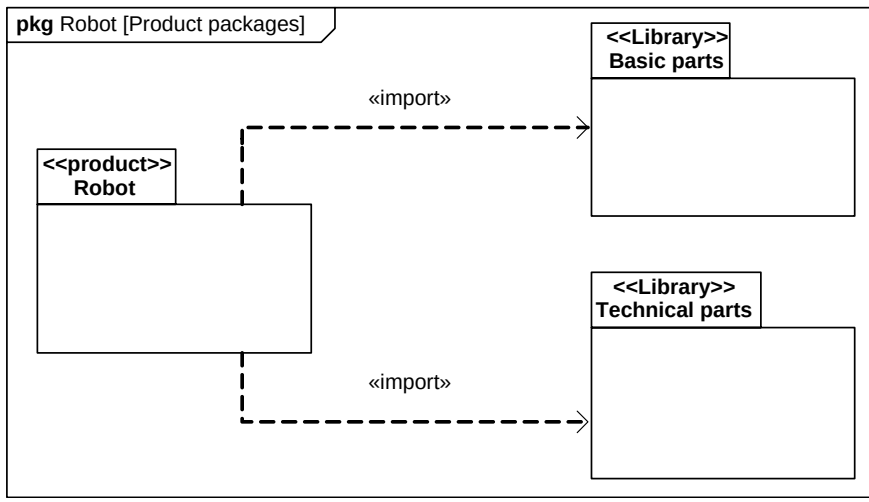

Figure 4.32 Packages and their relationships for the robot challenge

The diagram in Figure 4.32 shows a package named 'Robot', which happens to be a product that uses parts from two libraries in its construction: a 'Basic parts'

library and a 'Technical parts' library. The use of these parts libraries is shown by the ≪import≫ dependencies. The specification of the parts that make up these libraries could be defined on a block definition diagram. Examples of such library definitions are shown in Figure 4.33.

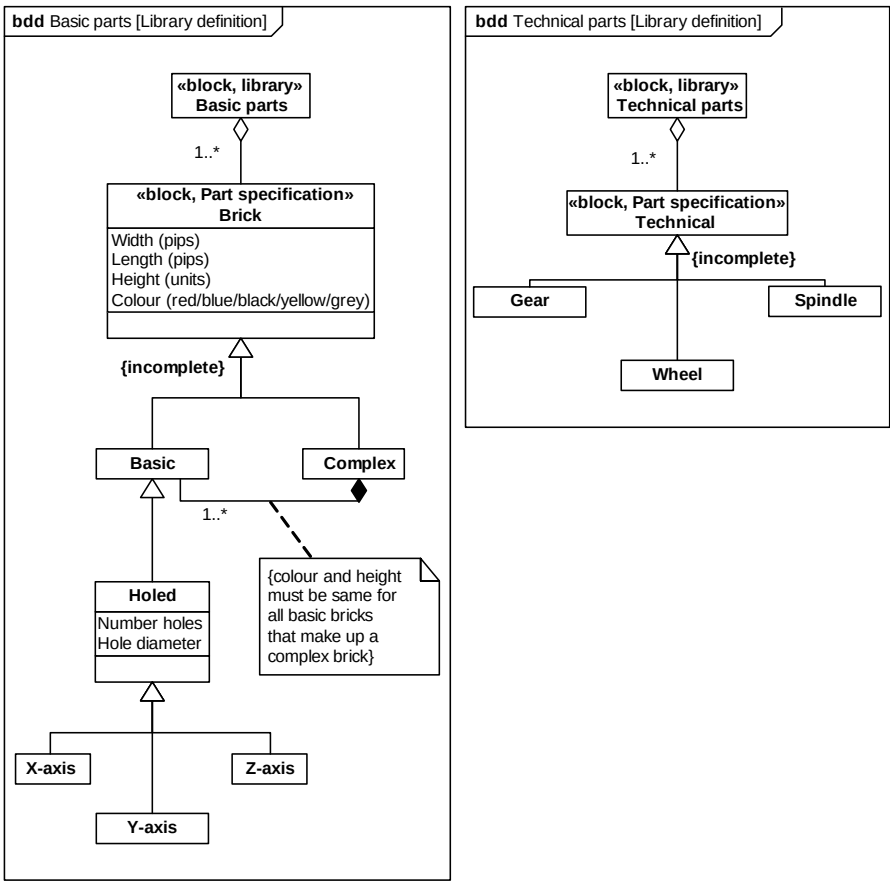

Figure 4.33 Example of the definition of a library of parts

The diagram in Figure 4.34 shows another package diagram that is being used to show which profiles are being used for a particular project. A profile is a set of additions, such as stereotypes, constraints and diagram extensions, which are used to tailor the SysML language for a particular application or domain. Indeed, the SysML can be considered to be a profile of UML, tailoring it for the systems engineering domain.

It can be seen that there is a package named 'Robot' (which happens to be a project) that imports packages called 'SysML' and 'Process model', which both happen to

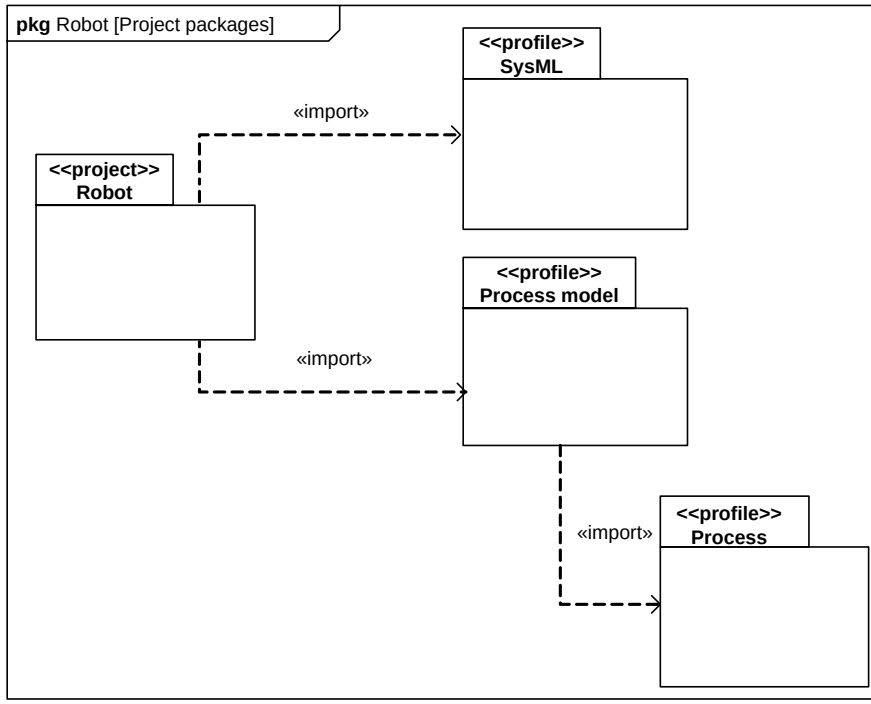

Figure 4.34 Example of a package diagram used to represent various profiles

be a profile. In turn, the Process model' package imports another package named 'Process', which happens to be a profile.

4.7.4 Using package diagrams

The main use for package diagrams is to show high-level relationships between groups of things in a model – particularly powerful for reuse of patterns and declaring profiles. Of possibly greater use is the use of packages on other diagrams, such as block definition diagrams, to show groupings of model elements. Figure 4.11 is such an example.

4.8 Parametric diagrams (structural)

4.8.1 Overview

SysML has introduced a new concept, that of the *constraint block* and the associated 'parametric diagram', that is not present in UML. Constraint blocks 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. The following sections give an overview of the concepts of constraints.

4.8.2 Diagram elements

Constraint blocks are defined on block definition diagrams and used on parametric diagrams. The parametric diagram is a restricted form of the internal block diagram.

To define a constraint block a special form of a block is used, as shown in Figure 4.35.

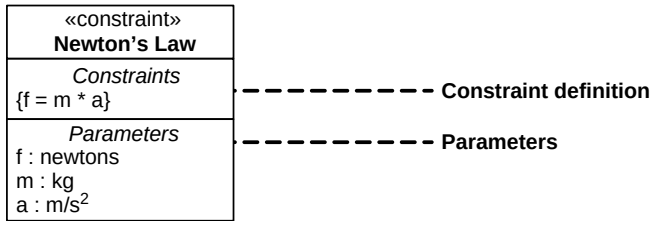

Figure 4.35 Graphical representation used in the definition of a constraint block

The constraint block is defined using a block stereotyped ≪constraint≫ and 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. The example in Figure 4.35 defines a constraint called 'Newton's Law', which 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.

Parametric diagrams are made up of one or more 'Constraint block', zero or more 'Part' and one or more 'Connector'. The 'Constraint block' is used to show which constraints are being used. These constraint blocks are linked to zero or more 'Part' or 'Constraint block' via a *binding* 'Connector'. Linking constraint blocks to parts allows the source of *input* parameters to be specified, with the value that a part takes supplying the value for an input parameter, or allows the values of *output* parameters to constrain the values that a part can take. Linking a constraint block to another one allows *networks* of constraints to be created, with the outputs of one constraint block forming the inputs to others. The partial meta-model for the parametric diagram is shown in Figure 4.36.

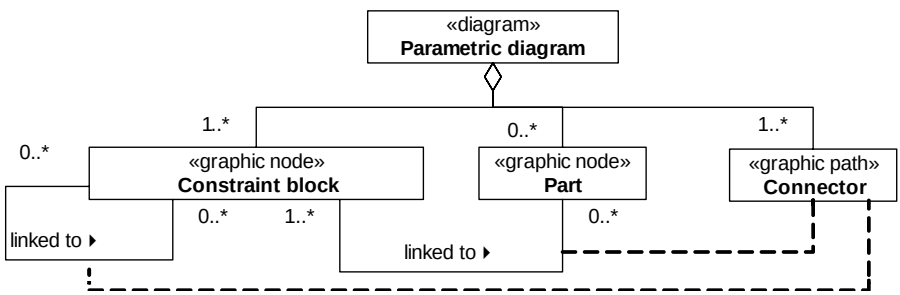

Figure 4.36 Partial meta-model for parametric diagram

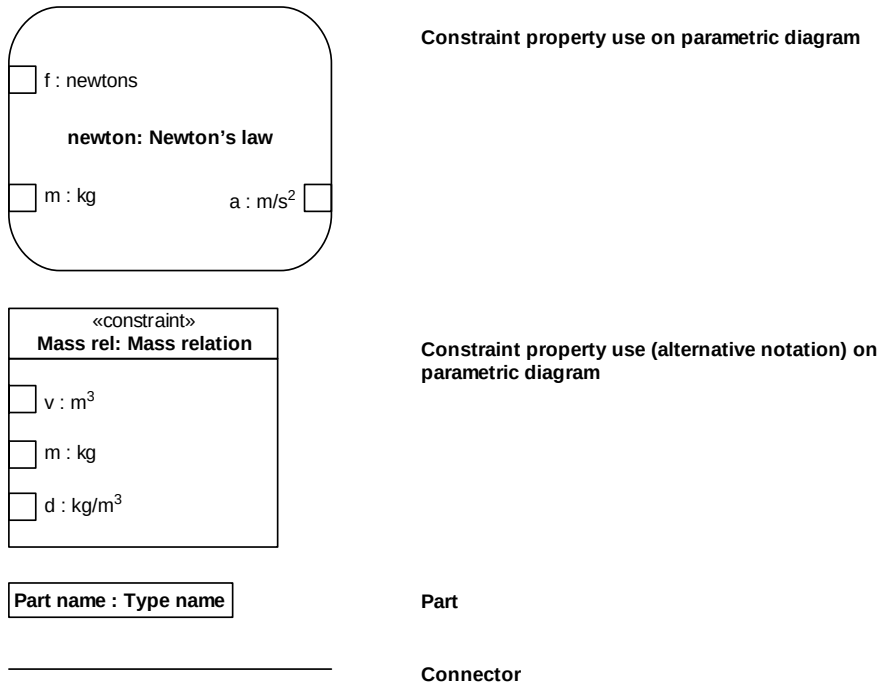

Figure 4.37 Graphical representation of elements in a parametric diagram

The notation used on parametric diagrams is shown in Figure 4.37.

The use of a constraint block on a parametric diagram can be shown in two ways, either as a round-cornered rectangle or as a rectangle with the stereotype ≪constraint≫. There is no difference in meaning. The symbols simply offer alternative notations. Small rectangles attached to the inside edge of the constraint represent each parameter and provide connection points when linking constraints to parts or other constraints. When used on a parametric diagram, a constraint block is referred to as a constraint *property* and each constraint property should be named *name*:*Constraint name*. This allows multiple copies of a constraint to be used on a diagram.

4.8.3 Examples and modelling – parametric diagrams

The robot project can be used to provide examples of the definition and use of parametric constraints. Figure 4.38 gives a definition of two constraints, which are used in Figure 4.39.

The motors driving the robot's wheels have small gears attached, which drive gears attached to each wheel axle. Different sizes of wheels can be used. Constraints can be defined that allow the linear speed of the robot to be calculated from properties of the motor, the gears used and the size of the wheels.

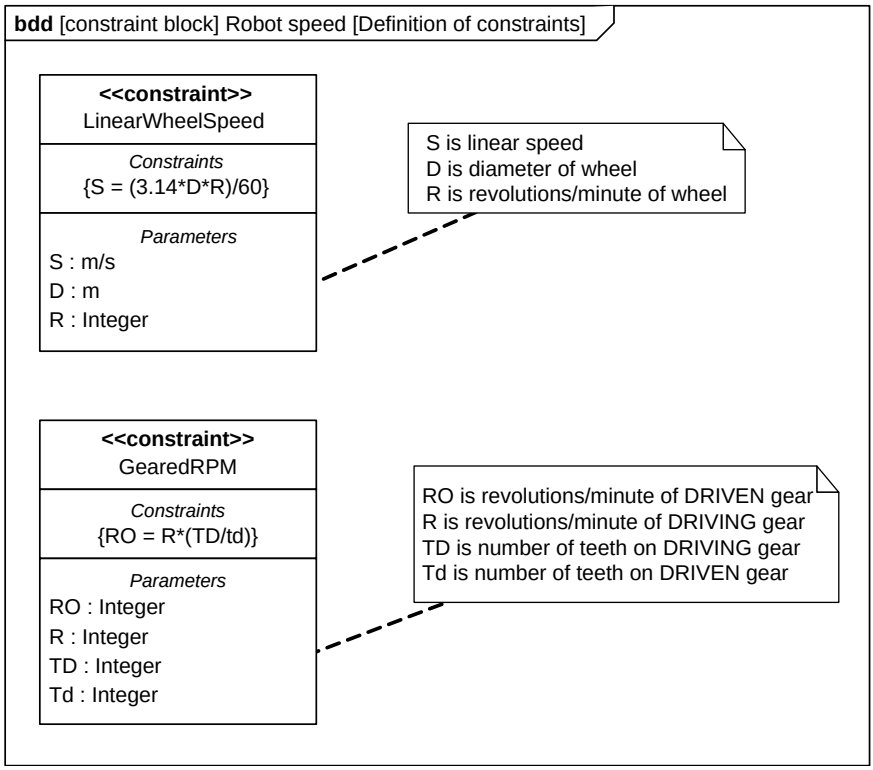

Figure 4.38 Block definition diagram showing the definition of two parametric constraints

Figure 4.38 defines two constraints. 'LinearWheelSpeed' defines how linear speed can be calculated from the diameter of a wheel and its rotational speed. 'GearedRPM' shows how the number of rotations per minute of a driven gear can be calculated from the rotational speed of the driving gear and the sizes of the two gears.

The two constraints defined in Figure 4.38 are used as shown in Figure 4.39. The number of teeth on an axle supply the input value for parameter 'Td', with the motor speed and number of teeth on the motor supplying the input values for parameters 'R' and 'TD'. These parameters are used by the constraint property 'AxleRPM:GearedRPM' to calculate the rotational speed, 'RO', of the driven gear connected to a wheel. This output forms an input to the parameter 'R' of the 'Robot-Speed:LinearWheelSpeed' constraint property, which uses the diameter of the wheel, 'D', to calculate the linear speed of the wheel, 'S', and hence the robot.

Note that SysML does not distinguish between *fixed* and *free* variables, that is between the parameters that are used as *inputs* and those that are considered *outputs*. In the example above, parameter 'S' can be considered as an output from the network of constraints. Another use of the same constraints could be to determine the size

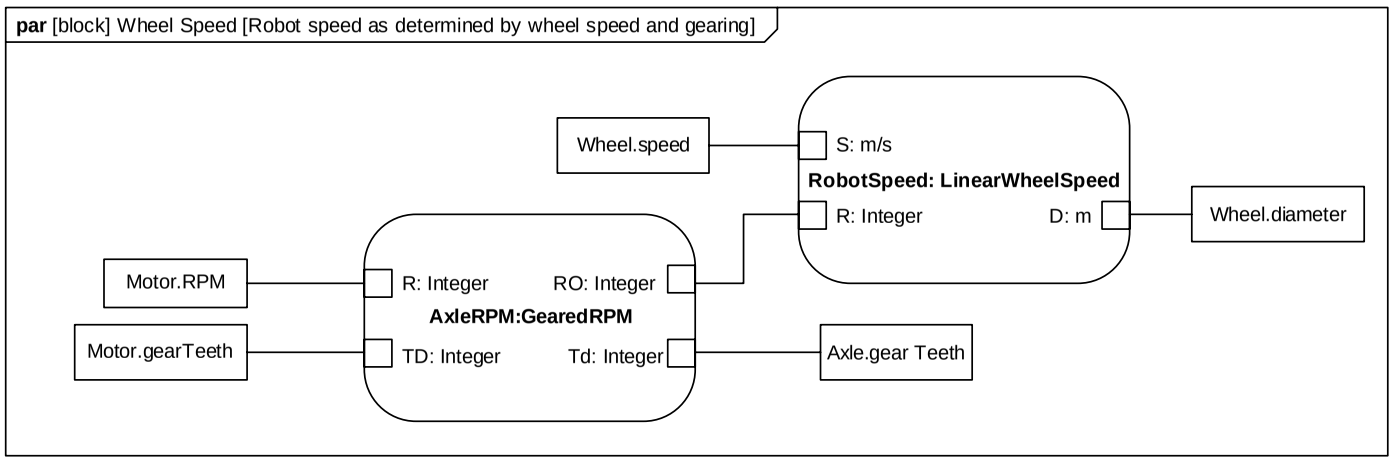

Figure 4.39 Parametric diagram showing the use of parametric constraints

of wheel needed to achieve a given linear speed from given motor speeds and gear sizes. In this case parameter 'S' would be an input with 'D' being an output. If it is important that the fixed and free nature of parameters be shown in a model, then stereotypes could be used to indicate this on parametric diagrams. Such an example is left as an exercise for the reader.

4.8.4 Using parametric diagrams

Constraint blocks and parametric diagrams provide a powerful notation that allows aspects of a system to be constrained in a rule-based fashion that can relate the properties of different parts of a system. While the SysML specification describes constraint blocks in terms of physical constraints that are represented by mathematical equations, the concepts behind constraints can be extended to cover general rules that constrain system properties and behaviour, and that can also be used in the verification and validation of a system's requirements. This is considered in much greater detail in Chapter 5, which gives a worked example showing different uses of constraints and discusses the links between constraints and other aspects of a system's model.

4.9 Requirement diagrams (structural)

4.9.1 Overview

One of the new diagrams added to SysML that are not present in UML is the 'requirement diagram', which is intended to be used to represent system requirements and their relationships. Before getting excited over this new addition, it is worth pointing out two things: first, the requirement diagram is little more than a tailored block definition diagram consisting of a stereotyped block and a number of stereotyped dependencies and fixed-format notes; secondly, SysML (and the UML) already has a diagram that has been used by many practitioners for modelling requirements for a number of years, namely the use case diagram. For this reason the usefulness of the requirement diagram is unclear at present.

It is also worth noting that the SysML specification [1] does not class the requirement diagram as a structural diagram, but rather classifies it as a 'crosscutting construct' since the elements that can appear on a requirements diagram can also appear on other diagram types. While this is true, requirements, as noted above, are essentially blocks. For this reason the requirement diagram has been classed as a structural diagram in this book.

4.9.2 Diagram elements

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

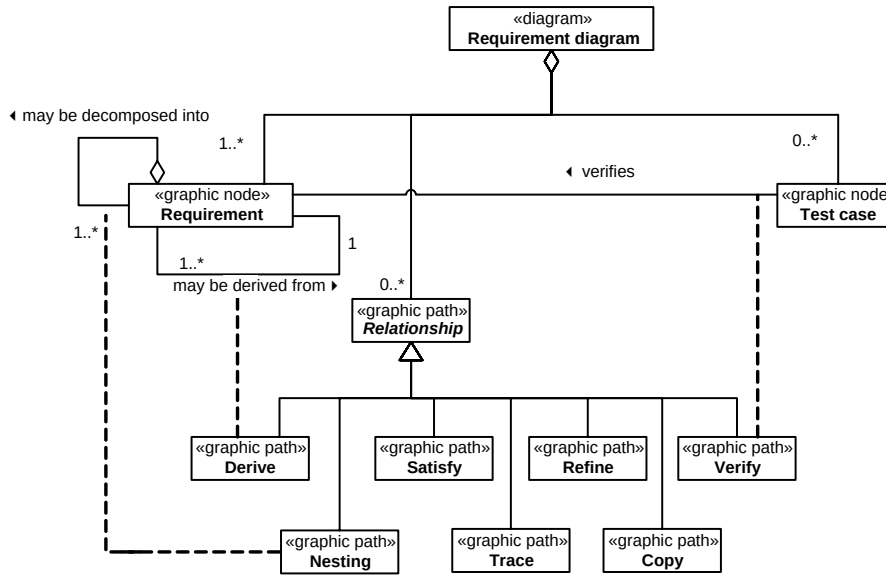

Figure 4.40 Partial meta-model for requirement diagram

Figure 4.40 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'.

Three of the 'Relationship' types are used to show how one 'Requirement' is related to another, namely the 'Nesting', 'Derive' and 'Copy' relationships.

- *'Nesting'* – a 'Requirement' may be decomposed into one or more 'Requirement' and this decomposition relationship is shown with a 'Nesting' relationship.
- *'Derive'* – one or more 'Requirement' may be derived from another 'Requirement' and this derivation is shown with a 'Derive' relationship.
- *'Copy'* – it may be that a 'Requirement' needs to be reused in a different context. The copied 'Requirement' can be related to the original using the 'Copy' relationship to indicate that the original is the master. In effect, the copied 'Requirement' is a read-only version of the original requirement: the copy cannot be changed but, if the original changes, then so will the copy.

The 'Verify' relationship is used to show how a 'Test case' verifies a 'Requirement' and so can be used only 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 that 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 other types of 'Relationship' can all be used to connect any type of model element to a 'Requirement' and are used in the following ways.

- 'Satisfy' 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.
- 'Refine' is used to show how a model element can be used to further refine a 'Requirement'. For example, a use case that is developed to refine a requirement would be linked to the requirement using a 'Refine' relationship.
- 'Trace' is used to show that a model element can be traced to a 'Requirement'. This provides a general-purpose relationship that allows model elements and requirements to be related to each other.

These various types of relationship allow the modeller to relate explicitly different parts of a model to the requirements as a way of ensuring the consistency of the model. It must be noted that the SysML specification urges modellers to use the other types of relationship in preference to the '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.

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes and illustrated in Figure 4.41.

Although the various types of 'Relationship' are all shown as being graphical paths in Figure 4.40, they can, with the exception of the 'Nesting' relationship, also be shown as graphical paths using the 'note' symbol as shown in Figure 4.41. This 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, but it can lead to inconsistency, particularly when modelling is not being done using a tool (or is being done using a tool that does not enforce consistency). Using the stereotyped dependencies gives an immediate and consistent indication of the relationship, since the two elements are explicitly connected by the dependency. Using the notes method 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.

4.9.3 Examples and modelling – requirement diagrams

This section presents some examples of requirement diagrams and related diagramming elements, again using examples from the robot project.

The first example shows how a requirement can be decomposed into lower-level requirements.

Figure 4.42 is a requirement diagram that shows the requirement 'Meet project constraints' decomposed into three lower-level requirements using the 'Nesting' relationship. Within the 'Robot project context' package, a use case 'Meet project constraints' and associated actor 'Sponsor' have been modelled, which can be traced to the main requirement as shown by use of the ≪trace≫ dependency. The figure

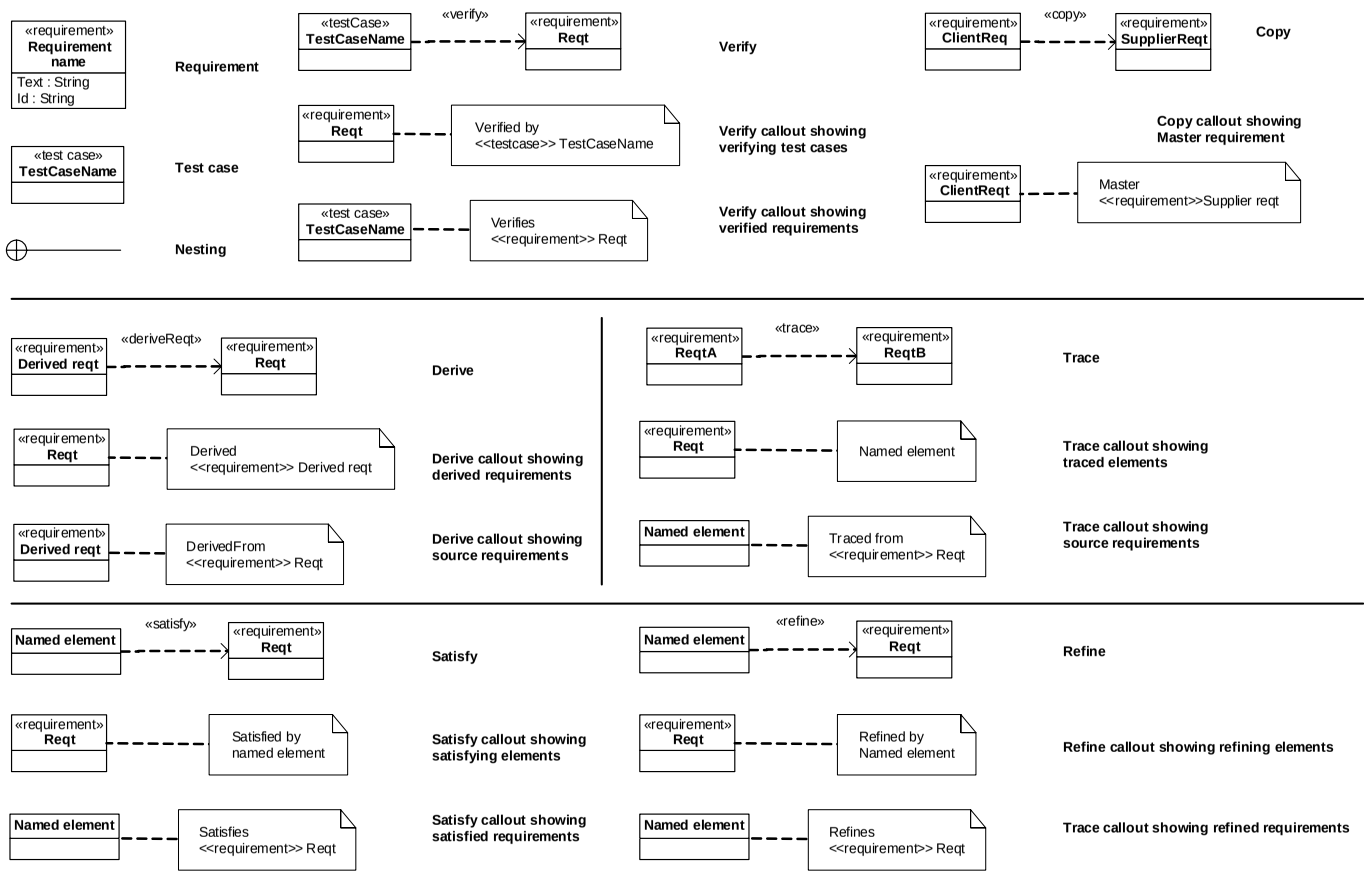

Figure 4.41 Graphical representation of elements in a requirement diagram

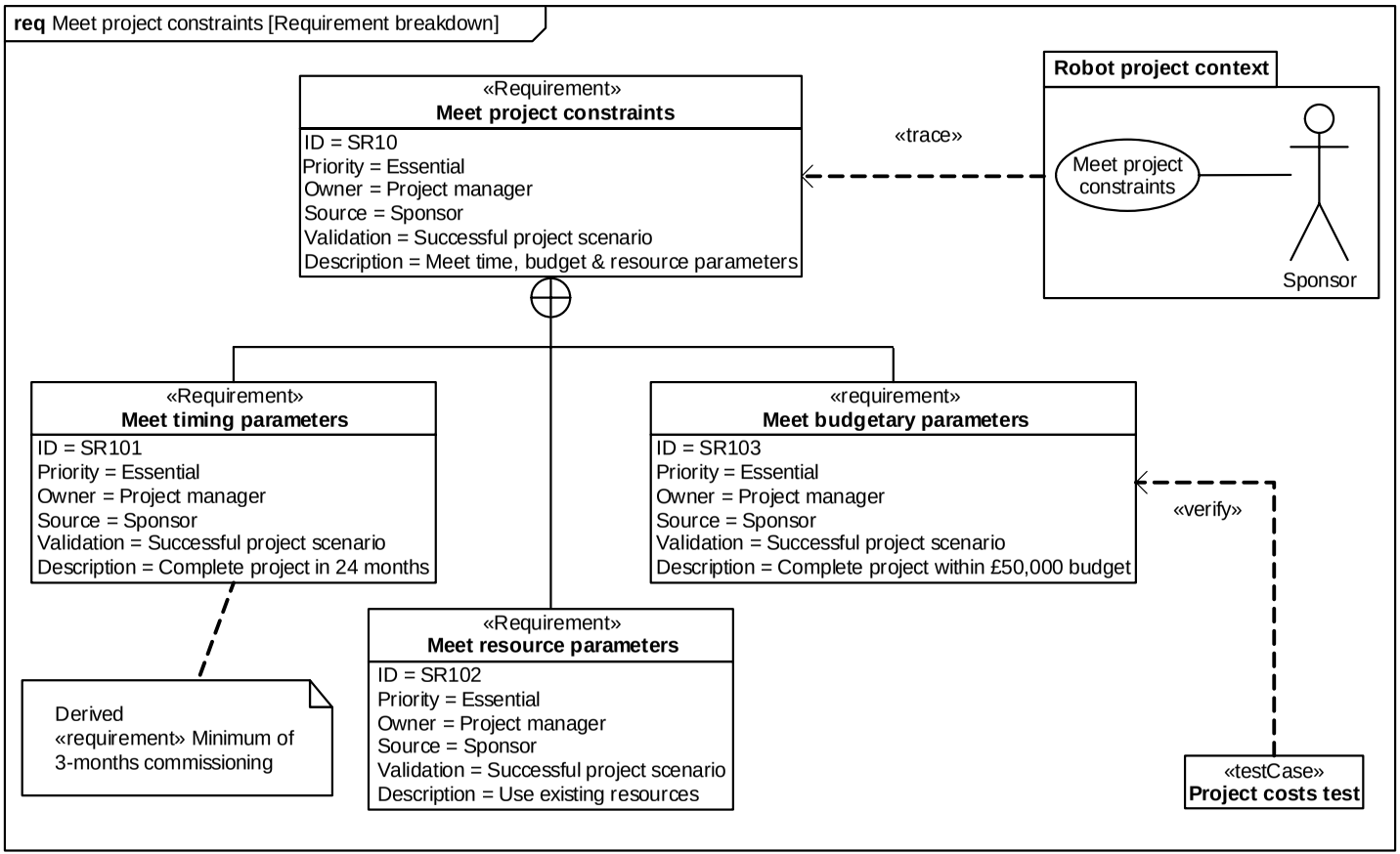

Figure 4.42 Requirement diagram for project constraints

also shows that the requirement ‘Meet timing parameters’ has a requirement derived from it called ‘Minimum of 3-months commissioning’. This derived requirement is shown in a separate requirement diagram, Figure 4.43.

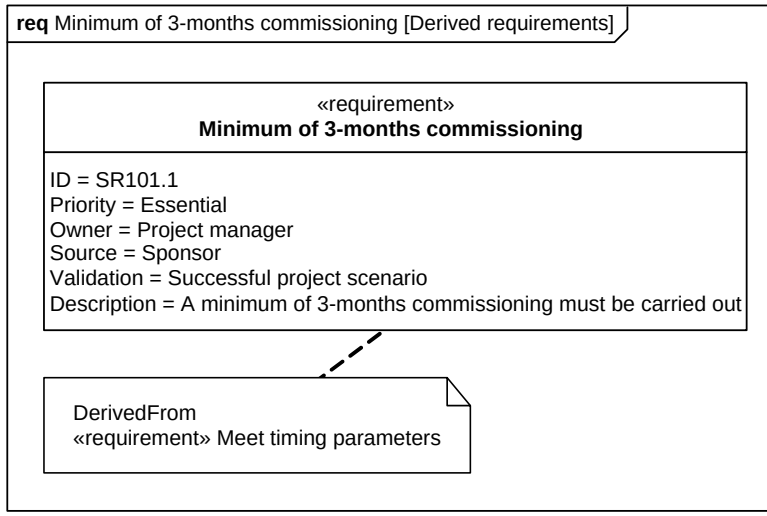

Figure 4.43 Requirement diagram showing derived requirement

In Figure 4.43, the derived requirement referenced in Figure 4.42 is shown, and is linked to the original requirement from which it is derived by use of the ‘DerivedFrom’ note. As discussed above, it is essential that, when using this ‘callout’ notation, both ends of the implicit relationship be indicated in the model in order to ensure consistency.

Care should be taken when using the ‘Derived’ and ‘DerivedFrom’ notations, as some people may find them unclear. Seeing a requirement with the word ‘Derived’ attached to it may lead readers of the diagram to assume that the requirement is a derived requirement, which is not the case.

The ≪testCase≫ ‘Project costs test’ from Figure 4.42 is shown in Figure 4.44.

Figure 4.44 shows an activity diagram (activity diagrams are discussed in Section 4.12 below) that is marked with the ≪testCase≫ stereotype to show that the behaviour modelled in the diagram is intended to verify a requirement. Any model element marked with the ≪testCase≫ stereotype *must* return a result of PASS, FAIL, INCONCLUSIVE or ERROR (known formally within SysML as being of type VerdictKind) to indicate the result of the verification it performs.

The diagram also makes use of the ‘Verify’ callout notation to show that the test case is verifying the requirement ‘Meet budgetary parameters’. This helps to ensure the consistency of the model and corresponds to the use of the ≪verify≫ dependency connecting the test case to the requirement in Figure 4.41.

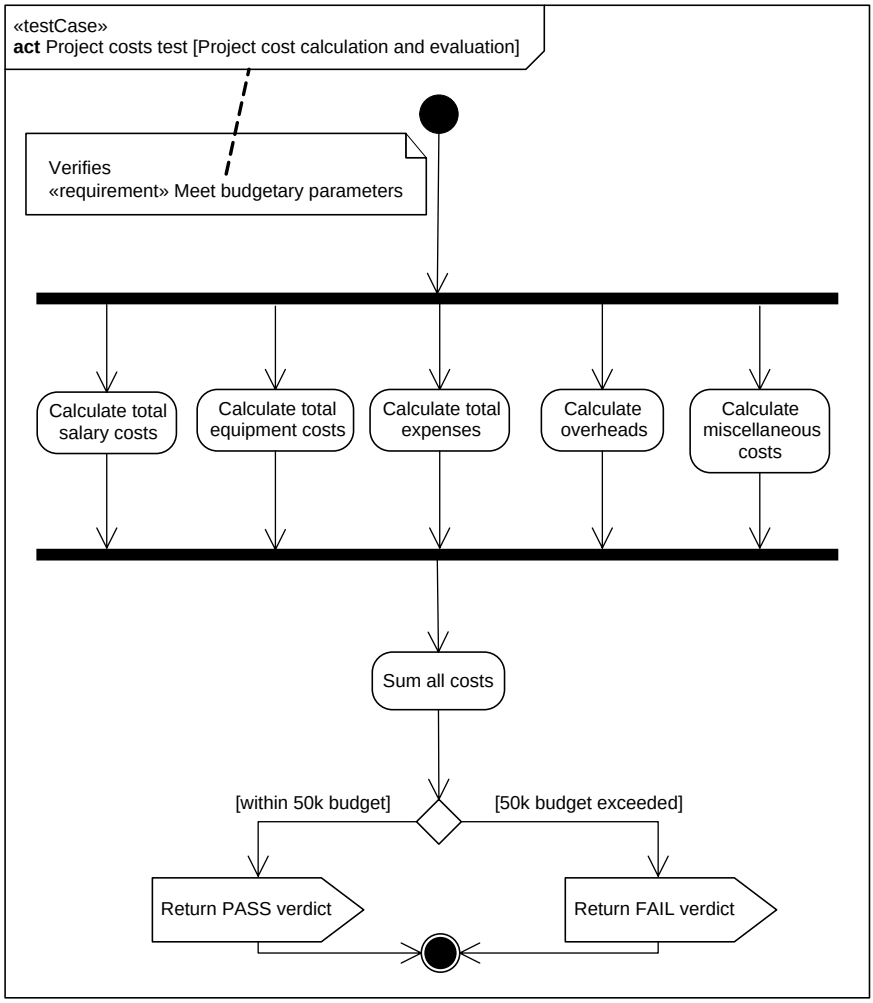

Figure 4.44 A test case realized as an activity diagram

4.9.4 Using requirement diagrams

Since SysML is a new language, the usefulness and, indeed, the use of requirement diagrams remains to be seen. While many systems engineers brought up on UML have successfully used use case diagrams for requirements modelling for many years, it is true that some practitioners have adopted an approach and notation similar to that given by the requirement diagram, using class diagrams and stereotyped classes to model requirements and the relationships between them and other model elements. Indeed, many early methodologies, such as Rumbaugh's OMT, used a similar approach (class diagrams) to modelling requirements. Since this can be done

equally well in SysML using stereotyped blocks, the need for a separate diagram type seems somewhat spurious.

A more detailed discussion of the use of requirement diagrams, along with use case diagrams, is given in Chapter 7, where requirements modelling is covered in much greater depth.

4.10 State machine diagrams (behavioural)

4.10.1 Overview

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 3 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 realize a behavioural aspect of the model and are sometimes referred to as *timing models*, which may, depending on your background, be a misnomer. In SysML, the term 'timing' refers purely to logical time rather than real time. By 'logical time' here it means the order in which things occur and the logical conditions under which they occur.

State machine diagrams model the behaviour during the lifetime of a block. It has already been stated that state machine diagrams model the order in which things occur and the conditions under which they occur. The 'things' in this context are, broadly speaking, activities and actions. Activities should be derived from block definition diagrams, as they are equivalent to operations on blocks. Indeed, one of the basic consistency checks that can be carried out is to ensure that any operations on a block are represented somewhere on its associated state machine as activities or actions.

4.10.2 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 object typed by the block is active. States may show when an activity is being carried out or when the properties of an object 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 of the block is waiting for something to happen.

When discussing state machine diagrams we again have a problem with SysML and its omission of instances. As has been discussed earlier, SysML does not include the concept of objects, unlike the UML on which it is based, in which objects represent actual examples, instances, of classes. However, state machine diagrams are related closely to objects. Every instance of a block has its behaviour described by the state machine diagram for that block. Indeed, the SysML specification itself states, 'The state machine diagram represents behaviour as the state history of an *object* in terms

of its transitions and states' (emphasis added). So we have the concept of objects occurring in SysML, but no way of expressing them directly in SysML!

Figure 4.45 shows the partial meta-model for state machine diagrams. State machine diagrams have a very rich syntax and thus the meta-model shown here omits some detail – for example, there are different types of action that are not shown.

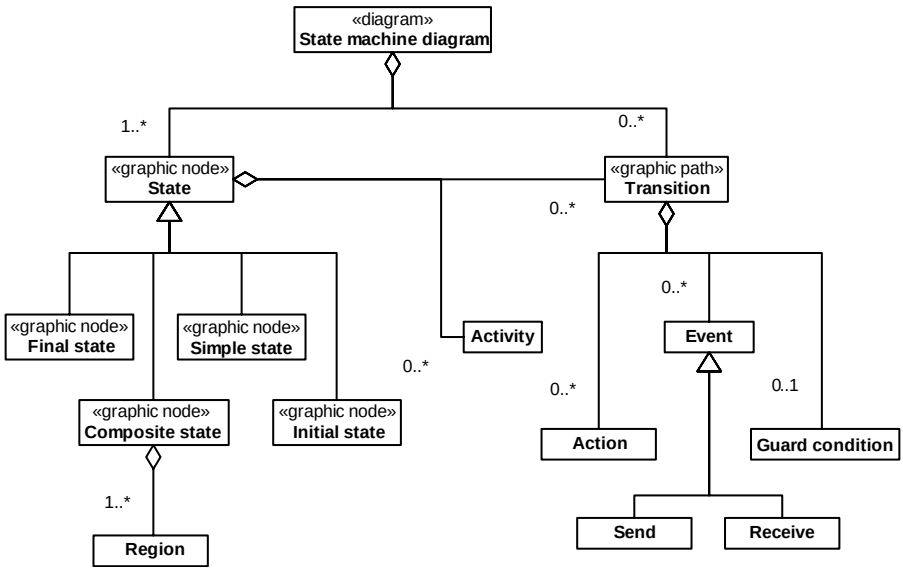

Figure 4.45 Partial meta-model for state machine diagrams

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': 'Simple state', 'Initial state', 'Final state' and 'Composite state'. Each state is made up of zero or more 'Activity'. An activity describes an ongoing, non-atomic unit of behaviour and is directly related to the operations on a block.

Each 'Transition' may have zero or more 'Guard condition', which explains how a transition may be crossed. A condition is a Boolean condition that will usually relate to the value of an attribute.

A 'Transition' may also have zero or more 'Action'. An action is defined as an activity that takes no time, and one that, depending on one's viewpoint, could be perceived as impossible. What is meant here is that the behaviour is *atomic*. That is, once started, it cannot be interrupted and will always complete. Activities, on the other hand, are *non-atomic* and can be interrupted.

Finally, a 'Transition' may have zero or more 'Event'. An event is, basically, the passing of a message from one state machine to another.

There are two main types of 'Event': 'Send' and 'Receive'. 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 message. Obviously, for each send event there must be at least one corresponding receive 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. This is analogous with component and integration testing, as it is possible to (and, in some cases, difficult not to) verify completely the functional operation of an element, yet, when it comes to integrating the elements into a system, they do not interact correctly together.

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes, as illustrated in Figure 4.46.

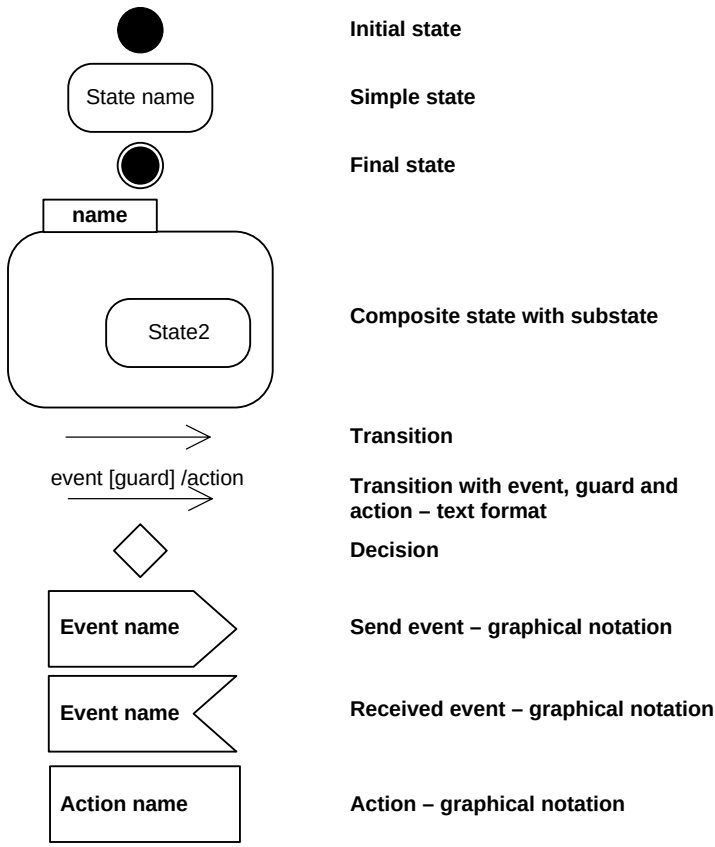

Figure 4.46 Graphical representation of elements in a state machine diagram

Perhaps the simplest and most common source of errors can be avoided by applying a simple consistency check between the state machine and its associated block. This relationship can be seen in Figure 4.47.

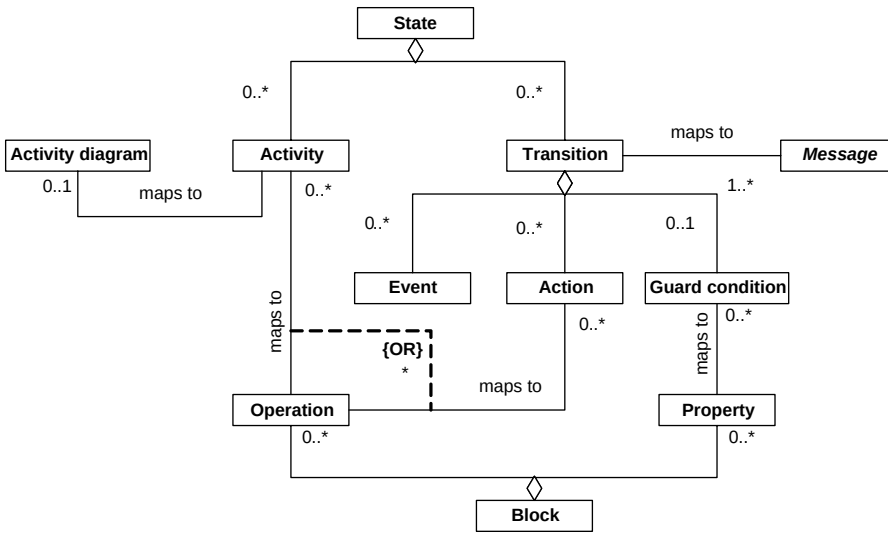

Figure 4.47 Meta-model relationship between blocks and state machine diagrams

Figure 4.47 shows the relationship between a state machine and its associated block.

Each 'Activity' in a 'State' maps to zero or one 'Activity diagram', as any activity may be decomposed into its own diagram. Each 'Transition' in a 'State' maps to one or more 'Message'. Both an 'Activity' and an 'Action' may map to an 'Operation' from a block, and a 'Property' from a 'Block' maps to zero or more 'Guard conditions'.

Therefore, the state machine diagram has consistency relationships, and hence checks, to block definition diagrams, activity diagrams and sequence diagrams.

A simple rule of thumb that will go some way to ensuring that all blocks in a system are fully defined is to say that any block that exhibits behaviour (that has operations) must have an associated state machine diagram.

4.10.3 Examples and modelling – state machine diagrams

This section looks at some practical examples of modelling using the robot challenge as examples. Unlike the discussion of state machines in Chapter 3, the emphasis here is on the actual mechanics of using state machine diagrams, rather than on the behavioural concepts that were covered previously.

Figure 4.48 shows the state machine for the 'ControllerSoftware' block from the robot challenge project. The 'ControllerSoftware' block is shown in Figure 4.15.

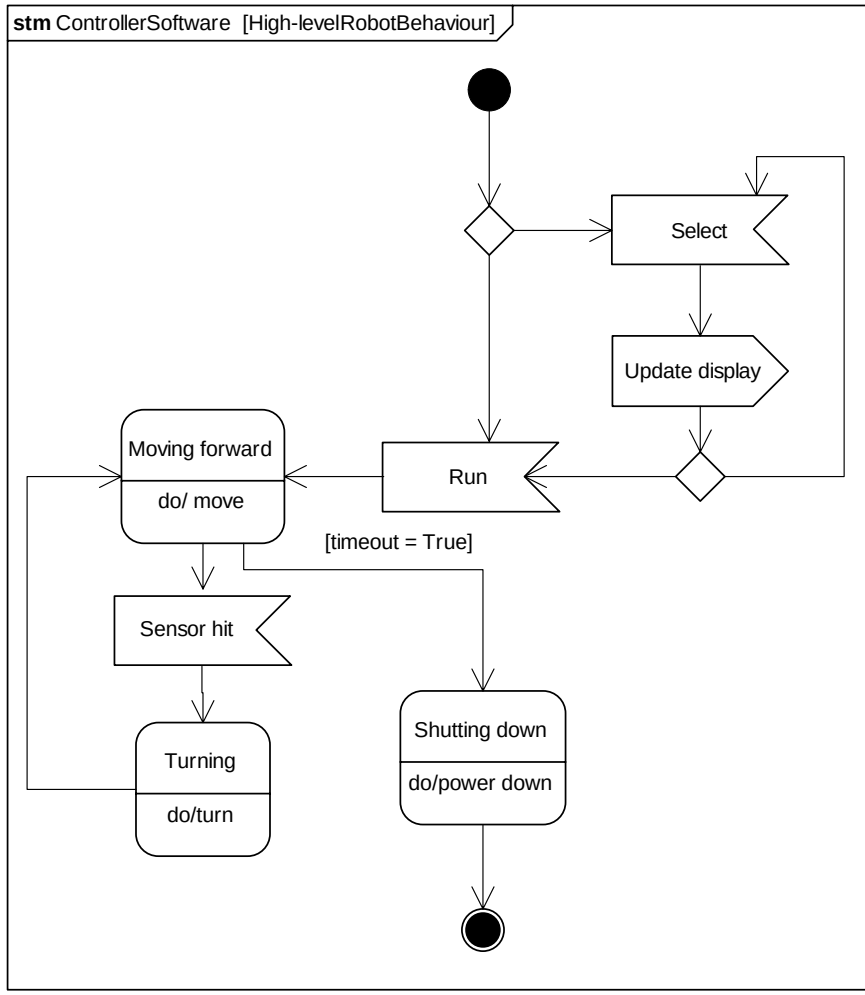

Figure 4.48 State machine for the 'ControllerSoftware' block, part of the 'Robot' block, showing high-level behaviour

Three types of state are shown here.

- An 'Initial state', realized by a filled-in circle, which indicates the creation of an instance of the block.
- A 'Final state', realized by two concentric circles, which indicates the destruction of the instance of the block.
- Simple states, indicated by rounded boxes, show what is happening in the instance at any particular point in time. It is important to ensure that the rounded boxes actually have vertical straight sides so that it is just the corners that are rounded.

The diagram also shows some additional notation.

- Three 'Receive' events, showing message coming in from outside this state machine.
- A 'Send' event, showing a message being sent out of this state machine to be used elsewhere in the system.
- Two decisions, indicated by the diamonds. A decision is also known as a *choice node* or a *choice pseudo-state*. It is a notational shorthand used where a decision about which transition to follow has to be made. Decisions can always be replaced with a combination of normal states and transitions. Figure 4.49 shows the same state machine diagram, but this time with the decision replaced with a normal state.

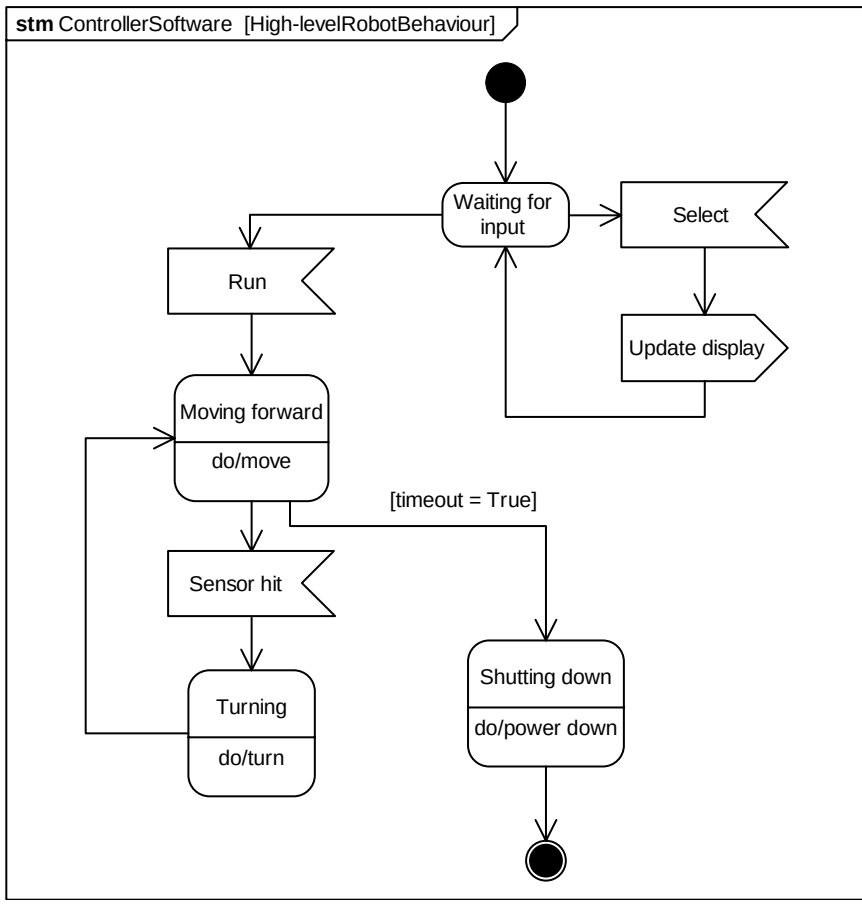

Figure 4.49 State machine for the 'ControllerSoftware' block, part of the 'Robot' block, showing high-level behaviour – no decisions

Normal states may or may not have activities or actions associated with them. Any activities or actions that are present should correspond directly to operations from the block definition diagram. Each operation from a block must exist as an activity on its associated state machine and, likewise, any activities that exist on the state machine must be present as operations on the governing block. Examples of both are shown here, where the 'moving forward' state has the 'move' activity, while the 'waiting for input' state has no activity.

The only way to go from one state to another is to cross a transition. Transitions are realized by directed lines that may have events and conditions on them. Transitions are unidirectional and, if a return path of control is required, so too is another transition.

Reading the diagram, the 'ControllerSoftware' begins its life by entering the 'waiting for input' state. It remains in this state until it receives either of the 'Run' or 'Select' messages.

If it receives the 'Select' message, indicating that the 'Select' button has been pressed (see Figure 4.11), it sends out an 'Update display' message and then returns to the 'waiting for input' state.

If it receives the 'Run' message, it enters the 'moving forward' state and executes the 'move' activity, which corresponds to the 'move' operation on Figure 4.15. Note that the behaviour of the 'move' activity is not defined here, only when it is invoked. The most commonly used method in SysML of defining the behaviour of activities (i.e. operations defined on a block) is via the *activity diagram*. These are described in Section 4.12 below, where the 'move' operation will be defined.

From the 'moving forward' state, the state machine can enter either the 'shutting down' or the 'turning' state. A condition is used to control whether the transition to the 'shutting down' state occurs. The condition is a Boolean expression and, in this case, the left-hand side of the condition relates directly to a property. This provides rather a nice consistency check back to the state machine diagram's governing block. The transition to the 'turning' state is triggered by the receipt of the 'sensorHit' message.

An important point to remember here is that activities are non-atomic and can be interrupted. Looking at Figure 4.49, we can see that this means that either of the transitions out of 'moving forward' can occur *while the 'move' activity is still taking place*. If this happens, the 'move' activity is stopped and the transition to the next state takes place. If the transition re-entered the same state (i.e. it is from a state back to that same state) then any interrupted activity would restart from the beginning.

Another 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 guards 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 is said to be *nondeterministic* and the exact behaviour is impossible to determine.

The state machine diagrams used so far have had an emphasis on using states where something happens over a period of time that is represented as an activity. There is another approach to creating state machine diagrams that is more event-driven and uses far more actions than activities. Indeed, in many such state machine

diagrams there are no activities at all, only actions. An example of an event-driven state machine can be seen in Figure 4.50.

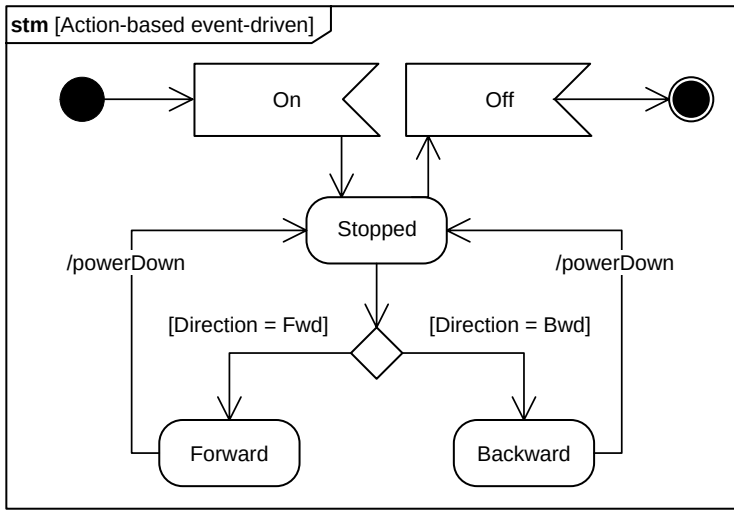

Figure 4.50 Event-based state machine

Figure 4.50 shows an event-based state machine. It can be seen here that the state names do not have the 'ing' ending, as, this time, they are representing states when particular conditions are met, rather than when something is actually happening. The things that happen here are shown as actions on the transitions between states. Actions actually have many types in the full SysML meta-model, such as: assignments, call, clear, create, destroy. Actions may be related to operations in a block definition diagram or may even refer to a simple assignment, or very low-level aspect of a design or specification.

There is a fine line between whether something that happens should be an action or an activity, but the following guidelines should give some pointers as to which are which.

- Actions are assumed to be non-interruptible – that is, they are atomic in nature and, once they have been fired, they must complete their execution. Activities, on the other hand, may be interrupted once they are running.
- As a consequence of the first point, actions are assumed to take zero time. This, however, refers to logical time rather than real time, which reflects the atomic nature of the action. Activities, on the other hand, take time to execute.
- Activities may be described using activity diagrams, whereas an action, generally speaking, may not.

It is important to differentiate between these activities and actions as they can have a large impact on the way in which the system models will evolve.

4.10.4 Using state machine diagrams

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

- All blocks that exhibit behaviour (have operations) should have an associated state machine diagram, otherwise the system is not fully defined. This reflects the fact that there must always be two views of the system, and forms a fundamental consistency check for any model.
- All operations in a particular block 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.

One question that is often asked concerns which of the two approaches to state machine diagrams is the better: the activity approach or the event-driven (action) approach? There is not much difference between them, as it really is a matter of personal preference. However, this said, the consistency checks for an activity-driven state machine are far stronger than those for the event-driven state machine, as the activities are related directly back to the governing block. Many books will use examples only where activity-driven state machine diagrams are considered, rather than both types. It should also be borne in mind that it is easier to transform an activity into an action than it is to transform an action into an activity. Therefore, when creating state machine diagrams, it is suggested that activities are used on the first iteration of the model.

4.11 Sequence diagrams (behavioural)

4.11.1 Overview – sequence diagrams

This section introduces and discusses sequence diagrams, which realize 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 moviemakers may draw up a storyboard. A storyboard shows the sequence of events in a film before it is made. Such storyboards in the SysML 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).

4.11.2 Diagram elements – sequence diagrams

Sequence diagrams have a very rich syntax, much of which is outside the scope of this book. The partial meta-model for the sequence diagram is shown in Figure 4.51.

Figure 4.51 shows that a ‘Sequence diagram’ is made up of a single ‘Frame’, zero or more ‘Gate’, one or more ‘Message’, one or more ‘Life line’ and zero or

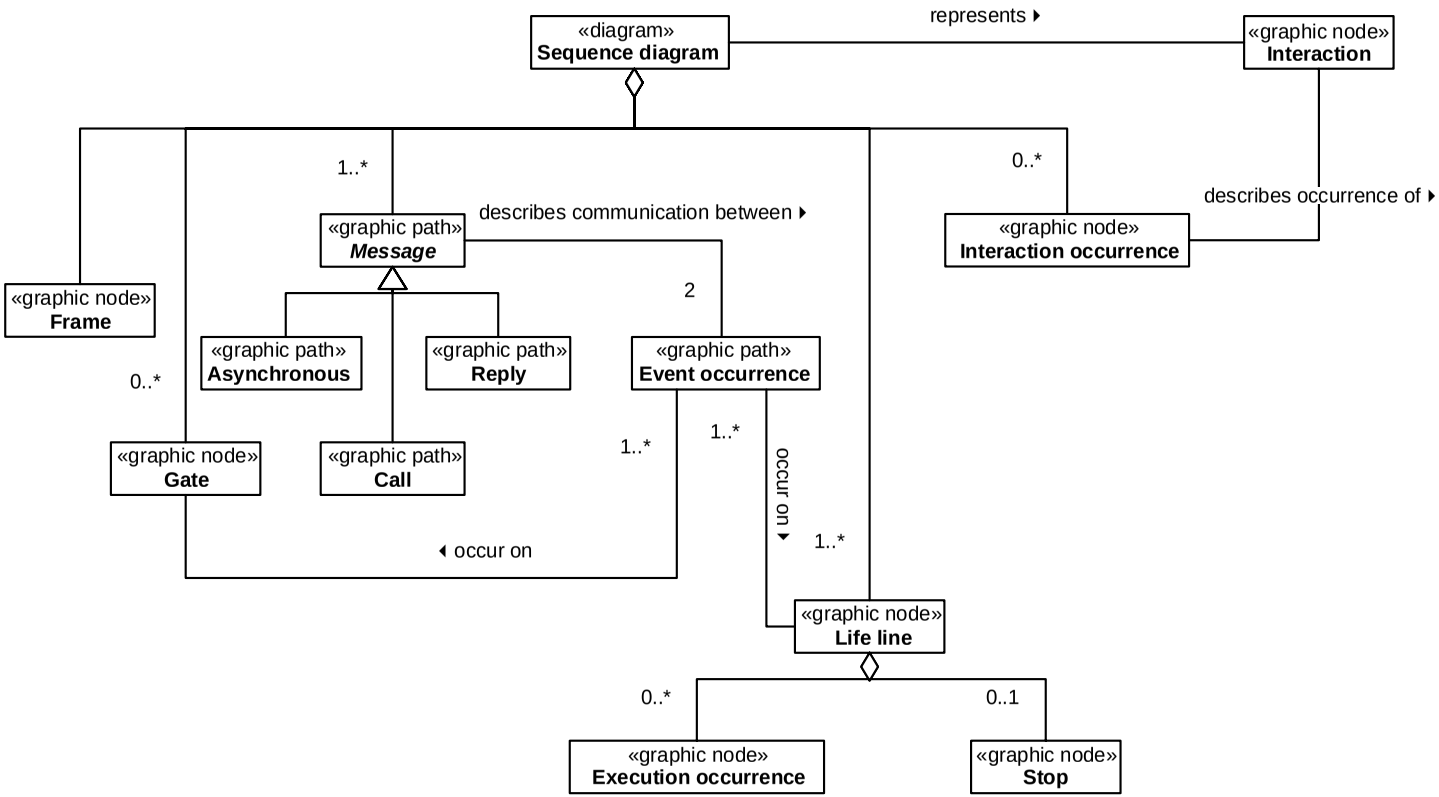

Figure 4.51 Partial meta-model for sequence diagrams

more ‘Interaction occurrence’. Each life line in a sequence diagram has a dotted line underneath that indicates the order in which things happen (such as messages) and shows when the life line is active via an ‘Execution occurrence’. The death or termination of a life line is indicated by a ‘Stop’. An interesting piece of syntax that is included here is the ‘Interaction occurrence’ that allows, effectively, another scenario to be inserted into a sequence diagram. This means that scenarios that show repeated behaviour can be bundled into a scenario on a separate sequence diagram and then inserted at the relevant point in time into the life line sequence.

Another useful element here is the ‘Gate’ that allows a message to enter into a sequence diagram from somewhere outside the diagram. This is useful for showing the message that invokes, or kicks off, a particular interaction without specifying exactly where it has come from (although this must be specified at some point in the model).

The element ‘Event occurrence’ shows when a message occurs on the life line. It has no notation, but is represented by the intersection of a message arrow and a life line.

These diagram elements are realized by either graphical nodes or graphical paths, as indicated by their stereotypes, and are illustrated in Figure 4.52.

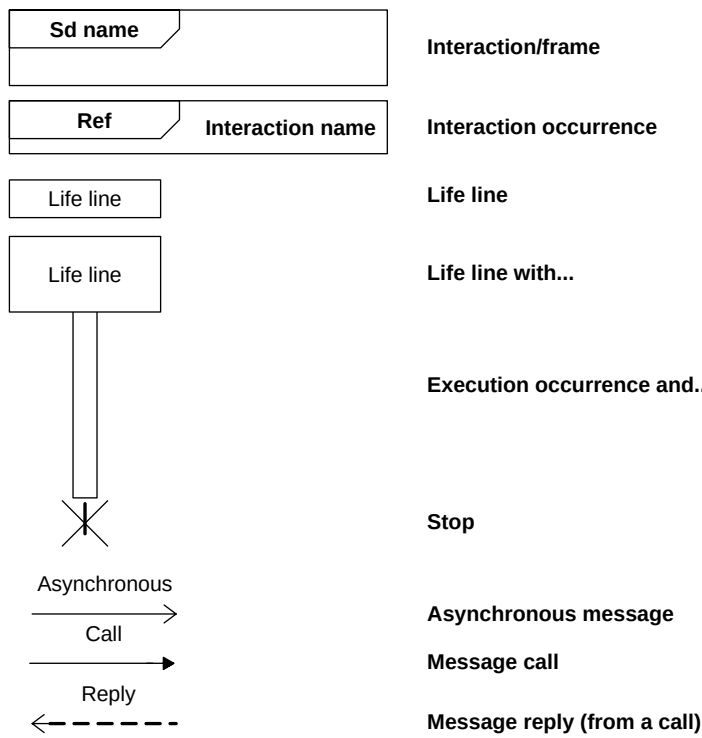

Figure 4.52 Graphical representation of elements in a sequence diagram

The diagram in Figure 4.52 shows the graphical notation used in sequence diagrams. The main symbol here is the life line with the dotted line underneath it, which represents the life of the life line. It is from such life lines that all the behaviour of the interaction is shown.

4.11.3 Examples and modelling – sequence diagrams

This section looks at some examples of using sequence diagrams, again using the robot challenge example that we have been using throughout.

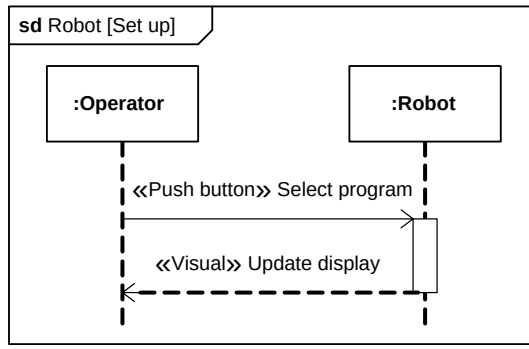

Figure 4.53 Sequence diagram for the robot challenge – set-up

Figure 4.53 shows a sequence diagram for the robot challenge. The life lines in the system are realized as: life line name, colon, block name. The life line name is optional and is used to identify a specific occurrence of a block that is taking part in the interactions. If no specific occurrence needs to be identified, then the life line name can be omitted.

Two life lines are shown here: 'Operator and 'Robot'. In the context of the robot challenge, they can be considered to come into existence at the same time – when the challenge begins – and, as such, both life lines appear at the same height on the diagram.

The diagram in Figure 4.53 shows the sequence diagram for the set-up scenario of the robot challenge. Of note here is the use of stereotypes on the messages to indicate the nature of the messages. The box shown on the 'Robot' life line indicates when the life line is executing something or, to put it another way, is busy. Each time a message is sent, there is an event occurrence at the point where the message joins the dotted life line. The dashed message indicates a response by the 'Robot' to the incoming 'Select program' message from the 'Operator'.

This scenario can be combined with another using the *interaction occurrence* notation. This is illustrated in Figure 4.54.

The interaction occurrence frame 'Set up' indicates that the scenario starts with the 'Set up' scenario shown in Figure 4.53 before continuing with the rest of the

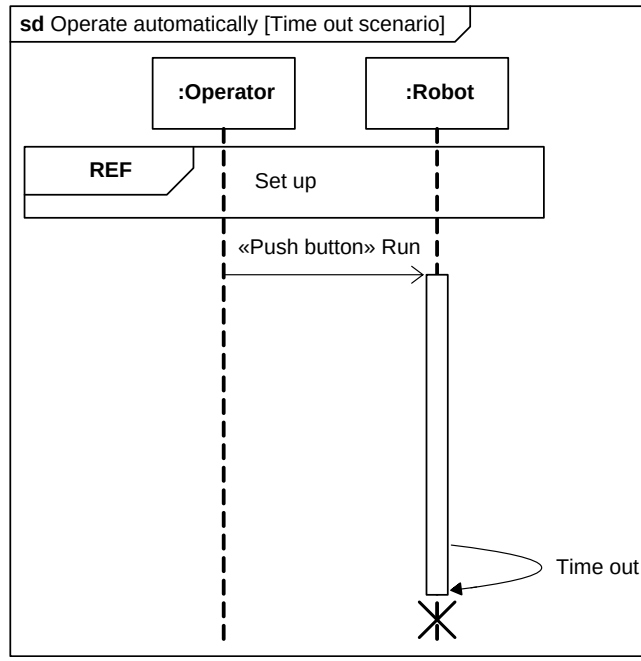

Figure 4.54 Sequence diagram for the robot challenge – time-out scenario

interactions shown on the diagram. In this scenario the 'Robot' is active from the time it receives the 'Run' message until it receives the 'Time out' message. At this point its life line terminates as indicated by the 'Stop' symbol – the large 'X' on the dotted line. Also of note here is the 'Time out' message. This is a message from the 'Robot' *to itself* and indicates that the 'Robot' has determined that it should time out and has instructed itself to do so.

The next example is that of a lower-level scenario that is showing interactions between elements that make up the 'Robot' (Figure 4.55).

In this scenario the first thing that happens is that the 'Sensor' sends a 'sensorHit' message to the 'Controller software'. On receipt of this message the 'Controller software' sends the 'turn' message to itself, initiating its turn behaviour.

The shaded area labelled 'par' is an example of a *combined fragment*. Combined fragments are a powerful part of sequence-diagram notation that allow such things as looping and, as in this case, parallel behaviour to be shown. In this example the 'Controller software' sends two 'activateDrive' messages (one to start and one to stop the 'Motor') to each of the two 'Motor' elements *in parallel*. Further discussion of combined fragments is outside the scope of this book. Of further note is the smaller *execution occurrence* rectangle overlying the main one on the life line of the 'Controller software'. This shows that the two 'activateDrive' messages are being sent as part of the behaviour that the 'Controller software' performs in response to the 'turn' message it sent to itself. The end of this smaller rectangular region indicates an end

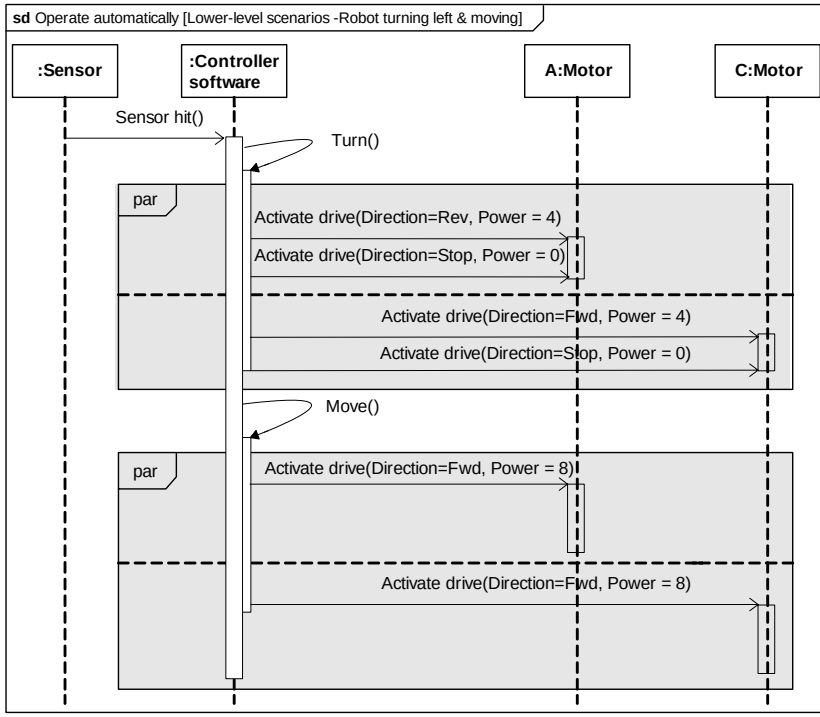

Figure 4.55 Sequence diagram for the robot challenge – turning and moving

to the 'turn' behaviour. At this point the 'Controller software' sends itself a 'move' message and again a parallel combined fragment is used.

Comparing this diagram with others also provides a very useful consistency check. Comparing the life lines against the blocks on Figures 4.11 and 4.12 shows that all the blocks responsible for the control and movement of the robot have been defined. Similarly, the messages sent by the 'Controller software' to itself correspond with the operations shown on Figure 4.15. The 'activateDrive' messages sent by the 'Controller software' to the two 'Motor' life lines correspond to the interfaces shown on Figure 4.12 and defined on Figure 4.13. The sequencing of a 'turn' followed by a 'move' can also be seen to be consistent with the defined behaviour of the 'Robot' by looking at its state machine diagram in Figure 4.48.

4.11.4 Using sequence diagrams

There are several rules of thumb that should be borne in mind when creating sequence diagrams.

- Life lines in a sequence diagram will always refer to another element in the model, such as blocks, ports or actors. This is true whether the other elements have been defined or not. If the elements have been defined, all is well and

good. If, however, elements have not been defined, sequence diagrams are a good approach to identifying elements such as blocks and parts based on example scenarios (perhaps created from particular use cases).

- In the same way, messages should always refer back to associations, connected interfaces and item flows. This can prove to be a good consistency check back to the block definition and internal block diagrams.
- Messages may be derived from state machine diagrams – if they already exist. If not, each message may show which send and receive events may be required for state machine diagrams.
- One final point that should be borne in mind is that sequence diagrams model behaviour according to logical timing or, to put it another way, the layout of life lines on the page is *not* important. However, it is considered good practice to position life lines so as to minimize the number of messages crossing life lines that they are not directed at.

Sequence diagrams are typically used to model scenarios; however, there is another use that is rarely mentioned in the literature, which is that of a consistency check for lower-level behavioural diagrams, such as activity diagrams and state machine diagrams. This point was raised in Chapter 3, when the state machine for the block 'Player' was able to be verified as working at one level, but it took a higher-level view to reveal that the messages defined on the state machine were incorrect. This also relates to Chapter 1, where it was stated that models should be looked at from more than one level of abstraction to ensure that the overall model is correct. Sequence diagrams are very powerful when used as a consistency check between various interacting life lines that already have their internal behaviour defined with state machine diagrams.

4.12 Activity diagrams (behavioural)

4.12.1 Overview

This section looks at another behavioural model of a system: the activity diagram. Activity diagrams, generally, allow very low-level modelling to be performed compared with 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, which is about as low as it is possible to go.

The other main use for activity diagrams, and certainly the most commonly seen usage of activity diagrams in other texts, is to model 'workflows' or processes. This will be discussed in some detail later in this chapter, since the level of abstraction of a workflow can vary. For a more detailed discussion of process modelling with SysML, see Chapter 6.

Activity diagrams are actually special types of state machine diagram and, as will be seen, the constructs are very similar between the two types of diagram. Activity diagrams are also similar to traditional flow charts (from which they were derived).

4.12.2 Diagram elements

The main elements that make up activity diagrams are shown in Figure 4.56.

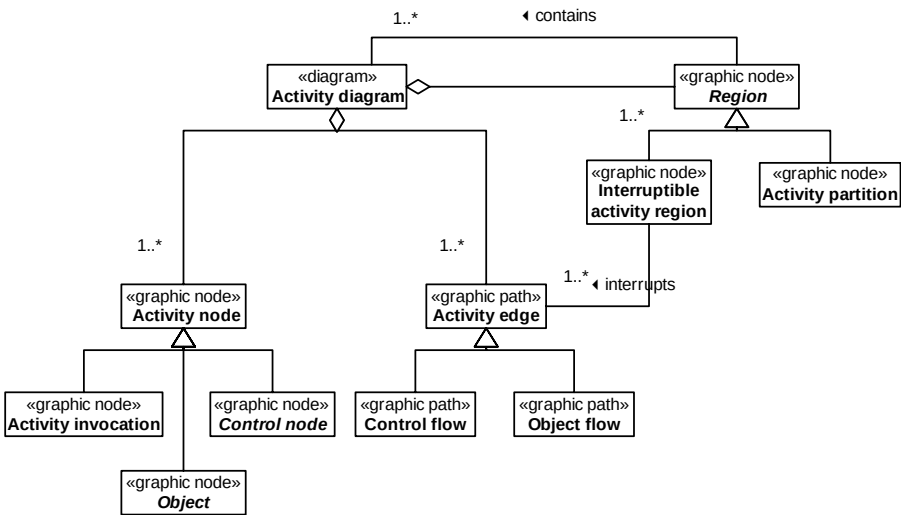

Figure 4.56 Partial meta-model for activity diagrams

Activity diagrams are made up of three basic elements: one or more ‘Activity node’, one or more ‘Activity edge’ and one or more ‘Region’. There are three main types of ‘Activity node’, which are the ‘Activity invocation’ and the ‘Object’ and ‘Control node’; the last two of these will be discussed in more detail later in this section. The ‘Activity invocation’ is where the main emphasis lies in these diagrams and it is through activity invocations that it is possible to establish traceability to the rest of the model via operations, activities and actions.

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 more ‘Activity node’. An ‘Object flow’ is used to show the flow of information between one or more ‘Activity node’ and does so by using the ‘Object’ type of ‘Activity node’. This is illustrated in Figure 4.56.

The other major element in an activity diagram is the ‘Region’, which has two main types: ‘Interruptible activity region’ and ‘Activity partition’. An ‘Interruptible activity region’ allows a boundary to be put into a diagram that encloses any activity invocations that may be interrupted. This is particularly powerful for software applications where it may be necessary to model different areas of the model that can be interrupted, for example, by a direct user interaction or some sort of emergency event. The ‘Activity partition’ is the mechanism that is used to visualize swim lanes that allow different activity invocations to be grouped together for some reason – in the case of swim lanes for responsibility allocation.

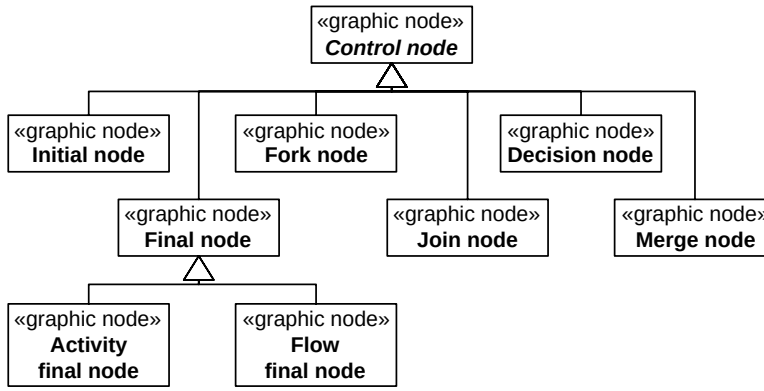

Figure 4.57 Expanded partial meta-model, focusing on 'Control node'

The diagram in Figure 4.57 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 and kicks off the diagram. 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 closing 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 'Join node' and 'Fork node' allow the flow in a diagram to be split into several parallel paths and then rejoined at a later point in the diagram. Forks and joins use the concept of 'token passing' that was used in Petri-net systems modelling, 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 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 rejoining the flows.
- The 'Decision' and 'Merge' nodes also complement each other nicely. A 'Decision' node allows a flow to branch off down a particular route according to a condition, whereas a 'Merge' node allows several flows to be merged back into a single flow.

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' and 'Time signal'. The 'Object node' is used to represent information that has been represented elsewhere in the model by a block and is forming an input to or an output from an activity. The 'Signal' symbol is used to show signals passed in and out of the activity diagram – the same as in a state machine diagram – whereas the 'Time signal' allows the visualization of explicit timing events. Again, the inconsistency of

the SysML in its treatment of instances can be seen here. In the UML, an object node allows an instance of a class – an object – to be used on an activity diagram. However, has as been discussed previously, SysML has no concept of an instance yet retains the 'Object node' on the activity diagram. The SysML specification is curiously quiet on just how an object node relates to a block. The types of 'object' node are shown in Figure 4.58.

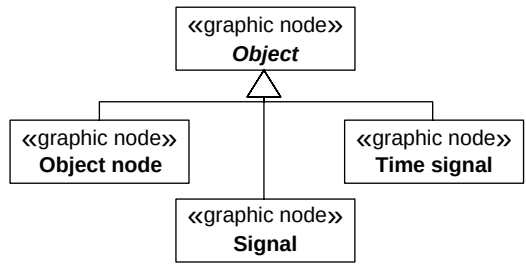

Figure 4.58 Expanded partial meta-model, focusing on 'Object' node

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes, as seen in Figure 4.59.

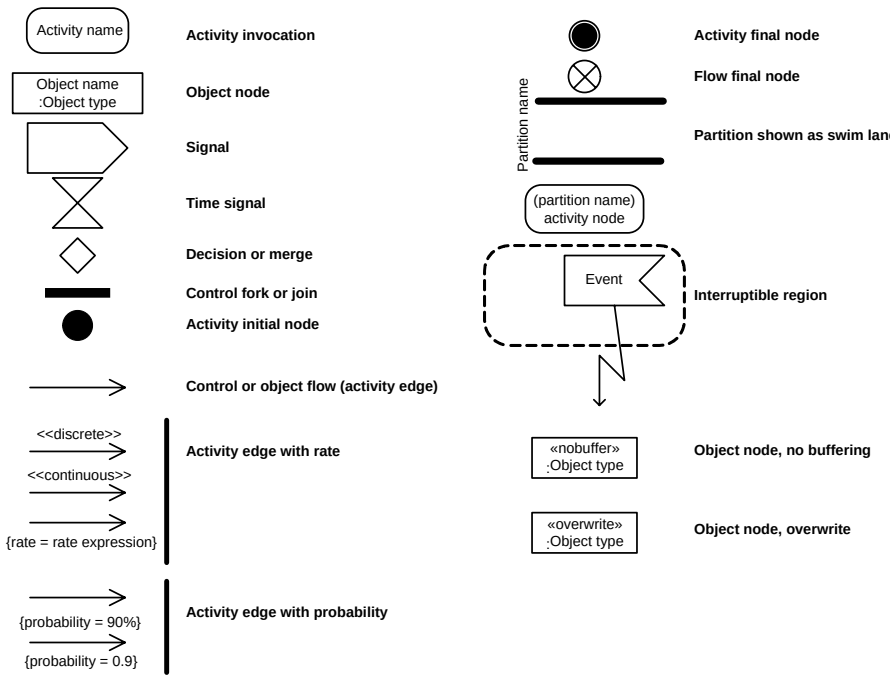

Figure 4.59 Graphical representation of main elements in an activity diagram

In addition to the elements mentioned so far, SysML has introduced some new notation that can be applied to an 'Activity edge' and an 'Object node'. This notation makes use of the existing constraint and stereotype notation already present in SysML (and its underlying UML) and simply defines some standard constraints and stereotypes for use of activity diagrams. Indeed, most of the differences between SysML and UML consist of predefined constraints and stereotypes, with associated meaning and guidelines, that can be applied to elements already present in UML. This is discussed further in Appendix B.

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. Unsurprisingly, this is shown by use of the ≪discrete≫ or ≪continuous≫ stereotypes placed on the edge. Alternatively, the actual rate can be shown using a constraint of the form: {rate = rateExpression}. For example, if data or material passed along an edge every minute, then this could be shown by placing the constraint {rate = per 1 minute} on the edge. The rate constraint can be used in conjunction with the ≪discrete≫ stereotype but not with the ≪continuous≫ one.

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 (as shown in Figure 4.59). All the probabilities on edges with the same source *must* add up to 1 (or 100 per cent). It is important to note that the actual edge traversed is governed by the guards 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 new notation modifies the behaviour of an 'Object node' and is indicated by the use of the stereotypes ≪nobuffer≫ and ≪overwrite≫. Their meaning is best explained by the examples shown in the three diagrams in Figures 4.60–4.62. In a brief departure from the world of robotics and to encourage unhealthy eating we will use the world of fast food to illustrate the notation.

Each diagram shows a fragment of an activity diagram for a burger shop that makes a new burger every minute and that takes an order every five minutes. The shop is rather strange in that it has space for only one burger under its heat lamps. This is illustrated by the use of the {upperBound = 1} constraint on the burger 'Object node'. The {upperBound} constraint is a standard SysML (and UML) constraint that can be applied to any 'Object node' to indicate how many of them can exist at a time.

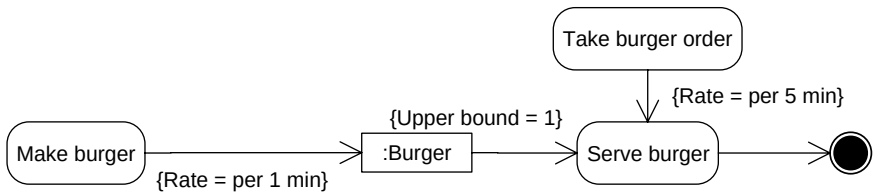

Figure 4.60 Burger example – normal 'Object node'

In Figure 4.60, once the burger has been made and is under the heat lamps, it will sit there until an order is placed. This will *block* the 'Make burger' action so that another burger will not be made until the burger under the heat lamp has been served. A burger will always be there for serving, but could be five minutes old.

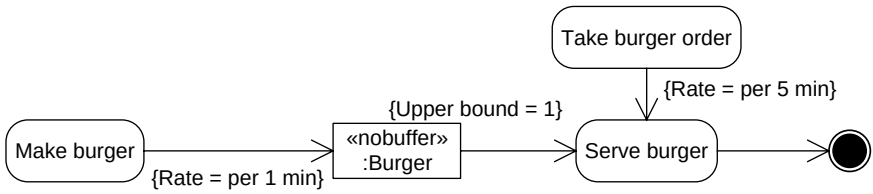

Figure 4.61 Burger example – ≪nobuffer≫ 'Object node'

In Figure 4.61, if there is no order when the burger is made, it is discarded as indicated by the ≪nobuffer≫ stereotype and there will never be a burger waiting under the heat lamp. The 'Make burger' action will not be blocked and after a minute another burger will be made. This means there would be a lot of waste, and a customer may have to wait for one minute to get the burger, but it will be fresh.

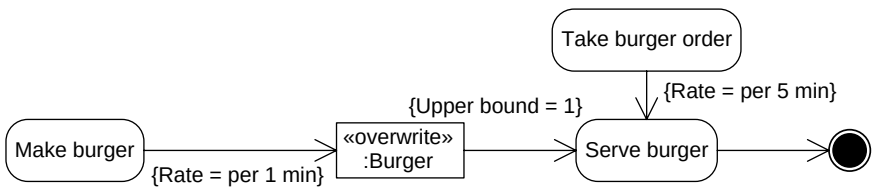

Figure 4.62 Burger example – ≪overwrite≫ 'Object node'

In Figure 4.62, if there is no order when the burger is made, the burger under the lamp is discarded and replaced with the new one as indicated by the ≪overwrite≫ stereotype. The 'Make Burger' action will not be blocked and after a minute another burger will be made. This means there would be a lot of waste (but not as much as in Figure 4.61), but a burger will always be there for serving although it could be one minute old.

Partitions can be shown in one of two ways: the traditional swim-lane way by drawing a box or two parallel lines enclosing the relevant region, or simply to write the partition name in brackets above the activity name in the activity invocation symbol. Interruptible regions are shown by a dashed line with an arrow coming out of it to show where the flow goes in the event of an interruption.

4.12.3 Examples and modelling

The two main uses for activity diagrams are to model workflows and operations. Conceptually, modelling operations is far simpler to understand than modelling

workflows. The first example, therefore, shows how to model the behaviour of an operation. This would be a procedure when applied to software, or a general algorithm when applied to general systems (perhaps a thought process or hardware design).

The first example shows how to model a simple software algorithm, in this case the 'move' operation of the robot that has to be implemented in the robot's control software.

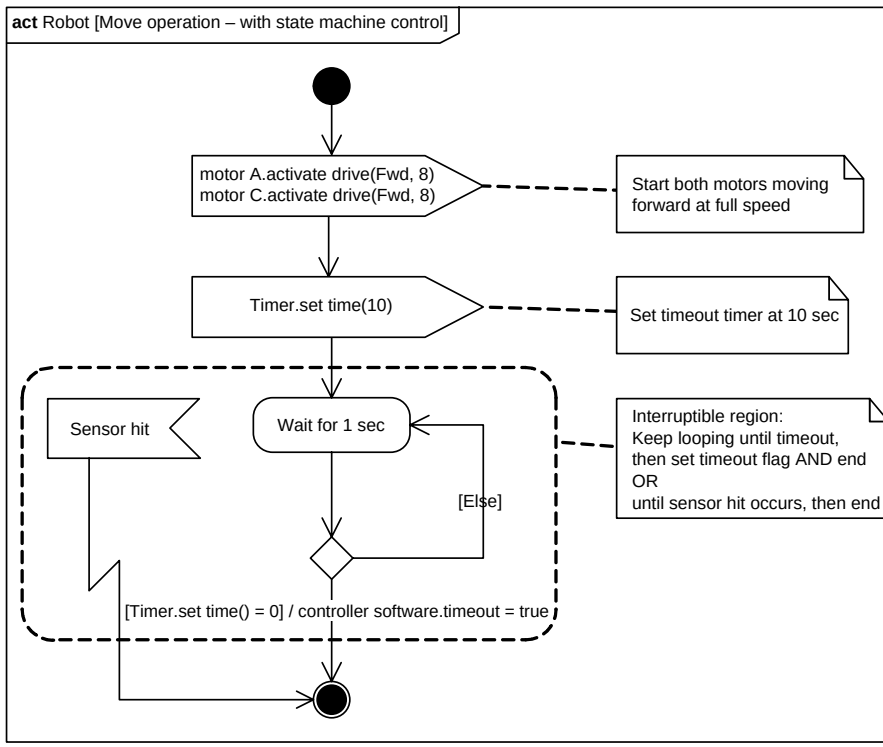

Figure 4.63 Activity diagram describing the behaviour of the robot's 'move' operation

Figure 4.63 shows how the robot's 'move' operation, defined in the 'Navigation' block (see Figure 4.15), could be implemented. The first thing that happens is that signals are sent out of this operation to activate both of the robot's motors using the 'activateDrive' operations defined on the 'Drive' interface of the motor (see Figure 4.13). Next a ten-second timer is started via a signal that invokes the 'setTime' operation of the 'Timer' block as shown on Figure 4.15.

The interruptible region is then entered via the main flow in which the timer is checked once per second to see if it has elapsed. This is done by invoking the 'getTime' operation of the 'Timer' block. If it has elapsed, then the 'timeout' property of the 'ControllerSoftware' block is set true. Note here that the property is set using the

action notation that has previously been seen for state machine diagrams. This is possible because an activity diagram is essentially a special type of state machine diagram.

The other path out of the interruptible region is part of the interrupt handler: if a 'sensorHit' event is detected, the interruptible region is immediately exited via the jagged path and, in this case, the end of the activity diagram is reached. This path can connect to *any* activity invocation outside the region, with this activity being responsible for doing any special processing or taking any special behaviour that is needed to handle the event. It is important to note that the 'normal' behavioural path in the interruptible region is *immediately* exited when an interrupt occurs; any activities that are running are terminated straightaway.

Activity diagrams are very useful for modelling complex algorithms with much iteration, but may be 'overkill' for modelling very simple operations, particularly if many exist within the design. Indeed, some sort of simple functional descriptions, such as informal pseudo-code, may be more appropriate for very simple operation definition.

The activity diagram can also be used to model any low-level conceptual operation that need not be software, as shown in Figure 4.64.

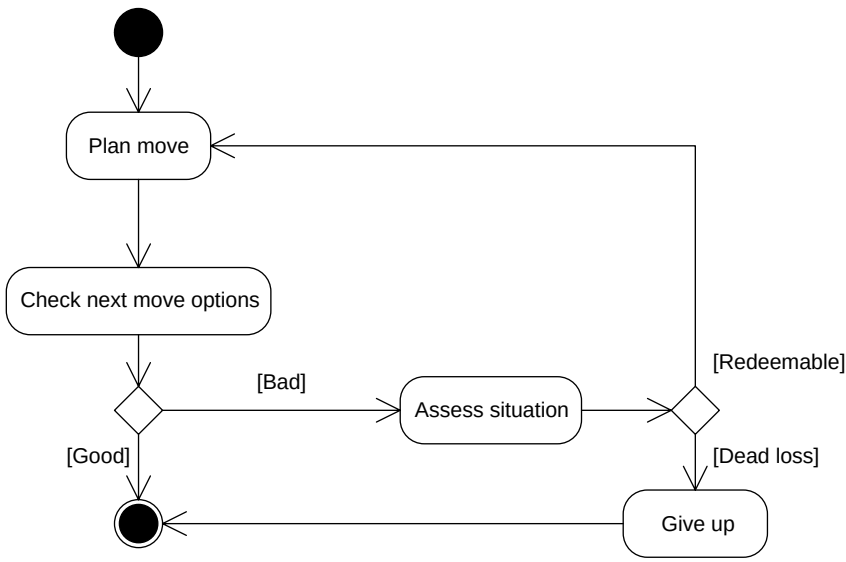

Figure 4.64 Activity diagram showing behaviour of a thought process of how to play chess (badly)

Figure 4.64 shows the 'move' operation from the original chess example. It can be seen that the first thing that happens is 'plan move'. The next is 'check next move options', which is then followed by a decision branch. There are two outcomes: 'bad', which leads to 'assess situation', and 'good', which leads to the end of the activity

diagram. Immediately following ‘assess situation’ is another decision branch where ‘redeemable’ loads back to ‘plan move’ and ‘dead loss’ leads to a ‘give up’ activity invocation.

The most widely used application for activity diagrams (certainly from most other literature’s point of view) is concerned with modelling workflows. This can cause some confusion, as the term ‘workflow’ is very much associated with the Rational Unified Process (RUP) [2, 3], which has its own distinct terminology, which can be different from some other terminology, such as the one adopted by ISO. Two examples of modelling workflows will be considered, one of an ISO-type interpretation of the term ‘workflow’ and one of the RUP interpretation of the term ‘workflow’.

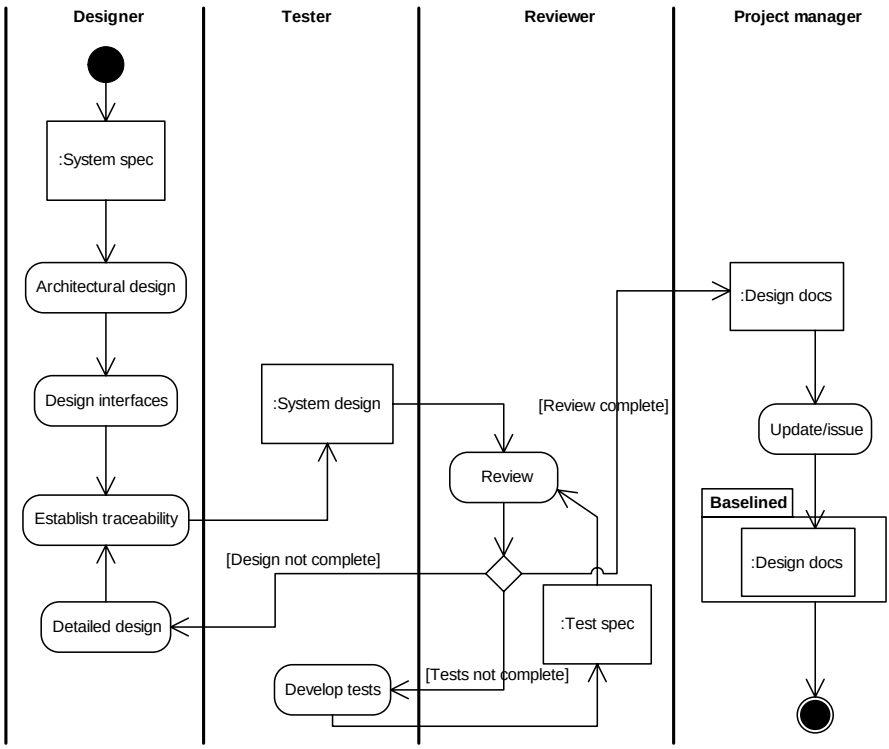

Figure 4.65 Activity diagram for an ISO work practice

Figure 4.65 shows how a design process behaves. Notice how this diagram uses partitions as swim lanes to show responsibility in the process. Also, note the cunning use of a package to indicate where one of the object nodes has been baselined.

The second example of modelling workflows is one that is taken from the definition of the RUP. For a more in-depth discussion of this concept of workflows, see [3, 4]. For now, however, it is enough to know that the activity diagram is recommended by the RUP to be the diagram that is used to model workflows.

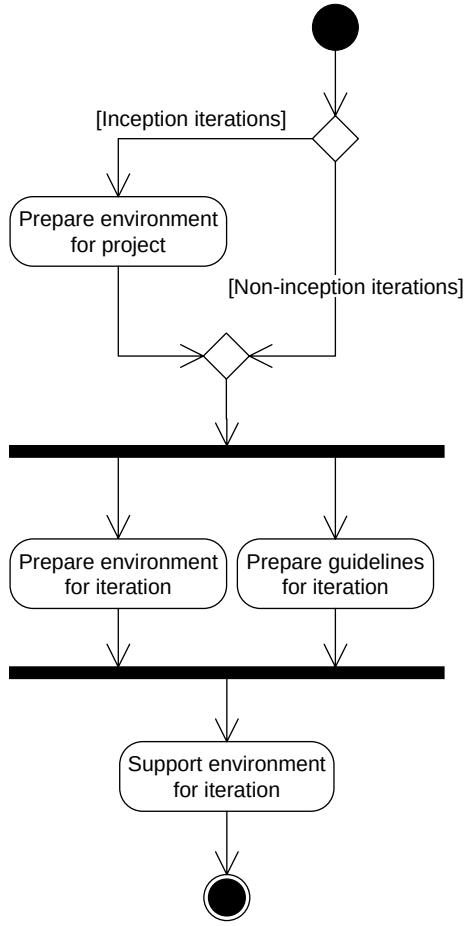

Figure 4.66 Activity diagram showing workflow from the RUP

Figure 4.66 shows an activity diagram that describes the behaviour of, or how to execute, a particular workflow. One point worth considering here is the use of the fork and join nodes. There are two activity invocations that are happening in parallel flows, but this does not necessarily indicate that the actual activities are being executed in parallel. What the diagram actually shows is that the flow is split into two flows and that both activities must complete before the flows can be joined. It could be the case that both activities do execute in parallel, or it could be that one is executed before the other, or any other permutation of this. The main point to remember here is that both flows must complete before they can be joined.

Also of note is the use of the merge node to bring the two alternate paths together so that a single control flow enters the fork node. Strict SysML syntax requires that a fork node have a single control flow entering with multiple flows leaving, whereas a

join node has multiple flows entering and a single flow leaving. It *is* possible to draw diagrams with multiple control flows entering a fork node but only when each of these flows represents an alternative path through the activity diagram such that only one of them could ever be 'active'. This is the case in the diagram in Figure 4.66. The merge node could be omitted and the two flows entering the merge could, instead, enter the fork node directly. Since each of these flows represents an alternative path resulting from the earlier decision node, this is allowed. However, adopting the strict syntax often makes the diagrams easier to follow at the expense of extra merge nodes.

4.12.4 Using activity diagrams

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 workflows. They are very good for helping to ensure model consistency, relating to state machine diagrams, sequence diagrams and block definition diagrams. However, activity diagrams must be used intelligently when used to represent particularly simple algorithms, since creating activity diagrams in these situations is often perceived as a waste of time.

Activity diagrams are actually defined as a special type of state machine diagram, but it is sometimes difficult to see why two types of diagram are required when, it may be argued, they are very similar indeed. The most fundamental conceptual difference between activity diagrams and state machine diagrams is that activity diagrams concentrate on activity flow and may thus show behaviour from more than one element, whereas a state machine shows the behaviour only for a single element.

As a consequence, state machine diagrams may show wait, or idle, states, whereas an activity diagram may not. The states in a state machine are normal states, which means that they may be complex and contain more than one internal transition, represented by a number of actions or activities, whereas activity invocation may contain only one. Activity diagrams may also use advanced syntax such as 'swim lanes' and 'object flow'. Swim lanes allow particular states to be grouped together and associated with a particular block, which is useful for assigning responsibility. Object flows allow the creation and destruction of blocks to be associated with activities, which may be used to show data flow.

4.13 Use case diagrams (behavioural)

4.13.1 Overview

This section introduces use case diagrams, which realize a behavioural aspect of the model. The behavioural view has an emphasis on functionality, rather than the control and logical timing of the system. The use case diagram represents the highest level of abstraction of a view that is available in the SysML and it is used, primarily, to model requirements and contexts of a system. Use cases are covered in greater depth in Chapter 7 and thus this section is kept deliberately short, emphasizing, as it does, the structure of the diagrams.

Use case diagrams are 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 model and accidentally to start analysis or design, rather than very high-level requirements and 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.
- Information on the practical use of use cases is surprisingly sparse, bearing in mind that many approaches that are advocated using the SysML are use-case-driven. In addition, much of the literature concerning use case diagrams is either incorrect or has major parts of the diagrams missing!

With these points in mind, the remainder of this section will describe the mechanics of use case diagrams. For an in-depth discussion concerning the actual use of use case diagrams, see Chapter 7.

4.13.2 Diagram elements

Use case diagrams are composed of four basic elements: use cases, actors, associations and a system boundary. Each use case describes a requirement that exists in the system. These requirements may be related to other requirements or actors using associations. An association describes a very high-level relationship between two diagram elements that represents some sort of communication between them.

An actor represents the role of somebody or something that somehow interacts with the system.

The system boundary is used when describing the context of a system. Many texts and tools ignore the use of the system boundary or say that it is optional without really explaining why. Think of the system boundary as the context of the system and its use becomes quite clear and very useful. System boundaries and contexts are discussed in more detail in Chapter 7.

Figure 4.67 shows that a ‘use case diagram’ is made up of zero or more ‘Actor’, one or more ‘Use case’, one or more ‘Relationship’ and zero or one ‘System boundary’. A ‘Use case’ yields an observable result to an ‘Actor’.

There are three basic types of ‘relationship’, which are ‘Extend’, ‘Includes’ and ‘Association’. An ‘Association’ crosses a ‘System boundary’ and, wherever this happens, it means that an interface must exist. The ‘Extend’ relationship allows atypical characteristics of a use case to be defined via an ‘Extension point’ that will actually define these conditions explicitly.

Each of these diagram elements may be realized by either graphical nodes or graphical paths, as indicated by their stereotypes, as illustrated in Figure 4.68.

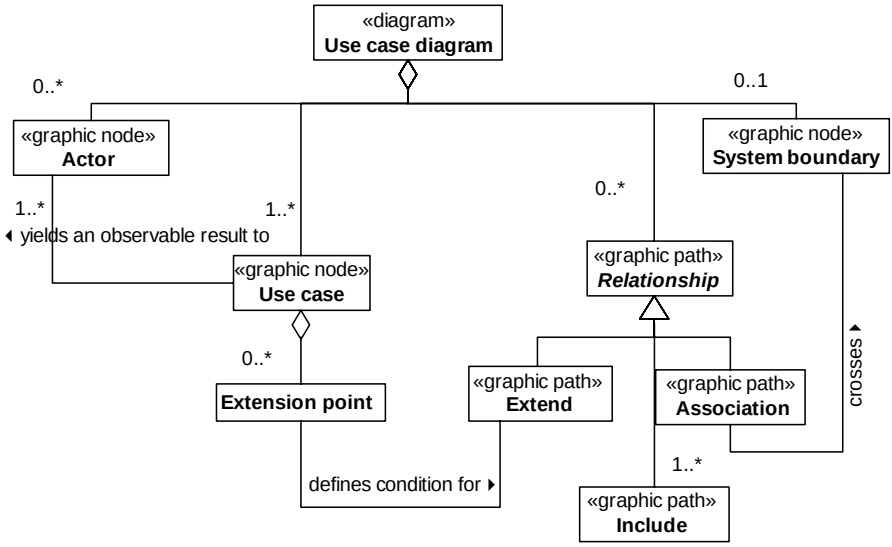

Figure 4.67 Partial meta-model for use case diagrams

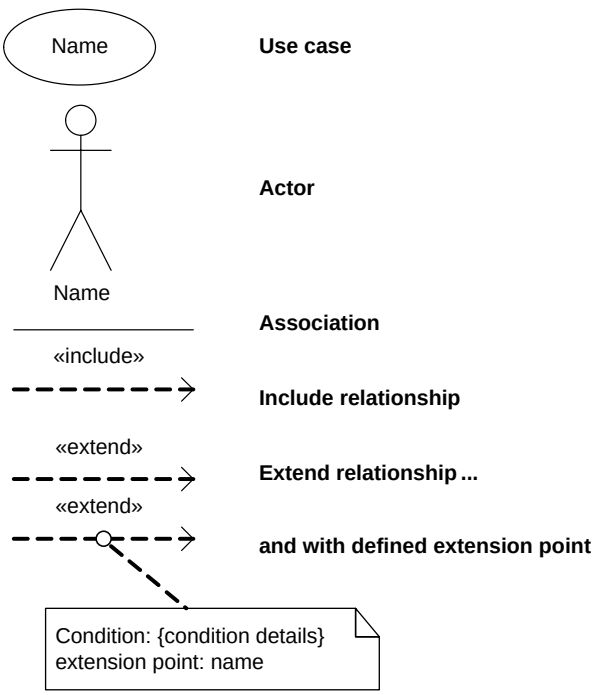

Figure 4.68 Graphical representation of elements of a use case diagram

One of the problems that many people have with using use case diagrams is how to tie them into the rest of the system model. It is not uncommon to be presented with a beautiful set of use case diagrams and then to be presented with a wonderful system model with no relationships whatsoever between the two! Figure 4.69, therefore, seeks to define this more clearly.

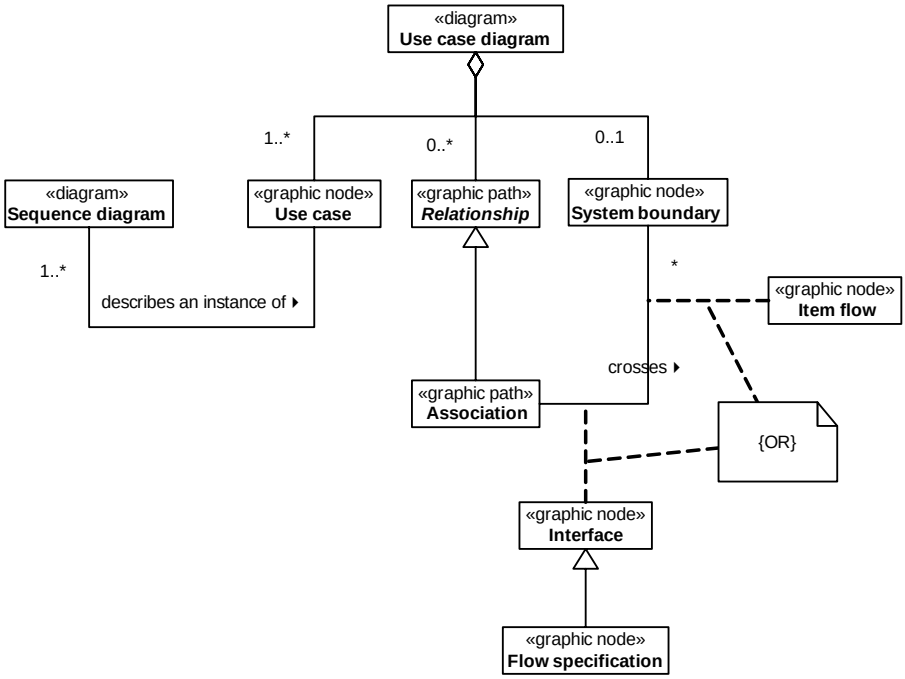

Figure 4.69 Meta-model showing relationships between use cases and other SysML elements

The diagram in Figure 4.69 shows the relationship between use case diagram elements and the rest of the SysML. Perhaps the most important relationship here is the one between ‘Sequence diagram’ and ‘Use case’, which states that one or more ‘Sequence diagram’ describes an instance of a ‘Use case’. Each use case will have a number of sequence diagrams associated with it or, to put it another way, each requirement will have a number of scenarios associated with it.

On a related note, each time an ‘Association’ crosses the ‘System boundary’, an interface, flow specification or item flow has been defined. This becomes important, as each will become the basis for message passing between different life lines in the interactions.

Once the interaction relationships have been established, it is then possible to navigate to any other diagram in the SysML and, hence, any other part of the system model.

4.13.3 Examples and modelling

Use case diagrams have two main uses: modelling contexts and modelling requirements. Although these two types of use should be related, they are distinctly different. For a full description of using use case diagrams, see Chapter 7. This section will concentrate purely on the mechanics and practical use of use case diagrams.

In addition, there are two types of context that can be modelled with regard to systems engineering: the system context and the business context. The first two example models show a business context and a system context for the ongoing robot challenge example.

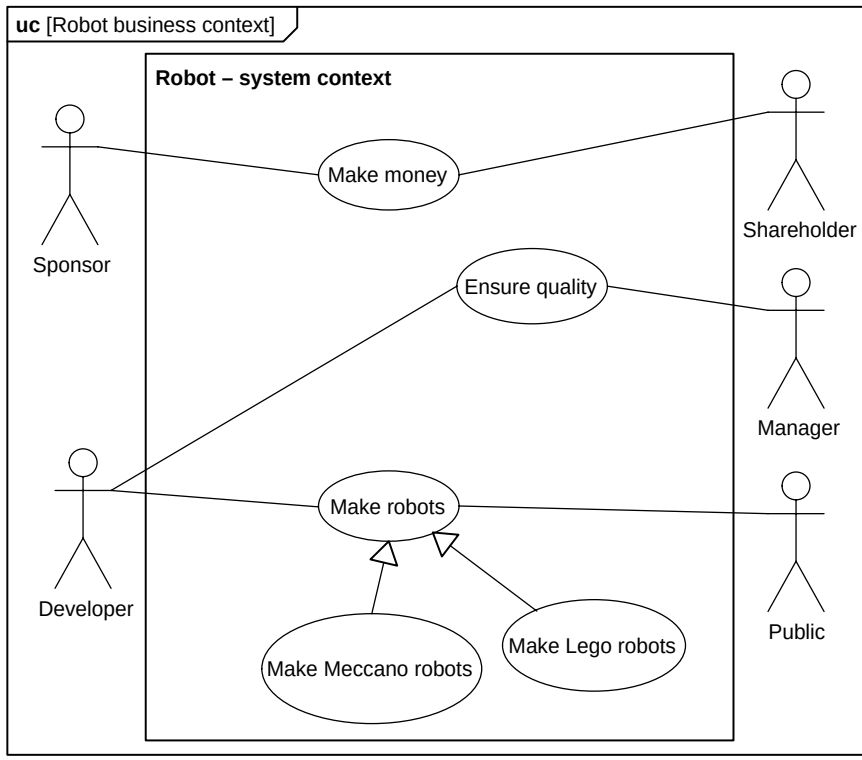

Figure 4.70 Use case diagram showing the business context for the robot challenge

Figure 4.70 shows the business context for the robot challenge. The business context shows the business requirements of the organization and identifies actors (see the discussion on stakeholders in Chapters 6 and 7) that are associated with the business requirements. The business requirements that have been identified are as follows.

- 'Make money', which is a business requirement that will be present in almost all business contexts in the world. This may seem obvious, but when one has asked

for funding for a project the response from management is, invariably, 'Make a business case and then we'll look at it.' By identifying the business requirements of your organization, it is possible to start to justify expenditure for future projects.

- 'Ensure quality' is a non-functional requirement that will impact on everything that the organization does. This is particularly true when organizations are trying to obtain, or have already obtained, some form of standard accreditation.
- 'Make robots' is basically what the organization does, which has two subtypes: 'Make Lego robots' and 'Make Meccano robots'.
- 'Make Lego robots' is the main business requirement that the system that has been used so far in this book has been concerned with. We can now see, however, how the requirement 'Make Lego robots' will help the organization 'Make money', which is very important.
- 'Make Meccano robots' is shown simply to indicate that the company may make more than one type of robot.

The actors that have been identified for the system must each have some sort of association with at least one use case. If an actor exists that does not have an association with a use case, there is something seriously wrong with the model.

Figure 4.71 shows the system context, rather than the business context. It is important that these two contexts can be related together in order to justify why a particular project should exist.

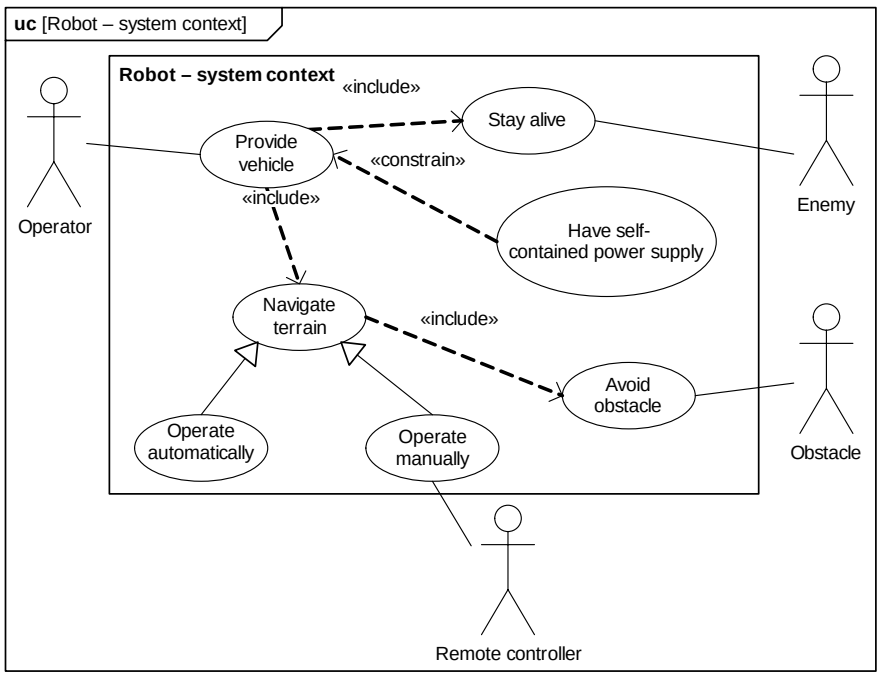

Figure 4.71 Use case diagram showing system context for the robot challenge

Figure 4.71 shows the system context for the robot challenge. The use cases that have been identified are, at this level, fairly simple and high-level. The use case 'Provide vehicle' represents the main aim of the robot challenge. The use cases 'Stay alive' and 'Navigate terrain' represent the main functionality that the robot vehicle has to display.

One argument that is often levelled at use case diagrams is that they are too simple and fail to add anything to a project. However, no matter how simple, a good use case diagram can contribute to a project in several ways.

- They can be traced back to a business context to help justify a project. Quite often it is the case that a simple one-sentence description is required to sum up a whole project and a good system context will show exactly that.
- They can give a very simple overview of exactly what the project is intending to achieve and for whom.
- They can form the basis of lower-level, decomposed use case diagrams that add more detail.

It is also worth remembering that, just because a use case diagram looks simple, it does not mean that it took somebody two minutes to create and did not involve much effort. This is true only when it comes to realizing the model using a CASE tool, but even the most simple of diagrams may take many hours, days or even months of work. Remember that what is being visualized here is the result of requirements capture, rather than the actual work itself.

Figure 4.72 shows how a single requirement may be taken and modelled in more detail, by decomposing the requirement into lower-level requirements.

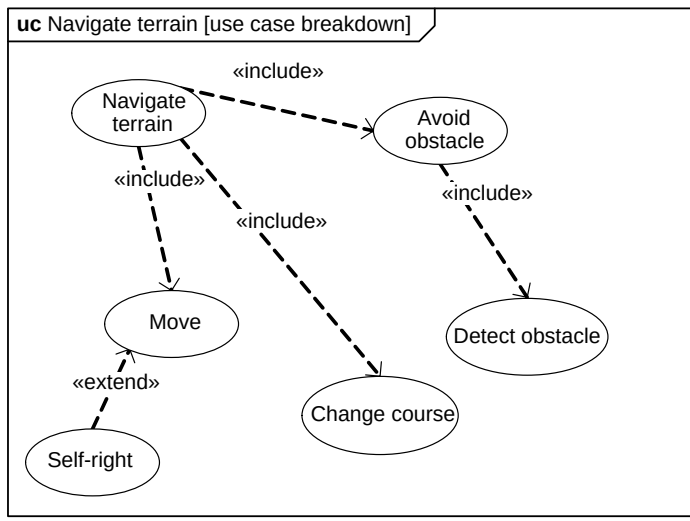

Figure 4.72 Use case diagram showing the modelling of a requirement in more detail

Figure 4.72 shows the 'Navigate terrain' requirement that has been decomposed into three lower-level requirements: 'Move', 'Change course' and 'Avoid obstacle', which is itself decomposed further into 'Detect obstacle'. Each of these requirements is a component requirement of the higher-level requirement, which is indicated by the special type of association '≪include≫', which shows an aggregation-style relationship.

The second special type of association is also shown here and is represented by the stereotype '≪extend≫'. The '≪extend≫' relationship implies that the associated use case somehow changes the functionality of what goes on inside the higher-level use case. In the example here, the extending use case is 'Self-right', which extends the functionality of 'Move' in the event that the robot falls over and forces 'Move' to behave in a different way.

Figure 4.73 shows what a use case diagram should not look like, which is a mistake that is very easy to make.

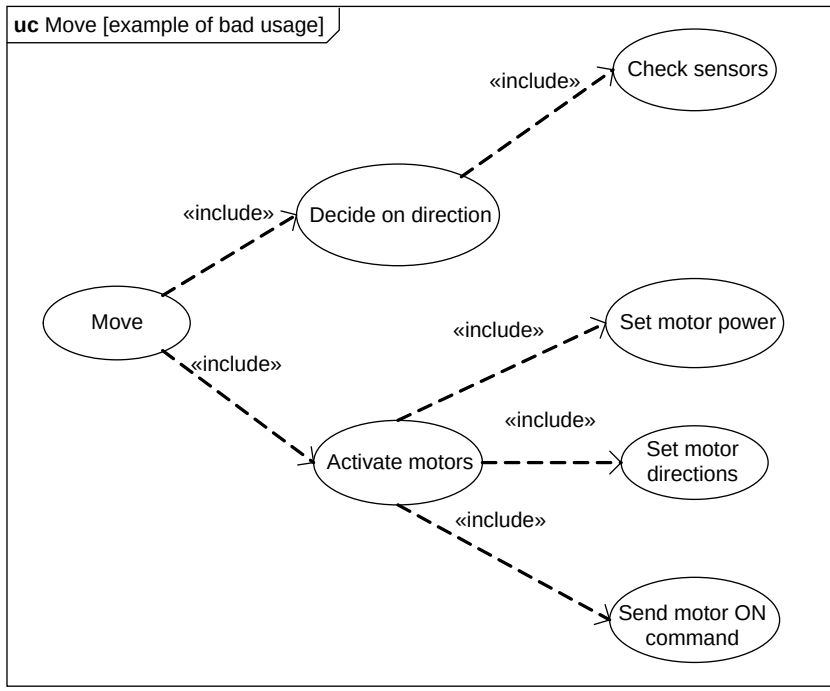

Figure 4.73 Use case diagram showing a poor example of modelling

The model in Figure 4.73 shows how a model should not be created, as it defies the objectives of use case diagrams. There are several points to consider here.

- The definition of a use case is that it must yield some observable result to an actor. In the example shown here, the decomposed use cases do not yield any observable result to the actors – they are modelled at too low a level.

- Remember that the aim of modelling requirements is to state a user's needs at a high level and not to constrain the user with proposed solutions. The example shown here has done exactly that and has started down the design road – which is inappropriate at this level.
- The diagram is turning into a data-flow diagram by starting at the highest level and decomposing until the problem is solved. Use cases are not the same as data-flow diagrams, despite appearances, and should not be used as such.

These points are discussed in more detail in Chapter 7.

4.13.4 Using use case diagrams

Use case diagrams are, visually, one of the simplest diagrams in the SysML, but they are also the most difficult to get right. Part of their downfall is this simplicity, which makes them easy to dismiss and easy to get wrong. Another problem with use case diagrams is that they look like data-flow diagrams and thus people tend to over-decompose them and start carrying out analysis and design rather than just simple requirements modelling.

One of the strengths of use case diagrams comes out only when they are compared with contexts. It is important to be able to trace any requirement back to the business context to justify a business case for the project. It is also important to be able to trace to the system context, as this provides the starting point for the project.

The meta-model shown here represents all of the basic SysML elements of use case diagrams, which indicates the simplicity of the diagram.

4.14 Summary and conclusions

4.14.1 Summary

This chapter has introduced each of the nine SysML diagrams in turn and has provided examples of their use. Each diagram has been discussed at a very high level, often missing out much of the diagram syntax, so that the modelling aspects of each diagram could be focused on, rather than being bogged down by all the syntax of each diagram.

The following models serve as a recap for everything that has been said in this chapter, which should be able to be read by anyone who has read (and understood) this book so far. Even with the limited amount of syntax that has been introduced, it is still possible to model many aspects of a system.

Figure 4.74 shows the nine types of SysML diagram that have been introduced in this chapter. These nine diagrams are often grouped into categories to represent different views of the system. Rather than be prescriptive with these groupings, the diagrams in this book are grouped into two very broad aspects of the model: behavioural and structural.

Figure 4.75 shows how the nine types of SysML diagram are grouped into the two aspects of the model. Many other texts on the market will show different 'views' of a system, such as logical view, physical view, use case view and so on, and this

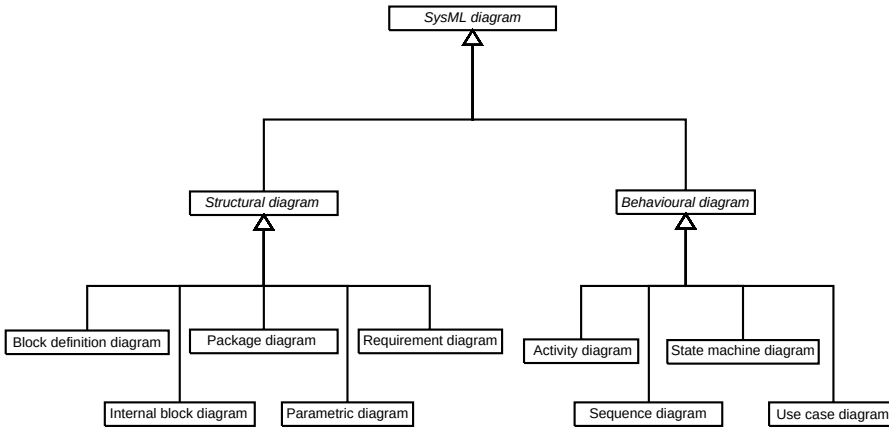

Figure 4.74 The nine types of SysML diagram

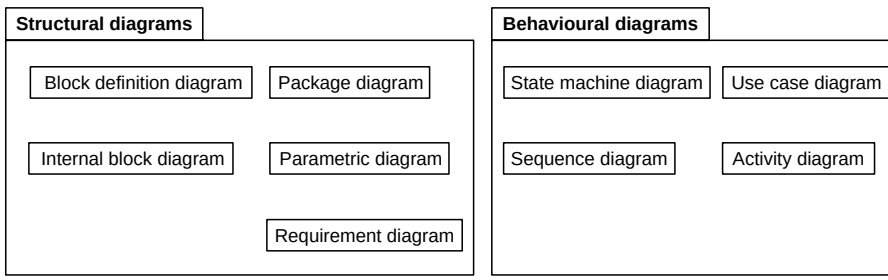

Figure 4.75 The types of diagram grouped into aspects of the model

is fine but it will not fit every application. These views are also consistent with the two aspects of the model shown here; each type of view that is defined may consist of a structural and behavioural element. Treat the two aspects here as the absolute highest level of aspects of the model and use the views introduced in other books as necessary.

4.14.2 Conclusions

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. These rules are not carved in stone, but experience has shown that they will be applicable to 99 per cent of all modelling.

- *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 limited 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 a living entity that will evolve over time and, as the model becomes more refined, so the connection to reality will draw close.
- *Keep all models*. Never throw away a model, even if it is deemed incorrect, since it will help you to document decisions made as the design has evolved.
- *Ensure that the system is modelled in both aspects*. In order to meet most of the above criteria, it is essential that the system be 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, as demonstrated in the robot example.

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.

4.15 Further discussion

1 Take one of the partial meta-models shown in this chapter and then, using the SysML specification [1], populate the meta-model further to gain a greater understanding of the type of diagram chosen for this example.
2 Expand on the robot challenge game example by adding new operations called ‘detect’, ‘evade’ and ‘attack’ to the block definition diagram introduced in Figure 4.15. Which other diagrams will this affect? What other diagrams will be needed?
3 Expand on the robot challenge game example by adding new versions of the motor, for example perhaps an electromechanical version. What interaction points must this new motor have? Show how this new type of motor can be connected to the controller.
4 Take any prerecorded CD-ROM and model the contents, based on what can be seen, for example, from a typical Windows-style browser.
5 Now model the steps that you took in order to put the CD into the computer and read the contents of the CD-ROM. Think about block definition diagrams for the structural view and sequence diagrams and state machine diagrams to model how this was achieved.

4.16 References

1 Object Management Group. *OMG Systems Modeling Language (OMG SysML) Specification* [online]. Available from http://www.omgsysml.org [Accessed August 2007]
2 Booch G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language User Guide*. 2nd edn. Boston, MA: Addison-Wesley; 2005
3 Kruchten P. *The Rational Unified Process: An Introduction*. 3rd edn. Boston, MA: Addison-Wesley; 2004
4 Rumbaugh J., Jacobson I. and Booch G. *The Unified Modeling Language Reference Manual*. 2nd edn. Boston, MA: Addison-Wesley; 2005

4.17 Further reading

Holt J. *UML for Systems Engineering: Watching the Wheels*. 2nd edn. London: IEE Publishing; 2004 (reprinted, IET Publishing; 2007)

Hoyle D. *ISO 9000, Pocket Guide*. Oxford: Butterworth Heinemann; 1998

Chapter 5
Physical systems, interfaces and constraints

> ‘There ain’t no rules around here. We’re trying to accomplish something.’
>
> Thomas Alva Edison (1847–1931)

> ‘The more constraints one imposes, the more one frees one’s self. And the arbitrariness of the constraint serves only to obtain precision of execution.’
>
> Igor Stravinsky (1882–1971)

5.1 Introduction

As discussed in Chapter 4, SysML has introduced some very useful new elements and diagrams not present in the original UML, namely flow ports, flow specifications and parametric constraints. This chapter looks at these concepts in greater detail and discusses how the connectivity between parts of a system can be modelled, how parametrics can be used to constrain system behaviour and to help with validation of requirements.

After revisiting concepts covered in Chapter 4, each is brought together in a new worked example – a problem from the world of escapology.

5.2 Connecting parts of the system

Modelling a system in terms of the blocks that make up the system does not give a complete structural representation of the system. What is also needed as part of the structural aspect of the model is a set of diagrams that shows how those blocks are connected. This can be done in SysML using *flow ports* and *standard ports*.

Just as with physical connections on consumer electronics – for example, a USB port on a personal computer – these ports show the *interaction points* of the blocks. Just identifying these interaction points is not enough: it is necessary to show also the *information* that can pass in and out of these ports. Figure 5.1 shows the two kinds of

port and the SysML elements that are used to define the information associated with a port.

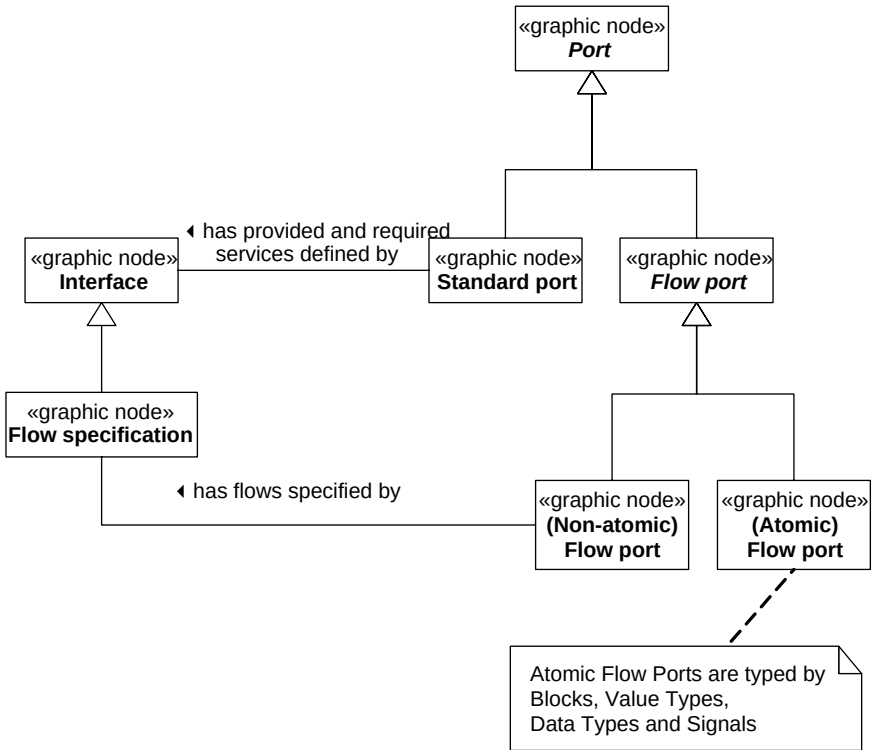

Figure 5.1 Partial meta-model showing types of ports, interfaces and flow specifications

The information flowing in and out of ports is specified using blocks, flow specifications and interfaces. The modelling elements shown in Figure 5.1 are discussed further in the following two sections.

5.2.1 Flow ports and flow specifications

If a block needs to pass or receive material, energy or data, then those interaction points are modelled using flow ports. Each flow port is typed to show what *can* flow through that port. If a single type of material, energy or data is passed then the flow port is considered to be *atomic* and is typed using a block. If multiple types are passed, possibly in opposite directions, then the port is considered to be *non-atomic* and is typed by a *flow specification*.

Revisiting the diving pump example from Chapter 4 illustrates these points. The example is reproduced here as Figure 5.2.

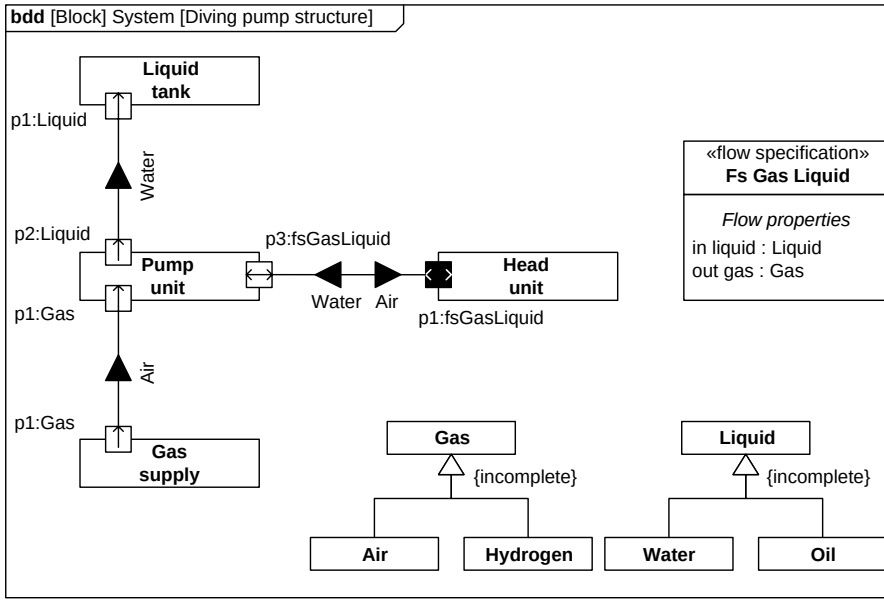

Figure 5.2 Block definition diagram showing the use of flow ports, flow specifications and item flows

Consider the 'Pump unit'. In this system it needs to be connected to a 'Gas supply', a 'Liquid tank' and a 'Head unit'. It receives gas from the 'Gas supply', which it sends down to the 'Head unit'. The 'Head unit' returns liquid to the pump unit, which then sends it off to the 'Liquid tank'.

The 'Pump Unit' therefore has two *atomic* flow ports that are typed using blocks; these are ports 'p1' and 'p2' on the 'Pump unit' in Figure 5.2, typed by the blocks 'Gas' and 'Liquid' respectively. Because its connection to the 'Head unit' requires it to send out gas and receive liquid, this requires the use of a *non-atomic* flow port typed by a flow specification; this is port 'p3' on the 'Pump unit', which is typed by the flow specification 'fsGasLiquid'. A port typed by 'fsGasLiquid' can send out 'Gas' and take in 'Liquid' as shown by the 'out' and 'in' annotation on the types defining the flow properties of the flow specification. Because the flow directions at the 'Head unit' are reversed from the point of view of the definition of the flow specification, a *conjugated* flow port is used, which shows that the directions should be reversed for this port.

Now the types of each port model what *can* flow through the port, not necessarily what *does* flow. In this example it is not 'Gas' and 'Liquid' that are flowing, but 'Air' and 'Water'. The *item flows* connecting the flow ports can model what *does* flow, which is shown by the type next to the black triangle on the item flow. As long as the type of the material, data or energy that does flow is the same as, or a *specialization* of, the type associated with the port, it can be used on the item flow. For example, in

Figure 5.2 'Air' can be used to type the item flow between the 'Gas supply' and the 'Pump unit' because 'Air' is a type of 'Gas'.

This ability to differentiate between the types of material, data or energy that can flow through a port and that which does flow is very useful. It means that it is possible to model the structural aspects of a system using blocks with generic interaction points and then reuse those blocks to solve specific problems. In the example above, the four elements have ports that pass generic 'Gas' and 'Liquid'. When assembled to build a diving pump they actually pass 'Air' and 'Water. Another part of the system, or even a different system, could reuse these blocks but pass say 'Hydrogen' and 'Oil' rather than 'Air' and 'Water'. The blocks and ports are defined only once, but the item flows used to connect them supply the specific information for a given usage context.

5.2.2 Standard ports and interfaces

If a block needs to communicate with another block by invoking operations on that block, then it is considered to be communicating via *services*; the operations that can be invoked are the services that the second block provides. Such communication is modelled using *standard ports* and *interfaces*.

Standard ports specify the interaction points through which the communication takes place. The block that initiates the communication does so via a *required* interface. This required interface is connected to a *provided* interface on the block providing the services invoked. The types of both the required and provided interfaces must match. The interfaces are attached to standard ports on each block using the 'cup' and 'ball' notation, shown in Figure 4.9 in Section 4.5.2 of Chapter 4. A block may have multiple ports that all use the same interface, and a port may also have both required and provided interfaces attached. This is often done when communication can be invoked by either block. In this case the required and provided interfaces will be of the same type.

As well as showing the interfaces that each block requires and provides – that is the *usage* of the interfaces – it is also necessary to *define* the services provided by the interface. This is done with a block that is stereotyped ≪Interface≫ and that has no properties and only operations. The operations identify the services that are available via that interface. How those operations are implemented is not defined by the interface definition. Rather, any block that provides that interface must implement those operations and may do so in any way it requires. This means that, if multiple blocks all provide the same interface, the way they implement the interface can vary from block to block. In essence, an interface is a *contract* that any block providing that interface must fulfil. A block can also provide many different interfaces; if so, then the block must implement the operations of *all* the interfaces it provides.

The example below, previously seen in Section 4.5.3 of Chapter 4 and reproduced here as Figure 5.3, gives an example of the use of standard ports and interfaces.

Figure 5.3 defines two interfaces, 'SensorIF' and 'Drive'. The 'Controller' block *provides* the 'SensorIF' via three standard ports, and *requires* the 'Visual' and 'Drive' interfaces (with the 'Drive' interface again connected to three standard ports). The

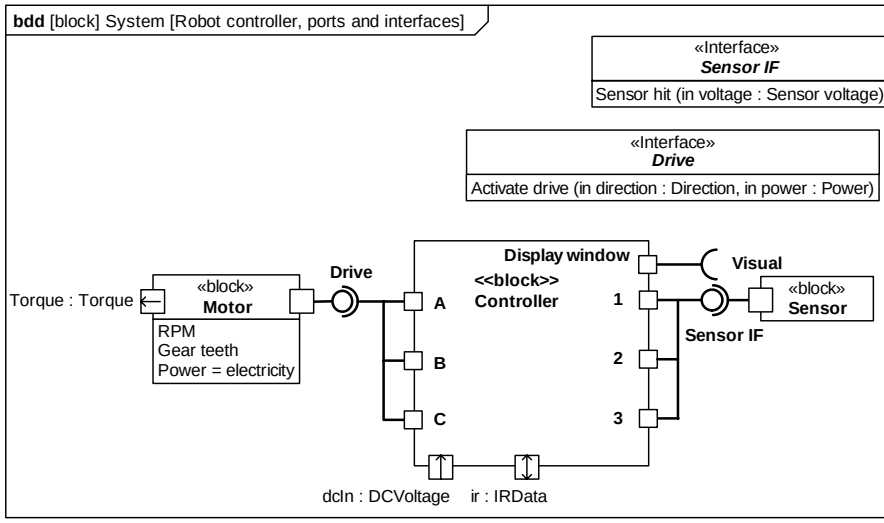

Figure 5.3 Block definition diagram showing the definition and use of standard ports and interfaces

'Sensor' block *requires* the 'SensorIF', and the 'Drive' block *provides* the 'Drive' interface. This means that 'Controller' must implement the 'sensorHit' operation defined in the 'SensorIF' interface and 'Drive' must implement the 'activateDrive' operation defined in the 'Drive' interface.

Because an interface defines a contract, blocks can be substituted as long as they have the necessary interface. For example, if another type of motor was defined that also provided the 'Drive' interface, then this could be attached to the 'Controller', which would be able to communicate with this new motor since the interface is the same: the new motor has fulfilled the contract specified by the 'Drive' interface. This is shown in Figure 5.4.

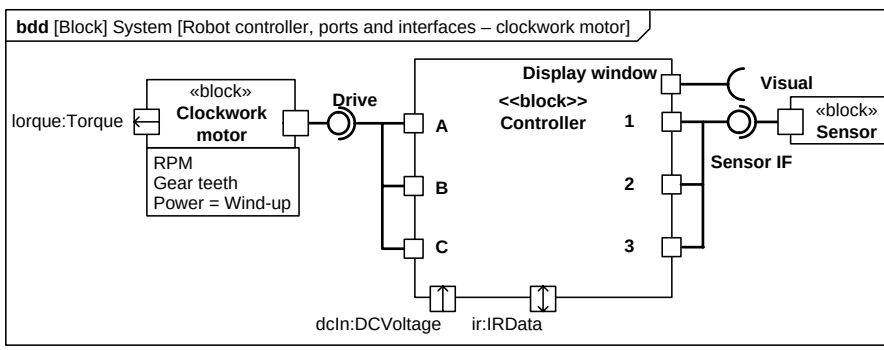

Figure 5.4 Block definition diagram showing different type of motor

It should be noted in the diagrams that the ‘Controller’, ‘Motor’ and ‘Clockwork motor’ blocks have standard ports, interfaces *and* flow ports. This is entirely typical of complex systems. Systems that use only standard ports and interfaces are examples of what are termed *service-oriented architectures*. Typical examples of pure service-oriented architectures are software-only systems.

5.3 Allocations

As noted in Chapter 4, the UML, on which SysML is based, has a diagram called the *deployment diagram*, that shows how various elements of a system are deployed. Inexplicably, this diagram has been omitted from the SysML. Instead, the SysML specification defines an *allocation* notation using stereotypes and versions of the comment to show how various elements are allocated to and 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.

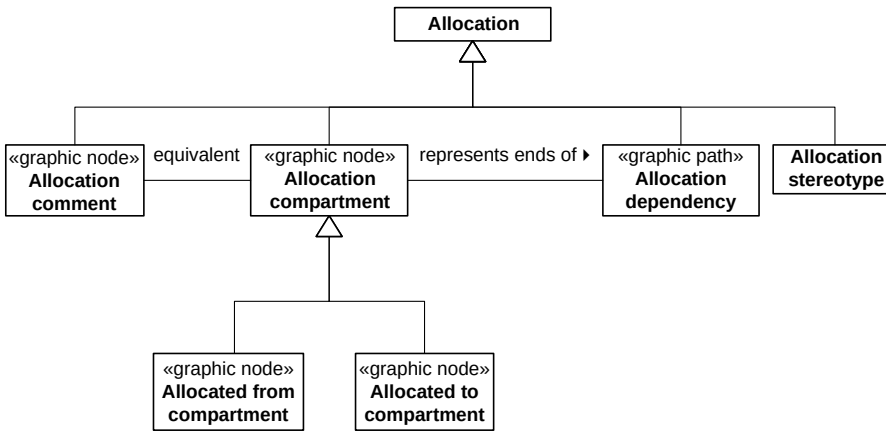

Figure 5.5 Partial meta-model for allocations

Figure 5.5 shows the partial meta-model for allocations and shows that allocations can be represented in four different ways: as an ‘Allocation comment’; an ‘Allocation compartment’ (either an ‘allocatedFrom compartment’ or an ‘allocatedTo compartment’) on an existing graphic node; as an ‘Allocation dependency’ between system elements (with each end of such a dependency equivalent to one of the two types of ‘Allocation compartment’; or as an ‘Allocation stereotype’. An ‘Allocation comment’ and an ‘Allocation compartment’ are essentially equivalent as can be seen in Figure 5.6, where the notation used for allocations is shown.

The notation for allocation compartments and allocation comments in Figure 5.6 shows the use of the stereotype ≪elementType≫. In an actual use of this notation ≪elementType≫ is replaced with a stereotype indicating the actual SysML element

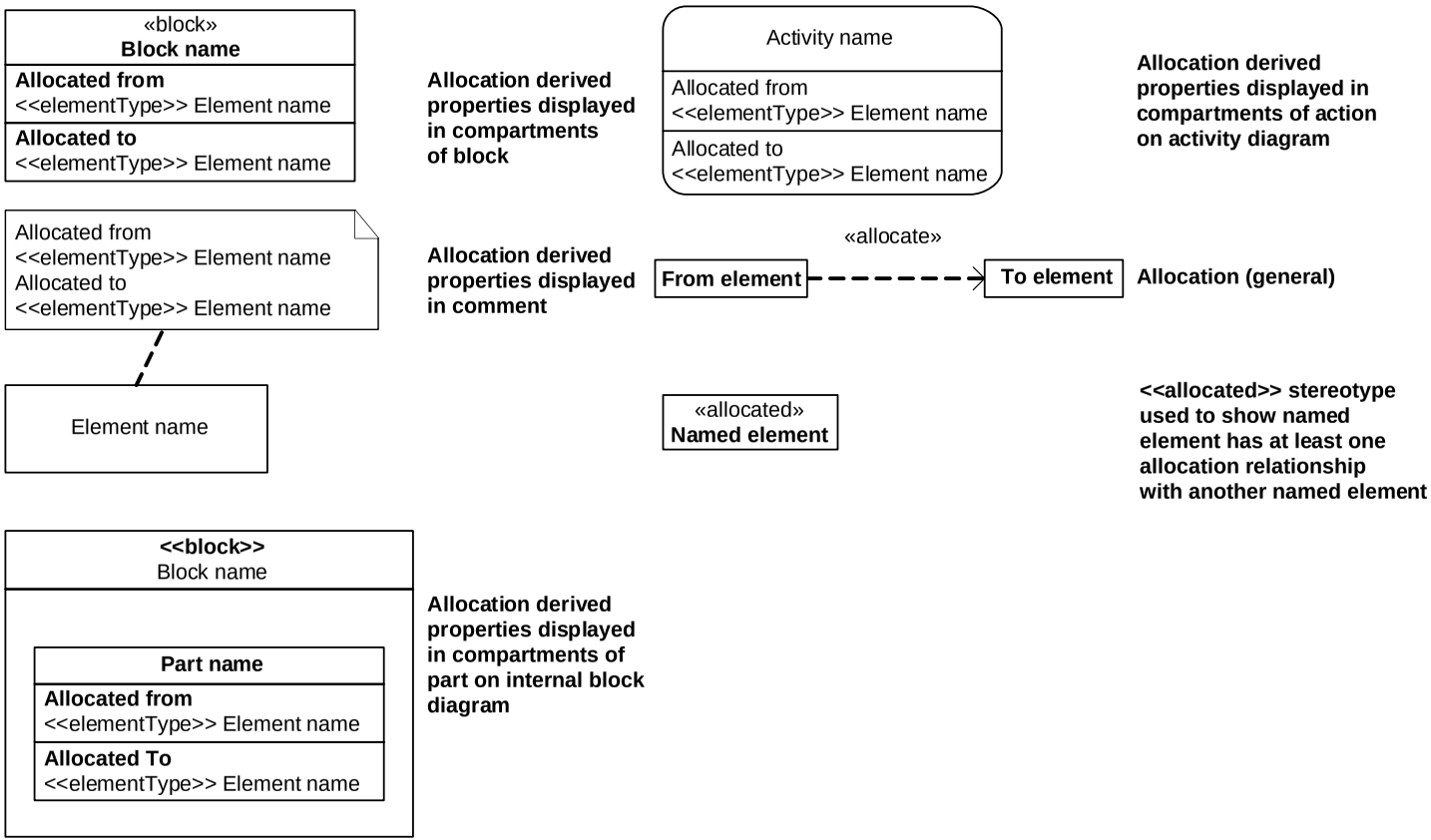

Figure 5.6 Graphical symbols for allocations

involved in the allocation. For example if the element being allocated from is a block, say, then ≪elementType≫ would be replaced with ≪block≫.

An example showing use of the allocation dependency is given in Figure 5.7. This diagram was seen previously in Chapter 4 and shows the use of the allocation dependency to model the concept of deployment.

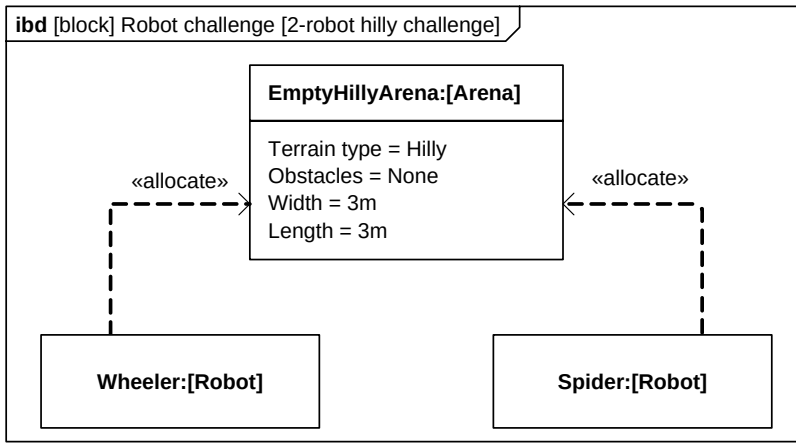

Figure 5.7 Allocations – dependency notation

The same information is repeated in Figure 5.8, but this time with one of the allocation dependencies replaced by two comments, one on the 'Robot' showing what it is allocated *to*, the other on the 'Arena' showing which system element it has an allocation *from*.

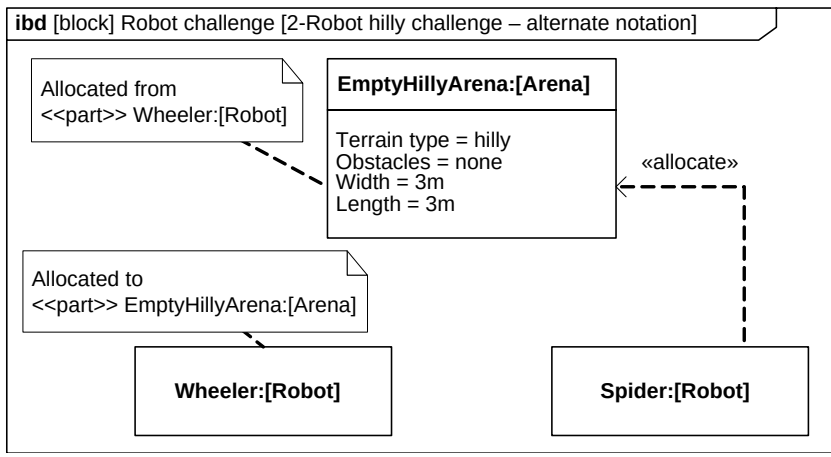

Figure 5.8 Allocations – comment notation

A word of warning should be given here. The allocation dependency gives a strong visual link between the related elements, and, because the two elements are directly connected, consistency of the model is ensured. When using allocation compartments or allocation comments this direct link is missing. It is therefore necessary to take care that the elements at *both* ends of the allocation are marked with either allocation compartments or allocation comments. If one end is not marked, an inconsistent model will result.

5.4 Parametric constraints

Parametric constraints provide a powerful new modelling notation not present in the UML. However, parametric constraints are not used in isolation and must be related to blocks both for their definition and their usage. While Section 4.8 of Chapter 4 covers the meta-model and notation for parametric constraints, along with guidance on how they are used, this section discusses the relationship that constraints must fulfil in order to produce a consistent model. It also considers a possible classification of constraints that the authors have found to be very powerful in modelling real-world systems. The remainder of this chapter discusses their use in greater detail than found in Chapter 4.

5.4.1 Relationships to blocks

In order to realize constraints fully, three views are needed:

- a *system view*, showing the structure of the system, in order to understand *what* is being constrained;
- a *constraint definition view*, showing the constraints that will be used and how they are *defined*; and
- a *constraint usage view*, showing how the constraints are *used* to constrain the elements defined in the system view.

These views, and the relationships between them, are discussed further below.

Parametric constraints are related very closely to the blocks that are used to define the structure of a system. Consider Figure 5.9.

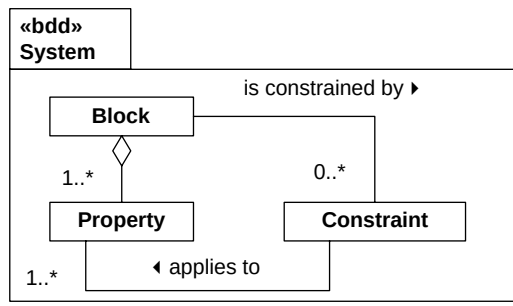

Figure 5.9 The relationship between blocks and constraints – system level

In Figure 5.9 the system is represented as a package of block definition diagrams (indicated by the ≪bdd≫ stereotype). The system elements are modelled using a number of 'Block', each of which is made up of one or more 'Property' (and, of course, other things such as operations, which have been omitted for clarity). Each 'Block' is constrained by one or more 'Constraint'.

These constraints may be 'ordinary' constraints represented by an expression in the constraint compartment of the block, but of interest to us here are constraints represented by parametric constraints. Again, such parametric constraints can be *explicitly* shown in a constraint compartment of the block as references to parametric constraint definitions modelled elsewhere. The constraints can also be *implicitly* related to blocks through their use on parametric diagrams. In either case a 'Constraint' applies to one or more 'Property' of a block.

Now consider Figure 5.10.

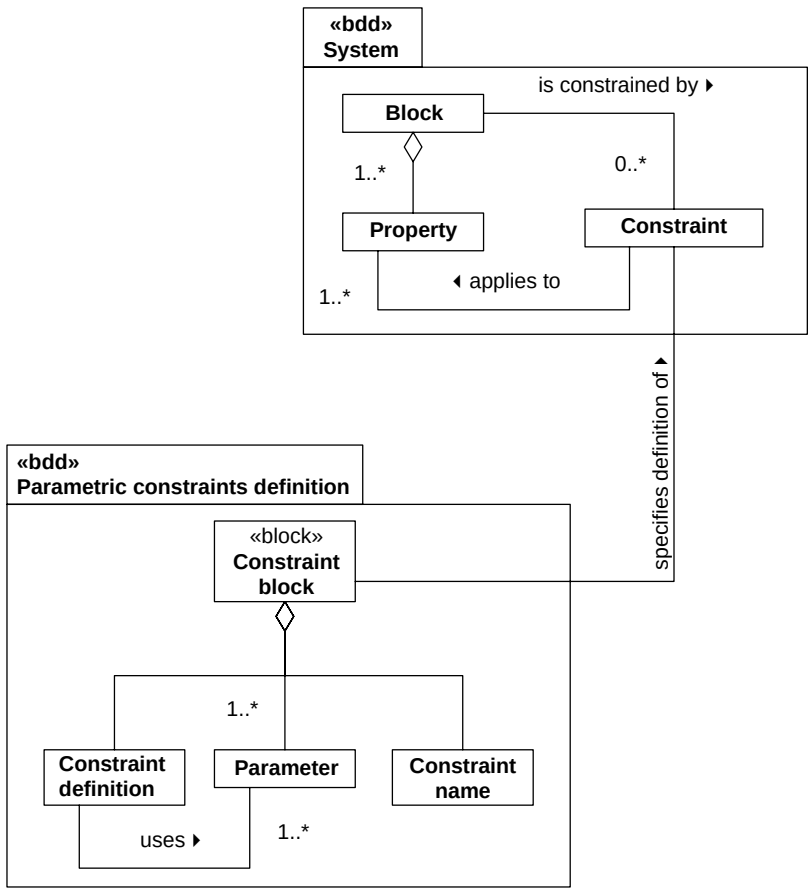

Figure 5.10 The relationships between blocks and constraints – constraint definitions

In Figure 5.10 another package, again stereotyped ≪bdd≫, has been added to the diagram. This new package represents the block definition diagrams used to *define* the parametric constraints that apply to the system.

These block definition diagrams contain a number of 'Constraint block' (modelled as blocks), each of which is made up of a 'Constraint name' (the name of the block), a 'Constraint definition' and one or more 'Parameter'. The 'Constraint definition' uses one or more of these 'Parameter'. (In fact the definition should use *all* of the parameters. If any are *not* used by the definition, then they should not *be* parameters and should be removed from the block.) Each 'Constraint block' specifies the definition of a 'Constraint' that is used to constrain the system. That is, it defines a constraint used by a block modelling a system element.

Now let us turn to the usage of parametric constraints. Consider Figure 5.11.

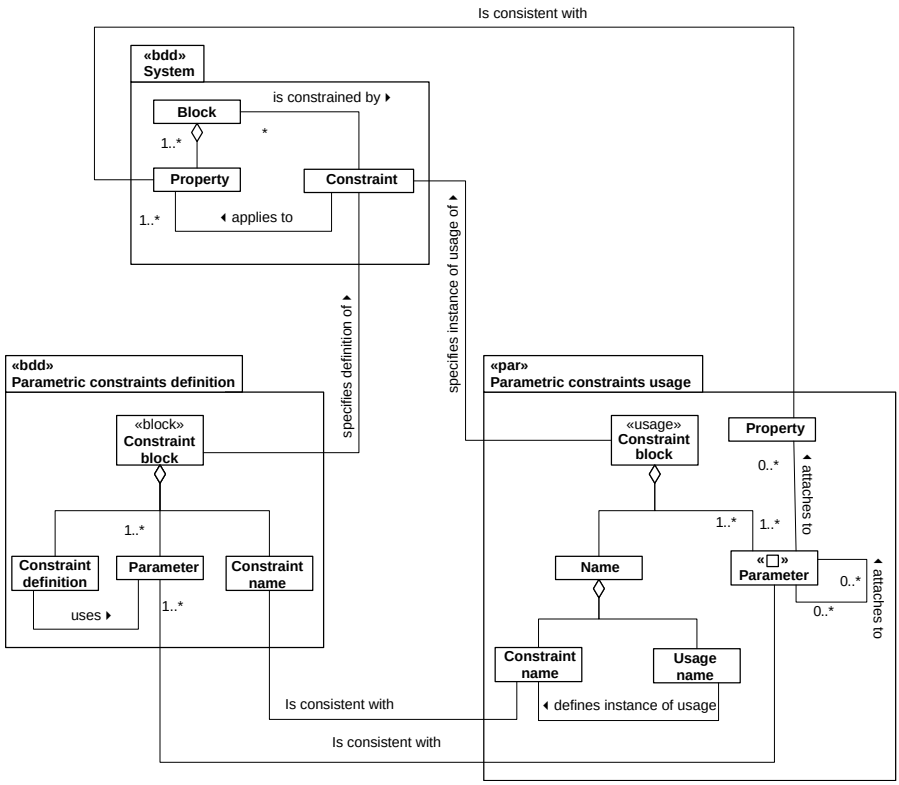

Figure 5.11 The relationship between blocks and constraints – constraint usage

Again, another package has been added. This time the stereotype ≪par≫ has been used to denote a collection of parametric diagrams, which model the usage of the parametric definitions. Each parametric diagram consists of a number of 'Constraint

block' (this time modelled as a usage), which are made up of a 'Name' and one or more 'Parameter'. The diagrams also contain a number of 'Property'.

The 'Name' is itself made up of a 'Constraint name' and a 'Usage name'. The 'Usage name' simply names an instance of usage of the constraint named by the 'Constraint name' – a parametric diagram can have many instances of a given parametric constraint and the 'Usage name' identifies each of these instances.

Each 'Constraint block' on a parametric diagram specifies an instance of usage of a 'Constraint' that is used to constrain the system. That is, it shows the usage of a constraint used by a block modelling a system element. For this reason the 'Constraint name' of the usage must be consistent with the 'Constraint name' of the constraint definition.

Each 'Parameter' that is part of a 'Constraint block' usage must be consistent with a 'Parameter' that is part of its 'Constraint block' definition – parameters appearing on a parametric diagram must match those appearing in the definitions of those parameters. Each 'Parameter' may be attached to zero or more other 'Parameter' – this is because constraint-block usages may be connected to other usages. If a 'Parameter' is *not* connected to another then it *must* be connected to one or more 'Property' appearing on the diagram. (There is one exception to this which is explored in 5.5.4.2 below.) Each of these 'Property' must be consistent with a 'Property' of a block defining a system element.

In summary, Figure 5.11 provides a high-level meta-model that illustrates the main consistency relationships between blocks modelling system elements and the definitions and usages of parametric constraints that constrain those blocks.

5.4.2 Types of constraint

The SysML specification defines a single type of constraint, the constraint block, as indicated by the stereotype ≪constraint≫. All the examples in the specification that show how to use constraint blocks consist of formulae representing physical laws. However, the concept of the constraint block is much more powerful than this. Consider Figure 5.12.

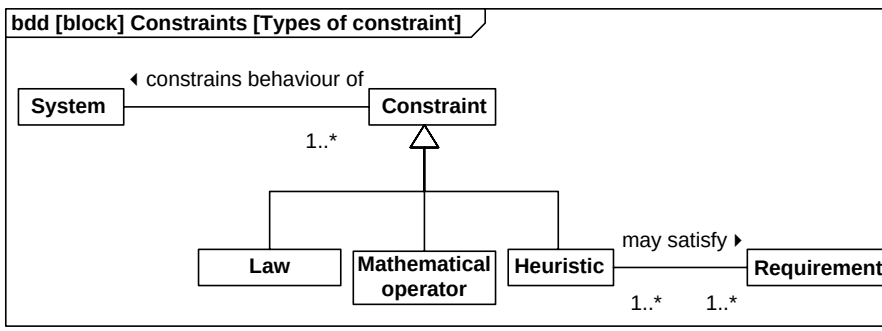

Figure 5.12 Types of constraint

Figure 5.12 shows three types of constraint: 'Law', 'Mathematical operator' and 'Heuristic'. These are intended to allow different types of constraint to be represented in a model, conveying extra information about the constraints. They are intended to be used in the following way.

- 'Law' constraints are essentially the same as the examples given in the SysML specification, representing physical laws or other formulae.
- 'Mathematical operator' constraints represent, unsurprisingly, mathematical (and logical) operators. When using constraints on a parametric diagram, 'Mathematical operator' constraints make it easier for other constraints to be connected together in a constraint usage network.
- 'Heuristic' constraints are used to represent decisions rather than calculation-type constraints, evaluating input parameters against some criteria and returning a result that could be, for example, a 'yes/no', 'true/false' or 'go/no-go'. These are often used to show that requirements can be satisfied.

Examples of definitions of each type of constraint are shown in Figure 5.13. Additional examples are defined and used in Sections 5.5.3 and 5.5.4 below.

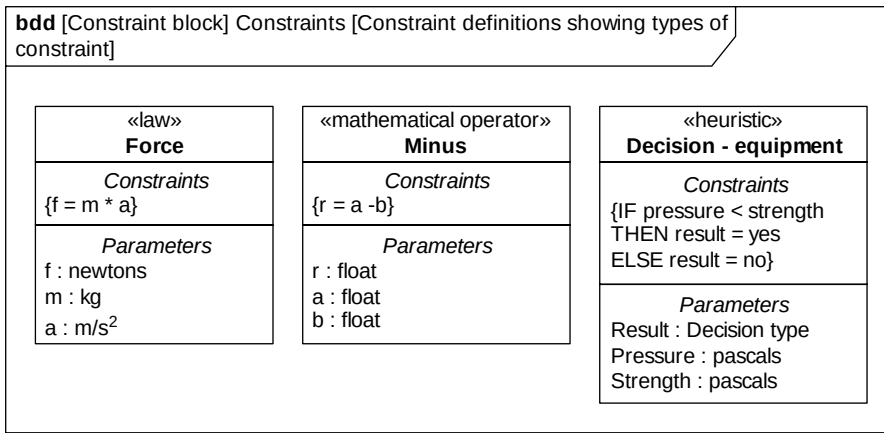

Figure 5.13 Examples of each type of constraint

It is important to note that these three types of constraint are *not* part of the SysML specification, but represent types that have been used by the authors in day-to-day practical application of the SysML. Also, they represent only one *possible* set of constraint types. The exact number of types of constraint and the names used can be modified as required, and the reader is encouraged to attempt this in the context of the types of system they are developing.

Nevertheless, the authors feel that this approach of defining different types of constraint adds significantly to the modelling power of the constraint block. Of particular power is the 'Heuristic' constraint, as these may be used to test whether requirements

have been or indeed *can* be satisfied. This is discussed further in the context of the escapology example below.

Also of note in Figure 5.13 are the types of the parameters in the various constraint definitions. While the 'minus' constraint simply has parameters typed as 'float', the other two constraints have types such as 'newtons' and 'm/s^2'. The intention here is that these types also indicate the units involved. The SysML specification contains definitions of many such SI units. Alternatively, their definition is left as an exercise for the reader.

A block definition diagram should be produced that defines these types. This is left as an exercise for the reader.

5.5 Putting it all together – the escapology problem

It is now time to put all these concepts together, which will be done by looking at an example from the world of escapology. In this example an escapologist is placed in a rectangular coffin, which is then placed in 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 5.14 shows the setup for the escape.

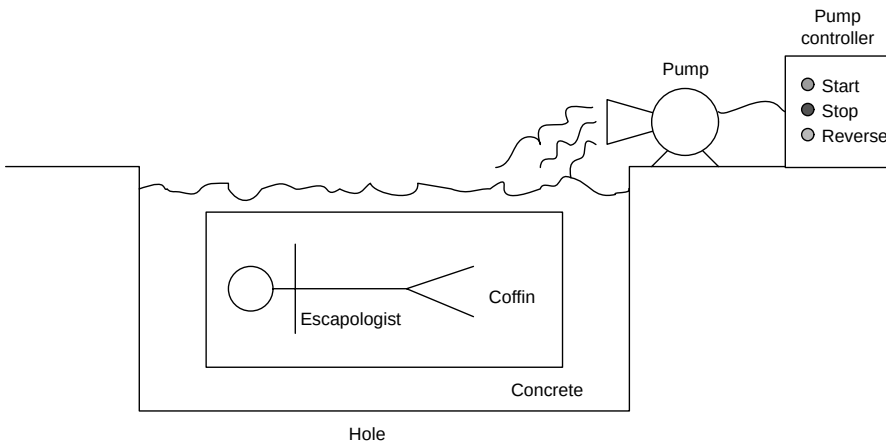

Figure 5.14 The coffin escape

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 constraints were not properly understood or evaluated. One such performer was Joe Burrus, who died on 30 October 1990, when the weight of the concrete crushed the coffin he was in.

How this escape can be modelled in SysML is discussed in the following sections.

5.5.1 Requirements

The requirements for the escapology stunt are shown in Figure 5.15.

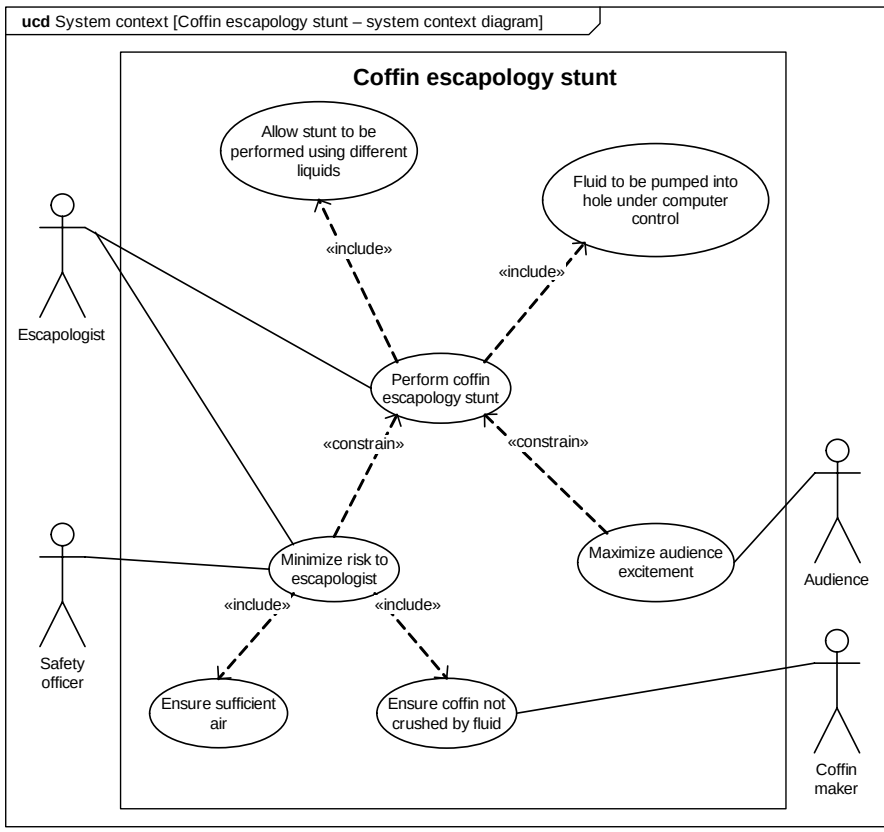

Figure 5.15 The coffin escape – requirements

The main requirement, of interest to the 'Escapologist', is to 'Perform coffin escapology stunt', which includes the requirements 'Allow stunt to be performed using different liquids' and 'Fluid to be pumped into hole under computer control'. Although the stunt is usually performed using concrete as the fluid, the escapologist wants the flexibility to be able to perform using different liquids, such as water. In order to remove potential human error when operating the pump, the escapologist requires that the pump be under computer control so that it can be activated for a precise amount of time.

There are two requirements that act as constraints on the main 'Perform coffin escapology stunt' requirement. The first of these is 'Maximize audience excitement', which is of interest to the 'Audience' stakeholder, as there is no point in performing any escapology stunt without an element of danger (even if only perceived danger) to

captivate the audience. The second constraint is 'Minimize risk to escapologist'. This is of interest to the 'Escapologist' and also to the 'Safety officer'. Both want to ensure that the stunt can be performed with as low a risk to the escapologist as possible, but possibly for different reasons: the escapologist doesn't want to be injured or killed and the safety officer doesn't want the venue hosting the stunt to get sued should anything go wrong.

'Minimize risk to escapologist' includes two further requirements: 'Ensure sufficient air' and 'Ensure coffin not crushed by fluid'. This latter is of interest to the 'Coffin maker' who supplies coffins used in the stunt. Any coffin used must be strong enough to withstand the pressure of the fluid filling the hole above the coffin. It is worth re-emphasizing here that roles may be taken by single or multiple people, and that a single person may have multiple roles. If the performer not only enacts the role of 'Escapologist', but also builds his own coffins, then he would also take on the role of 'Coffin maker'.

5.5.2 Definition of the system

With the requirements for the stunt defined it is now possible to define the components of the system and the interfaces between them. This is the system view discussed in Section 5.4.1. Figure 5.16 shows the main components.

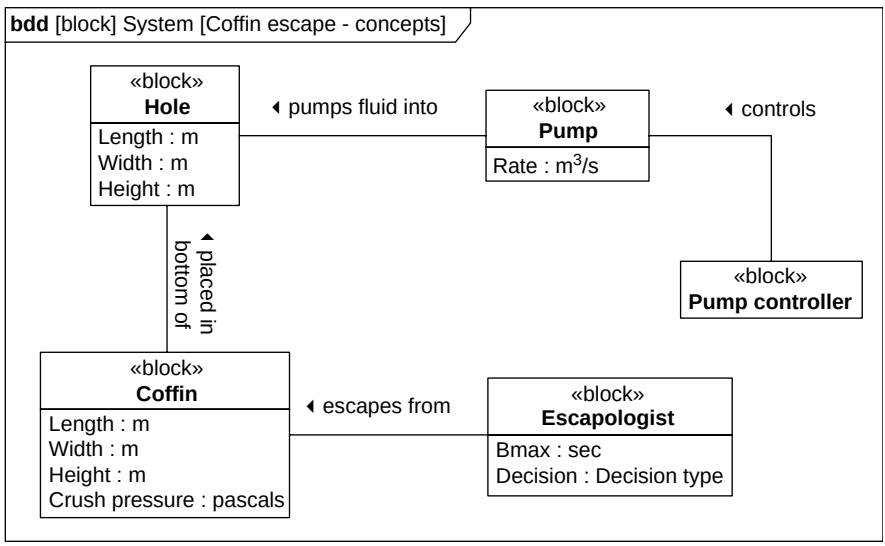

Figure 5.16 The coffin escape – system components

In Figure 5.16 we see that the main system components are the 'Pump controller' controlling the 'Pump', which pumps fluid into the 'Hole'. The 'Escapologist' escapes from the 'Coffin', which is placed in the bottom of the 'Hole'. Various properties have

been defined for the blocks representing the system components. These properties will be connected to the various parametric constraints as shown in Figure 5.21 in Section 5.5.4 below.

The deployment of the 'Escapologist' to the 'Coffin' and the 'Coffin' to the 'Hole' is shown in Figure 5.17, making use of the ≪allocate≫ dependency to show this.

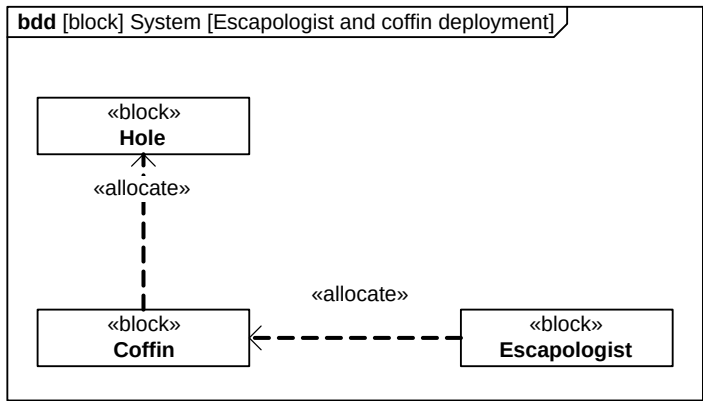

Figure 5.17 The coffin escape – deployment

With the blocks representing the main components defined, the interfaces between the blocks can be defined. This has been done in Figure 5.18.

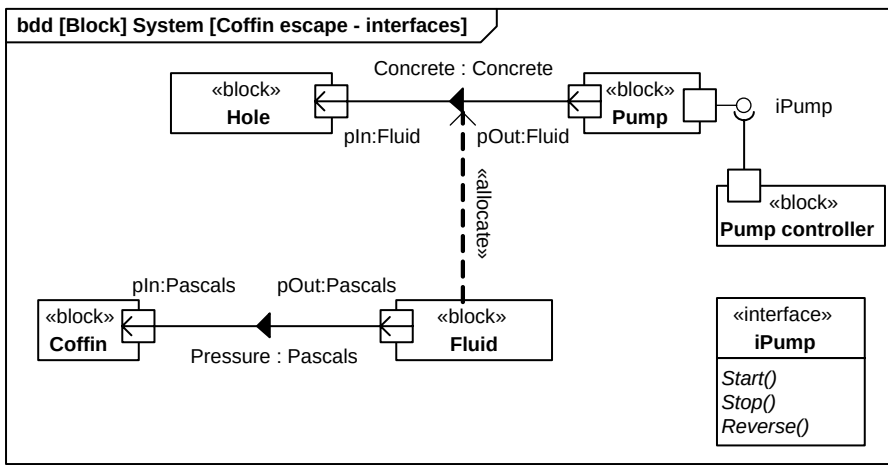

Figure 5.18 The coffin escape – system interfaces

The 'Pump controller' is connected to the 'Pump' via the 'iPump' interface attached to service ports on the blocks. The 'iPump' interface block defines the

operations supported by this interface. The ‘Pump’ and the ‘Hole’ each have a flow port of type ‘Fluid’, with an item flow linking these ports. This item flow shows that it is actually ‘Concrete’ that flows from the ‘Pump’ to the ‘Hole’ rather than a generic ‘Fluid’. In order to ensure consistency it is necessary to model the types of ‘Fluid’ that the system can support. This is done in Figure 5.19, which defines a number of types of ‘Fluid’, including ‘Concrete’. The {incomplete} constraint is used to show that further fluid types may need to be defined.

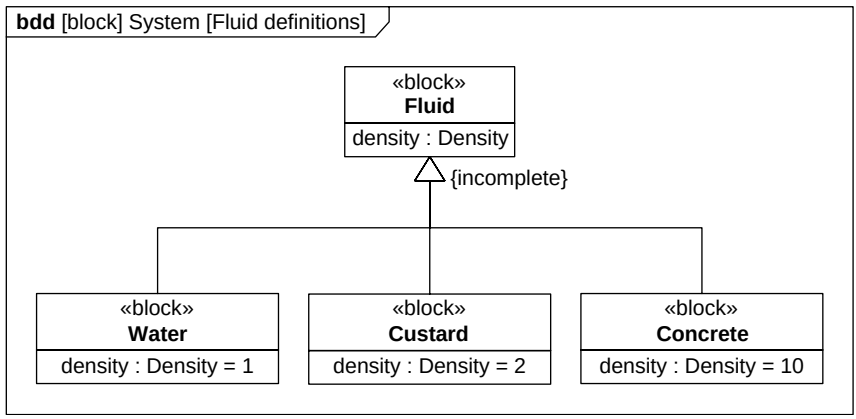

Figure 5.19 The coffin escape – types of fluid

Figure 5.18 uses flow ports to show that the ‘Fluid’ and the ‘Coffin’ are connected by the pressure that the ‘Fluid’ exerts on the ‘Coffin’, with an item flow showing the transfer of pressure in pascals from the ‘Fluid’ to the ‘Coffin’. Finally, an ≪allocate≫ dependency is used to relate the ‘Fluid’ block to the item flow between the ‘Pump’ and the ‘Hole’.

The three diagrams in this section, along with the diagrams in subsequent sections showing the definition and usage of the parametric constraints, are presented in a logical manner intended to show a move from the concrete (showing the elements that make up the system) to the more abstract (showing interfaces between elements, then the constraints placed on those elements).

However, it is important for the reader to understand that it is rarely possible actually to develop the diagrams in this fashion. Indeed, in producing the diagrams for this example the authors worked on all five diagrams in parallel. Thinking about the constraints that were needed and the way they had to be connected (see Figures 5.20 and 5.21) helped determine the definitions of the constraints and also the properties of the blocks shown on Figure 5.16. Knowing how the components were connected helped in the modelling of the interfaces and the types of fluid shown in Figures 5.18 and 5.19 respectively. It cannot be emphasized enough that such a way of working is entirely normal when modelling with SysML – a change to one diagram often means that other diagrams have to be revisited and even new diagrams created. Anyone

using SysML who says, 'I have finished the X diagrams, now I have to do the Y diagrams' has not understood how to model with the language.

5.5.3 Definition of the constraints

With the system defined and the requirements understood it is possible to start thinking about the parametric constraints that need to be defined. One possible set of constraint definitions is given in Figure 5.20.

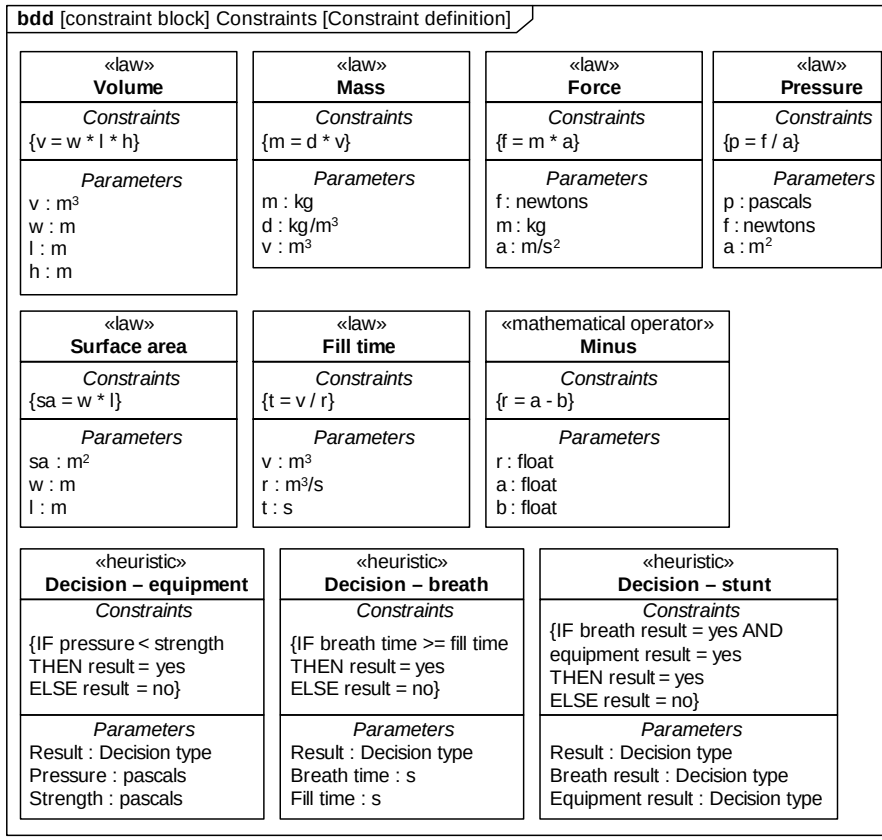

Figure 5.20 The coffin escape – parametric definitions

Looking at the requirements for the coffin escape shown in Figure 5.15, we can see that the stunt is constrained by the need to 'Minimize risk to escapologist' and, in particular, to 'Ensure sufficient air' and to 'Ensure coffin not crushed by fluid'. This suggests the need for a heuristic constraint that checks whether the escapologist can hold his breath long enough while the hole is being filled and for one that checks whether the coffin can survive the pressure of the fluid on top of it. A further heuristic

can then be used to check the results of these two heuristics and provide a final decision on whether or not the stunt should be performed. These three heuristics are modelled as the parametric constraints 'Decision – breath', 'Decision – equipment' and 'Decision – stunt'. In Figure 5.20, these three parametric constraint definitions are stereotyped ≪heuristic≫ to show the type of constraint they represent. These heuristics are related directly to the requirements for the escapology stunt, enabling the requirements to be tested to ensure that they have been or can be satisfied. Different requirements for the stunt would require the definition of different heuristics.

Having determined the decisions that need to be taken, it is then possible to define the other parametric constraints that are needed to support these heuristics. In practice, the most effective way of determining these additional constraints is, as noted in Section 5.5.2, by working on the definitions and usage in parallel, working backward from the final heuristic on Figure 5.21 and adding the necessary constraints to the network, creating the parametric constraint definitions as their usages are added.

Doing this results in the other constraints shown in Figure 5.20. The stereotypes ≪law≫ and ≪mathematical operator≫ have been added to indicate the nature of the constraints. Some of these parametric constraint definitions, when used, will be connected to properties of the blocks representing the system elements. It is essential that the two are consistent. Indeed, defined block properties may determine the way the parametric constraints can be defined or, conversely, the needed definitions may result in additional properties being required in some of the blocks.

The next step is to show how these are used in the system.

5.5.4 Using the constraints

Defined parametric constraints can be connected together with block properties on a parametric diagram that shows a usage of the constraints. This is shown in Figure 5.21, where the constraints are connected together with properties of the blocks that define the escapology stunt system in a usage network that supports the decision as to whether or not the coffin escape can be attempted.

The sizes of the hole and the coffin are calculated and used to determine the amount of concrete needed to fill the hole. This volume and the pumping rate of the pump are used to determine how long it will take to fill the hole. This forms an input to a usage of the 'Decision – breath' heuristic constraint, along with the length of time that the escapologist can hold his breath, which returns a 'yes/no' decision indicating whether the escapologist can hold his breath long enough.

The volume of concrete needed is used, along with a constant defining the acceleration due to gravity, to calculate the amount of force exerted by the concrete. This force is converted to an exerted pressure using the surface area of the coffin, with the pressure then being compared against the coffin crush pressure in a usage of the 'Decision – equipment' heuristic constraint to return a 'yes/no' result that indicates whether or not the coffin is safe to use.

Finally, the outcomes of these two decisions are used in a usage of the 'Decision – stunt' heuristic to decide whether the stunt should be performed, setting the 'decision' property of the 'Escapologist' block. In this way the parametric constraints are used

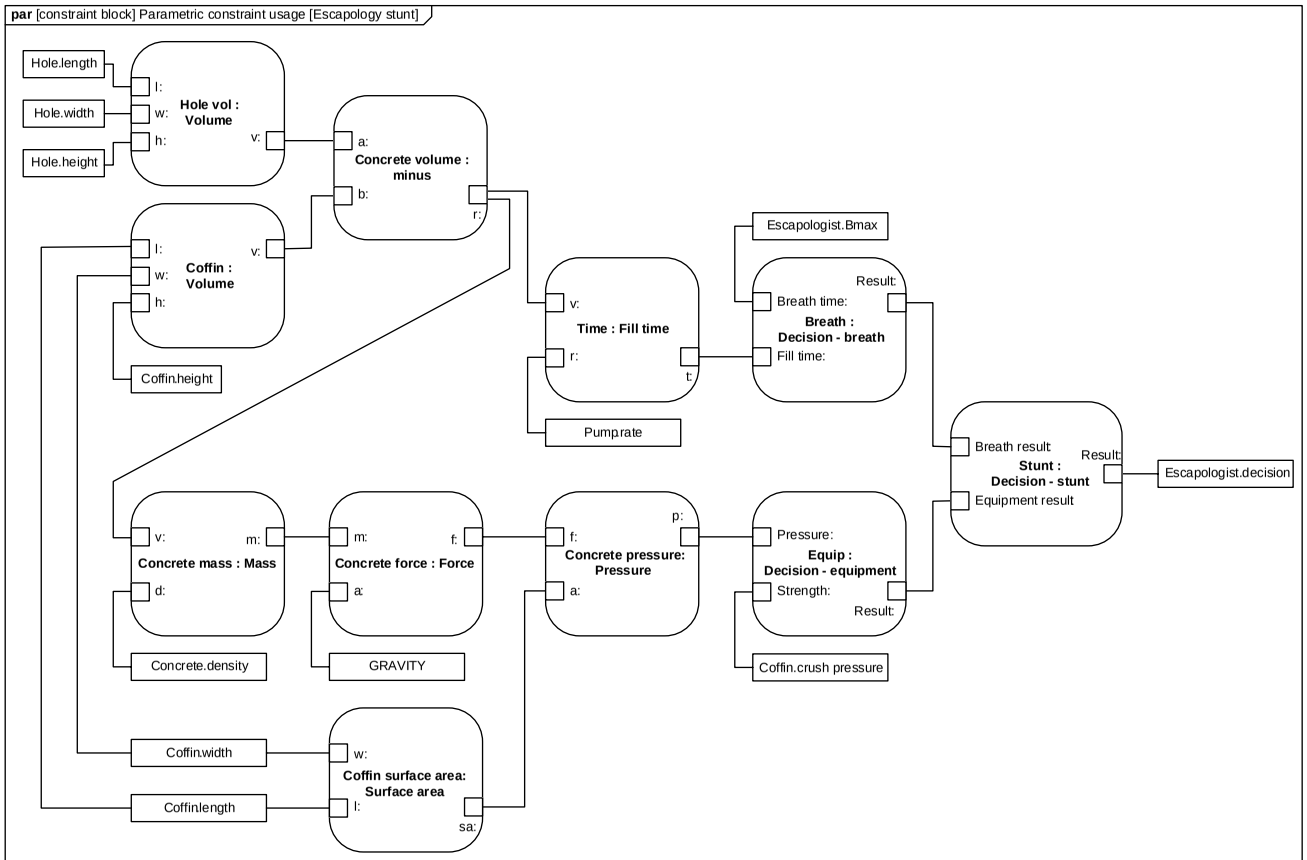

Figure 5.21 The coffin escape – parametric usage

not only to specify constraints on the system but also to allow system requirements to be validated. Indeed it may be possible, if parametric constraints can be developed at an early stage of a project, to use them to establish whether a project is even possible long before detailed and costly development work has been undertaken.

5.5.4.1 Different contexts

The parametric usage shown in Figure 5.21 has been drawn and explained from the point of view, the context, of only one of the stakeholders – that of the escapologist. But what about the other stakeholders? What about the coffin maker or the safety officer, for example?

Each parametric constraint defined in Figure 5.20 can be thought of as consisting of n variables, one 'output' variable and $n-1$ 'input' variables. Indeed, this is the way that a constraint is defined by the text in the *constraints* compartment of the defining constraint block. The way that the constraints have been drawn in Figure 5.21 reflects this, with the input variables drawn on the left of the parametric usage and the output variable drawn on the right. However, this representation is only a convention adopted by the authors. See Figure 5.22.

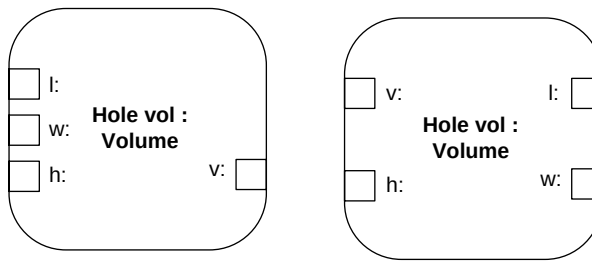

Figure 5.22 Example of notation convention

The diagram on the left of Figure 5.22 shows this convention, with the 'inputs' on the left and the 'output' on the right. However, the parameters can be positioned anywhere around the edge of the parametric usage. The diagram on the right of the figure represents exactly the same usage but with the parameters arranged differently. This is just one of many possible arrangements.

In adopting this convention the usage diagram is drawn as though all the 'inputs' (i.e. the free variables) are already determined or fixed. However, the diagram could be used in the 'other' direction.

For example, going 'backwards' through Figure 5.21 we can 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 the 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.

As another example, if, say, a maximum hole size was always used, then the parametric usage diagram could be used by the coffin maker to determine the crush pressure of the coffins he makes. Again, we can think of this usage being run 'backwards' through the network in Figure 5.21. Alternatively, we could draw additional usage diagrams specifically for the other stakeholders. An example is given in Figure 5.23.

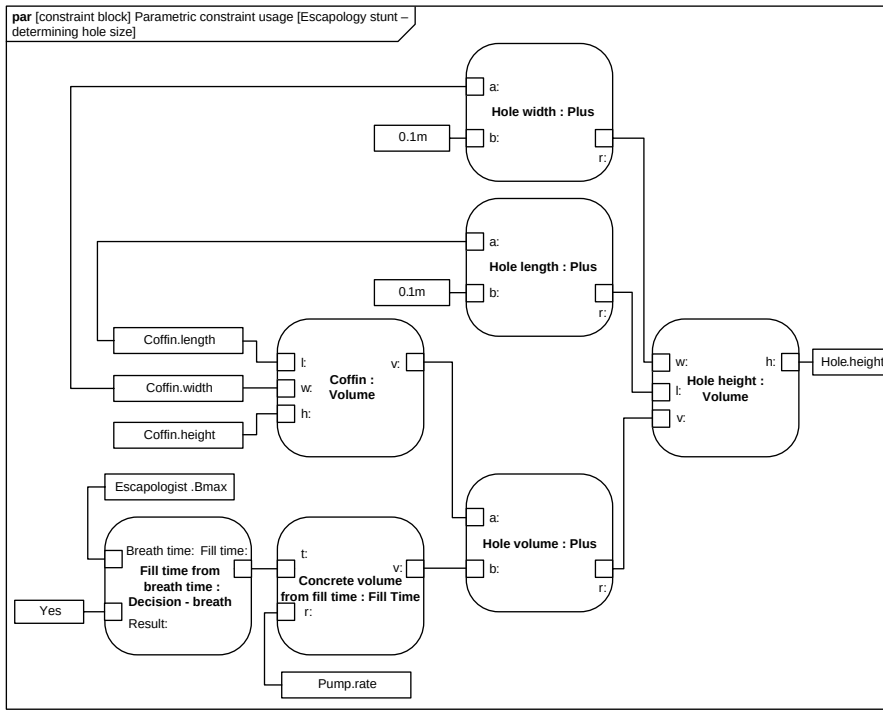

Figure 5.23 The coffin escape – parametric usage diagram – safety officer's context

Figure 5.23 gives an alternative parametric usage diagram for the safety officer, showing how the size of the hole could be calculated, as described previously. The parameters have been rearranged within each parametric usage element to conform to the notation convention described above, with inputs on the left of each element and the output on the right. Note also the use of a new constraint, 'plus', which is a mathematical operator that outputs the result of adding its two inputs. The definition of this constraint is left as an exercise for the reader.

One possible change to parametric usage diagrams would be to stereotype the parameters on the usage diagram as ≪input≫ and ≪output≫ or ≪fixed≫ and ≪free≫ to show how they are intended to be used on that diagram. A parametric usage diagram could then be reused with the same shape and connections,

but with the relevant stereotypes changed to indicate what is being constrained or calculated by the usage network.

5.5.4.2 Nesting constraints

Parametric constraints can also be 'nested', that is, they can be grouped into higher-level constraints that make use of existing constraint definitions. Consider the three parametric constraints in the top left of Figure 5.21 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. First we define the new constraint as shown in Figure 5.24.

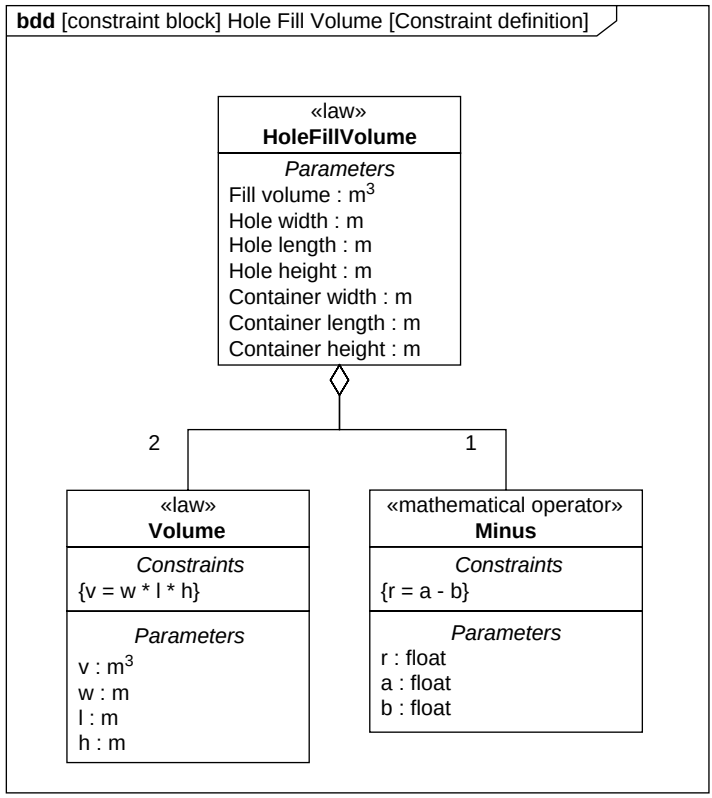

Figure 5.24 The coffin escape – parametric definition showing how higher-level constraints can be constructed

'HoleFillVolume' is defined as being made up of two 'Volume' constraints and one 'minus' constraint and has a number of parameters defined. However, the actual constraint is *not* defined on this diagram. For this we need a special parametric diagram that shows how the component constraints are used. This is shown in Figure 5.25.

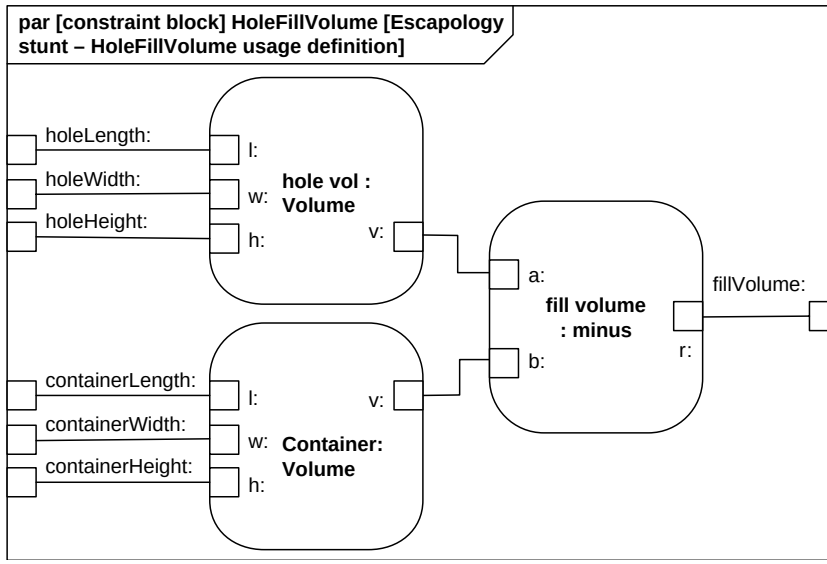

Figure 5.25 The coffin escape – parametric usage showing how higher-level constraints can be constructed

It must be noted here that the SysML specification is unclear on the exact syntax to be used when defining such nested constraints, and that the example presented here is the best interpretation that the authors can make based on the information given in the specification. Unlike the definitions of individual constraints (as shown in Figure 5.20), the definition in Figure 5.24 does not show how the parameters are related to each other or how they are related to the parameters of the component constraints. The usage diagram in Figure 5.25 is needed to define fully this nested constraint and must be considered as part of the definition.

Note how, in Figure 5.25, the parameters of the high-level constraint are attached to the diagram frame with binding connectors used to connect these to the parameters of the internal constraints.

Having defined this high-level 'HoleFillVolume' constraint, Figure 5.21 can now be redrawn to show how it can be used. This is shown in Figure 5.26.

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

It would be expected that, over time, an organization 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 Other ways to use constraints

The power of parametric constraints comes not from the parametric constraint definitions but from the various usages, that is the parametric diagrams, which are created

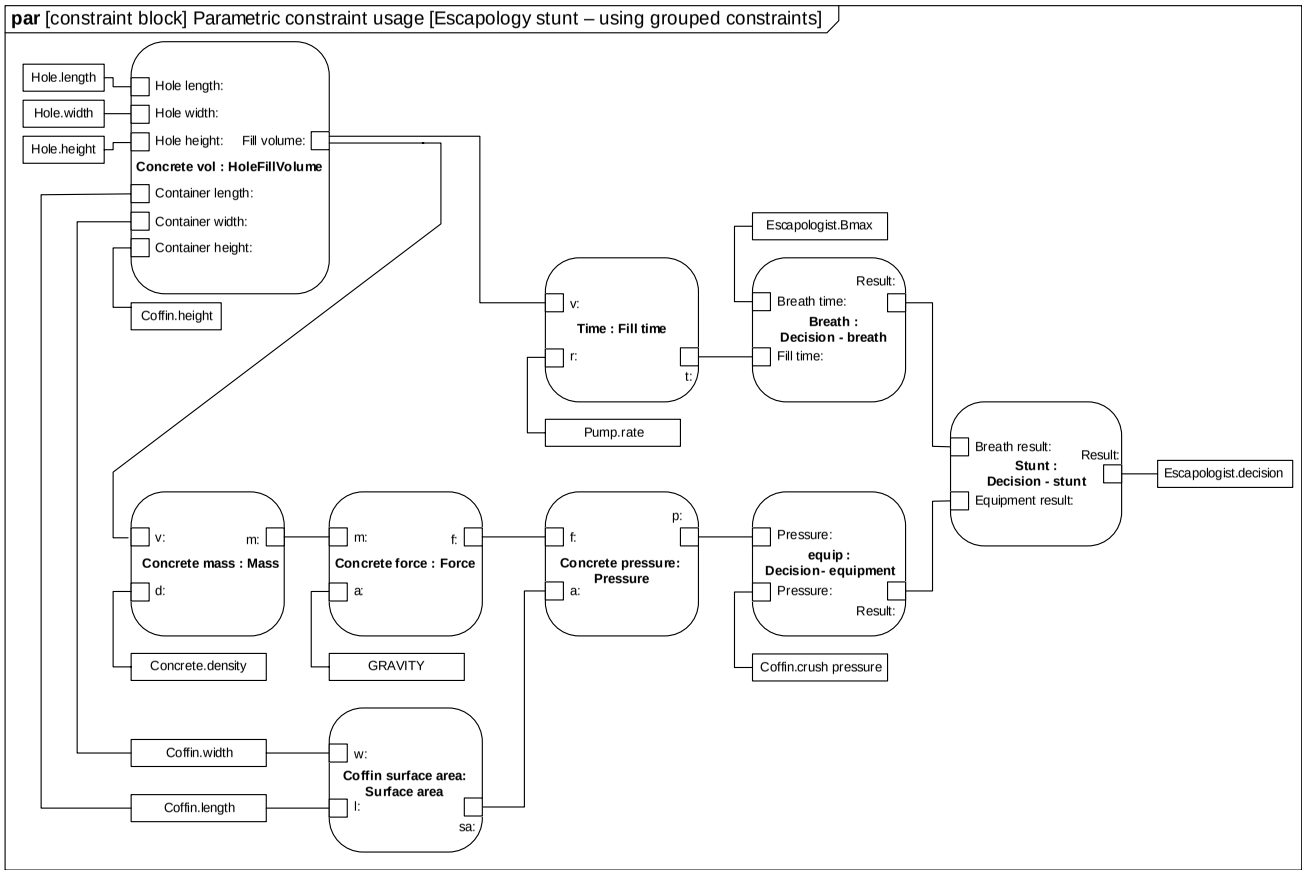

Figure 5.26 The coffin escape – parametric usage showing use of a high-level 'grouped' constraint

using the definitions. As has been shown in 5.5.4.1 and 5.5.4.2 above, these usages allow the constraints on the system to be defined, requirements to be validated for different stakeholders and libraries of constraint definitions and usages to be created.

There are also two other powerful ways of using parametric constraints:

- using parametric constraints as guards;
- constraining system behaviour at different levels of accuracy.

These are discussed below.

Using parametric constraints as guards

As well as using parametric constraints to constrain system behaviour and to help validate requirements, one could also use them in activity and state machine diagrams. These diagrams use *guard conditions* (logical expressions) that determine which branch is taken from a decision node on an activity diagram or that show the conditions that must be satisfied in order to allow the associated transitions to occur on a state machine diagram. Generally, these guards use properties of blocks in their definition.

However, the use of parametric constraints can also be applied to these guards. By defining a parametric diagram that uses a ≪heuristic≫ constraint to constrain the final output (as has been done in Figure 5.21), the usage defined by the parametric diagram can be used as a guard condition. The guard references the *model element name* associated with the parametric diagram (see Section 4.2.1 in Chapter 4), which means that when the guard is evaluated the network of constraints on the parametric diagram is exercised and the result of the final ≪heuristic≫ constraint determined. As this ≪heuristic≫ constraint effectively returns a Boolean value (a 'yes/no', 'true/false' or 'go/no-go' result as discussed in Section 5.4.2), it can be used as the conditional test in the guard. An example is shown in Figure 5.27.

Figure 5.27 shows a state machine diagram for the behaviour of the escapologist. It starts with the escapologist 'waiting to perform'. The escapologist remains in this state

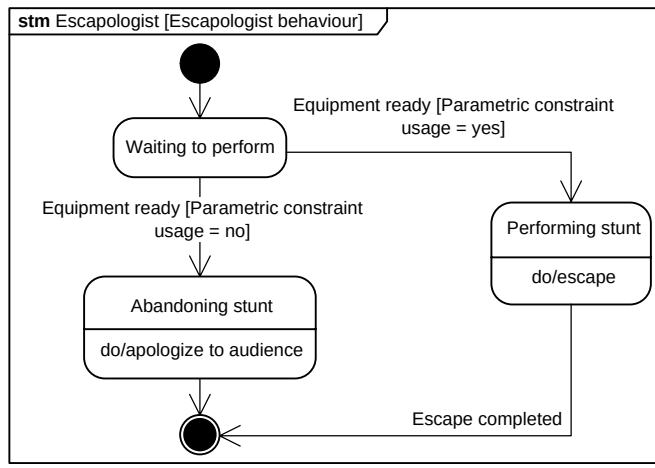

Figure 5.27 State machine diagram for escapologist showing use of parametric constraint

until a signal is received that the equipment is ready. What happens next depends on the result of the parametric constraint usage that determines the go/no-go decision for the stunt. This usage is that shown in Figure 5.21 and the output is used in the guards in the state machine diagram above to determine what the escapologist does next. If the parametric constraint usage returns a 'yes', the stunt goes ahead and the escapologist moves on to 'performing stunt', which terminates when the escape is completed. If the result returned is 'no', then the escapologist moves on to 'abandoning stunt' and then completes his act once he has apologized to the audience.

In this way, complex conditionals can be used when modelling the behaviour of operations or life cycles of blocks with activity and state machine diagrams without the need to use complex expressions in the guards.

Constraining system behaviour at different levels of accuracy

One way of refining system behaviour constrained by parametric usages is to keep the *pattern* of use the same but to change the *definition* of some of the constraint used.

Consider a set of parametric definitions and associated usages that determine how a system responds to its environment, for example whether or not the system has to instigate some collision-avoidance behaviour. In some circumstances it may be necessary to determine the avoidance behaviour – that is the output of a parametric usage – very quickly. In order to do this the parametric diagram might use parametric constraint definitions that are modelled as simple heuristics, for example as lookup tables. As long as no false negative results are returned this is fine. The system can engage its avoidance behaviour quickly without having to resort to complex and lengthy calculations, but with perhaps a number of false positives triggering the avoidance behaviour unnecessarily.

Under other circumstances the system may have more time to determine its response and may be able to do so in a way that still returns no false negatives but minimizes the number of false positives returned. In these circumstances the parametric usage may well have exactly the same *pattern* as before and may use many of the same parametric definitions. However, in this new usage the parametric definitions that were modelled as simple lookup tables are now replaced with alternative definitions that take much longer to evaluate, perhaps being slow but accurate mathematical simulations. Figures 5.28 and 5.29 illustrate this.

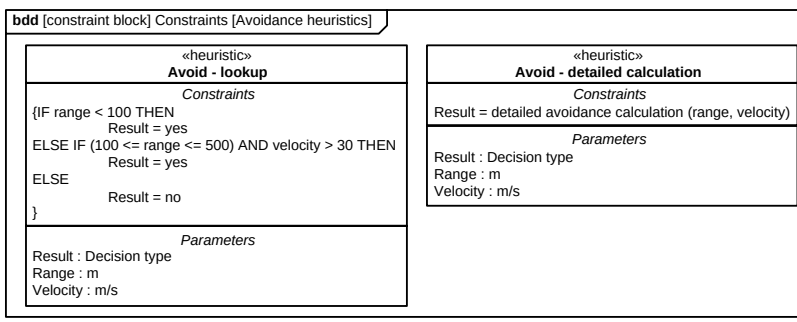

Figure 5.28 Alternative constraint definitions

Figure 5.28 shows two alternative constraint definitions that determine whether or not an avoidance behaviour should be taken. 'Avoid – lookup' is a quickly determined heuristic, whereas 'Avoid – detailed calculation' is a heuristic that invokes a more detailed calculation that takes some time to return a result.

These two constraint definitions can be used with the same *pattern* of constraint usage as shown in Figure 5.29.

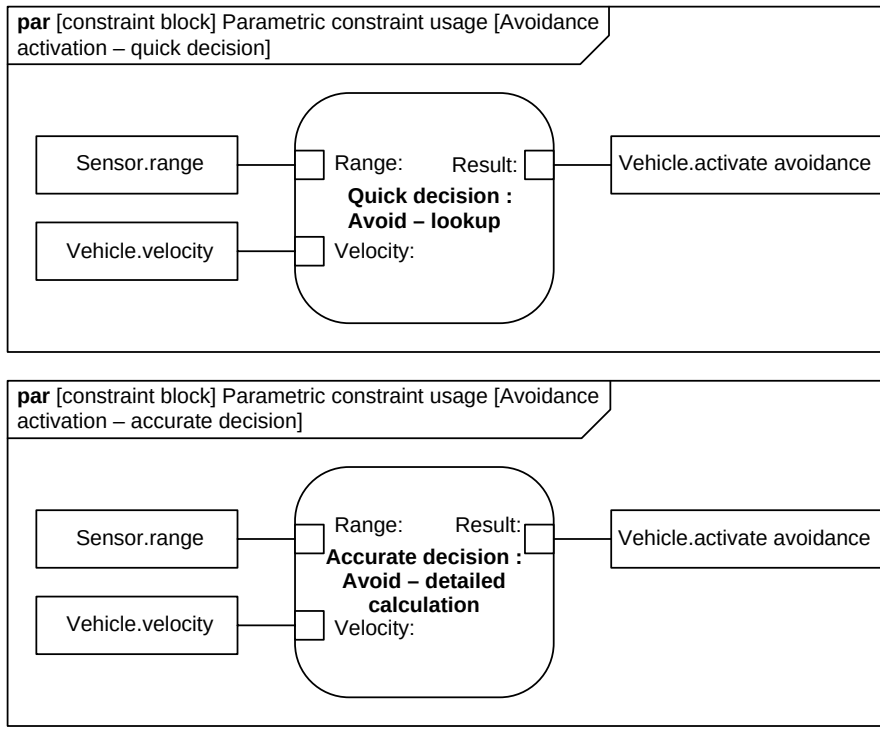

Figure 5.29 Parametric usage – same pattern, different constraints

The top diagram in Figure 5.29 makes use of the 'Avoid – lookup' constraint definition to return a quick result based on distance to obstacle and vehicle velocity. The bottom diagram has the same shape, using the same inputs and generating the same output, but this time making use of the 'Avoid – detailed calculation' constraint to return a more accurate result that takes longer to determine.

In this way what is essentially the same *usage*, albeit with some of the *definitions* changed, can be used to constrain system behaviour at different levels of accuracy. Other parametric usages, in conjunction with the activity and state machine diagrams that define the system's behaviour, might be used to determine which of the parametric diagrams governing avoidance behaviour should be used in a given set of circumstances.

5.6 Conclusions

Interaction between system elements can be modelled in SysML using flow ports and item flows where the interaction involves the transfer of material, data or energy. Where the interaction is service-based, with one block invoking operations that are provided by another block, such interactions can be modelled using standard ports and required and provided interfaces.

In order to show how elements of a system are deployed, or to show how different design elements are allocated to others, SysML provides the concept of the allocation. Such allocations can be represented using stereotypes, dependencies, special compartments on the elements or comments with a defined format. Care must be taken when using the compartment or comment notation to ensure the consistency of the model.

A powerful concept introduced in SysML that is not found in the UML on which it is based is that of the parametric constraint. These parametric constraints allow properties and behaviour of a system to be constrained and may also be used to test whether system requirements have been or indeed *can* be satisfied.

By using stereotypes, different types of constraint can be represented allowing extra information about the constraints to be represented in the model. Three such types are presented; these are not part of the SysML but are types that have been used by the authors in day-to-day practical application of the SysML.

Parametric constraints can be nested, allowing complex, high-level constraints to be developed. It would be expected that, over time, an organization would develop a library of constraint definitions, with lower-level constraints being grouped into higher-level ones for particular application usages.

Parametric constraints can also be used as guard conditions on activity and state machine diagrams. In addition, by using different definitions of a constraint (such as a lookup table and a complex mathematical simulation) but the same usage pattern, system behaviour can be constrained at different levels of accuracy or with constraints that can be evaluated at different speeds.

Chapter 6
Process modelling with SysML

‘Oo-bi-doo, I wanna be like you-oo-oo,
I wanna walk like you, talk like you too’

King Louis, *The Jungle Book*

6.1 Introduction

This chapter looks at using SysML for the application of process modelling. Understanding processes is key to successful systems engineering for a number of reasons.

- A process describes an approach to doing something, in this case systems engineering.
- Systems engineering best practice is usually presented by identifying the key processes that are involved; for example, both *The INCOSE Systems Engineering Handbook* and ISO 15288 are defined in terms of processes.
- When demonstrating quality in a system, it is the process that is assessed or audited.
- Capability is demonstrated through having effective processes defined and having the ability to execute them.

It makes sense, therefore, that our processes be defined using the same notation as we are using for executing our processes.

6.1.1 Modelling processes using SysML

In order to understand the requirements for process modelling, the *process-modelling meta-model* will be introduced, which introduces the concept and necessity of having a process model. This process-modelling meta-model is then expanded to include the definition of a number of views that are essential for process modelling. Each of these views will be realized using SysML. In fact, only a subset of the SysML diagram set is used: block definition diagrams, activity diagrams, use case diagram and sequence diagrams.

6.1.2 The process-modelling meta-model

This section introduces the seven-views process-modelling meta-model, as defined in Reference 1. The process-modelling meta-model comprises two basic views – the *conceptual view* and the *realization view*. The conceptual view defines the basic concepts that must be understood and accepted for process modelling and may be thought of as defining basic requirements for process modelling.

The realization view looks in more detail at each of the seven views that are used in the meta-model and shows which SysML constructs may be used to realize each view visually.

6.1.3 The process meta-model – conceptual view

Before it is possible to start applying modelling to any application, it is first important to look at what is going to be modelled – this can be achieved through modelling! The block definition diagram in Figure 6.1 shows the basic concepts involved with process modelling.

The diagram shows that 'Process knowledge' is made up of one or more 'Process'. This represents any source for process knowledge and may be as varied as being tacit information inside someone's head, or may be more formally realized, such as standards, processes, procedures, etc.

The 'Process model' organizes the 'Process knowledge' and is made up of three main elements, as follows.

- The 'Process description', in whatever format it may take, which describes the process in a common format – in this case using SysML.
- The 'Requirements set', which represents the need for the process in the first place. This may seem like stating the obvious but experience has shown that this view is very often ignored or missed out altogether. The 'Requirement set' is also owned by a 'Stakeholder' or stakeholder set.
- The 'Process validation', which ensures that the process descriptions satisfy the requirements set.

On the right-hand side of the diagram, there is a block named 'Process document', which is formatted according to a 'Document template'. This represents the final output of the process, for example, a standard, a process, a website.

One of the biggest problems that exist in the world of process modelling is that there is a 'short circuit' between the 'Process knowledge' and the 'Process output'. What quite often happens is that some sort of domain expert will be sat down with a document template and asked to pour out their domain knowledge to form the standard. The assumption often made here is that, because the process document follows a template, it must be well written. This could not be further from the truth. In reality, what happens is that a document is produced with no inherent structure – simply headings – and that very few people can actually understand what is written down. Although all the technical content may be there, the domain expert will often make assumptions about what the reader knows or the document will be biased from a particular point of view.

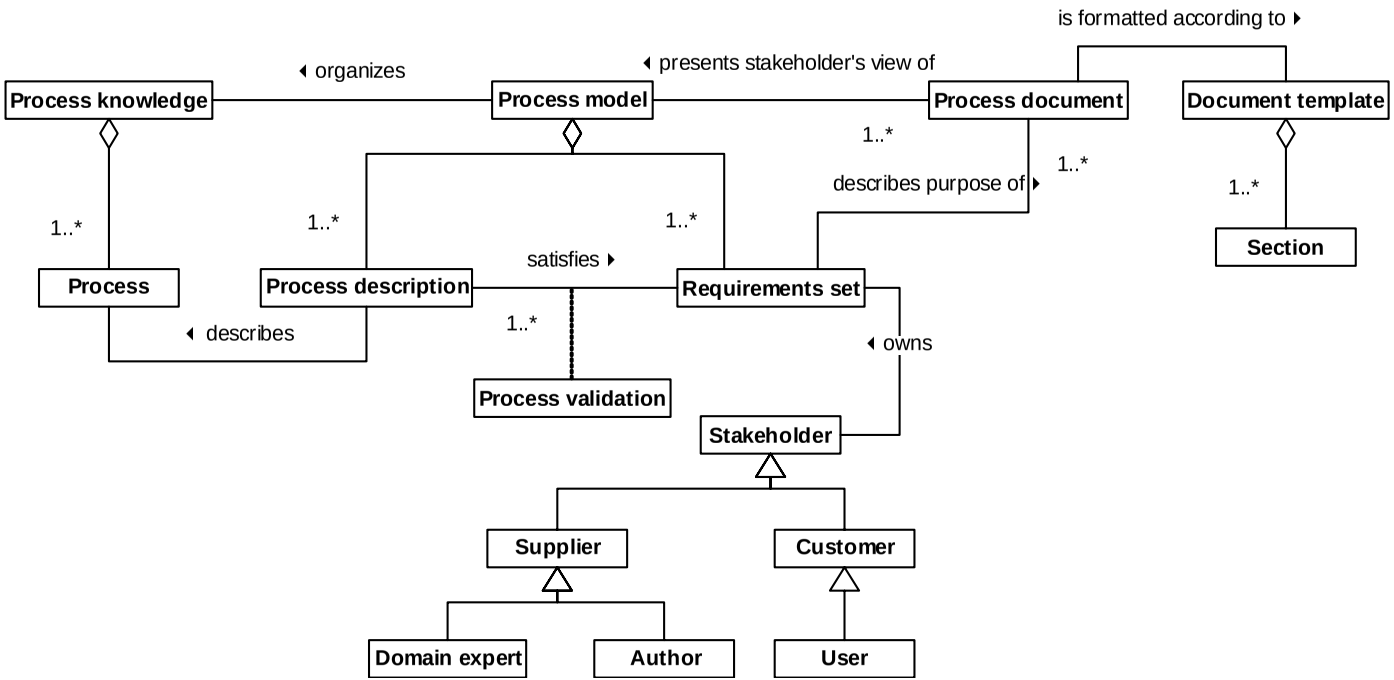

Figure 6.1 Process meta-model – concept view

Sometimes, a technical author will write the document and the opposite happens, where a beautifully written document is created with no real content.

Therefore, the basic theory behind this diagram is that it is essential to have a process model that sits between the process knowledge and the process document. This process model will usually contain more information than will appear in any single process document, as the process document usually represents only a single context that will represent the system from the point of view of a single stakeholder or group of stakeholders.

If the need for a process model can be accepted, the actual process model itself can be broken down into a number of views that make up what is known as the *process meta-model*.

The diagram in Figure 6.2 shows an expansion of the three major elements of the process meta-model – 'Requirements set', 'Process description' and 'Process validation'. Each of these elements is then broken down into a number of views, seven in total, each of which is described below.

- 'Process-structure view'. The process-structure view defines the basic concepts and terminology used on a process model, and is realized in SysML using a block definition diagram. One of the main problems with many processes is a lack of consistency in the language that is used. One of the benefits that a good process-structure view will bring is the definition of a common vocabulary for the process model and, hence, any projects that use it. Each block in this view represents a particular concept or term in the process model's vocabulary, and the associations between these blocks help to give each one meaning with regard to the other terms. This may then be used as a basis for a taxonomy, glossary or statement of definition of terms.
- 'Requirements view'. The requirements view defines what the process is intended to do, and is realized in SysML using a use case diagram. One major problem with many process models is that the actual requirements for the process model are not defined. As in all systems engineering, it is impossible to validate any system if there is not a well-defined set of requirements – and a process model is a type of system. A requirements view will consist of a set of requirements, represented visually by use cases, and a set of stakeholders, represented visually by actors. The requirements are drawn inside a system boundary, whereas the actors exist outside the boundary. Various relationships are drawn between use cases and use cases, and between use cases and actors. The requirements view should be revisited periodically to ensure that it is still an accurate representation of the process needs. The process requirements, in particular the constraints, change over time due to external factors, such as business evolution, changes in legislation and standards.
- 'Process-content view'. The process-content view shows the processes available in a particular process model, and is realized in the SysML using a block definition diagram. A good way to think of the process-content view is to think of it as a process library. Each block in this view represents a single process, with its properties representing artefacts and its operations representing activities

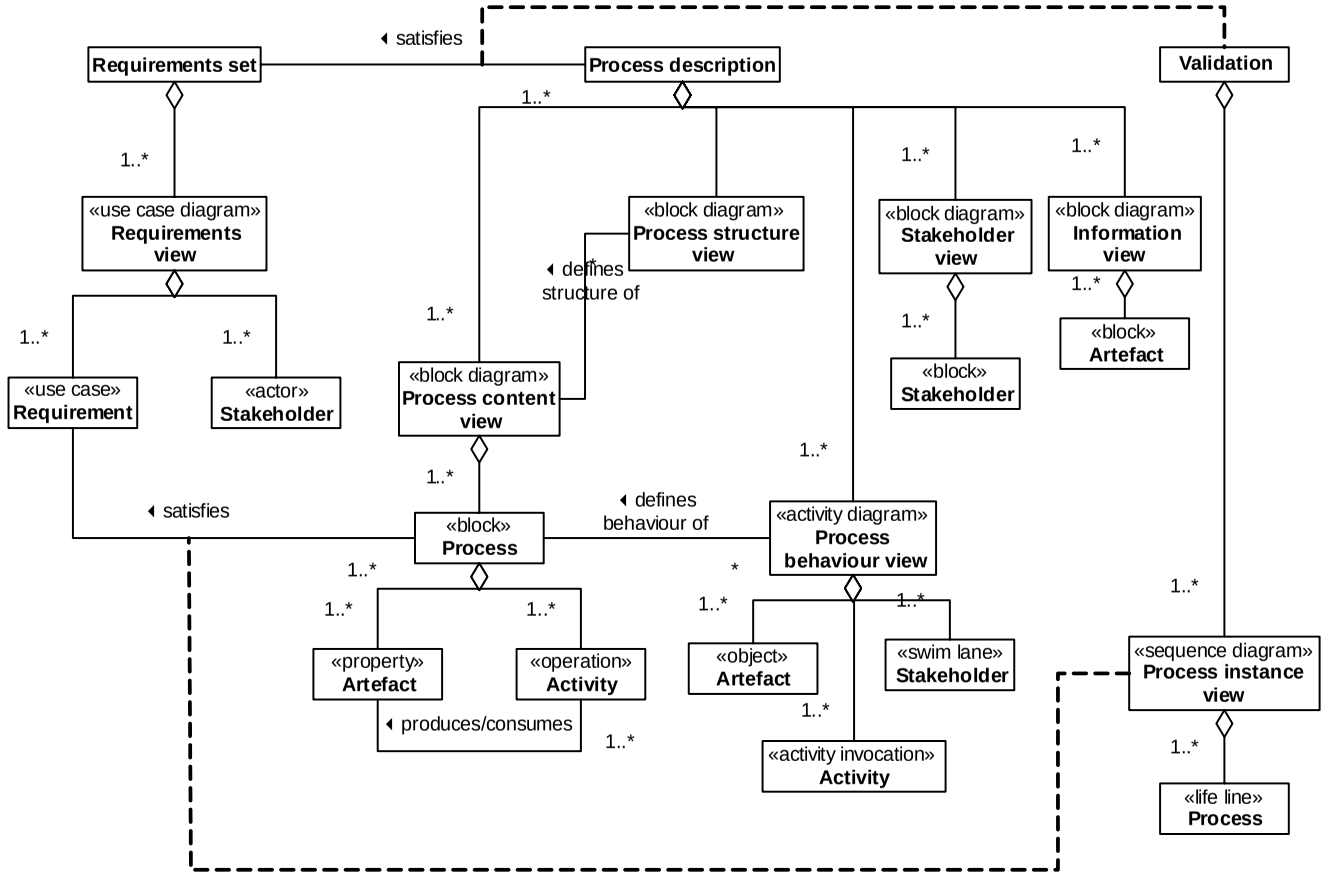

Figure 6.2 Process meta-model – realization view

in the process. Each process is self-contained within a single block, and any inputs or outputs to the process (the artefacts) must interface to the outside world.

- 'Stakeholder view'. The stakeholder view identifies the main stakeholder roles in a process model, and is realized in SysML using a block definition diagram. Each block on the diagram represents a stakeholder role. Stakeholders are shown in a classification hierarchy, where different types of stakeholder are shown using generalizations. It is also possible to use regular associations on the stakeholder view to show how the stakeholder roles relate to one another.
- 'Information view'. The information view identifies the process artefacts, the relationships to other process artefacts and the relationships between a particular process model and any others. The information view is realized in SysML using a block definition diagram. Each block on this view represents an artefact or a part of an artefact. Associations between blocks represent traceability paths between artefacts
- 'Process behaviour view'. The process behaviour view defines the internal behaviour for a particular process, and is realized in the SysML using an activity diagram. This is the view that is often associated with process modelling and, all too often, the only view associated with process modelling. The process behaviour view shows the order of execution of the activities and the information flow between them in terms of the artefacts. Also, this view shows the responsibility for each activity in the process by grouping them into swim lanes. Each swim lane has a stakeholder associated with it that shows responsibility
- 'Process-instance view'. The process-instance view validates the process requirements by showing the execution of a number of processes for a particular requirement. This is realized in the SysML using a sequence diagram. Each life line represents the execution of a process, and the messages between the life lines represent the interactions between the processes.

Not all views will be necessary for every process model – it will depend on the application and why the modelling is being carried out. Indeed, in the three example applications that are presented in this chapter, only a subset of the views is used in two of them.

6.1.4 Consistency between views

As has been stated previously in this book, consistency is essential for any good model – a model without consistency is a set of pictures. As well as the usual SysML consistency checks between diagrams, the process meta-model also identifies a number of other checks There are two types of check: structural and mechanical. Wherever a like term appears on the meta-model, this indicates a consistency check, which will be referred to as a *mechanical check*. For example, a 'Stakeholder' appears on the requirements view, but also appears on the stakeholder view. Therefore,

there must be consistency between the stakeholders in both views. Also, stakeholders appear in the process behaviour view, which provides another consistency check.

The structural checks refer to checks that may be applied based on the structure or pattern of the meta-model, particularly with respect to their relationships. Many of these checks can be identified based on the relationships in the meta-model. Table 6.1 contains a list of structural checks to apply.

Table 6.1 Table showing structural consistency checks

Check description	Meta-model reference
View check. Do all the views exist?	All blocks that describe diagrams, for example: 'Information view' is realized by a ≪block definition diagram≫
Process behaviour check. Does each process in the process-content view have its behaviour defined?	'Process behaviour view' defines behaviour of each 'Process'
Is each requirement validated? Does each requirement have at least one scenario defined to ensure that the requirement is met?	'Process-instance view' validates each 'Requirement'

Table 6.1 shows the specific structural consistency checks that should be applied that are based on the main associations in the process meta-model.

The second type of check is the *mechanical check.* All that is involved with applying the mechanical consistency checks is to select an element from the actual process model, identify its corresponding block on the meta-model, and then look for other occurrences of this block name on the meta-model. For example, consider the case where it is required to apply consistency checks to stakeholders in the stakeholder view. First of all, look to the meta-model and find the block named 'Stakeholder' in the stakeholder view. The diagram indicates that the 'Stakeholder' in the 'Stakeholder view' is realized by a ≪block≫ in SysML. Now, it is simply a matter of looking for other occurrences of stakeholder on the meta-model, which can be seen to be in the 'Process-content view', where a 'Stakeholder' is realized by a ≪Life line≫, and in the 'Requirements view', where a 'Stakeholder' is realized by an ≪Actor≫ in SysML.

This information is captured in Table 6.2.

Table 6.2 shows the specific mechanical checks that should be applied, based on the common elements within the process meta-model.

Table 6.2 Table showing mechanical consistency checks

Concept	View	Realized in SysML by
Stakeholder	Requirements view	≪actor≫
	Process behaviour view	≪swim lane≫
	Stakeholder view	≪block≫
Activity	Process-structure view	≪block≫
	Process-content view	≪operation≫
	Process behavioural view	≪activity invocation≫
Artefact	Process-structure view	≪block≫
	Process behavioural view	≪object≫
	Process-content view	≪property≫
	Information view	≪block≫
Process	Process-structure view	≪block≫
	Process-content view	≪block≫
	Process-instance view	≪life line≫

The seven views have been described, along with consistency checks that can be applied to ensure that the process model is correct. The only thing that remains, therefore, is to discuss how these seven views can be used effectively.

6.1.5 Using the seven-views meta-model

The seven-views process meta-model is realized by a block definition diagram that is itself a structural diagram. Therefore, there is no concept of *order* of generating the views – provided that the full complement of views exists (depending on the application) and they are consistent, then it is unimportant what order they are created in.

There are a number of ways in which the meta-model may be used. For the purposes of this book, only three uses of the meta-model will be considered – life-cycle modelling, modelling standards and defining a new process.

6.2 Modelling life cycles

6.2.1 Introduction

This section looks at one particular application of the seven-views meta-model, that of systems engineering life cycles and life-cycle models. This is one of the areas where the most common conceptual mistakes are made with systems engineering. The idea of the iterative life cycle will be introduced in this section, which represents the current best-practice approach to project life cycles and should be thought of as the latest evolution in the history of life cycles.

Everything has a life cycle, as mentioned in Chapter 1. For the purposes of systems engineering it is usual to look at two types of life cycle: the product life cycle and the project life cycle. In order to illustrate the main concepts of life cycles and life-cycles models, three of the seven views will be used – the process-structure view, the process-instance view and the process-content view.

6.2.2 The life cycle

Regardless of the whether we are considering a project or product, which standard we are following or what application domain we are working in, one common theme emerges when considering life cycles. This is the concept of a stage or phase. The term *stage* will be used in this book, rather than *phase*, simply because that is the term that is used in the main references for this book – ISO 15288 and the INCOSE development manual.

In order to understand the basic structure of the life cycle, a process-structure view will be drawn up that shows the main concepts and terminology to be adopted. A life cycle is realized by a structural diagram – the SysML block definition diagram.

The diagram in Figure 6.3 shows that a 'Life cycle' is made up of one or more 'Stage' and that one or more 'Process' is executed over each stage.

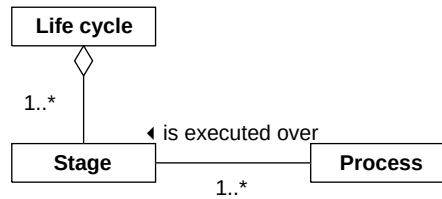

Figure 6.3 Process-structure view for life cycles

At this point, the actual stages that will be defined will depend upon the standard or process model that is being followed. For the sake of the point to be made in the next section, two different approaches for defining the life cycle will be shown in this section. These will then be compared and contrasted when life-cycle models are discussed. The diagram in Figure 6.4 shows two different definitions of life-cycle stages.

The diagram in Figure 6.4 shows two sets of stages that are defined for a life cycle. On the upper part of the diagram (a) there are nine stages that have been defined: 'Requirements', 'Analysis', 'Planning', 'Design', 'Implementation', 'Integration', Transition', 'Operations' and 'Decommission'. On the lower part of the diagram, six stages have been defined: 'Concept', 'Development', 'Production', 'Utilization', 'Support' and 'Retirement'.

These two examples have not been chosen arbitrarily. The example chosen in (a) is typical of the way that stages would be defined in a classic life cycle that will employ a linear life-cycle model, such as a Waterfall approach [2]. This represents

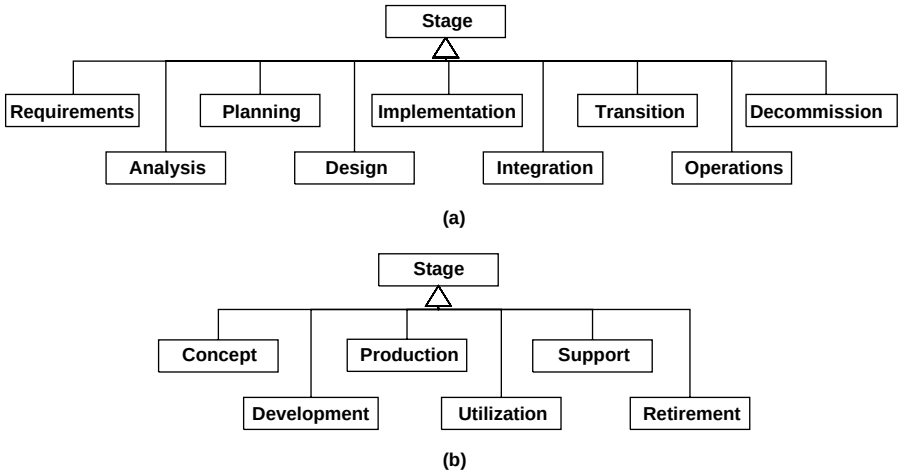

Figure 6.4 Process-structure view – types of stage

an approach that was used heavily in the 1980s and 1990s and was used extensively in the software-engineering community – not without some success. The example chosen in (b) is typical of the way that stages would be defined in a more modern, systems-engineering-style approach that employs an iterative life-cycle model, such as advocated in many systems standards, e.g. ISO 15288, and some more modern software-engineering approaches, e.g. the unified process [3].

At first glance it may seem that the only real differences between the two are that the approach in (a) has more stages than the one in (b). The differences become more apparent, however, when the process-structure views are expanded to include the relationship between stages and processes, as shown in Figure 6.5.

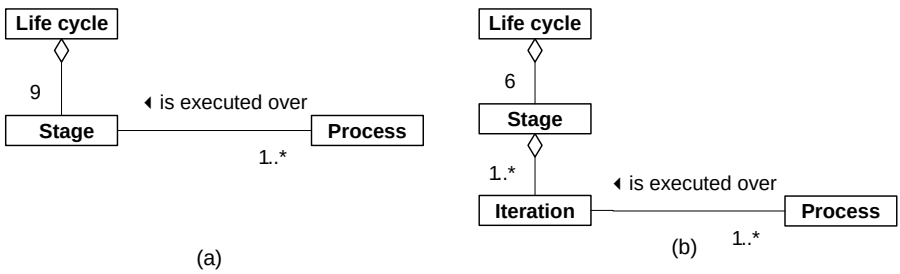

Figure 6.5 Process-structure view – linear versus iterative

Figure 6.5 shows how processes relate to each of the life-cycle structures. In the case of (a), there is a direct relationship between the ‘Process’ and the ‘Stage’. In this case, the relationship is a one-to-many relationship, but, as will be seen later, this is very often not the case, where just a one-to-one approach was taken.

In the case of (b) there is no direct relationship between the stage and the process, as there is an in-between concept of an 'Iteration'. An iteration is simply a collection of processes that are executed for a specific purpose – this will be expanded upon in the next section.

The difference between these two approaches is still not fully apparent until the life-cycle models are looked at when the differences become more immediately apparent.

6.2.3 The process library

The next view that will be modelled will be the process-content view, where a set of processes will be identified that will be used by both of the approaches identified in the diagrams as (a) and (b). The process-content view here has been simplified and shows only the process name, rather than any of the activities or artefacts that one would usually find in this view. Think of the process-content view as the process library that is available for use by both (a) and (b), as shown in the diagram below.

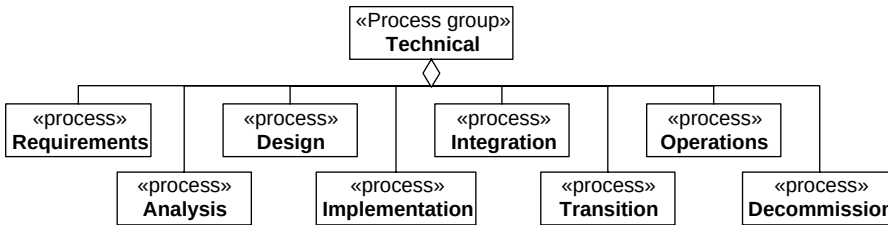

Figure 6.6 Process-content view – process library

Figure 6.6 shows the set of processes that is available in the process library. The fact that the same set of processes is used by both approaches is significant, since it shows that the life cycle and life-cycle model, although strongly related to process, are very separate concepts. Note that this diagram shows only the processes that are defined in the technical process group and not any processes from the other process groups (enterprise, project and agreement – for more information on process groups, see the section concerning ISO 15288 later in this chapter).

6.2.4 Life-cycle models

A project *life cycle* identifies the stages for a project and is realized by a structural view, whereas the *life-cycle model* defines how these stages are executed and is realized by a behavioural view. When visualizing the life cycle, the process-structure view was used (in the form of a block definition diagram), but when visualizing the life-cycle model, the process-instance view (in the form of a sequence diagram) is used.

The life-cycle model is where the true fundamental differences between the approaches shown in (a) and (b) come to the fore.

A life-cycle model describes how the stages in the life cycle are executed, and by this we mean in what order and under what conditions we move between the stages. Life-cycle models have been applied to projects for many years and have undergone quite an evolution since the original linear-type approaches were first seen in the software world in the 1970s. There are many examples of linear life-cycle models but it is not really in the scope of this book to compare and contrast lots of different approaches, so the discussion here will be limited to a single linear approach and the iterative approach.

Perhaps the most widely recognized life-cycle model is known as the 'Waterfall model' [4] and, ironically, it is also the most widely criticized model. The Waterfall model adopted a linear execution of stages, but the real problem with the whole approach is not necessarily this linearity but stems from the fact that a *single* process was executed in each stage and the process name was usually the same as the stage name. This leads to an awful lot of confusion between the terms *stage* and *process* and is still, to this day, one of the most common conceptual errors that people make when it comes to life cycles and life-cycle models. This is illustrated in Figure 6.7.

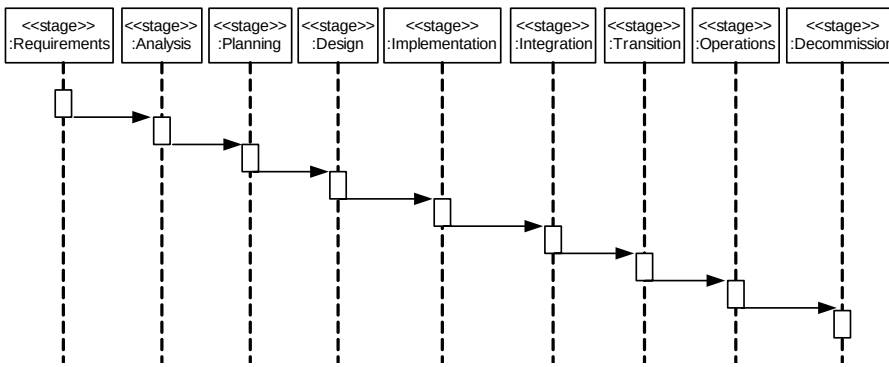

Figure 6.7 Process-instance view – the Waterfall life-cycle model

Figure 6.7 shows how the stages are executed in a linear Waterfall-style approach and Figure 6.8 shows that there is a single process executed in each stage. The process names (taken from the process-content view in Figure 6.6) and the stage names (taken from the process-structure view in Figure 6.4) are identical.

When the terms *stage* and *process* are confused, several undesirable consequences can occur. First, as people apply the same life-cycle model to several projects (which in itself is questionable in the case of the Waterfall), it means that there is always the same set of processes being executed on every project. Secondly, if there is no differentiation between stage and process, there is very little room for flexibility. Even when thinking about requirements, it is almost always the case in systems engineering that some other aspects of the system may need to be looked at that will contribute to the requirements exercise. In reality, any number of processes may need to be executed in order to have a full set of requirements. Also, there is no possibility

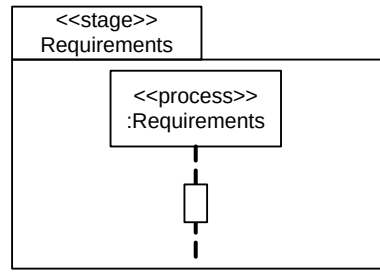

Figure 6.8 Process-instance view – process execution inside a single stage

of revisiting the requirements later in the life cycle, which means that there is no mechanism to allow for changing requirements or requirements creep. This is very unrealistic in all but the shortest of projects and assumes that the requirements can be frozen very early in the life cycle. In fact, the larger the project, the less well suited the Waterfall approach is.

That said, however, the Waterfall approach has been used very successfully on a number of projects over the years, but it does lend itself to particular types of project. In fact, it has been established that the Waterfall approach is indeed effective for short-duration projects, where the requirements are well understood.

This brings us rather neatly round to the second type of life-cycle model – the iterative life-cycle model. The iterative approach differs from the linear approach in that it tends to have fewer stages and each stage has a more generic name than its linear counterpart. Each stage in an iterative life-cycle model does not correspond to a single process, but to one or more groups of process executions, known as *iterations*. The theory is that, if a linear life-cycle model works well for small, well-understood projects, a number of these life-cycle models can be used for a large complex project.

Figure 6.9 shows the execution of the stages in the process-instance view, using a sequence diagram.

At first glance, the diagram in Figure 6.9 does appear to be much different from the one in Figure 6.7 – the only real difference seems to be that the stages are fewer in

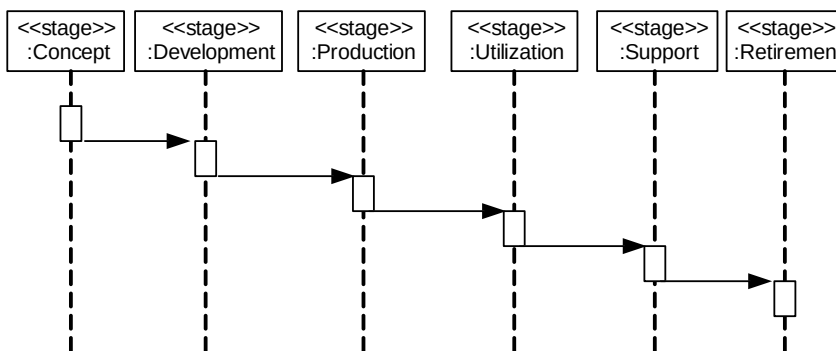

Figure 6.9 Process-instance view – the iterative life-cycle model

number and that the stages have different names. However, it is what is going on *inside* each stage that reveals the true iterative nature of this approach. Inside each stage, rather than just a single process, a number of *iterations* are executed. An iteration is simply a logical grouping of processes that are executed for a particular purpose. For example, there may be several teams of people who are working on various aspects of design during the 'Development' stage of the life cycle. The example in Figure 6.10 shows the case where three different teams are working in parallel before an integration team take over.

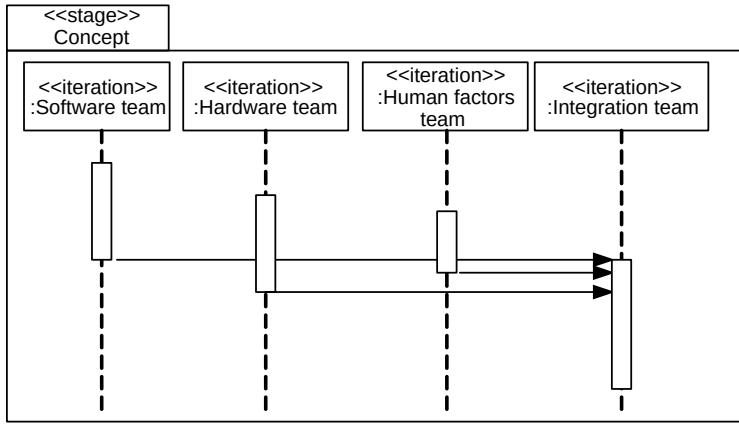

Figure 6.10 Process-instance view – iterations within a stage

In the example shown in Figure 6.10, each iteration has been named according to the team that is carrying out the iteration: 'Software team', 'Hardware team', 'Human factors team' and 'Integration team'. It can be seen here that three iterations are executed in parallel before the integration team takes over. Within each of these iterations there will be a number of processes executed – in this case, the 'Human factors team' iteration is considered (Figure 6.11). The processes in each iteration are taken from the process library defined in the process-content view in Figure 6.6.

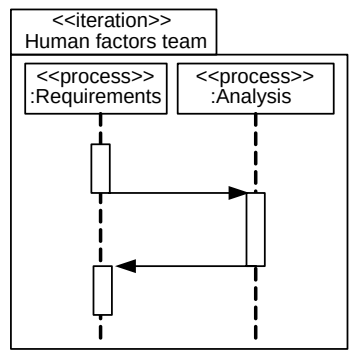

Figure 6.11 Process-instance view – processes executed in an iteration

The iterative life-cycle model allows for a far more flexible working approach that is more realistic than traditional linear life-cycle models. It is possible to execute any process during any stage, which means that, for example, requirements may be revisited at any time during the project. It also means that, for example, some initial design work or analysis may be carried out during the concept stage that may then feed into the requirements process and contribute to the formal requirements of the project.

This iterative approach also lends itself to including processes from different process groups. When using the linear approach, there was an explicit stage named 'Planning', where a planning process would be executed. In reality, however, planning activities will be carried out throughout the project and may be revisited on an iteration and stage basis.

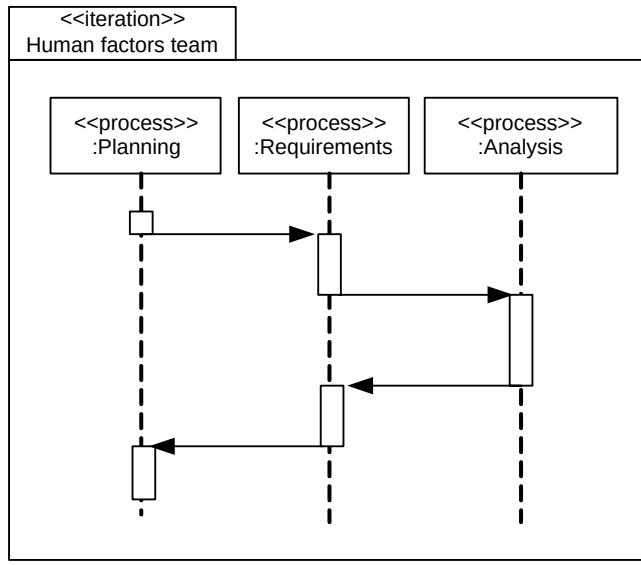

Figure 6.12 Process-instance view – expanded iteration showing a planning process

The diagram in Figure 6.12 shows a different view of the iteration. This time, a 'Planning' process has been added in. Note that this process would be selected from the 'Project' process group, rather than the 'Technical' group, hence it does not appear in Figure 6.6. The iterative approach, therefore, allows a rich mixture of processes from any process group to be included in any iteration, which makes the whole approach far more flexible than the linear approach.

This approach also has the advantage of being more scaleable and can be applied equally well to small projects, medium projects and massive projects in terms of the time, resource and complexity of the project.

The iterative life-cycle model is the approach that has been adopted by both ISO 15288 and the INCOSE development manual.

6.2.5 Using life cycles and life-cycle models

It is essential that a life cycle and life-cycle model be defined for all projects. The iterative approach allows for the life cycle (the actual stages) to be defined at the beginning of the project and then the life-cycle model to be defined in an incremental way as the project progresses.

There is a clear link between project scheduling and life-cycle modelling, and it is essential that, when any project plans are drawn up, such as Gantt charts, they accurately reflect the life-cycle model of the project, including iterations, process execution and, where appropriate, what happens inside each process (see the process definition section for details on modelling individual processes).

6.2.6 Summary

It has been shown that the seven-views approach can be used to illustrate life cycles and life-cycle models. In this application, only three of the seven views were used – the process-structure view, the process-content view and the process-instance view.

The next section looks at applying the same approach to a standard – ISO 15288.

6.3 Applying the seven views to a standard

6.3.1 Introduction

The importance of having a well-defined process was discussed in Chapter 1. However, it is not good enough simply to have a defined process, as it is also necessary to demonstrate compliance with a norm, demonstrate usage of the process on projects and demonstrate that the process is effective. It is possible to use the SysML in order to achieve these aims, and this chapter sets out to do exactly that.

The first of the points raised here, demonstrating that the process is compliant with an existing norm, is crucial to allow the process to be audited or assessed according to such a norm. In real terms, these norms are standards, procedures, best-practice guides, textbooks, etc.

This may seem perfectly reasonable and fit in with our definition of systems engineering being the implementation of common sense, but this is far easier said than done. In order to demonstrate compliance with a standard it is important to have a good understanding of that standard. In addition, it is necessary to be able to map between the relevant part of the chosen standard and the process under assessment.

6.3.2 Introduction to standards

Unfortunately, there are many problems associated with standards that hinder their understanding, or that lead to incorrect implementation of standards, and that were introduced briefly previously in this book.

One problem with standards is that there are so many of them – in fact, in some ways, there are too many standards, which may lead to all sorts of problems.

Here is a list of some of the more popular standards in use today in the world of systems engineering.

- EIA 632 Processes for engineering a system. This is a popular standard that has seen a lot of implementation in the defence and aerospace industries.
- EIA 731 Systems engineering capability model. This is a widely adopted standard that is concerned with capability determination and uses the processes defined in EIA 632 Processes for engineering a system.
- ISO 9001 Model for quality assurance in design, development, production, installation and servicing. This is arguably the most widely used standard in the world as it provides the international benchmark for system quality. This includes all aspects of systems except for software-based systems! This has led to the development of a set of guidelines specifically for software systems, which is described in the next point.
- ISO 9000-3 Guidelines for the application of 9001 to the development, supply and maintenance of software. This is not a standard but a set of guidelines that describe how to apply ISO 9001 to software systems. This has also led to the definition of formal accreditation guidelines under the TickIt scheme.
- IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems. This standard is aimed very much at programmable devices, such as PLCs, and is used heavily in the signalling world.
- ISO 15288 Life cycle management – system life cycle processes. The most recent standard for systems engineering and, arguably, the most widely adopted standard.
- ISO 12207 Software life cycle processes. This standard provides an input to many of the other standards mentioned here and is one of the most widely accepted standards. In fact, most standards will make explicit reference to ISO 12207 and cite it as a basis for life-cycle definition.
- ISO 15504 Software process improvement and capability determination (SPICE). This very long and well-written standard defines a complete set of processes within an organization that can be used as the basis for process improvement and/or capability determination.
- IEEE 1220 Application and management of the systems engineering process. This US standard is a widely used international standard that has a strong track record for systems development.

These standards represent just a small selection that exist in the world of systems and do not include any industry-specific standards. Anybody familiar with standards should recognize at least one of those in the list above. Anybody who has ever tried to read and understand any of these standards (or any standard, for that matter) may have encountered some difficulty in understanding them, which may be for any number of reasons.

- Standards are often very difficult to understand individually and, it may be argued, some are not terribly well written. As will be demonstrated later in this chapter, some standards are inconsistent with themselves and contain many errors and much ambiguity. This does not help when trying to read and understand standards

(a lack of understanding), which puts us on the rocky road to complexity and communications problems. If a standard is not fully understood, then this may impact other processes, such as training and assurance.

- If standards are difficult to understand individually, they are almost impossible to understand when they are read in groups. It is often the case that an organization wants to comply with a generic standard, such as ISO 9001, but also wants to comply with a more industry-specific standard (e.g. EN 50128 Railway applications, software for railway control and protection systems) or even a more field-specific standard (e.g. ISO 15288 Life cycle management – system life cycle processes).
- Understanding the information may also be hindered by the fact that some standards are too long (some in excess of 300 pages), which can seem insurmountable when first encountered. Length is not necessarily an indication of complexity but in many cases will lead to such. Indeed, some of the longer standards are actually less complex than those with a far lower page count. For example, ISO 15504 is far less open to ambiguity than ISO 9001, despite having a page count that is over ten times higher.
- Some standards are too short and written at such a high level that they bear little or no resemblance to what actually occurs in real life. Take, for example, ISO 9001, which, it can be argued, is the most widely used standard in the world today. It is perhaps because it is applicable to so many different areas that the standard is so generic. However, there is an inherent danger in this, as the more generic the standard, the more open to ambiguity it becomes. These ambiguities occur here, as two different people reading the same paragraph of the standard may come up with two completely different interpretations.
- Standards are often written by committees and thus tend to end up somewhat less than comprehensible. Standards often take years to be prepared, and trying to keep all of the people happy all of the time can often lead to the production of a camel rather than a horse!

Assuming, then, for one moment, that the standards can be understood, the next issue is that of demonstrating compliance. Compliance is demonstrated through audits or assessments. Audits are performed by an external, registered organization and yield a straight pass-or-fail result. Assessments, on the other hand, may be carried out internally or externally, but usually yield more useful results, such as a capability profile. As a simple example of this, ISO 9001 requires a formal audit, which yields a yes-or-no outcome. Assessment-based standards, such as ISO 15504 and CMMI, on the other hand, require an assessment (as opposed to an audit) that yields a potentially more useful capability profile that can be used to improve existing processes.

Compliance is actually achieved through having a defined process that is then the subject of an audit or assessment. In almost all cases, standards explicitly state that a defined process is essential to demonstrate compliance.

Most audits and assessments will call for some sort of mapping to be established between the process and the standard that is being used as a basis for the assessment or audits. In order to map between two of anything, it is important that they be both represented in a language or format that may be directly compared – otherwise mapping is impossible.

This then means that, if any of these standards are to be met, they must be able to be related to a defined process. In addition, if more than a single standard is read, they must be able to be compared and contrasted so that a common understanding may be achieved.

In summary, therefore, it is possible to draw up a set of requirements that are needed in order to use standards in a practical and effective manner.

- There is a need to be able to *understand standards*. This means that any problem areas such as ambiguities or inconsistencies can be sorted out and that everyone who reads the standard has a common understanding.
- It is vital to be able to *compare standards with a defined process*. It is no use at all if both the standard and the process can be understood individually if they cannot be compared in an effective manner.
- It is also necessary to be able to *compare different standards with one another*. Again, standards often refer to other standards, or are required to be met in conjunction with some other standard.

It is essential, therefore, that all standards and processes can be represented in a common format or language. The SysML represents such a common language and, as will be demonstrated, is most effective at modelling processes and standards. This all boils down to being able to effectively analyse any standard or process by modelling them using the SysML.

This section applies the seven-views approach to a specific standard, in this case ISO 15288.

6.3.3 Process-structure view

Figure 6.13 shows the process-structure view for ISO 15288 using a block definition diagram. The basic terminology of the standard is identified at this point. This

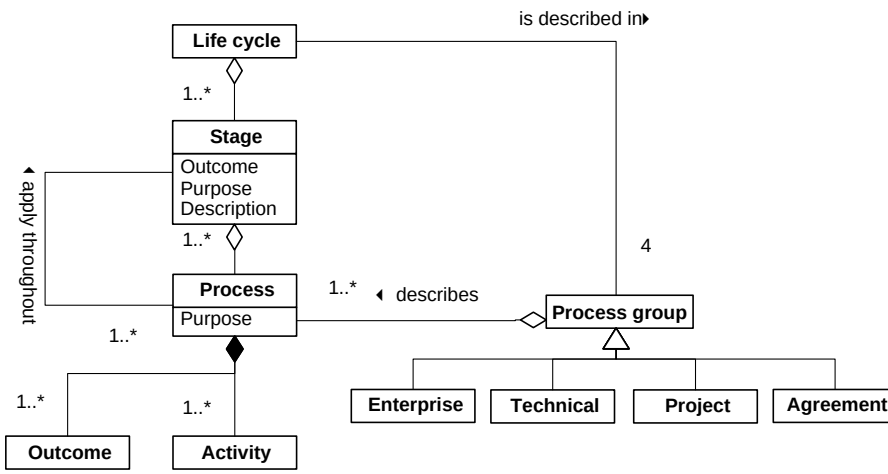

Figure 6.13 Process-structure view for ISO 15288

diagram should be familiar by now, as it has been used as one of the main inputs to the systems engineering conceptual diagram that was introduced in Chapter 1. As will be seen, however, there is some more specific terminology and concept definition used in this standard compared with the generic conceptual view shown previously.

The diagram in Figure 6.13 shows that a 'Life cycle' is made up of one or more 'Stage' and that each 'Stage' can be defined by its: 'Outcome', 'Purpose' and 'Description'. Each 'Stage' is made up of one or more 'Process', each of which has a 'Purpose' and is composed of one or more 'Outcome' and one or more 'Activity'.

The life cycle is described in four 'Process group', each of which describes one or more 'Process'.

- 'Enterprise', which collects together all processes that apply across the whole organization, all staff and all projects.
- 'Technical', which collects together the processes that most people will associate with systems engineering and that cover areas such as requirements and design.
- 'Project', which collects together processes that are applied on a project-by-project basis, such as project planning, risk management, etc.
- 'Agreement', which collects together processes that describe the customer-and-supplier relationship in the project.

The same diagram can now be further populated with other standard-specific terms but with a focus on the types of life-cycle stage. For the sake of clarity, not all terms will be repeated in Figure 6.14.

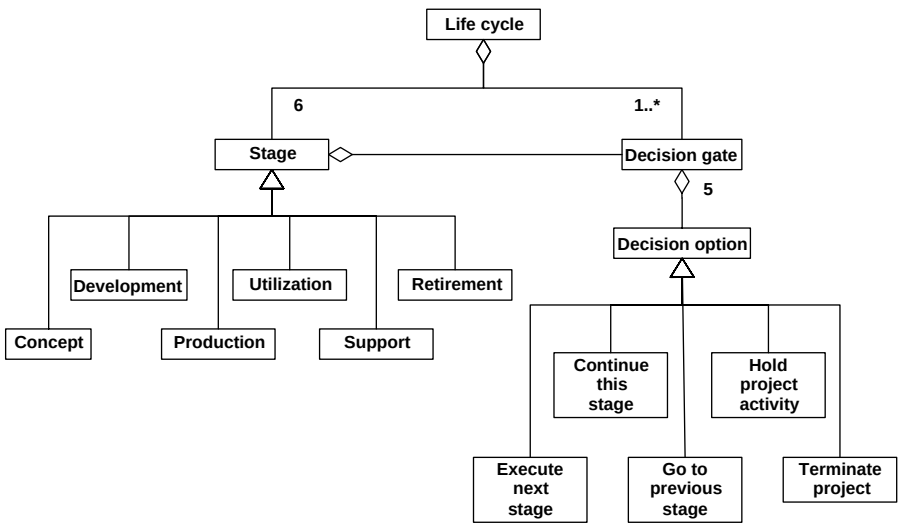

Figure 6.14 Process-structure view with an emphasis on stages

The diagram in Figure 6.14 shows the process-structure view but this time with the emphasis on the types of stage. It can be seen that there are six stages defined here.

- The 'Concept' stage, which covers the identification, analysis and specification of all the requirements for the project. This may also include prototyping, trade studies, research or whatever is necessary to formalize the requirements set.
- The 'Development' stage, which covers all the analysis and design for the project and results in the complete system solution.
- The 'Production' stage, which covers the production of the system of interest, either as a one-off in the case of bespoke systems, or full manufacture, in the case of volume systems. This stage also includes the transition to operation.
- The 'Utilization' stage, which covers the operation of the system at the target environment.
- The 'Support' stage, which provides the maintenance, logistics and support for the system of operation while it is being operated.
- The 'Retirement' stage, which covers the removal of the system of interest and all related processes or support mechanisms from service.

Also shown here is the introduction of a stage, 'Decision gate'. A decision gate is executed before moving on to the next stage of the life cycle. This may take the form of a review or some sort of assessment. There are five basic 'Decision option' that can be made.

- 'Execute next stage', which would reflect a project that is progressing well with few or no problems.
- 'Continue this stage', which would reflect a project that has perhaps been delayed or has not fully completed the current stage. In the event of this decision being made, then another gate review would have to be held before progressing onto the next stage.
- 'Go to previous stage', which would reflect some significant issues with the project. Perhaps the requirements have changed or the design solution that has been chosen is not acceptable.
- 'Hold project activity', which would reflect serious issues with the project, such as significant requirements change, problems with the customer, problems within the organization and legislation change.
- 'Terminate project', which would reflect the worst-case scenario where the project is stopped altogether.

Like all aspects of standards, these will need to be tailored for a specific organization, but this set of decision options covers most scenarios in a project.

6.3.4 Requirements view

The requirements view tells us what the standard is all about – what is it setting out to do, who are the major stakeholders, and so on.

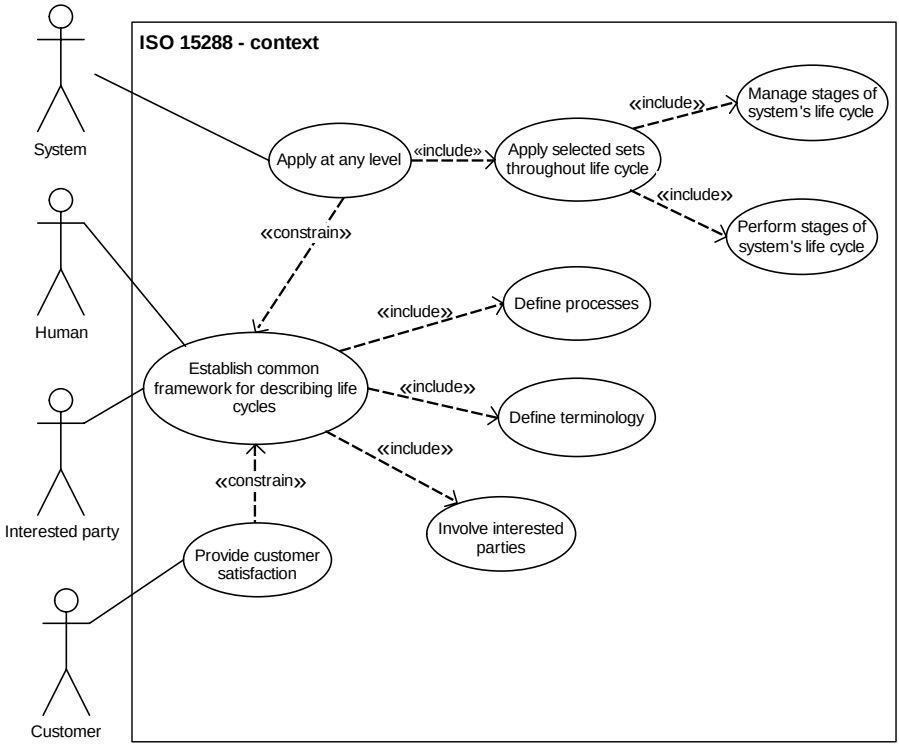

Figure 6.15 Requirements view for ISO 15288

The diagram in Figure 6.15 shows the requirements view for ISO 15288. It can be seen that the main requirement is to 'Establish a common framework for describing life cycles', which includes the following.

- 'Define processes', which forms the core of the standard itself and will form the basis for the process-content view.
- 'Define terminology', which defines all concepts and terms in the standard and forms the basis of the process-structure view.
- 'Involve interested parties', which includes all stakeholders who are involved with systems engineering, and will form the basis for the stakeholder view.

This main requirement is also constrained by the following two requirements.

- 'Apply at any level', which relates to how this standard may be applied to any level of system or system of system, as defined in Chapter 1 of this book. This includes 'apply selected sets throughout life cycle', which refers to sets of processes, in terms of management and actually performing the processes.
- 'Provide customer satisfaction', which relates to the generic requirement of the standard to keep all the stakeholders happy.

The processes that are defined in the standard must satisfy these requirements, as shown in the process meta-model in Figure 6.2.

6.3.5 Process-content view

The process-content view shows the processes that are available for use on projects and may be thought of as the *process library*. It has already been stated in the process-content view that there are four types of process group, so a process-content view may be generated for each group. For the sake of brevity in this book, only a single process group will be shown, the project group.

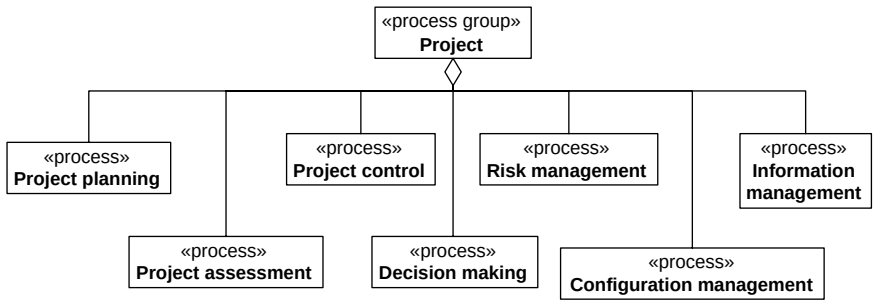

Figure 6.16 High-level process-content view for the 'Project' group

The diagram in Figure 6.16 shows a high-level process-content view for the 'Project' process group. In this view, only the processes have been identified and there is no more detail shown. Figure 6.17, on the other hand, shows the full set of activities (represented as SysML operations) and outcomes (represented as SysML properties).

The diagram in Figure 6.17 gives an idea of the sheer volume of information that is contained within the standard. This also provides an insight into the complexity of the main processes in this group. This is the view where mapping will be performed between this standard and any other standard or process model.

6.3.6 Information view

The diagram in Figure 6.18 shows an information view for the standard. In this case, the information view is being used to show the relationships between ISO 15288 and a number of other standards. The term used to represent this type of information view is a 'standards quagmire' – a term that was coined by the hugely talented Sarah Sheard in her often-referenced paper on technical standards [5].

Each block on this view represents a particular standard, and then a set of associations is drawn between them. In this case, it is possible to see how the standards have evolved from the original military standards to the current ISO 15288.

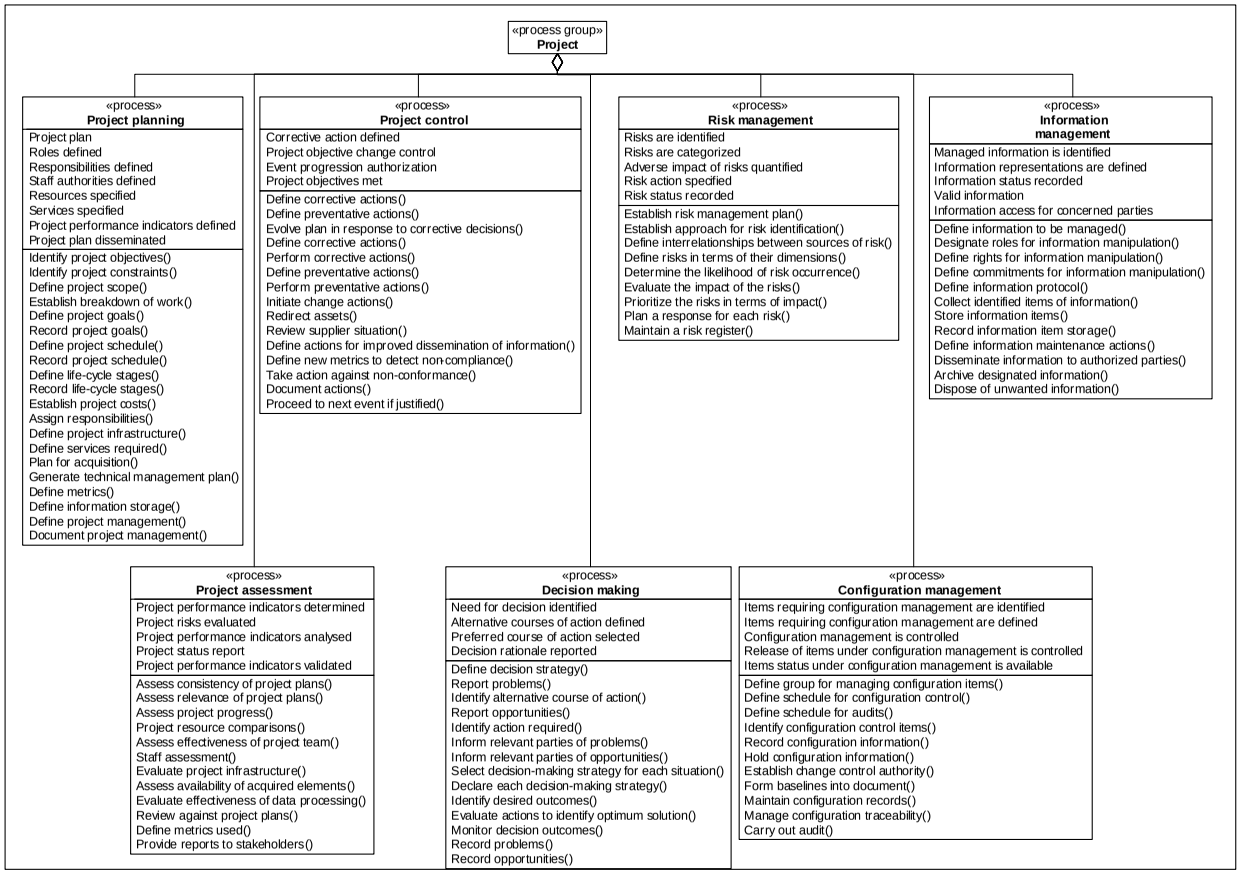

Figure 6.17 Low-level process-content view for the 'Project group'

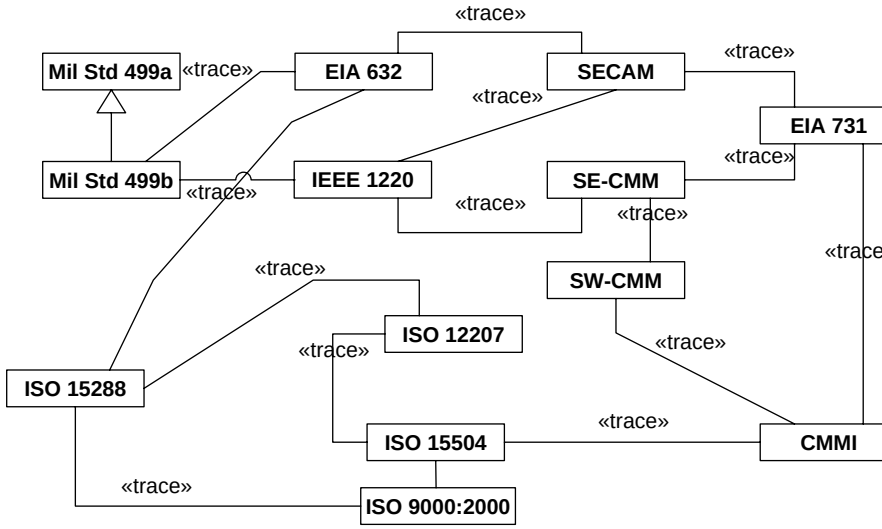

Figure 6.18 Information view – a partial standards quagmire

6.3.7 Using the views

Once these views have been generated, it is then possible to use them as a basis for compliance mapping for other processes and standards. The two main views that are used for mapping are the process-structure view, where the terminology is mapped, and the process-content view, where the actual processes are mapped.

This mapping can be carried out between any number of standards; for example, the standards that were identified in the information view in Figure 6.18 could be used.

These views are also very useful from a training point of view and form an ideal way of presenting and communicating the high-level concepts and content of the standard to project members.

6.3.8 Summary

The seven-views approach was applied to an existing standard, in this case ISO 15288. The views that were generated were the requirements view, process-structure view, process-content view and the information view.

The model generated can be used as a basis for compliance mapping and makes an excellent training tool.

6.4 Apply the seven views to a process

6.4.1 Introduction

All seven views should be defined, ultimately, to make a process useful. In the following example, a process model will be considered. The process model is known

as *STUMPI* (STUdents Managing Projects Intelligently – apologies for the partial acronym) which is an ISO-15288-compliant process model that was developed as an educational tool for training and teaching. The main idea behind STUMPI was to develop a very short process model that was able to be used in very short projects but that could be used as a tool for demonstrating the main concepts of process, life cycles and standards.

An example of each of the seven views for the STUMPI process model will be shown.

6.4.2 The STUMPI life cycle

The STUMPI model defines a life cycle that can be used as a basis for projects.

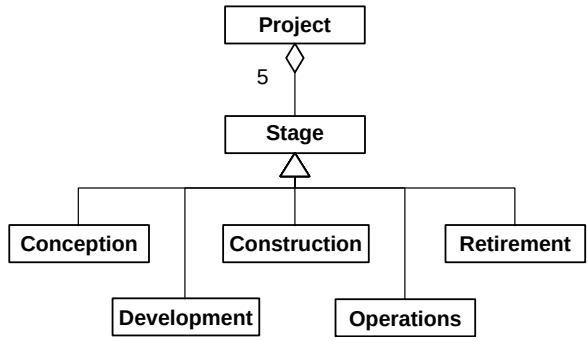

Figure 6.19 Process-structure view – the STUMPI life cycle

Figure 6.19 shows the life cycle defined as part of the STUMPI model. It can be seen that each 'Project' is made up of five 'Stage', which are: 'Conception', 'Development', 'Construction', 'Operations' and 'Retirement'. It is interesting to note that many student projects will never actually implement all of these stages. For example, many projects will be mainly concerned with the conception and development stages, where the end result of the project is a deliverable. It may also be that a project is a feasibility study, in which case the life cycle may have only a single conception stage. Of course, some projects will have a complete life cycle – it depends on the nature of the project, and this forces people to think about the nature of the project at the outset and ask a few important questions, such as, 'How far through the life cycle are we going with this project?' and 'Will we be doing all the work?'

The life cycle shows the 'what' of the project, but it is also vital to show the 'how' of the project using a behavioural view. This behavioural view is known as the *life-cycle model* and will be discussed in the next section.

6.4.3 The STUMPI life-cycle model

The life-cycle model shows how the life cycle is executed over the course of a project. It shows which stages are executed and in what order. The actual order of the execution

of these stages will vary depending on the type of project being run, which goes to show that it is possible to have many different life-cycle models that relate to a single life cycle.

The diagram in Figure 6.20 shows an example of a STUMPI life-cycle model in the form of a sequence diagram.

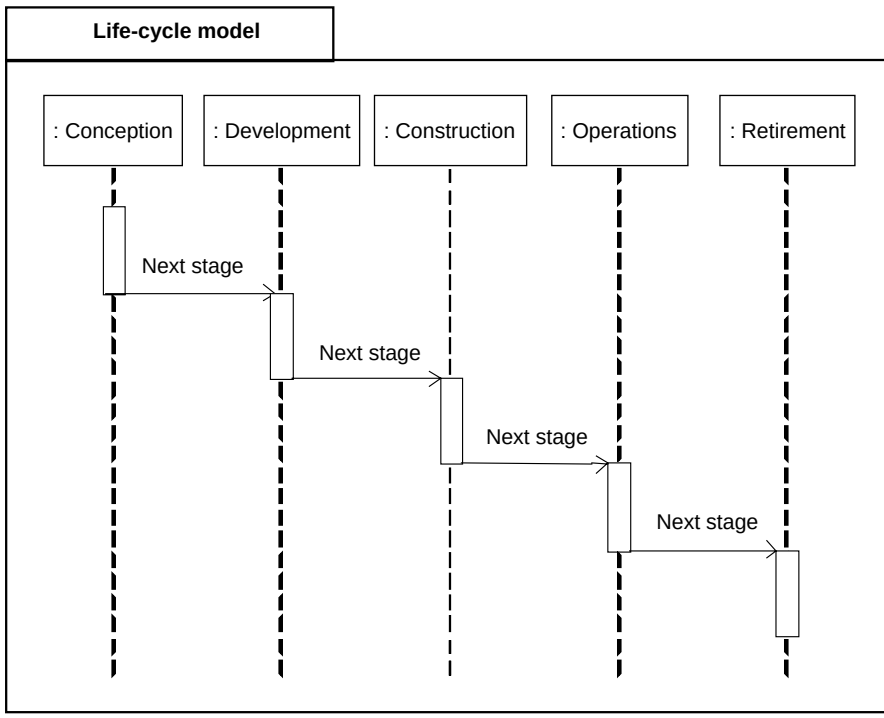

Figure 6.20 Process-instance view – an example STUMPI life-cycle model

It can be seen that, in this example, the life-cycle model is following a simple linear execution of stages with no examples of iteration between stages. Iteration can be shown easily on such a diagram by adding in more messages between stage instances.

The diagram in Figure 6.21 shows an example of a different life-cycle model (behavioural – the sequence diagram) for the same life cycle (structural – block diagram). In this life-cycle model, the system is developed in two increments rather than one. For more information on the different types of life-cycle model, see Reference 2.

It is important here not to confuse this sort of life-cycle model with a more traditional linear life-cycle model, as each stage here will be made up of a number of iterations, which is not necessarily true for linear models. This can be seen more clearly in Figure 6.22.

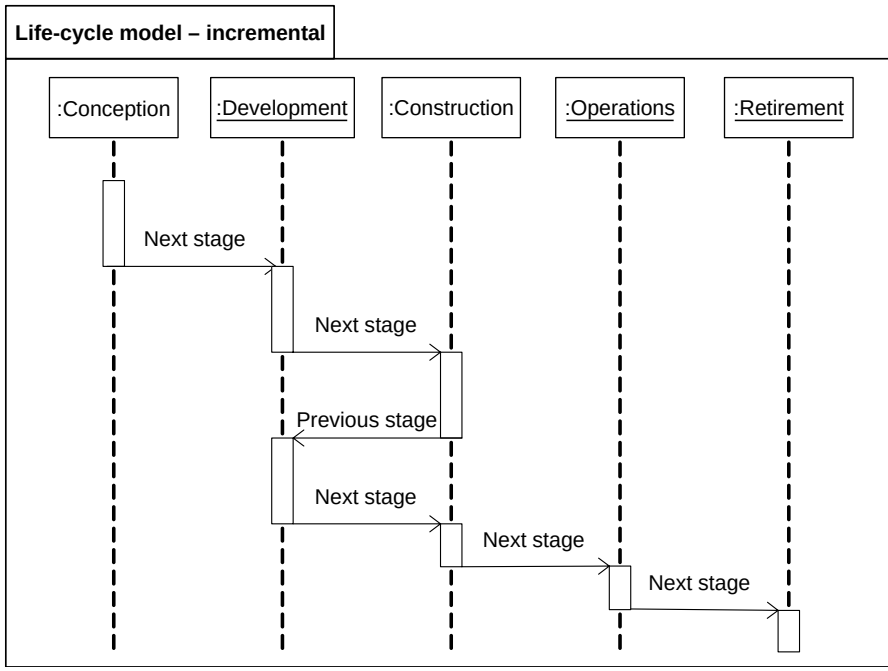

Figure 6.21 Another life-cycle model – incremental

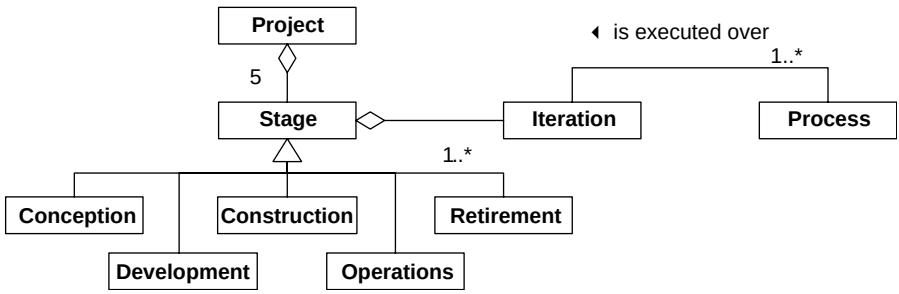

Figure 6.22 Process-structure view – relating stages to iterations and processes

Figure 6.22 shows how stages are related to other STUMPI concepts – the iteration and the process. This diagram is an extended version of the diagram shown in Figure 6.19, but this time it can be seen that each ‘Stage’ is made up of one or more ‘Iteration’, each of which has of a number of ‘Process’ executed over it. This is a crucial relationship to identify, as it defines the conceptual link between the stages and the processes that will be executed on a project – something that is very often confused.

These processes will now be defined in the next section.

6.4.4 The STUMPI process model

The structure of the processes is key to understanding the nature of life-cycle management, so it is important that this structure be defined and well understood by all people involved in the project. The process structure for STUMPI is based on that for ISO 15288 but has been quite severely cut down to enable its use for student projects.

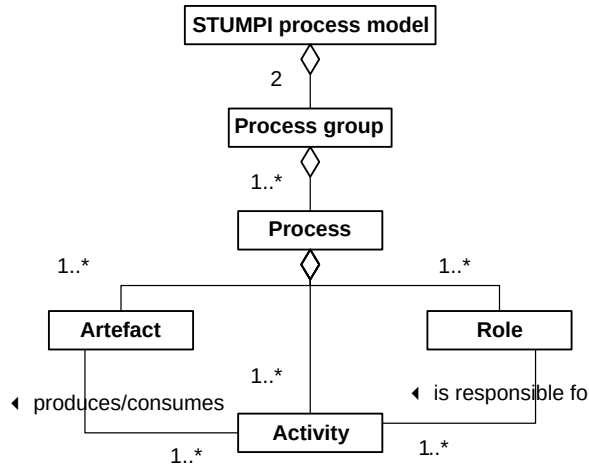

Figure 6.23 Process-structure view – the STUMPI process model

The diagram in Figure 6.23 shows the process structure for the STUMPI model. This is based directly on ISO 15288 and it can be seen that the 'STUMPI process model' is made up of two 'Process group', each of which is made up of a number of 'Process'. Each 'Process' is made up of a number of 'Artefact', a number of 'Activity' and a number of 'Role'.

The source standard, ISO 15288, identifies four different process groups, which are: 'Technical', 'Enterprise', 'Project' and 'Agreement'. The STUMPI model, on the other hand, has only two process groups identified, which can be seen in Figure 6.24.

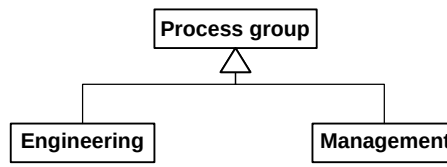

Figure 6.24 Process-structure view – types of process group

The diagram in Figure 6.24 shows that there are two types of 'Process group' – 'Engineering' and 'Management'. The STUMPI 'Engineering' process group maps

directly onto the 'Technical' process group from ISO 15288, whereas the STUMPI 'Management' process group maps to the 'Project' process group in ISO 15288.

The processes for each of these process groups can be seen in Figures 6.25 and 6.26.

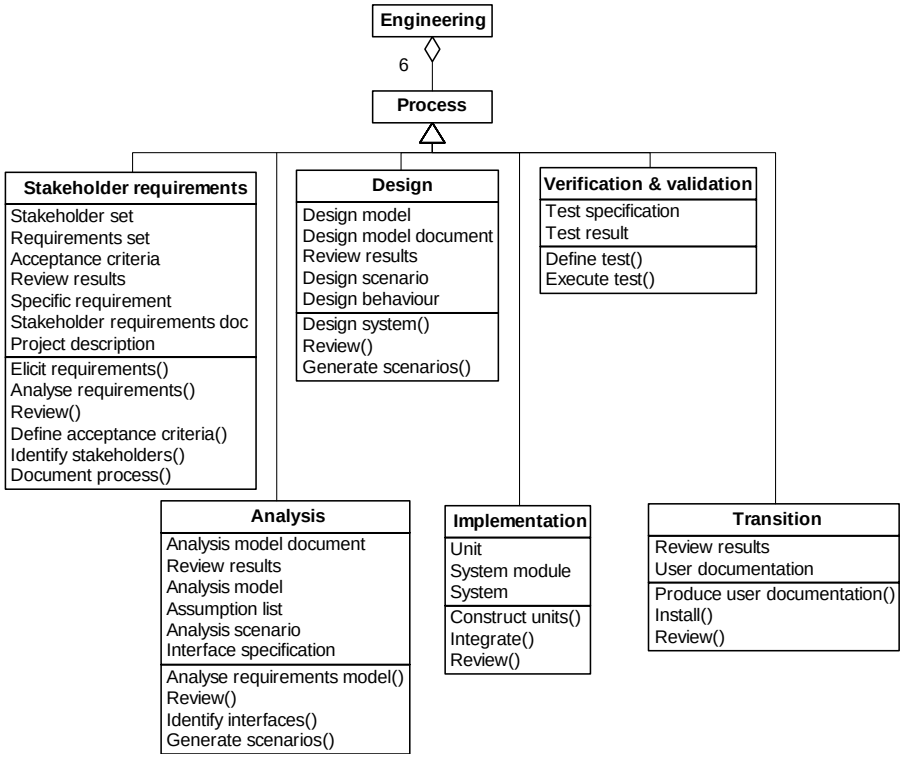

Figure 6.25 Process-content view – the 'Engineering' processes

The diagram in Figure 6.25 shows the processes that have been identified for the 'Engineering' process group. Each block that represents a process has the following three things defined.

- The process *name*, which is shown by the block name.
- The process *artefacts*, which are shown as properties on the process block. These represent the sort of information that must be recorded in a log book to demonstrate that the activities associated with this process have been executed.
- The process *activities*, which are shown as operations on the process block. These represent the things that must be done in order to execute this process successfully.

It can be seen that six processes have been identified here: 'Stakeholder requirements', 'Analysis', 'Design', 'Implementation', Verification and validation' and 'Transition'.

These processes are based directly on those in ISO 15288, but each one has been simplified in order to make it more usable on small-scale projects. Some of the names have been changed to maintain consistency with concepts that are covered within other subject areas, such as 'Analysis' and 'Design'.

The processes associated with the 'Management' process group are shown in Figure 6.26.

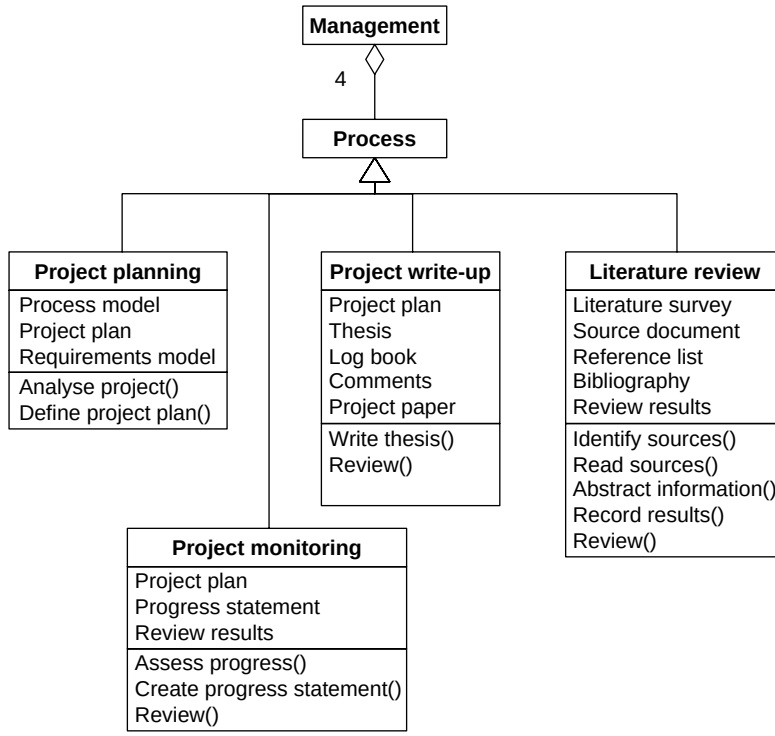

Figure 6.26 Process-content view – the 'Management' processes

Figure 6.26 shows the processes associated with the 'Management' process group, which are: 'Project planning', 'Project monitoring', 'Project write-up' and 'Literature review'. These processes are also based on the source standard but, in this case, two processes have been explicitly introduced, as they relate specifically to students. These processes are: 'Project write-up', where a thesis or dissertation is produced at the end of a project; and 'Literature review', where all current literature in the project application domain is reviewed to demonstrate expert knowledge. Although these two processes do not explicitly exist within ISO 15288, it is still possible to trace individual activities back to activities from various processes from within ISO 15288.

It should be stressed that the processes identified here are intended to be used as a starting point for identifying processes for a particular project. Clearly, in the event that a particular project requires a specific process, this process must be defined in the same way as those shown here.

The only area that has not yet been covered by the process model is that of the stakeholder diagram, which can be seen in Figure 6.27.

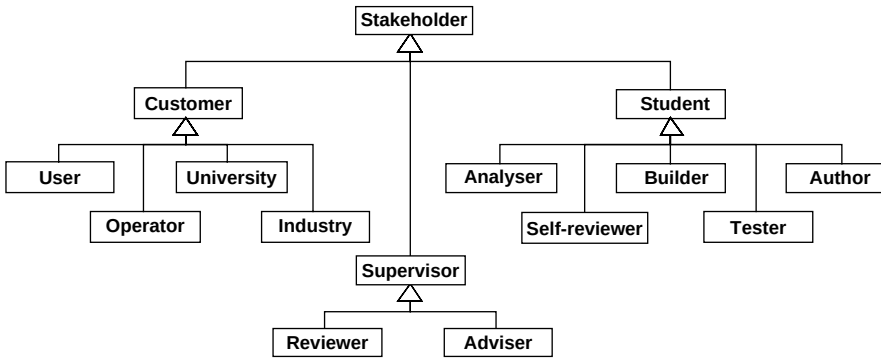

Figure 6.27 Stakeholder view – key stakeholders

The diagram in Figure 6.27 shows the stakeholders that have been identified within the STUMPI model. There are three main types of 'Stakeholder'.

- The 'Customer' stakeholder, which represents the roles that want something from the project and which includes: 'User', 'Operator', 'University' and 'Industry'. Again, it may be the case that there is no industrial involvement, in which case there will be one fewer stakeholder shown here. Conversely, it may be the case that several more stakeholders are identified here.
- The 'Supervisor' role, which represents the roles taken on by a supervisor of the project. In this case, there are two roles identified: 'Reviewer' and 'Adviser'. It may be the case that both of these roles are taken by a single individual or, in another case, it may be that there are several supervisors (maybe both academic and industrial) that are involved in the project.
- The 'Student' role, which represents the various roles that the student, or group of students, will take on the project. Again, consider a project with a single student, in which case all roles will have one name associated with them; or, on the other hand, it could be a group project, where the roles can be 'shared out' among the project team.

This stakeholder diagram should be tailored for a particular project and then the responsibility for each role considered. It is known from the process structure in Figure 6.23 that each 'Role' is responsible for a number of 'Activity', and this is shown using an activity diagram to describe the behaviour of each process, as shown in Figure 6.28.

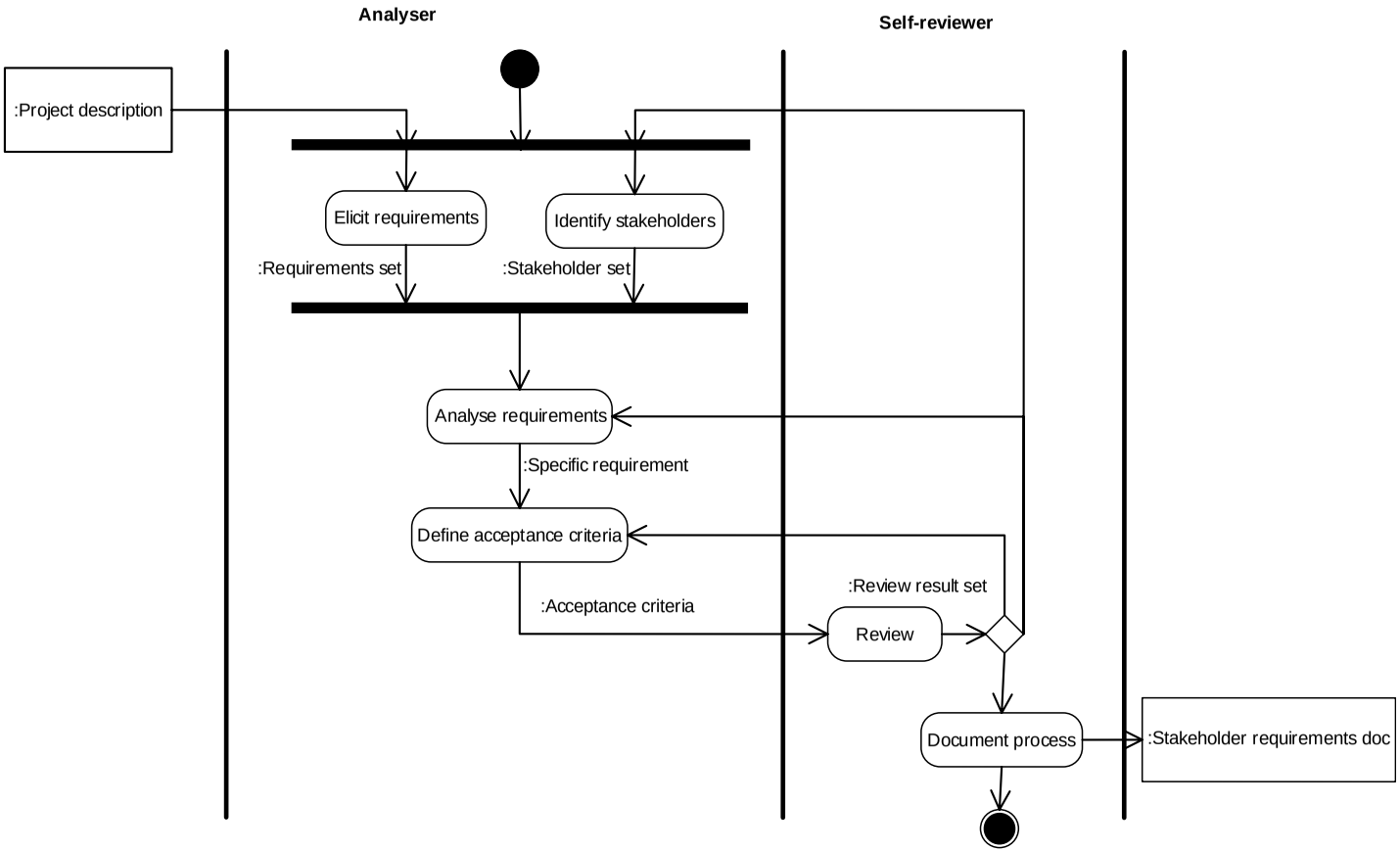

Figure 6.28 Process behaviour view – the 'Stakeholder requirements' process

The diagram in Figure 6.28 shows the behaviour for a single process from the STUMPI process model, in this case the 'Stakeholder requirements' process. This diagram ties together the role responsible for each activity using swim lanes and also shows the information flow, as well as the ordering of the activity execution.

Now that the life cycle has been identified and the process model has been defined, it is possible to go back to the life-cycle model and consider how each one behaves when it is executed. This will be done by considering 'iterations' within each stage and their associated process execution.

6.4.5 Stage iterations

Each stage is made up of a number of iterations, as specified in Figure 6.22. Each of these iterations consists of the execution of a sequence of processes, as can be seen in Figure 6.29, which shows an iteration that takes place in the conception stage.

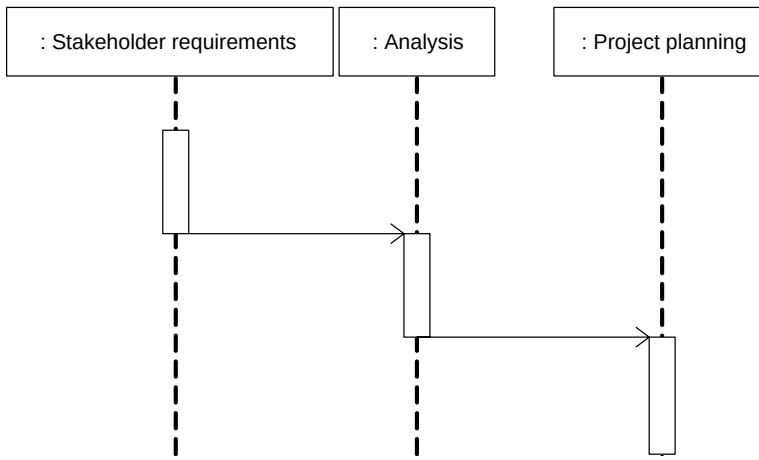

Figure 6.29 Process-instance view – an example of an iteration

The diagram in Figure 6.29 shows an example of a typical iteration that is taking place inside a stage – in this case the conception stage. The diagram shows that three processes are being executed: 'Stakeholder requirements', 'Analysis' and 'Project planning'. In this scenario, each process is being executed in a simple sequential manner, but there is no reason why there would not be more iterations or, indeed, more processes, between the process executions. In the case of the iteration shown here, it represents a simple iteration at the beginning of a project where some requirement engineering is taking place, followed by some analysis of the requirements model. This then enables some project management to take place based on the requirements and analysis models.

Another example of an iteration is shown in Figure 6.30, but this time there is more of an emphasis on analysis and design.

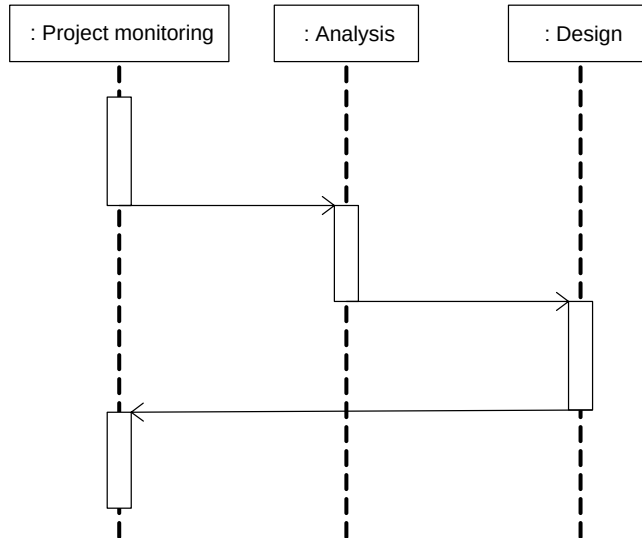

Figure 6.30 Process-instance view – another example of an iteration

Figure 6.30 shows another iteration that has come, this time once again, from the conception stage. Notice how, in both examples, there is a mixture of processes from both the 'Management' and 'Engineering' processes, which is typical of an everyday project. Seeing these two iterations, however, raises quite a fundamental point associated with iterations, which is, 'Why do we need iterations?' If these two iterations were being executed in a sequential order, then why bother showing them as separate iterations, rather than just a larger, single iteration? There are several answers to this but, to start with, think about if the iterations were being executed in parallel, rather than in sequence, then this would make perfect sense. It may very well be that two teams are working on the project at the same time, in which case the iterations must be considered separately rather than as a single iteration. In fact, thinking about the way that projects run in real life, it is unusual if everyone involved in the project is working on exactly the same process at all times and if these processes are operated in a sequential manner. In fact this is one of the reasons why the iterative approach has evolved from the linear approach of life-cycle modelling. Therefore, a good way to think about iterations is to consider different groups of people having their own iterations that can be executed sequentially or in parallel. To relate this back to the sort of project that STUMPI is designed for, it is easy to imagine how an individual project with a single worker would have a number of sequential iterations, whereas a project with a team of people working on it would suit a set of parallel iterations.

6.4.6 Process behaviour

So far, the behaviour of the life cycle (the life-cycle model) has been discussed along with stages and their associated iterations. It is now time to look at a lower-level

behavioural view of an individual process by defining an activity diagram for each process.

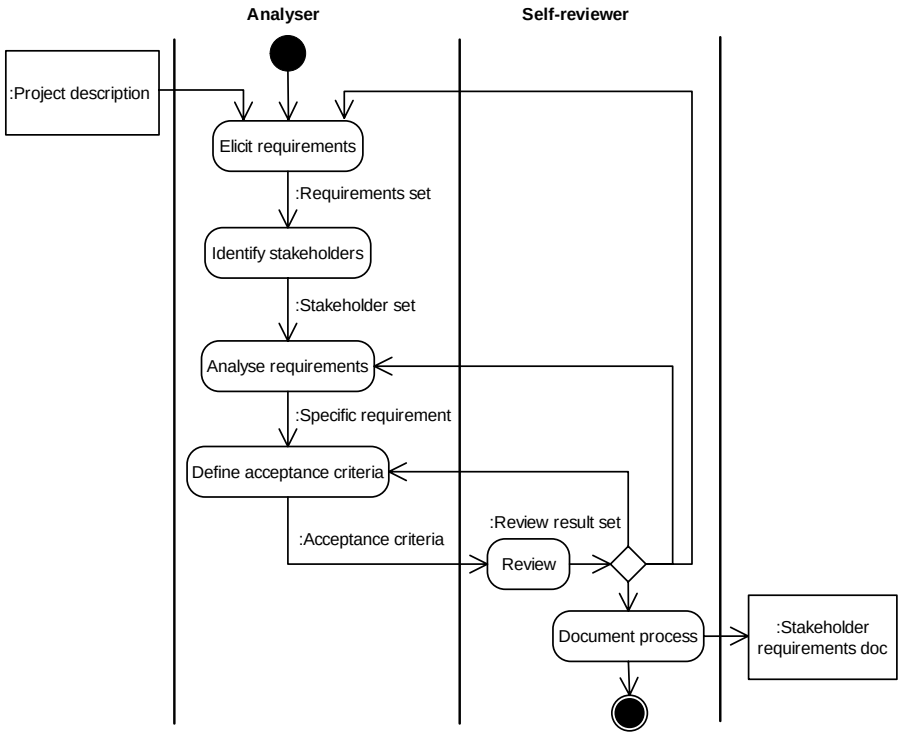

Figure 6.31 Process behavioural view of the 'Stakeholder requirements' process

The diagram in Figure 6.31 shows a behavioural view of one of the STUMPI processes, in this case the 'Stakeholder requirements' process. Note that this interpretation of the process is slightly different from the one defined in Figure 6.28, which shows that a single structural definition of a process can actually be executed in a number of different ways, depending on how the behaviour is defined. The activity diagram shows the order of execution of the activities that are carried out in the process, along with the production and consumption of the outcomes for each activity. Also shown here is the stakeholder responsibility for each activity by use of swim lanes.

An activity such as the one shown here should be created for each of the processes in the STUMPI model, which will lead to a fully defined set of basic processes.

6.4.7 Information view

For the purposes of this example, consider a development project that is being carried out according to a particular process and particular life-cycle model. The actual process and life-cycle model being followed are not too important for this example, as the emphasis will be on the artefacts of the project and the relationships between

them. It is crucial, however, that the artefacts have been identified and defined in an information model that can be represented as a block definition diagram.

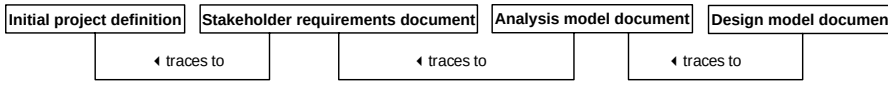

Figure 6.32 High-level view of a simple information model

Consider the diagram in Figure 6.32, which shows four main blocks and three relationships between them to represent a partial information model for the project. Each block represents an artefact for the project; in this example they are as follows.

- The 'Initial project definition', which is the rough-and-ready source of the requirements and is the output of one of the customer–supplier processes. This is not a formal requirements document and is more like a set of statements concerning the project.
- The 'Stakeholder requirements document', which is the formal statement of requirements for the project and is one of the outcomes of some sort of requirements process. The 'Stakeholder requirements document' must be traceable back to the 'Initial project definition'.
- The 'Analysis model document', which is the formal definition of the analysis, or specification, of the project and describes the problem, based on the requirements. This artefact is the outcome of some sort of analysis or specification process, and the 'Analysis model document' must be traceable back to the 'Stakeholder requirements document'.
- The 'Design model document', which is the formal definition of the design, based on the analysis, and describes the solution to the problem, based on the requirements. This 'Design model document' must be traceable back to the 'Analysis model document'.

The traceability paths between artefacts are shown as simple association on the diagram.

This diagram may now be examined in more detail, by looking at each of the artefacts in turn and considering the lower-level relationships between different parts of each artefact. It has already been established that there is a high-level relationship between artefacts, but now it is time to look at a lower level and explore exactly what the nature of the relationships is.

The diagram in Figure 6.33 is very similar to that shown in Figure 6.32, except this time it can be seen that the 'Initial requirements definition' is made up of a number

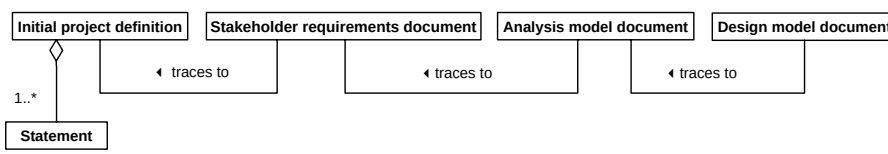

Figure 6.33 Lower-level view of the 'Initial project definition'

of ‘Statement’. Each statement will be analysed and, eventually, will evolve into the full, formal ‘Stakeholder requirements document’. However, there is more to the ‘Stakeholder requirements document’ than a fuller description of each statement, so this can now be looked at in more detail and is shown in Figure 6.34.

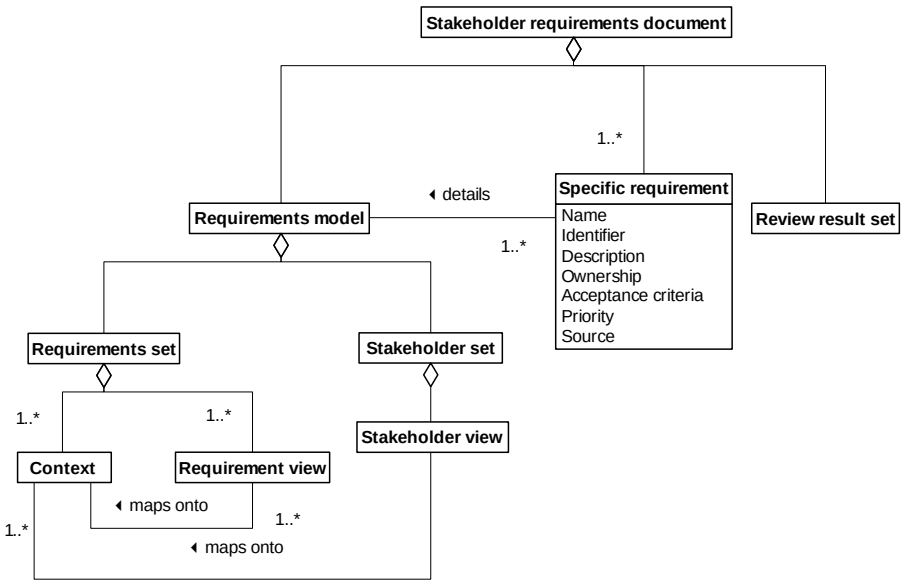

Figure 6.34 Lower-level view showing the structure of the ‘Stakeholder requirements document’

The diagram in Figure 6.34 shows a lower-level description of the ‘Stakeholder requirements document’. This is not intended to be a definitive model of what constitutes a stakeholder requirements document, but is based on good practice and is shown here for illustration purposes only.

The ‘Stakeholder requirements document’ is made up of a ‘Requirements model’, one or more ‘Specific requirement’, which detail the ‘Requirements model’, and a ‘Review result set’. Each ‘Specific requirement’ has a number of attributes (properties) that contained detailed information about the requirement.

The ‘Requirements model’ is made up of a ‘Requirements set’ and a ‘Stakeholder set’. The ‘Requirements set’ is made up of one or more ‘Context’ and one or more ‘Requirement view’, each of which maps onto a ‘Context’. The ‘Stakeholder set’ is made up of a ‘Stakeholder view’, which maps onto one or more ‘Context’ from the ‘Requirements set’.

To complete the information model, similar diagrams should be drawn for all the artefacts of the STUMPI processes. As well as showing the structure of these artefacts, the information model may relate the lower-level components of the various artefacts together, as appropriate.

6.5 Conclusions

This chapter introduced the application of SysML to process modelling. This was done by showing an example of how process modelling can be realized using the best-practice seven-views approach. This was illustrated by introducing the conceptual and realization views of the seven-views meta-model.

In order to illustrate the application of the seven views, three examples were chosen for the modelling: life-cycle modelling, standards modelling and tailored process modelling.

The world of process modelling is vast and modelling languages such as SysML and UML can be used extensively for many aspects of process modelling. This chapter has really only scratched the surface of process modelling using visual modelling. For a fuller explanation of this and the seven-views approach, see Reference 1.

6.6 References

1 Holt J. *A Pragmatic Guide to Business Process Modelling*. Swindon: British Computer Society; 2005
2 Pressman R. *Software Engineering: A Practitioner's Approach: European Adaptation*. Maidenhead: McGraw-Hill Publications; 2000
3 Jacobson I., Booch G. and Rumbaugh J. *The Unified Software Development Process*. Boston, MA: Addison-Wesley; 1999
4 Holt J. *UML for Systems Engineering: Watching the Wheels*. 2nd edn. London: IEE; 2004 (reprinted, IET Publishing; 2007)
5 Sheard S. 'The Frameworks Quagmire, a Brief Look'. *Proceedings of the Seventh Annual International Symposium of the International Council on Systems Engineering*; Los Angeles, CA: INCOSE; 1997

Chapter 7
Modelling requirements

> ‘If you don’t know where you’re going, you’re unlikely to end up there.’
>
> Forrest Gump

> ‘Father Brown laid down his cigar and said carefully: “It isn’t that they can’t see the solution. It is that they can’t see the problem.” ’
>
> G.K. Chesterton, ‘The Point of a Pin’, in *The Scandal of Father Brown*

7.1 Introduction

This chapter is concerned with modelling requirements. Requirements engineering is the discipline of engineering that is concerned with capturing, analysing and modelling requirements. In this book we are looking purely at modelling requirements with some degree of analysis. How these requirements are arrived at in the first place is entirely up to the engineer. Indeed, many different techniques for requirements capture will be mentioned, but will not be covered in any detail, since this is beyond the scope the book.

Before the SysML can be related to requirements, it is important to understand some of the basic concepts behind requirements engineering. These concepts will be introduced at a high level before any mention of the SysML is made. In this way, it is possible to set a common starting point from which to work with the SysML. Newcomers to the field of requirements engineering should use this section of the book as an introduction and should consult some of the books referred to at the end of the chapter for more in-depth discussion about the whys and wherefores of requirements engineering. Experienced requirements engineers should treat this section as a brief refresher, but it should be read at least once purely to determine the common starting point for the SysML work.

7.2 Requirements engineering basics

7.2.1 Introduction

This section introduces some fundamentals of requirements engineering. The emphasis in this chapter is on how to use the SysML to visualize requirements rather than to preach about requirements engineering itself; this section is therefore kept as brief as possible, while covering the basics. The basic concepts that are covered here will all be addressed practically in subsequent sections with respect to the SysML.

Getting the requirements right is crucial for any project for a number of reasons.

- The requirements will drive the rest of the project, and all other models and information in the project should, therefore, be traceable back to its original requirement. If you look at a typical information model, such as the one discussed in Chapter 6, this can turn out to be a large amount of documentation, all of which will be driven by the requirements.
- The requirements are the baseline against which all acceptance tests are defined and executed; the success of the project will be based on the project passing the acceptance tests, which rely entirely on the project meeting the original requirements. It follows, therefore, that the success of the project is directly based on the requirements.
- One of the definitions of quality that ISO uses is: 'conformance to requirements'. Therefore, requirements are absolutely crucial to achieving and then demonstrating quality.

One point that will emerge from this chapter is that much of the art of analysing requirements is about organizing existing requirements so that they can be understood in a simple and efficient manner.

For a more in-depth discussion of requirements engineering in general, see References 1 to 3.

7.2.2 The 'requirements stage'

Most people, when thinking of requirements, will associate requirements modelling with a 'requirements stage' in a project. Traditionally, this has always taken place towards the beginning of a project. However, in more recent times, it has become common practice to avoid the term 'requirements stage' and use a more generic term, such as 'concept stage', which is leading a project towards a more iterative than linear approach to its execution. This was seen in Chapter 6, where ISO 15288 was briefly considered in the context of process modelling.

One reason for this is that people will often confuse the terms 'stage' and 'process', and, indeed, when many people refer to the 'requirements stage', what they are actually referring to is the 'requirements process'. The requirements process will invariably be executed during an initial concept stage, but it should also be borne in mind that so may several other processes. The process that will be used in this section is shown in Figure 7.1.

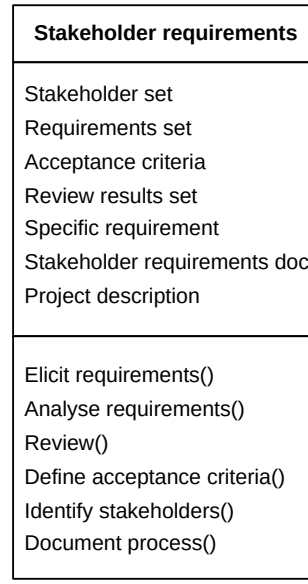

Figure 7.1 The stakeholder requirements process

Figure 7.1 shows a block that is called 'Stakeholder requirements', which represents a requirements process. This process is taken from an ISO-15288-compliant process called STUMPI, which was described in Chapter 6 and uses the generic term 'stakeholder' to qualify the term 'requirements'. Very often, the term 'user requirements' is used here, but the term 'stakeholder' is more accurate, since a user is simply a special type of stakeholder.

Adopting the notation used in Chapter 6, it can be seen from the diagram that there are seven artefacts associated with process (represented as properties): 'Stakeholder set', 'Requirements set', 'Acceptance criteria', 'Review results set', 'Specific requirement', 'Stakeholder requirements doc' and 'Project description'. Also, it can be seen that there are six main activities associated with the process (represented by operations) which are: 'elicit requirements', 'analyse requirements', 'review', 'define acceptance criteria', 'identify stakeholders' and 'document process'.

Now that the 'what' of the process has been defined, a logical next step is to define the 'how' of the process: that is, the behaviour of the process. The diagram chosen to represent the behaviour of the process is the activity diagram, which can be seen in Figure 7.2.

The diagram in Figure 7.2 shows the behaviour of the stakeholder requirements process in the form of an activity diagram. Each of the activity executions represents an activity from the process and is the equivalent of an operation from the parent block being executed. The main flow of control can be seen here by looking at the activity executions (represented by the 'round rectangle' shape), their order and the conditions between them.

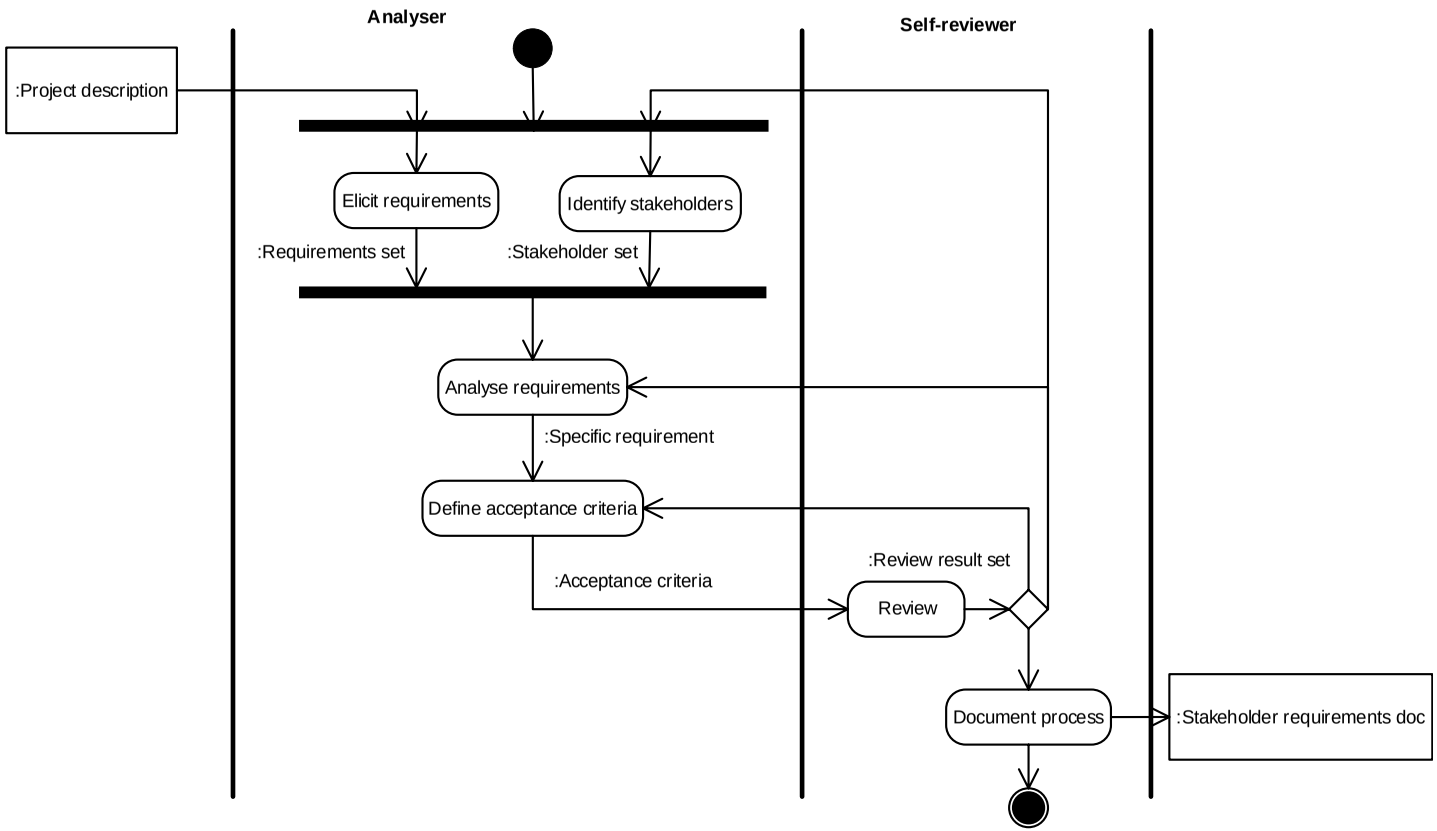

Figure 7.2 Behaviour of stakeholder requirements process

The flow of information around this process is shown by the object flows on the diagram, which provide an indication of which activity executions produce and/or consume which object nodes. These object nodes show the process artefacts, which are represented on the parent block as properties.

The swim lanes represent the roles, or stakeholders, that are responsible for various activities in the process. Any activity execution that falls within the boundary of a swim lane is the responsibility of the stakeholder named above the swim lane.

It should be stressed that the process shown here is simply an example and should be treated as such. The process itself is fully traceable back to ISO 15288, which provides the model of best practice for this example.

7.2.3 Capturing requirements

There are many ways to capture requirements, both formal and informal, and there is still much debate about which of the approaches yields the best requirements. Rather than enter into this debate, the following list will simply give an overview of the types of technique that may be adopted when capturing requirements.

- *Interviews, both formal and informal*. Formal interviews follow a predefined format and have predefined questions that yield results that are comparable. This may use predefined forms that define a set of questions and may indeed even include typical answers from which requirements may be drawn up. Informal interviews, on the other hand, start with a blank sheet of paper that will be used to record information and comments during an informal discussion. These results will then be analysed and, from these, a set of requirements may be drawn up. Each of these has its own set of advantages and disadvantages. For example, formal interviews may be conducted in such a way that the questions given are quantifiable so that they may be directly compared or may be analysed statistically. On the other hand, although not so easily quantifiable, informal interviews may pick up requirements that may have been missed out by the formal questionnaire-style approach.
- *Business requirements*. Requirements may be derived from business requirements. A business requirement is a very high-level requirement that drives the business rather than the product, yet may drive particular aspects of the project or set up some constraints. Business requirements are discussed in more detail in due course.
- *Comparable systems*. Other systems with similar features may be looked at in order to capture their functionality, which may be transformed into stakeholder requirements. These other systems may be similar systems within the same organization, legacy systems that may need to be updated, or they may even be a competitor's product.
- *Maintenance feedback*. The maintenance phase of a project should allow for customer feedback associated with a product. This is particularly important where the project is concerned with updating or replacing an existing system. The actual feedback may be in the form of error reports, customer wish lists or periodic surveys of customers' opinions of a product.

- *Working in the target environment*. Spending some time in the target environment can be invaluable when it comes to generating requirements. It may be that users of a system simply assume some functionality that may be overlooked when it comes to stating the system requirements. A fresh viewpoint from a nonspecialist may be exactly what is needed to draw out some obvious, yet crucial, system requirements.
- *User groups*. Many companies with a large customer base set up special user groups and conferences, where users can get together and exchange viewpoints and ideas. Although expensive, these meetings can prove invaluable for obtaining customers' requirements. These meetings also demonstrate to the customer that the supplier actually cares about what customers think and has put some effort and investment into constantly improving the product that they already use.
- *Formal studies*. Relevant studies such as journal and conference publications can often show up 'gaps' in products or provide a good comparison with other similar products. It is important, however, to assess how much detail the studies go into, as some can be superficial. It is also important to find out the driving force behind studies, as any study carried out or financed by a product vendor may lead to distorted facts and, hence, any derived requirements may be suspect.
- *Prototypes*. Prototypes give the customer some idea of what the final product will look like and hence can be useful to provide valuable early-life-cycle feedback. Indeed, this approach is so well promoted by some that it has led to the definition of a very popular life-cycle model: the *rapid prototype model*. The use of prototypes is particularly tempting for systems with a large software content, or those that use a computer-based interface to control or operate a system. User interface packages are very simple (and cheap to come by) and can give potential users a perfect picture of what they will be getting at the end of a project. Non-software prototypes are also very useful but can be very expensive if physical models have to be constructed.
- *User modifications*. These are not applicable to many systems, but some systems are designed so that they can be extended or modified by the customer. In order to relate this to a real-life example, think of a drawing or CAD package on a computer. Many of these packages allow the user to specify bespoke templates or drawing elements that may make the life of the user far simpler. Whenever a modification is made, either by user or supplier, there is generally a good reason why it has been done. This reason will almost invariably lead to a new requirement for the system, which can be incorporated into subsequent releases of the system.

This is by no means a complete set of requirements-capture techniques, but they give a good idea of the sheer diversity of techniques that have been adopted in the past. It may be that a completely different technique is more appropriate for your system – in which case use it.

Capturing requirements is all very well, but what in fact are requirements and what properties should they possess? The next section seeks to address some of these issues generally, before the SysML is used to realize them.

7.2.4 Requirements

7.2.4.1 Overview

The previous section has introduced a number of techniques for capturing requirements, but there is more to requirements than meets the eye. For example, requirements are often oversimplified and one of the main reasons for this is that there are actually three types of requirement. It has already been stated that much of the skill involved in modelling requirements is involved with organizing requirements into logical and sensible categories. These three categories are dictated by the types of requirement that are shown in Figure 7.3.

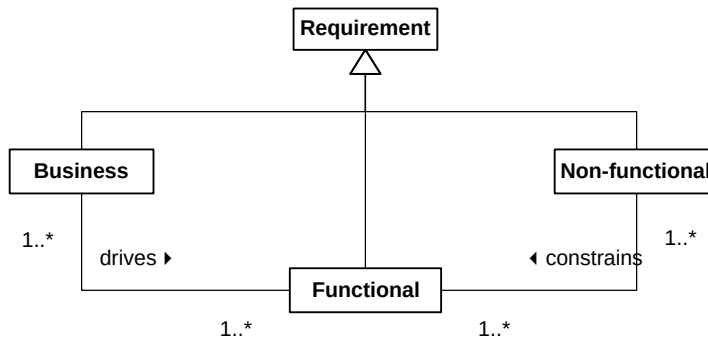

Figure 7.3 Types of requirement

Figure 7.3 shows that there are three types of 'Requirement': 'Business', 'Functional' and 'Non-functional'. One or more 'Business' requirement drives one or more 'Functional' requirement, and one or more 'Non-functional' requirement constrains one or more 'Functional' requirement.

'Functional' requirements are what are thought of as traditional stakeholder requirements and 'Non-functional' requirements are often referred to as 'constraints' or 'implementation requirements'. Each of these types of requirement will be discussed in more detail in the following sections.

7.2.4.2 Business requirements

The first type of requirement to be discussed is the business requirement. Business requirements, as the name implies, relate to the fundamental business of the organization. Business requirements tend to be oriented more towards high-level business concerns, such as schedule and cost, rather than towards development itself and, as such, often fall outside the context of the system.

Business requirements and functional requirements are often confused, but are not quite the same thing. Business requirements will relate to things that drive the business, such as ‘keep the customer happy’, ‘improve product quality’. Because of this, business requirements are often implied, but rarely stated explicitly. They are normally aired from a point of view that ties them into business processes. Chapter 6 introduced the subject of process modelling and showed that there were other concerns apart from engineering. When looking at a process such as ISO 15288, we see that business requirements often relate heavily to the ‘Enterprise’ process group, rather than ‘Technical’.

Figure 7.4 shows a set of business requirements that have been generated for a systems engineering organization, which may help to clear up some of these points.

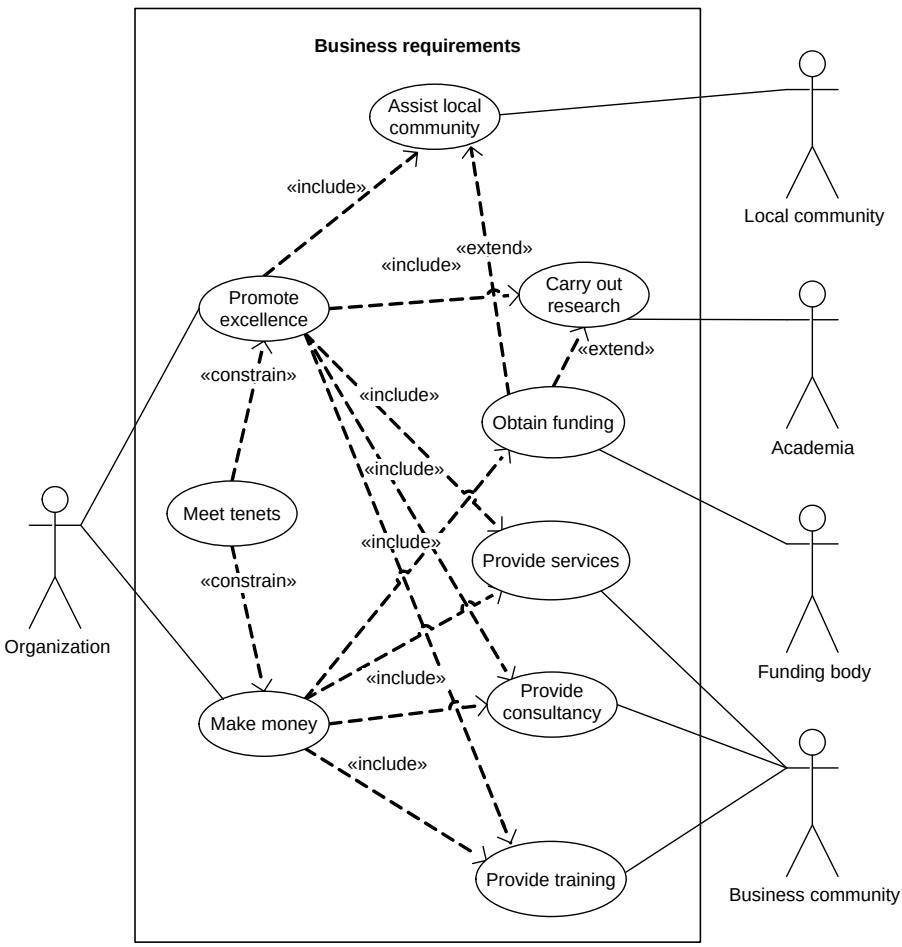

Figure 7.4 Business requirements for a systems engineering organization

Figure 7.4 is actually a use case diagram that is being used to describe the business context (discussed in more detail later) for a real-life systems engineering organization. All the use cases on this model are actually representing business requirements, rather than functional requirements. Look at the nature of these requirements and it can be seen that they exist at a very high level conceptually. For example, 'promote excellence' just happens to come directly from this organization's mission statement, which must be a business requirement for the organization, otherwise they are not fulfilling their mission. The second high-level business requirement is to 'make money', which must be an underlying factor associated with almost every business in the world.

The 'make money' requirement has an association with four other requirements: 'Obtain funding', 'Provide services', 'Provide consultancy' and 'Provide training'. This relationship is an 'include' relationship, which will be discussed in more detail later in this chapter, although its meaning is self-explanatory.

The 'Promote excellence' requirement also has an association with several other requirements: 'Assist local community', 'Carry out research', 'Provide services', 'Provide consultancy' and 'Provide training'.

The 'Obtain funding' requirement extends both 'Carry out research' and 'Assist local community'. The 'extend' relationship will be discussed in more detail later in this chapter.

The actors in this model, which are shown graphically by stick people, actually represent project stakeholders. Stakeholders represent any role, or set of roles, that has an interest in the project, and will be discussed in more detail later in this chapter.

The model that is used here will be used throughout this chapter and will drive the functional requirements for projects within the organization. Therefore, any projects that are used as examples from this organization must be traceable back to the company's original business requirements.

7.2.4.3 Functional requirements

Functional requirements are the typical stakeholder requirements for a system, in that they define the desired functionality of a system. Stakeholder requirements should have some observable effect on a particular stakeholder of a system, otherwise they are possibly being defined at too low a level.

One of the relationships that were established in Figure 7.3 was that 'Business requirements' drive 'Functional requirements'. This means that all functional requirements should, in some way, be traceable back to the organization's business requirements. If this is not the case, perhaps it should be questioned why this requirement is necessary.

7.2.4.4 Non-functional requirements

The third and final type of requirement that was introduced in Figure 7.3 is the 'non-functional' requirements, which are also sometimes referred to as 'implementation

requirements' or 'constraints'. These 'non-functional' requirements constrain functional requirements, which means that they may limit functional requirements in some way. Examples of non-functional requirements include the following.

- *Quality issues*. A common non-functional requirement is 'comply with standard', where the word *standard* may refer to some sort of established norm, such as an international standard or process. This is particularly relevant when read with Chapter 6, which discusses the importance of standards compliance. This may be a fundamental requirement of the system and may make all the difference between meeting acceptance tests and failing them.
- *Implementation issues*. It may be desirable to use a particular technique or language to implement the final system. As an example of this, imagine a system whose target environment may contain other systems and will require a particular platform or protocol to be used.
- *Life-cycle issues*. This may include the way in which a project is carried out. It may be that a particular project-management technique should be used. An example of this is the PRINCE system, which is often cited in government projects as an essential part of the project. This may also include other constraints, such as design techniques and methodologies and even the modelling language (such as the SysML!).

There are many more types of non-functional requirement; the above is a small sample.

It is essential that these non-functional requirements be treated in the same way as both functional and business requirements, and that they exist on the requirements models.

7.2.4.5 Properties of requirements

Once requirements have been identified, it is important that they be classified in some way so that their status may be assessed at any point in the project and they may be organized and manipulated properly. In order to classify requirements, they often have their features, or properties, defined – see References 4 to 6. This may be represented graphically by adding properties to a block that represents a requirement. Figure 7.5 shows such a block with some example properties added.

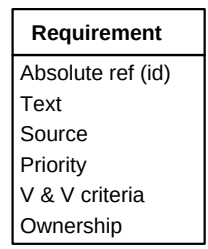

Figure 7.5 Properties or a requirement shown as properties

Figure 7.5 shows a block that represents a requirement with several properties of a generic requirement defined. These requirements would be inherited by each of the three types of requirement – business, functional and non-functional. As described in Chapter 4, SysML defines a requirement block as part of the language. However, SysML *requires* such requirement blocks only to have the 'id' and 'text' properties. The SysML specification does, however, state that 'Additional properties such as verification status can be specified by the user'. The block depicted here shows what the authors consider to be the minimum set of such properties.

The properties chosen here represent typical properties that may be useful during a project; however, the list is by no means exhaustive. The properties currently shown are as follows.

- 'Absolute reference (id)', which represents a unique identifier by which the requirement may be identified and located at any point during the project. This also forms the basis for any traceability that may be established during the project to show, for example, how any part of any deliverable during any phase of the project may be traced back to its driving requirement from the user requirement specification.
- 'Text', which contains a textual description of the requirement. Often an organization will define a standard sentence structure to be used when describing requirements.
- 'Source', which represents where the requirement was originated. This is particularly useful when there are many source documents that the stakeholder requirements have been drawn from, as it may reference the role of the person who asked for it.
- 'Priority', which will give an indication of the order in which the requirements should be addressed. A typical selection of values for this property may be 'essential', 'desirable' and 'optional'. This is similar to the concept of 'enumerated types' in software, where the possible values for a property are predefined, rather than allowing a user to enter any text whatsoever. This is important, as in many cases it is simply not possible to meet all requirements in a project, and thus they are prioritized in terms of which are essential for the delivery of the system. This also takes into account the fact that many functional requirements are user wishes that are no more than 'bells and whistles' and do not affect the core functionality of the system. There is often quite a large difference between what the customer wants and what the customer needs. Although both are valid requirements, it is important that the customer's needs be addressed before their wants so that a minimum working system can be delivered. Of course, it can be very hard to get a customer to rate a requirement as anything other than 'essential'.
- Verification/validation criteria, which give a brief idea of how compliance with the requirement may be demonstrated. This may be split into two different properties or may be represented as one, as shown here. It is crucial that there be some clear way to establish whether a requirement may be verified (that it works properly) and validated (that it has been met), as the validation will form the basis for the customer accepting the system once it has been delivered. This will be

high-level information and will not be a test specification, for example, although the information here will be used when producing test specifications. If it is not possible to define verification and validation criteria, this is often a sign that the requirement is not well formulated and should be investigated further.

- 'Ownership', which will be related directly to the responsible stakeholder in the system. If the requirement is not owned by one of the defined stakeholders, there is something wrong with the stakeholder model. The stakeholder model is discussed later in this chapter.

Many other properties may be defined depending on the type of project that is being undertaken. Examples of other requirement properties include: urgency, performance and stability.

It is important that requirements are written in a concise and coherent way in order to avoid:

- *irrelevancies*: it is very easy to put in too much background information with each requirement description, so that the focus is lost amidst irrelevant issues;
- *duplication*: it is important that each requirement is only stated once; this is particularly important when requirements affect other requirements, such as constraints, which are ripe for duplication; and
- *overdescription*: verbosity should be avoided wherever possible.

One of the keys to successful requirements engineering is the way in which the requirements are organized. Much of this chapter will deal with organizing the models of requirements into a coherent set of usable requirements.

7.2.5 Stakeholders

7.2.5.1 Overview

This section examines another fundamental aspect of requirements modelling, which is that of stakeholders. A stakeholder represents the role, or set of roles, of anyone or any thing that has a vested interest in the project, which can range from an end-user to shareholders in an organization, to the government of a particular nation.

It is very important to identify stakeholders early on in the project, as they can be used to relate to any type of requirement and can also be used to define consistent responsibilities.

7.2.5.2 Types of stakeholder

One of the practices identified in the requirements process was that of 'identify stakeholders'. This is often written as 'identify users' or 'identify roles', depending on which source is used for reference. The reason why these terms are often used interchangeably is that a 'user' is actually a type of stakeholder. A user will actually use the output of the project, the product, and may be thought of as having an interest in the project. Figure 7.6 shows this relationship from a SysML point of view.

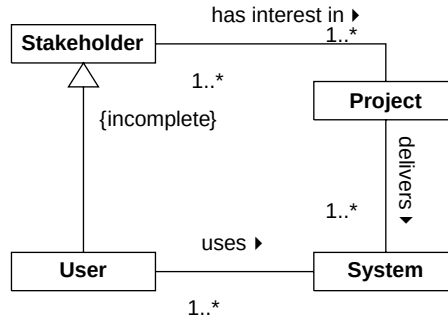

Figure 7.6 Relationship between users and stakeholders

From the model, ‘User’ is a type of ‘Stakeholder’. One or more ‘User’ uses the ‘System’, while one or more ‘Stakeholder’ has an interest in a ‘Project’. In addition, the ‘Project’ delivers one or more ‘System’.

Although it is impossible to list all possible stakeholders in an organization, it is possible to draw up a generic model that is very useful as a starting point for a stakeholder model. Figure 7.7 shows such a generic model that will not fit all projects or organizations, but is very useful as a starting point for identifying any potential stakeholders.

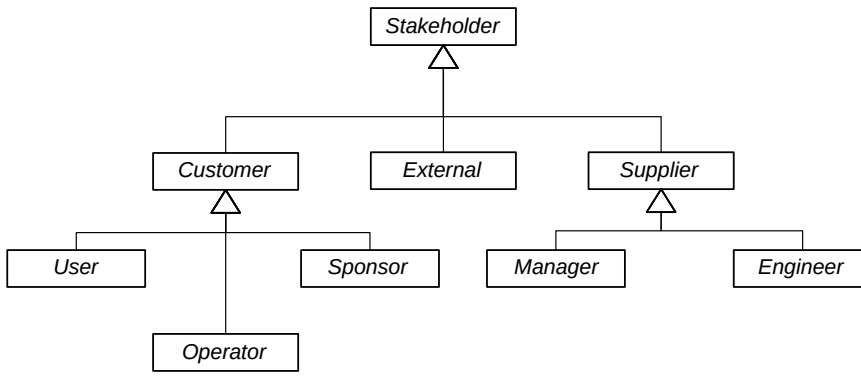

Figure 7.7 Types of stakeholder

It can be seen that there are three main types of ‘Stakeholder’: ‘Customer’, ‘Supplier’ and ‘External’. These are chosen as the highest-level stakeholders, as almost every project will have both a supplier and a customer and will normally have some external stakeholders, such as standards or external systems. It should be pointed out that the name of the block that represents each stakeholder represents a role that is being performed, rather than the name of an individual or particular organization. It may be that the ‘Customer’ and ‘Supplier’ stakeholders are actually within the same

organization, or may even be the same person in the case of one person who is creating a system for personal use.

The 'Customer' stakeholder has three main types: 'User', 'Operator' and 'Sponsor'. The 'User' will be the role that represents the end-user of the system – for example, in the case of a transport system such as rail or air, the 'User' role would be fulfilled by the actual passengers. The role of the 'Operator' in the same example would be taken by the staff who actually work at the stations or ports where the transport is taking place and the people who drive or fly the vehicles. The 'Sponsor' role represents whoever is responsible for paying for the project, which may be a large organization such as an airline or rail company in the example.

The second main type of 'Stakeholder' is the 'Supplier', which has two main types: 'Manager' and 'Engineer'. Again, these are generic roles that should be used as a starting point for the real stakeholder model.

'External' stakeholders can have many different types, too many to show here. Typical 'External' stakeholders include standards, professional bodies and other systems and projects with which the project under development must interact.

In order to take the concept of stakeholders further, let us consider an example that fits in with the business requirements that were shown in Figure 7.4, with regard to the 'provide training' business requirement. The idea here is to identify the stakeholders for this requirement by using the model shown in Figure 7.4 as a starting point for the reasoning.

Imagine a situation where a client organization has asked the organization whose business requirements have been defined to provide a training course at the client's premises. Therefore, a project is started that, for the sake of this example, will be referred to as the 'provide training course' project. The stakeholder model has been identified as shown in Figure 7.8.

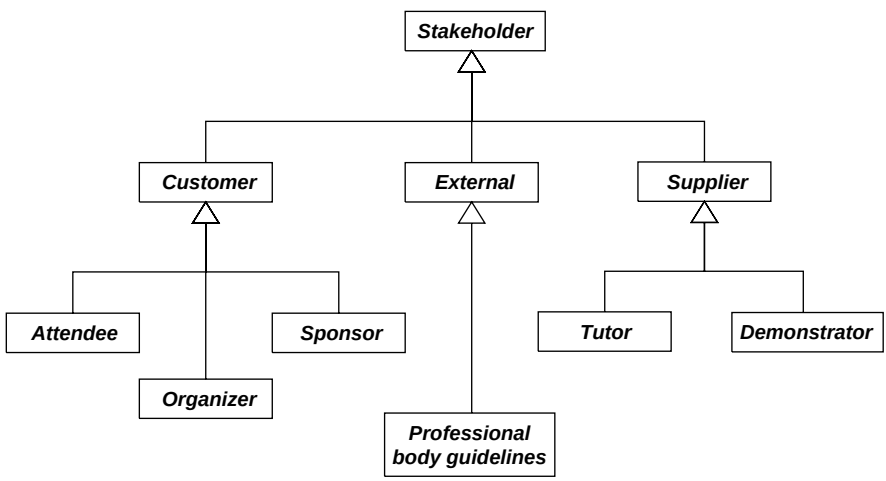

Figure 7.8 Example stakeholder model for 'provide training' requirement

The model in Figure 7.8 shows some stakeholders that have been identified for the 'provide training' business requirement. The main 'Customer' and 'Supplier' types of 'Stakeholder' remain the same, because, in order to deliver a training course, there must be someone who wants the course, the 'Customer', and someone who can supply it, the 'Supplier'.

The 'Customer' stakeholder has been split into three types in this example:

- the 'Organizer', whose role it is to set up the logistics of the course and ensure that there are enough attendees to justify the expense of the course;
- the 'Attendee', which represents the people who will be attending the course; and
- the 'Sponsor', which represents the role that actually pays for the course.

In the same way, the 'Supplier' stakeholder has two types that differ from the original generic model. These two types are:

- the 'Tutor', who is the person or team of people who teach the course; and
- the 'Demonstrator', who is the person or team of people that provide practical demonstrations that complement the tutor's teaching efforts.

The 'External' stakeholder is also present and has a sub-type defined, 'Professional body guidelines', since the provision of training courses is governed by an accrediting professional body.

It is useful to remember that the stakeholders represent the roles, rather than individuals, involved in the project. There are a number of main reasons for this.

- There is nothing to say that each role has to be a separate individual or entity, as it is possible for one person to take on more than one role. It is important that the roles be considered rather than the individual, as it may be that the system is split up with regard to the functionality attached to each role, rather than to a person.
- If an individual leaves a project and has a number of roles, it may be that more than one person can replace the initial individual. It may also occur that somebody moves position within a project, maybe up the management hierarchy, so it is absolutely crucial that the roles associated with that person should not be confused with any new roles. If all associations are with an individual, rather than a role, this will become confusing to the possible detriment of the project.
- One person may take on two roles. For example, the role of the 'Tutor' and the 'Demonstrator' may be taken on by one person for one instance of the course and this may change for another instance of the course. It is important, therefore, to know the skills and specialist knowledge required by the role so that, if a person who takes on both roles needs to be replaced, it can be ensured that the replacement can fulfil both roles.

7.2.6 Summary

The information that has been introduced and discussed so far in this chapter may be summarized by the SysML model shown in Figure 7.9. This diagram may be thought of as a meta-model.

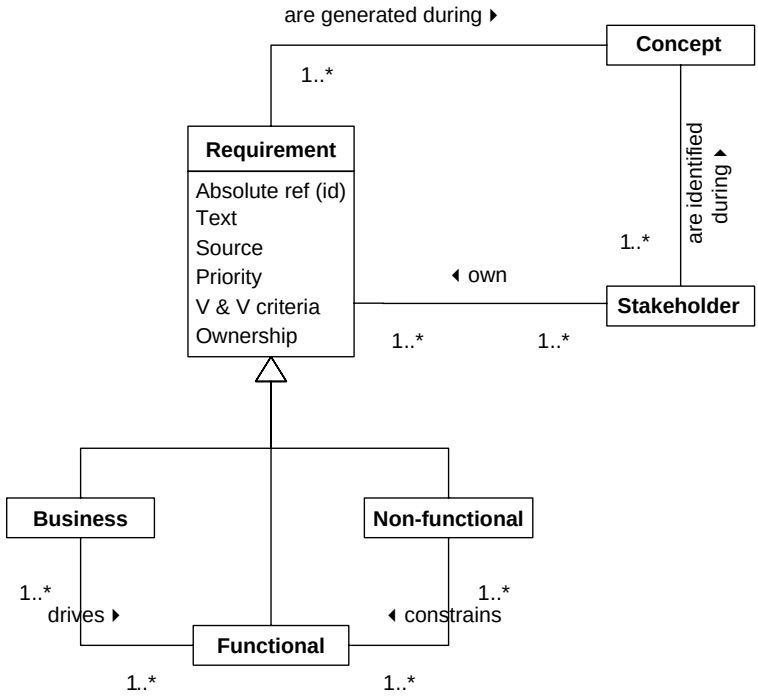

Figure 7.9 Summary so far – requirements meta-model

In summary, one or more ‘Requirement’ is generated during ‘User requirements’, which is a type of ‘Phase’. One or more ‘Stakeholder’ is identified during ‘User requirements’ and one or more ‘Stakeholder’ owns one or more ‘Requirement’. Each ‘Requirement’ has the properties ‘source’, ‘priority’, ‘V&V criteria’, ‘ownership’ and ‘absolute ref’.

There are three types of ‘Requirement’: ‘Business’, ‘Functional’ and ‘Non-functional’. One or more ‘Non-functional’ requirements constrain one or more ‘Functional’ requirements. One or more ‘Business’ requirements drive one or more ‘Functional’ requirements.

This covers the basic concepts associated with requirements engineering. These concepts will now be related to SysML issues in Section 7.3.

7.3 Using use case diagrams (usefully)

Now that the basics of requirements engineering have been introduced, albeit briefly, it is useful now to take another look at the SysML diagram that will be used for most of the requirements engineering models: the use case diagram. By way of a recap, the meta-model for use case diagrams is shown as Figure 7.10.

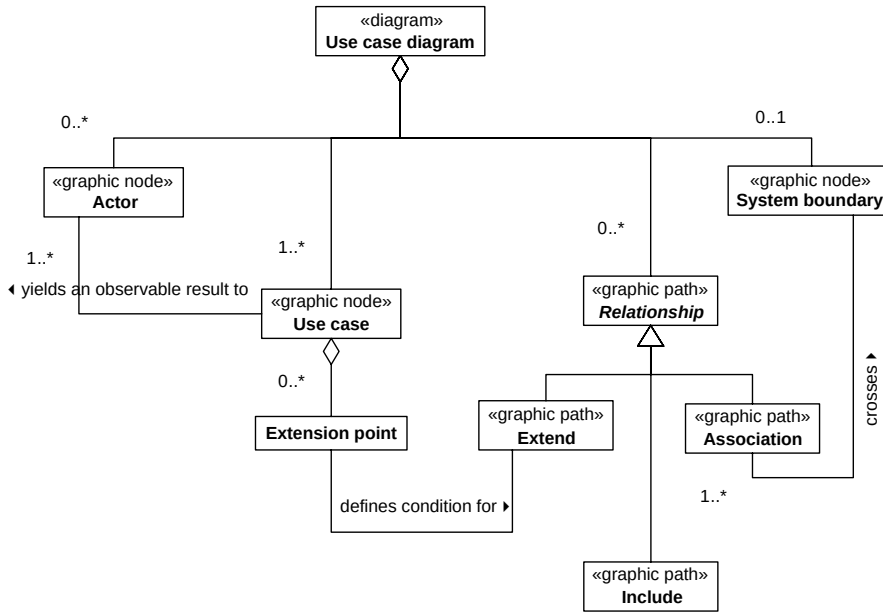

Figure 7.10 Partial meta-model for use case diagrams

Figure 7.10 shows the partial meta-model for use case diagrams, which was first introduced in Chapter 4. It can be seen that a 'Use case diagram' is made up of one or more 'Use case', zero or more 'Actor', one or more 'Relationship' and zero or one 'System boundary'. There are three special types of 'Relationship': 'Association', 'Extend' and 'Include'. A 'Use case' yields an observable result to one or more 'Actor' and is made up of zero or more 'Extension point', which defines the conditions for use cases related by the 'Extend' relationship. The 'Association' is used to connect actors to use cases, crossing the 'System boundary' in doing so.

The remainder of this chapter discusses how to apply these SysML concepts to requirements modelling.

7.4 Context modelling

7.4.1 Types of context

Several types of requirement have been introduced and the relationship between them defined. However, in reality, it is useful to separate these types of requirement and relate them to different types of stakeholder. This organization of requirements results in defining the 'context' of a system.

The context of a system represents what the system is and its boundaries from a particular viewpoint. This is absolutely critical, as it is possible to have many

contexts that relate to a single system, but that exist from different points of view. For example, the system context for a customer may be different from the system context of an operator. Something that is perceived as lying outside the context of one system may be included when looked at from a slightly different angle. Examples of this will be given in due course.

The context of the system shows the system boundary and any users or peripherals that communicate with the system. This may include 'who' interacts with the system and also 'what' interacts with the system. It should be noted that users are not necessarily people and may be peripherals, such as computers, hardware and databases, or, indeed, another system. In terms of what has been shown so far, these entities that interact with the system are equivalent to stakeholders. This means that if all stakeholders have already been identified correctly, they can be used to help model the context of the project. It is important to understand that every stakeholder may itself be a system that has its own context. Also, the product (*what* is being built) and the project (*how* it is being built) will have their own, and different, contexts.

Two commonly used types of context are the 'business context' and the 'system context'. Each context is defined by the system boundary, which is one of the basic elements of the use case diagram. The system boundary is actually a very interesting piece of syntax for the SysML, as it is either massively misused or underused. Many SysML texts, tools and references ignore the system boundary altogether or simply state that it is optional. If the system boundary was optional, with no guidelines issued for when it should and should not be used, what would be the point of it in the first place? An excellent use for the system boundary element in the SysML is to define a context in a system. In a nutshell, if a use case diagram has a system boundary, it is a context diagram. If it does not have a system boundary, it is simply showing the breakdown of requirements.

Each of these two types of context will now be looked at in some detail and the example used so far will be taken further and, in some cases, revisited with a new reflection on matters.

7.4.1.1 Business context

The business context shows the business requirements of an organization or a particular project. The business requirements that were shown in Figure 7.4 are really a very high-level business context that sets the context for the whole organization. It may be that new, additional business contexts are generated for each project, but it is absolutely crucial that these be consistent with the high-level business requirements, or context, otherwise the company will be straying from its original mission.

As a general rule of thumb, business requirements will exist within a business context, rather than non-functional or functional requirements. In the same vein, the roles that exist outside the context of the system are generally high-level stakeholders, rather than actual users of the system. The use case diagram in Figure 7.11 shows a generic business context that highlights these points.

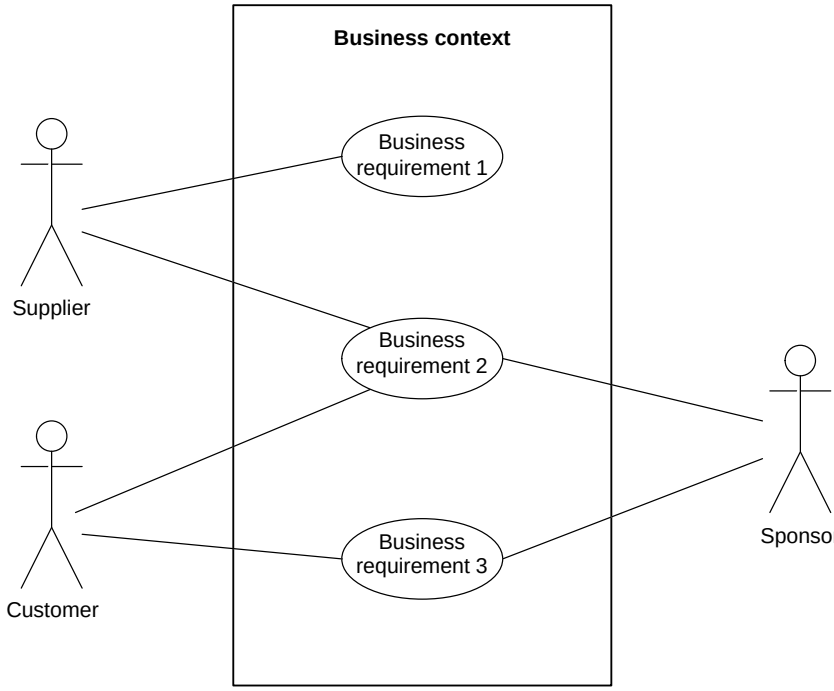

Figure 7.11 Generic business context

It can be seen from the diagram in Figure 7.11 that the various stakeholders – 'Customer', 'Supplier' and 'Sponsor' – are associated with various business requirements: 'Business requirement 1', 'Business requirement 2' and 'Business requirement 3'.

If this is related back to the original business requirements for the systems engineering organization in Figure 7.4, it can be seen that these rules of thumb hold true. All the actors in the diagram are high-level stakeholders rather than users and all the use cases are business requirements.

Note the explicit use of the system boundary, which shows that this model is a context model rather than a straight requirements model. The system boundary is represented very simply by a rectangular box that goes around the requirements (shown as use cases) in the context and keeps out the stakeholders (shown as actors).

The second type of context to consider is the 'system context', which is discussed in 7.4.1.2 below, before a full example is worked through, showing how to differentiate between the two types of context, and offers advice on how practically to create both.

7.4.1.2 System context

The system context relates directly to the project at hand and is concerned with the stakeholder requirements and constraints on them, rather than with the high-level business requirements. The stakeholders that exist as actors on the use case

diagram are typically the users and lower-level stakeholders, such as people who will be involved directly with the day-to-day running of the project – in other words, managers and engineers.

As a general rule of thumb, therefore, the actors on the system context will be users and low-level stakeholders and the use cases will be functional and non-functional requirements. Figure 7.12 shows a generic system context that represents these rules of thumb.

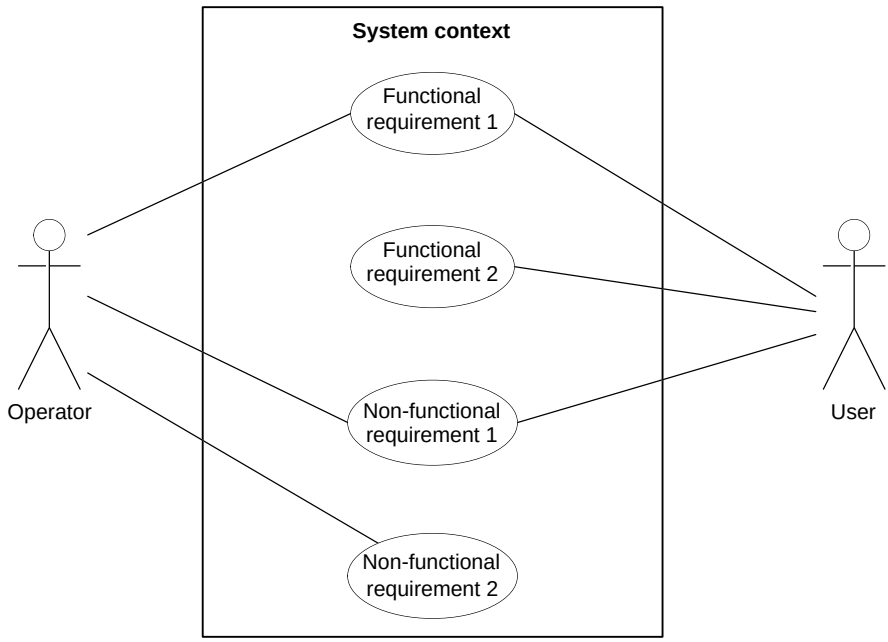

Figure 7.12 Generic system context

Figure 7.12 shows that the system context diagram consists of various types of functional requirement ('Functional requirement 1' and 'Functional requirement 2') and various types of non-functional requirement ('Non-functional requirement 1' and 'Non-functional requirement 2').

Now that the basic rules of thumb have been established, it is possible to apply these to the example that has been used previously in this chapter. This is discussed in Section 7.4.2 below, following a brief discussion of other contexts.

7.4.1.3 Other contexts and other points of view

Although we have been talking about the business and system contexts as though they were the only possible contexts that can be drawn, this is not the case. They are simply the most common types drawn.

When producing context diagrams it may be necessary to draw them from the point of view of the different stakeholders involved in the project.

Consider a railway project that has the following stakeholders: train, driver, signals, passenger. It is possible to draw a context diagram for the entire project, as in Figure 7.13. This is not incorrect: it gives the context of the project and shows all of the high-level project requirements.

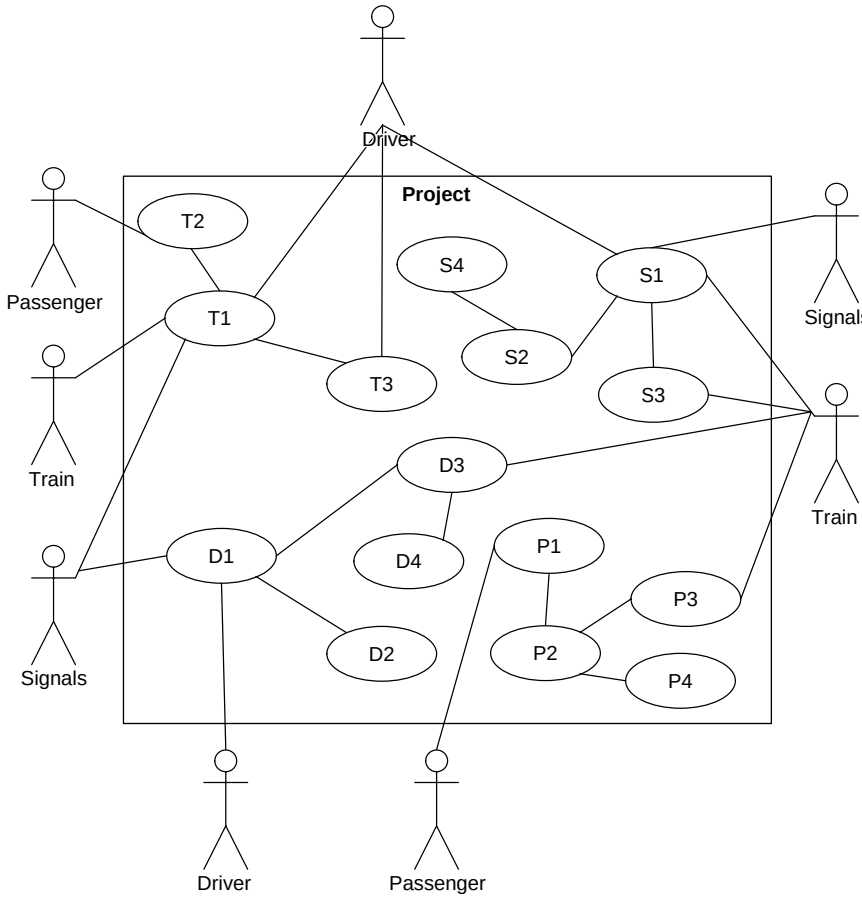

Figure 7.13 Full project context

However, for any reasonably sized project, such a context diagram is likely to contain a large number of use cases, making it very difficult to understand the context of the project as seen by the major stakeholders.

A better approach is to draw context diagrams for the major stakeholders. (It is possible to draw a context diagram for each stakeholder, but in practice this would generally be too low-level and of little value. A context diagram for each major stakeholder or stakeholder grouping is generally sufficient.)

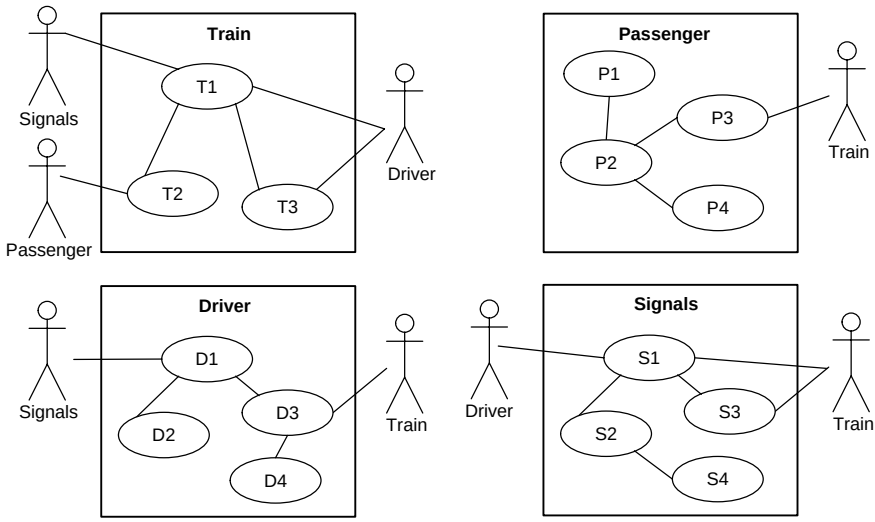

Figure 7.14 Context diagrams per stakeholder

This has been done in Figure 7.14, where four context diagrams have been drawn, each from the point of view of a different stakeholder. The requirements for the project as a whole are the sum of those on each of the context diagrams. Each diagram shows the context of the project from the point of view of a single stakeholder and is clearer to understand than a single context diagram.

7.4.2 Practical context modelling

Consider now an example where the systems engineering organization that has been modelled previously in Figure 7.4 wants to start up a project to run training courses. This was introduced previously and a set of stakeholders was identified. This example will now be taken one step further and the business and system contexts will be created for the 'provide training course' project. The first question that should be asked is whether or not the project is consistent with the organization's mission. In this case there is an obvious connection, as one of the business requirements for the organization is to 'provide training'. If there was no clear connection to the business requirements of the organization then one of two things must be done.

- The business requirements of the organization must be changed to reflect the new project. If this is not done, the project does not fit in within the defined scope of the company's activities.
- If it is the case that the project is outside the scope of the company's current activities and the company is not willing to add it to its list of business requirements, the answer is quite clear: the project should not be taken on in the first place!

The first step is to ensure that the project can be proven to be consistent with the organization's business requirements and then the next step is to identify the

requirements for the project and the stakeholders. This will, in reality, be an iterative process and will take several iterations to get right.

As a first pass, let us try to identify some general requirements, regardless of type. These requirements were generated by simply asking a group of stakeholders what they perceived the requirements to be. At this stage, the requirements were not categorized, but were simply written down as a list. The list below shows a typical set of requirements that is generated when this example is used to illustrate context modelling:

- 'ensure quality', which refers to the work that is carried out by the supplier organization;
- 'promote supplier organization', in terms of advertising what other services the organization can offer and hope for some repeat business or future business;
- 'improve course', which means that any lessons learned from delivering the course should be fed back into the organization in order to constantly improve courses as time goes on;
- 'teach new skills', from the point of view of the supplier organization;
- 'provide value', so that the clients will keep coming back for more business;
- 'organize course', which involves setting up, advertising, etc.; and
- 'deliver course', which consists of the delivery of the course, in terms of teaching and practical demonstrations.

The next step, once the initial set of requirements has been identified, is to categorize each requirement by deciding whether it is a business requirement, a functional requirement or a non-functional requirement. The list of business requirements is:

- 'promote supplier organization', which is clearly aimed at the business level and will relate directly back to the original business requirements of the organization; and
- 'provide value', which relates directly back to the supplier's business requirements.

The list of functional requirements is:

- 'teach new skills', which is what is expected to come out of the course;
- 'deliver course', which involves somebody turning up and teaching the course and providing course materials; and
- 'organize course', which involves publicizing the course, setting up the course and providing support for the course.

The list of non-functional requirements is:

- 'ensure quality', which means that the quality of the course must be maintained (in reality, this may equate to giving out course assessment forms and feeding them back into the system); and
- 'improve quality', which relates to how the course may be improved constantly in order to meet one of the original business requirements of 'promote excellence'.

It may be argued that these two non-functional requirements are one and the same and, hence, would be condensed into a single requirement. It may also be argued that they are separate, distinct requirements, and that they should remain as two requirements. The problem is, which of the two is correct? As there is no simple answer to this, there is a particular rule of thumb that should be applied under such circumstances. If the answer to whether to split up or condense requirements is unclear, keep them as two. This is because experience has shown that it is far easier to condense two requirements at a later time than it is to split a single requirement in two.

Now that a number of requirements have been identified, what about the stakeholders? The list of stakeholders has already been identified in a previous section and can thus be reused here. First of all, consider the business context and apply the rules of thumb that were discussed previously. The resulting business context is shown in Figure 7.15.

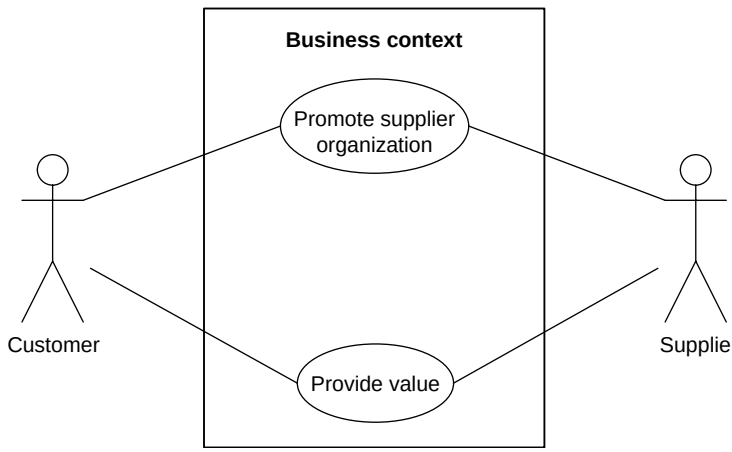

Figure 7.15 Business context for the training-courses example

Figure 7.15 shows the business context for organizing training courses. Note that the actors here are high-level stakeholders and that the use cases are business requirements. This model should be able to be traced directly back to the organization's business requirements model. It could be argued that both use cases relate directly to both the 'provide value' and 'promote excellence' business requirements. There is no problem with this; there is no reason why the mapping should be one-to-one. Note also that the actors in this model are high-level stakeholders from the stakeholder model that was defined in Figure 7.8, which maintains the rule concerning high-level stakeholders and business contexts.

The next step is to create the system context, which should, hopefully, make use of the remaining requirements (both functional and non-functional) and the remaining

stakeholders (user-level stakeholders). The resulting system context is shown in Figure 7.16.

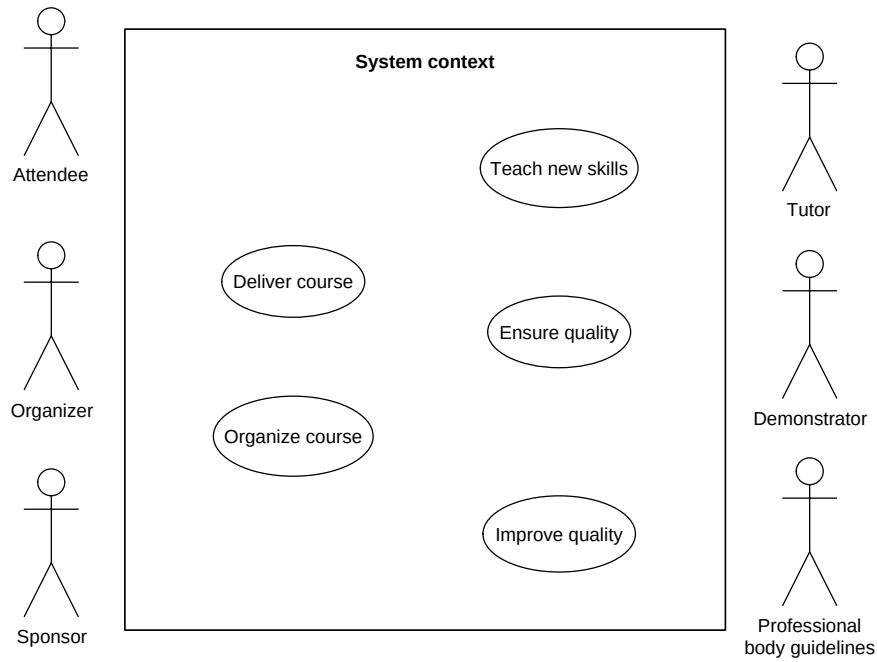

Figure 7.16 System context for the training-courses example

Figure 7.16 shows the system context for the training-courses project. This should be concerned with the nuts-and-bolts activities of actually running the course, rather than the high-level business requirements that appear on the business context.

The actors that have been included on this model are actually the same as the stakeholders that were identified back in Figure 7.1. By using the use case diagrams in conjunction with the original stakeholder model (realized by the block definition diagram) it is possible to build up some consistency between the models, even at this early point in analysing the project. One of the problems with using use case diagrams very early in the project is that there is very little to relate them to. By relating the use cases to an associated block definition diagram, all diagrams become more consistent and the connection to reality gets stronger. Relationships between actors and use cases have been omitted from this model, as they will be discussed in more detail in due course.

So far, it has been possible to generate both the business context and the system context for a new project. The system context for the project is also traceable to the business context for the project. Note that both of these contexts are related back to

the business requirements of the organization, which is very important when it comes to establishing a business case for the new project.

7.4.3 Summary

The work covered so far in this chapter may be summarized by the SysML diagram in Figure 7.17.

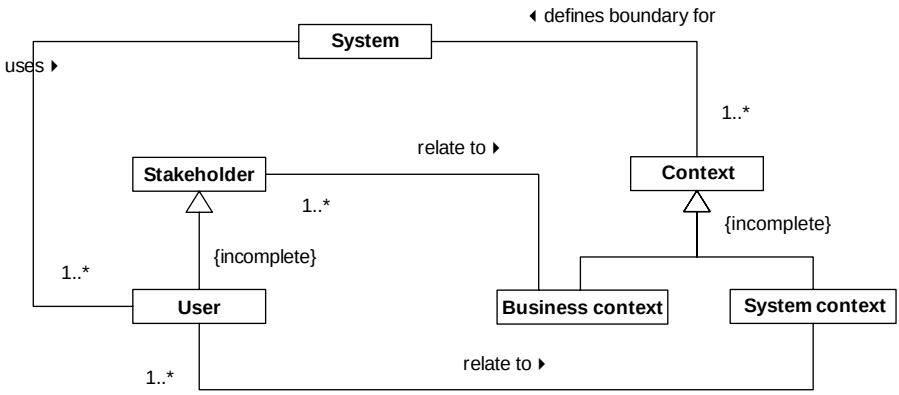

Figure 7.17 Relationship between context and system – meta-model

In summary, Figure 7.17 can be shown to state that one or more ‘Stakeholder’ relates to a ‘Business context’ and one or more ‘User’ relates to a ‘System context’. The ‘Context’ (of which there are two types) defines the boundaries of the ‘System’ and one or more ‘Stakeholder’ is identified for the ‘System’.

7.5 Requirements modelling

7.5.1 Introduction

Section 7.4 was concerned with modelling the business and system contexts of the system. These contexts actually represent requirements but at a very high level. For any project, the next step would be to look at these requirements in more detail and see if they can be broken down, or decomposed, into more detailed requirements. It should be pointed out that there is nothing to say that use case diagrams have to be used for requirements modelling. The SysML also provides the requirements diagram for this purpose.

Indeed, in the days before SysML and the UML on which it is based, when people used other techniques, such as Rumbaugh’s object modelling technique (OMT), class diagrams (which became block definition diagrams in SysML) were used to model requirements. In this way SysML has actually (re-)introduced a somewhat historical approach with the inclusion of the requirements diagram.

As with all aspects of the SysML, you should use whatever diagrams you feel the most comfortable with. It just so happens that use case diagrams were created in the UML specifically to model requirements and, therefore, tend to be the preferred technique despite the introduction of the requirements diagram into SysML. When working on projects involving software in which use case diagrams are used by the software team for modelling requirements, it is important that systems engineers use a similar technique in order to communicate with the software team in the same language. For this reason the authors advocate the use of use case diagrams for requirements modelling. In addition, this is the approach that was advocated by the old Objectory process, which has since evolved into the Rational Unified Process (RUP), a common software development process.

7.5.2 Modelling requirements

The starting point for modelling the requirements of a system is to take the system context and use the high-level requirements contained therein. This gives an indication of what the overall functionality of the system should be. The most natural thing to do is to take each requirement and to decompose it, or break it down into smaller or lower-level requirements.

Figure 7.18 shows the systems context as generated in the previous section, except that the relationships have now been added in. Types of relationship will be discussed

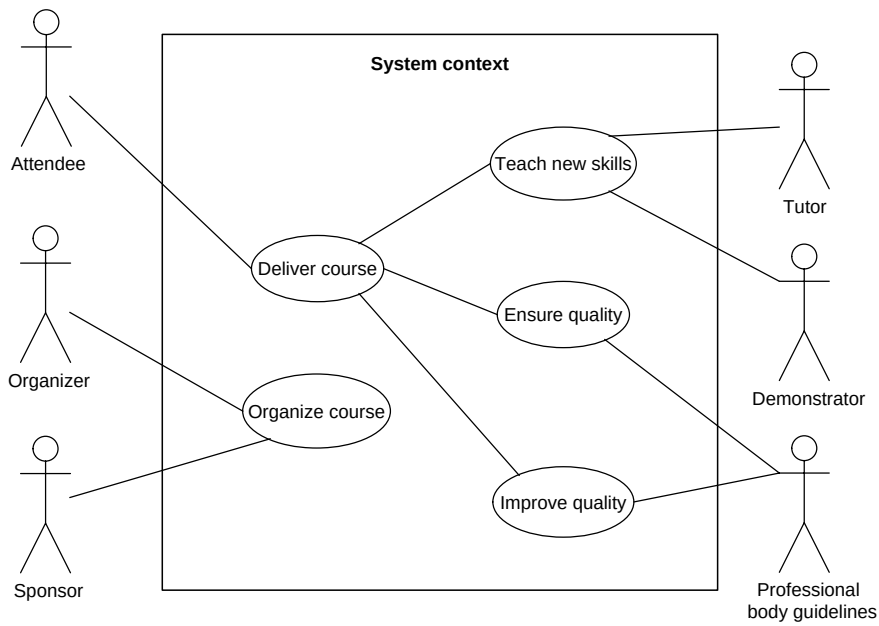

Figure 7.18 The system context with initial relationships

in some detail later in this chapter and this model will be revisited as part of the further discussion.

The first point that will be discussed, however, is that of decomposing high-level requirements into lower-level requirements.

The use case that will be taken and used for this example is that of 'organize course'. When thinking about how to decompose this requirement, it is very tempting to over-decompose. This can lead to all sorts of problems, as it is quite easy to go too far accidentally, and end up taking a data-flow diagram approach where the system is decomposed to such a level as to solve the problem, rather than state the requirements. This is compounded even further as use cases actually look a little like processes from data-flow diagrams. Indeed, data-flow diagrams were also used to define the context of a system, as are use case diagrams!

It is important, therefore, to know when to stop decomposing use cases. In order to understand when to stop, it is worth revisiting the definition of a use case, which states that a use case must have an observable effect on an actor. Therefore, consider each use case and then ask whether it actually has an effect on an associated actor. If it does not, it is probably not appropriate as a use case.

Another good way to assess whether the use case is at an appropriate level of functionality is to assess whether or not scenarios may apply directly to the use case. Remember again that a scenario is an instance of a use case, and thus look to see if any scenarios are directly applicable to that use case. If no clear scenarios are apparent, it may be that the use case is too high-level and needs to be decomposed somewhat. If there is only one scenario per use case, perhaps each one is too specific and the use case needs to be more generic. Scenarios will be discussed in more detail in due course.

Taking the example one step further, Figure 7.19 shows how the 'organize course' use case can be decomposed into lower-level use cases.

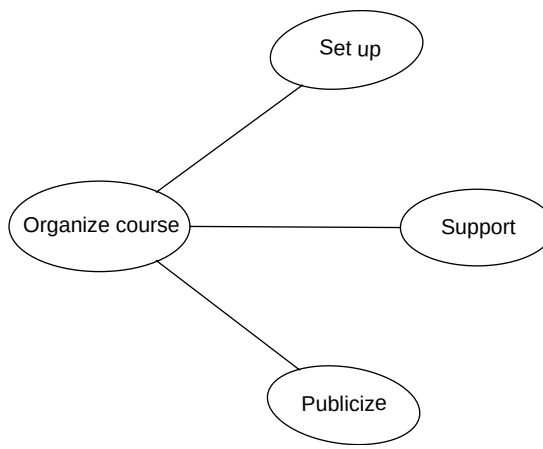

Figure 7.19 Simple decomposition of 'organize course' requirement

The model here is derived from the model in Figure 7.18, by selecting a single use case and decomposing its functionality. It can be seen now that the ‘organize course’ use case has been decomposed into three use cases: ‘set up’, ‘support’ and ‘publicize’.

When this model is considered, imagine that one of the team also comes up with a few new use cases that are also applicable to the decomposed model, but that have slightly different relationships compared with the ‘decomposed into’-style relationship that is shown in this model. The three new use cases are ‘cancel course’, ‘organize in-house course’ and ‘organize external course’, as shown in Figure 7.20, with their relationships to other use cases indicated by a simple associations.

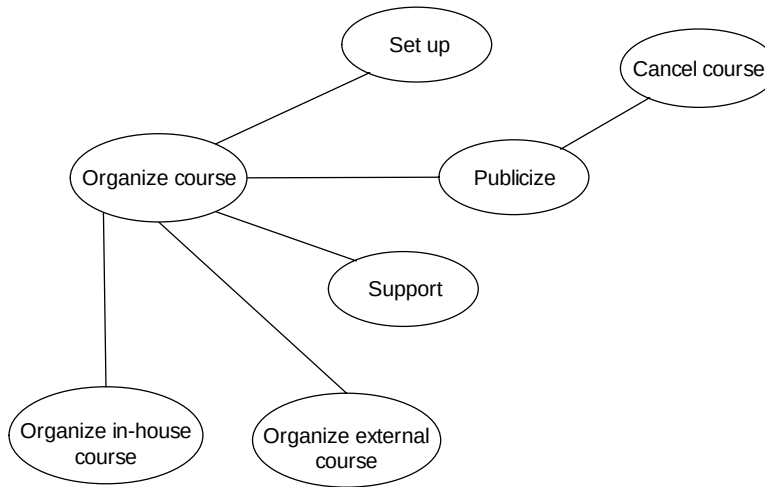

Figure 7.20 A more populated use case model

Figure 7.20 shows the updated use case model that has been populated with some new requirements. The new use cases are as follows.

- ‘Organize in-house course’ and ‘organize external course’: These represent two different ways in which the course may be organized, as sometimes the clients will come to the organization, while on other occasions the organization will go to the client. Although the courses will still be set up in similar ways, it is important to distinguish between the two types of course. These two requirements are directly associated with the original ‘organize course’ requirement.
- ‘Cancel course’: It is highly possible that the publicity will not be successful for the course and that too few people register, thus making the course economically unviable for the organization. In such an event, the course must be cancelled. This new requirement is related directly to the ‘publicize course’ use case.

Therefore, when added to the basic decomposition-style relationship, there are two more ways in which a relationship may be defined. Clearly, these are different

types of relationship and there is no way to differentiate between them, as seen in this model. This is all clearly leading somewhere and there just happen to be three types of relationship that are defined as part of the regular SysML (although two of these are inbuilt stereotypes). These types of relationship are shown in Figure 7.21.

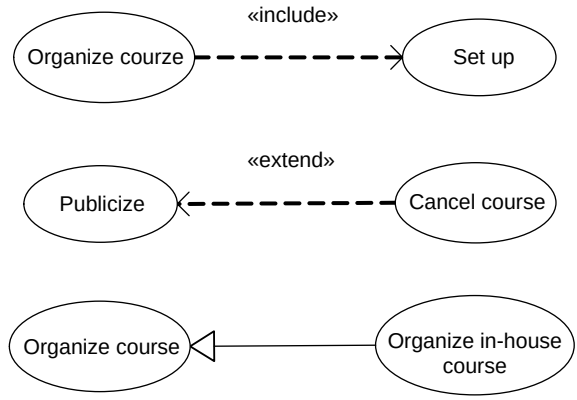

Figure 7.21 Types of relationship

Figure 7.21 illustrates the three basic types of relationship that may be used to add more detail to use case diagrams, which are part of the standard SysML.

The '≪include≫' association 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. The direction of the arrow should make sense when the model is read aloud. This part of the model is read as 'organize course' includes 'set up'.

The '≪extend≫' association 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 included use case as *sometimes* being part of the parent use case An example of this is when the 'cancel course' use case is implemented. Most of the time, or at least so the organization hopes, this use case is not required. However, there may be an eventuality where it is required. This is represented by the '≪extend≫' stereotype, but it is important to get the direction of the relationship correct, as it is different from the '≪include≫' direction. Always remember to read the diagram, and the direction of the relationship makes perfect sense. In addition, note that both '≪include≫' and '≪extend≫' are types of dependency rather than a standard association.

The final type of relationship is the generalization relationship, which is exactly the same as when used for block definition diagrams. In the example shown here, there

are two types of ‘organize course’, which have been defined as ‘organize in-house course’ and ‘organize external course’.

These new constructs may now be applied to the requirements diagram to add more value to the model, as shown in Figure 7.22.

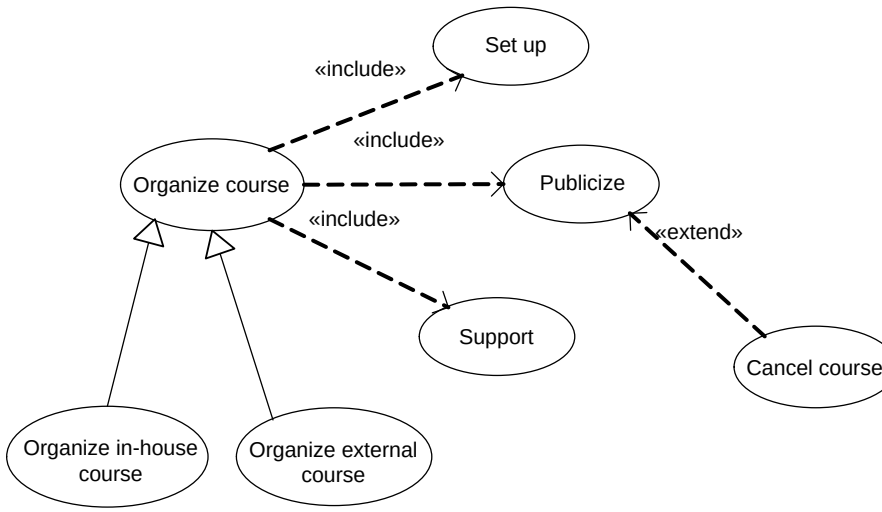

Figure 7.22 Complete use case model for ‘organize course’ use case

Figure 7.22 shows that the main use case ‘organize course’ has two types: ‘organize in-house course’ and ‘organize external course’. The ‘organize course’ use case includes three lower-level use cases: ‘set up’, ‘publicize’ and ‘support’. The ‘publicize’ use case may be extended by the ‘cancel course’ use case.

Because ‘organize in-house course’ and ‘organize external course’ are types of ‘organize course’, they too include the three use cases ‘set up’, ‘publicize’ and ‘support’ that are included by ‘organize course’.

There is nothing to stop new types of relationship being defined – indeed, there are mechanisms within the SysML that allow for this. However, let us consider a simple relationship that may be useful for requirements modelling and see how it may be applied to the example model.

It was stated previously that it is very important to distinguish between the different types of requirement that exist, whether they be ‘Functional’, ‘Non-functional’ or ‘Business’ requirements. However, it is unclear which of the requirements are which in some respects. We already know that anything on a business context will exclude the type ‘Functional’ and that any requirement in a system context will exclude ‘Business’ requirements, but what about ‘Non-functional’ requirements that may exist in both contexts?

In order to help to understand this, the original definition of what exactly a ‘Non-functional’ requirement is must be revisited. The original definition, according to

Figure 7.3, was that 'Non-functional' requirements constrain 'Functional' requirements. This will form the basis for the new type of relationship between use cases. SysML's stereotyping mechanism will be used to define a new SysML element. A full discussion of the definition and usage of stereotypes is beyond the scope of this book, but for the purposes of this example it is enough to assume that it is possible to define a new type of SysML element and then, whenever it is encountered, an assumption is made about its semantic meaning. This assumed meaning is conveyed using an 'assumption model', which is defined as part of a profile as shown in Figure 7.23. Such a diagram would be accompanied with a description of the intended meaning for the stereotypes.

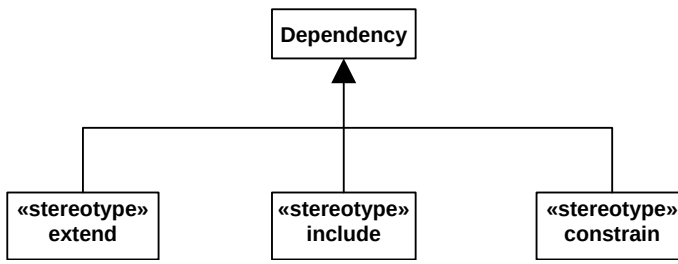

Figure 7.23 Defining new stereotypes for use case relationships

Figure 7.23 shows not only the existing stereotypes of relationships, but a new one called 'constrain'. Whenever the term '≪constrain≫' is encountered in a model, it is assumed that we are using a new type of SysML element whose definition must be explicitly recorded somewhere, ideally with the profile definition. This new type of relationship stereotypes the 'dependency' element from the SysML meta-model and is shown in Figure 7.24.

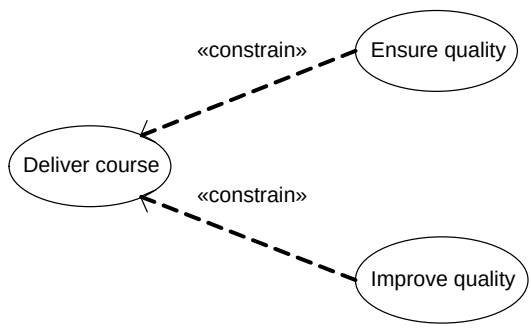

Figure 7.24 Making 'Non-functional' requirements explicit using the 'constrains' stereotype

Figure 7.24 shows how the new SysML element has been used to show that ‘improve quality’ and ‘ensure quality’ constrain the requirement ‘deliver course’. Based on the previous definitions in this chapter, it is now quite clear that both ‘improve quality’ and ‘ensure quality’ are ‘Non-functional’ requirements. That is, any use case connected to another with a ‘≪constrain≫’ dependency is a ‘Non-functional’ requirement.

7.5.3 Ensuring consistency

When developing use case diagrams common patterns are often seen that can guide the modeller in refinement of the use case diagrams. This section discusses some of these patterns.

7.5.3.1 Use case too high-level?

One common mistake is to model use cases at too high a level. Consider Figure 7.25.

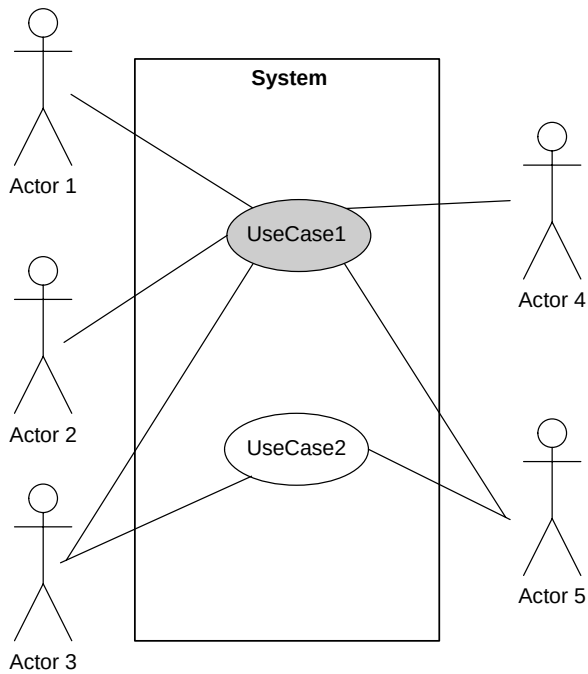

Figure 7.25 Use case too high-level?

Figure 7.25 shows a use case, ‘UseCase1’, which 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 the ≪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.

7.5.3.2 Actor too high-level?

Another common error is to model actors at too high a level. Consider Figure 7.26.

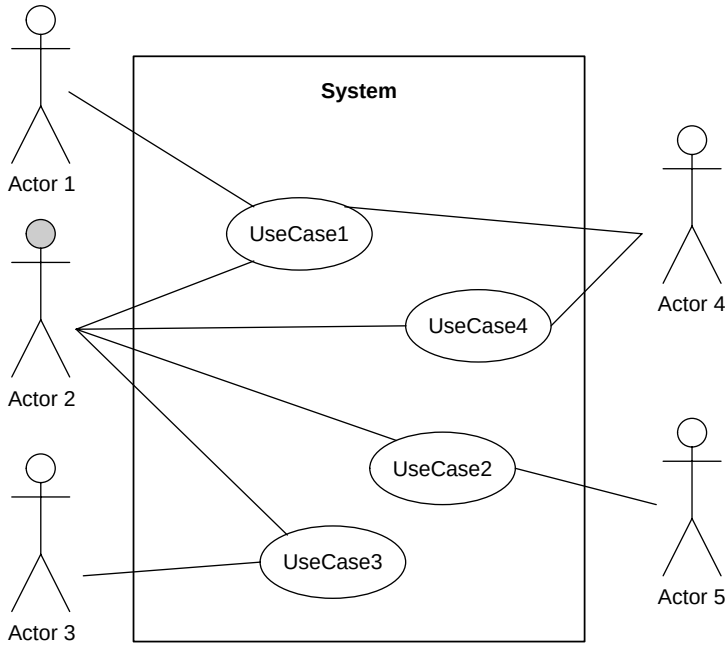

Figure 7.26 Actor too high-level?

Figure 7.26 shows an actor, 'Actor2', which 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 that actor.

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 that actor, then the actor can be removed from the diagram. See 7.4.1.3 above for a discussion of this.

7.5.3.3 Repeated actors?

Sometimes a pattern is seen in which two or more actors are connected to the *same* use cases. Figure 7.27 shows this.

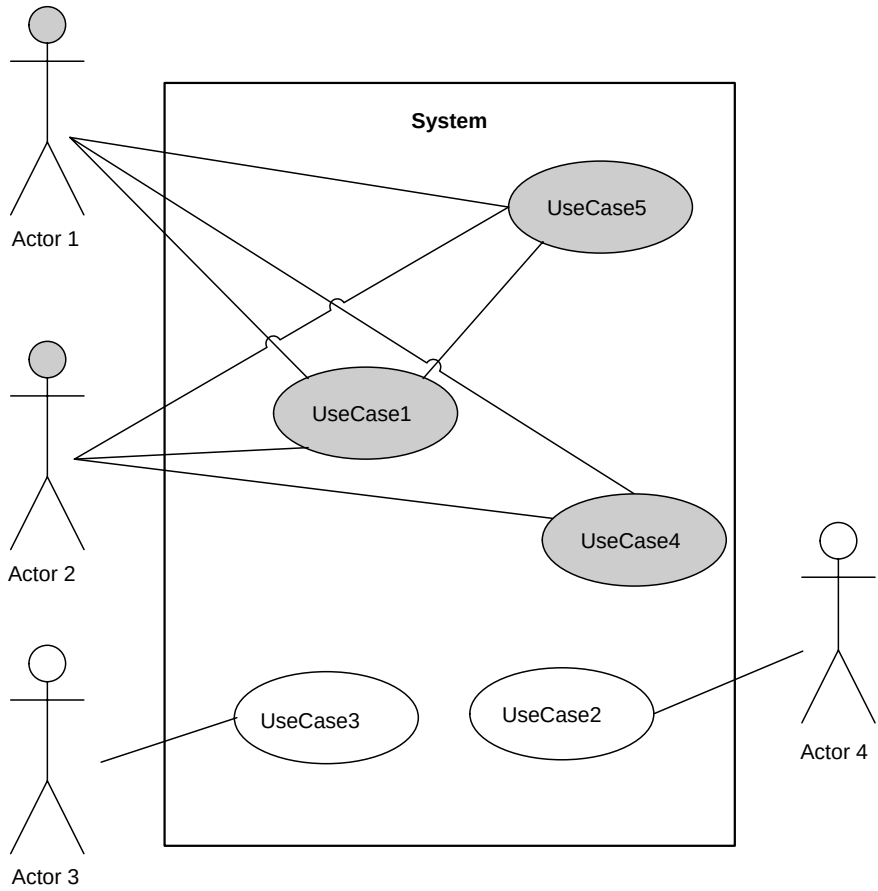

Figure 7.27 Repeated actors

Here we see two actors, 'Actor1' and 'Actor2', both connected to the same three use cases. This pattern may indicate that the actors are representing the same stakeholder. Alternatively, it may indicate that *instances* of stakeholders have been used (check for names of specific people, organizations, standards, etc.). Instances should *never* be used. Remember that a stakeholder 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.

7.5.3.4 Something missing? Use cases without actors and actors without use cases

What does it mean if we have use cases or actors that not related to anything? Consider Figure 7.28.

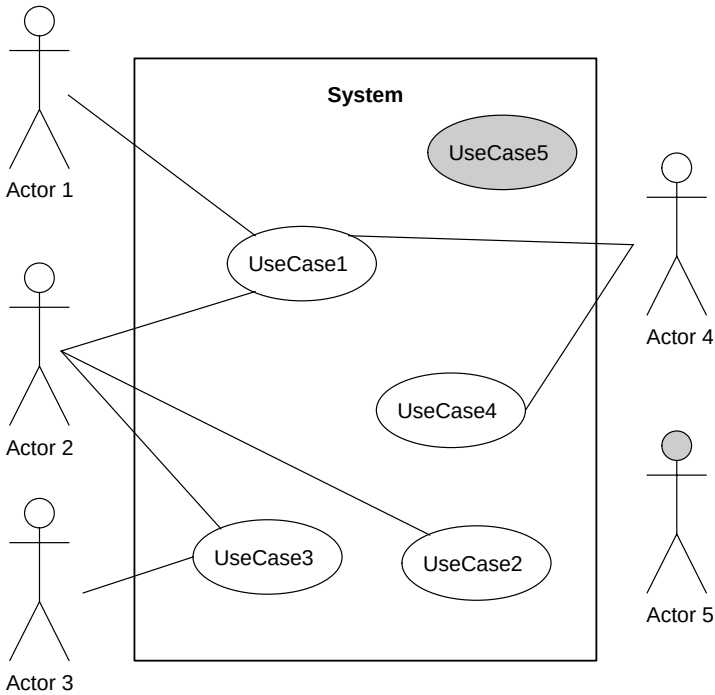

Figure 7.28 Something missing? Basic use case diagram checks

Figure 7.28 has a use case, 'UseCase5', and an actor, 'Actor5', that are not connected to anything else on the diagram.

'UseCase5' 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.

7.5.4 Describing use cases

7.5.4.1 Overview

One possible next step is to describe what happens within each use case. This may be done in one of two ways: by writing a text description or by modelling the use case visually.

A text description is exactly what it says it is. The text description should take the form of structured English text, much like pseudo-code descriptions from the software world. These descriptions should describe the typical sequence of events for the use case, including any extension points that may lead to unusual behaviour (relating to the '≪extend≫' relationships). The text should be kept as simple as possible with each describing a single aspect of functionality. Many textbooks advocate the use of text descriptions, which is fine, but it may be argued that this is going slightly against the SysML ethos, which is all about visualization.

For people who prefer visualization, there are, of course, SysML diagrams that allow the internal behaviour of a use case to be described. The most common SysML diagram to use for this is the state machine diagram, although activity diagrams can also be used.

However, this can lead to a certain amount of confusion when it comes to scenarios (described in more detail later in this chapter), as scenarios show an instance of a use case and they are modelled using sequence diagrams.

A single use case can be described by a state machine to show its internal operation. An example of a use case (an instance) is known as a *scenario*, which is modelled using a sequence diagram that describes the messages passed between the system and its actors. Thus we can conceptualize slightly different views that the different diagrams represent.

- The sequence diagram describes a scenario that focuses on the interactions between life lines by defining the messages that are passed between them. The key to scenarios is to show interactions between the system and anything that interacts with it. In SysML terms, this will equate to the system as defined by the system boundary and its external actors.
- The state machine describes what goes on inside a use case and, hence, within the system itself.

We shall now look at examples of describing a use case using text and state machine diagrams.

7.5.4.2 Text descriptions of use cases

The first approach to describe the internal operation of a use case is to write a simple text description of the sequence of steps that must be carried out in order for the use case to function. The example in Figure 7.29 shows a simple text description for the use case 'publicize'.

5.3.3.1 Publicize:	*Receive suggested dates for course* *Send out fliers for the course* *Wait for a predetermined length of time* *When time is up, assess the response* *If the response is sufficient, send details to attendees* *If response is insufficient, cancel the course*

Figure 7.29 Use case description using text description

It should be quite obvious to anyone looking at this description how the use case is intended to function. The advantage to using text descriptions is that they are very simple and, when written with care, are easy to understand. On the downside, however, a slightly more complex use case may be very difficult to represent in text terms. In addition, the formal tie-in to the rest of the SysML model is almost nonexistent, whereas a visual approach would have stronger ties.

The next way to describe a use case is to model it using a state machine diagram.

7.5.4.3 State machine description of a use case

The second approach to describe the internal operation of a use case is to model visually using a state machine or activity diagram. The model in Figure 7.30 shows a

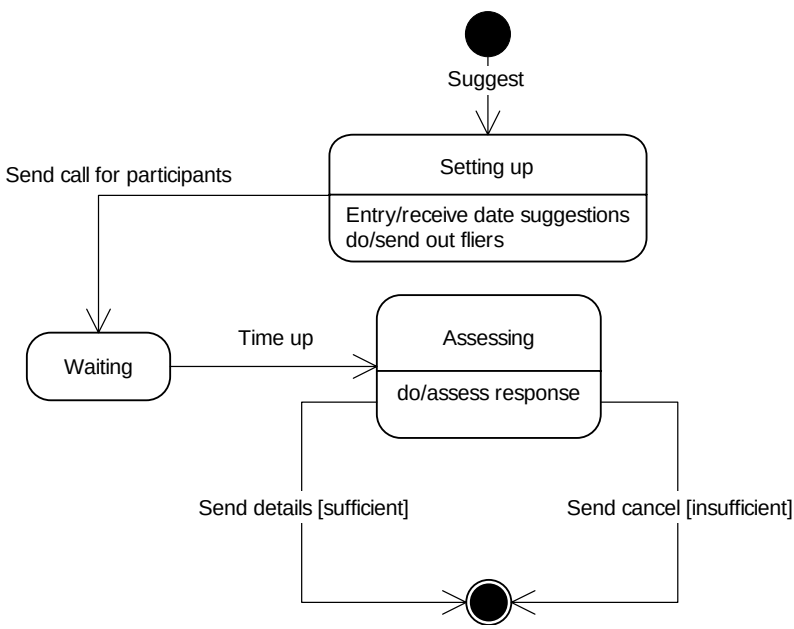

Figure 7.30 Use case description using a state machine

state machine representation of the 'publicize' use case that is equivalent to the text description defined in Figure 7.29.

Figure 7.30 shows a visual description for the 'publicize' use case. One thing that is much clearer using the visual approach is that loops and iterations are far easier to spot. These can often be obscured, lost or misconstrued using an informal text description, but are immediately apparent when they can actually be seen. From a SysML point of view, note that 'receive date suggestions' is an action whereas 'send fliers' is an activity. The keyword 'entry' applies to an action that takes zero time and is non-interruptible and so is correct for receiving a date. The keyword 'do', however, applies to an activity that indicates that the 'send fliers' will take time and may be interrupted.

7.5.4.4 The better approach

The immediate question that most people ask at this point is, 'Which is the better of the two approaches?' This can be answered with the standard SysML response of, 'Whichever is the more suitable for the application at hand.'

Text descriptions are often preferred, because they are easier to create and, it may be argued, can be very simple for anyone to understand. They also have the advantage of being in a text form, which will fit directly into a user requirements document, for example. However, anything written in plain English is more prone to error and misinterpretation, simply because any spoken language is so expressive and almost all words have more than one meaning.

A visual description of a use case may be preferred for someone wanting a more formal approach to describing use cases. In addition, it is easier to reference one model to another model than it is to relate a text description to another part of the model.

Some people may not even want to describe the use cases in any way, as perhaps they do not have enough information about the requirements. One way to derive more information about requirements is to consider a scenario that interacts with the outside world, from a single aspect of functionality. Defining scenarios will be covered in more detail below.

7.6 Modelling scenarios

Once a set of stakeholder requirements has been generated and they have been organized and analysed using use case diagrams, it is possible to go one step further and look at 'scenarios'. A scenario shows a particular aspect of a system's operational functionality with a specific goal in mind. The definition of a scenario from the SysML point of view is that it is an instance of a use case. This then gives good, strong traceability back to the original requirements.

Scenarios are often generated as part of the conception stage during an analysis process. This is not always the case, as it will depend upon the process that is being followed – indeed, it is not unheard of to have a set of scenarios defined during the requirements process. However, as we are following the STUMPI process introduced in Chapter 6, it is assumed that we have now satisfied all the criteria to exit the requirements process and have moved into the analysis process.

Scenarios have many uses and can be a crucial part of the analysis of the stakeholder requirements and will help to identify the intended functionality of the system. This has a number of practical uses.

- It is important to identify how the user will use and interact with the system. Bear in mind that the actual design will take its starting point as the system specification, and thus it is important to understand how the system will be used.
- Scenarios are a good source for system and acceptance tests for a system. A scenario represents a particular aspect of a system's functionality and models the interaction between the system and the outside world, and thus they are an excellent source for defining acceptance tests.

When scenarios have been created as part of the requirements or analysis processes, they will form the basis for the actual system design. Therefore, as they are traceable back to individual use cases and forward to the design, they enable a traceability path to be set up from the design right back to the requirements.

7.6.1 Scenarios in the SysML

A scenario models how the system interacts with the outside world. The outside world, in the case of our system, is actually a subset of the stakeholders that have already been defined during the requirements phase and exist both on the stakeholder model as blocks and also on the business and system contexts as actors. Potential paths for communication between a stakeholder and a stakeholder requirement are clearly indicated on the SysML diagrams as associations that cross the system boundary. Each time the system boundary is crossed, this indicates an interface between the stakeholder and the system.

The diagram that is used in the SysML to realize scenarios is the sequence diagram that shows the interactions between life lines from a logical timing point of view.

When considering scenarios, we think about actual instances in the system, rather than blocks, as we are looking at a real example of operation. This gives a strong link to block definition diagrams, as all life lines on a sequence diagram must have an associated block. This is one approach to identifying blocks for the analysis and design of the system.

7.6.2 Example scenarios

The first step when defining scenarios is to choose a use case from the original requirements model, which, for the purposes of this example, is shown in Figure 7.22. The use case that will be chosen for the initial example is 'publicize', which describes a requirement for the system to help the organizer publicize a training course. This is the use case that was described previously using both a state machine and a text description.

With the use case described, it is now time to look at some possible scenarios from that use case. Two spring immediately to mind: one where publicity is successful and the course goes ahead and one where the publicity is unsuccessful and the course does not go ahead.

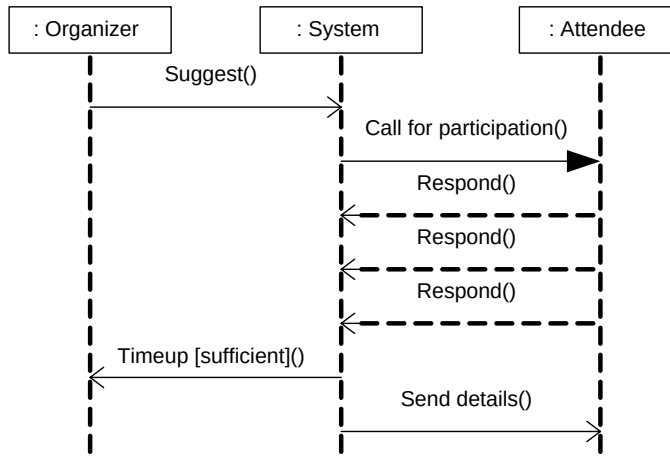

Figure 7.31 Scenario #1 – successful publicity, sequence diagram

Figure 7.31 shows a sequence diagram that represents a scenario where the publicity is successful.

The first thing to notice about the diagram is that the main components are life lines that are real-life instances of blocks. Sometimes, these life lines may be shown using the actor symbol if they represent stakeholders from the system context. It is usual to show the system as a single life line that will allow us to identify the interactions that must occur with the outside world in order for the system to meet its original requirements.

Remember that the life lines, represented graphically by the vertical dotted line, show logical time going down the page. Therefore, as time goes on, we would expect certain interactions to occur. These will already have been defined to a certain extent, as the use case will have been described, perhaps using text or a state machine as described in Section 7.5.4.

The first thing to happen is that the 'Organizer' suggests some details to the 'System'. These details may include the suggested time, date and location. The 'System' then sends out a call for participation to a number of 'Attendee'. In this scenario, a number of attendees respond to the call for participation by sending messages back to the 'System'. Eventually, the time allowed for people to respond expires and the system has to make a decision about whether to progress with the course or not. In this scenario, there is sufficient demand for the course and thus the system sends out details of the course to the attendees who have applied to attend the course. That is the end of that particular scenario.

Now consider a different scenario, one where the publicity is unsuccessful due to an insufficient number of responses. Although not frequent, this is certainly a scenario that must be taken into account when thinking about the system.

Figure 7.32 shows the second scenario. Note how the sequence of messages begins in exactly the same way as before, right up to the point when the call for participation

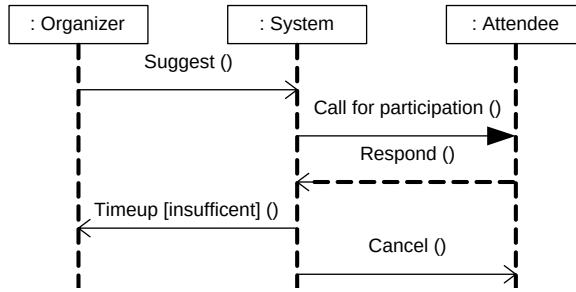

Figure 7.32 Scenario #2 – unsuccessful publicity, sequence diagram

is broadcast. The differences occur because there are not many responses this time – in fact, there is only a single response in this scenario. The same 'timeup' event occurs, but in this scenario the condition has changed. Rather than there being a 'sufficient' number of responses, there is an 'insufficient' number. The consequence of this is that the course must be cancelled, hence the 'cancel' message is sent out to any 'Attendee' who may have responded to the original call for participation.

The scenarios considered so far have been deliberately simple. It is often a good idea to start off with simple scenarios that are well defined before complex scenarios are considered.

Figure 7.33 shows a new scenario that exhibits far more complexity than the previous two scenarios. This scenario relates to the 'set up' use case and describes the

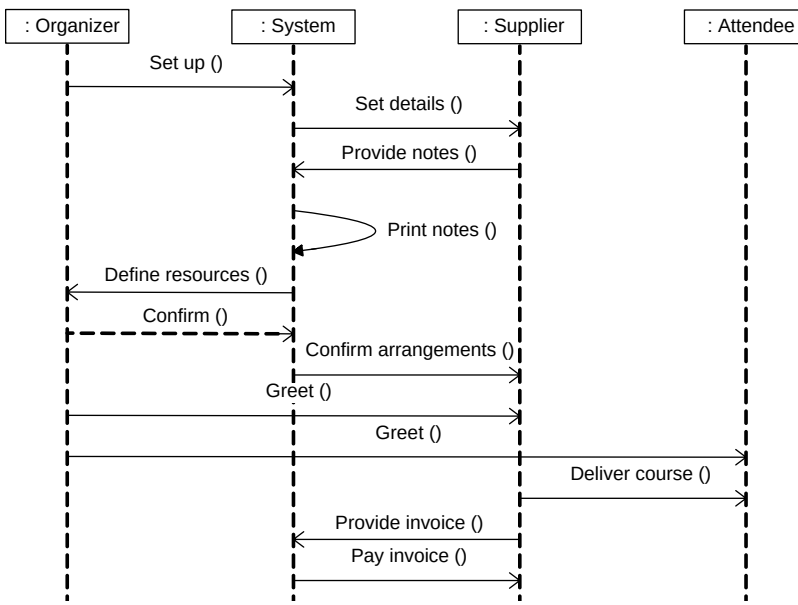

Figure 7.33 A more complex scenario

normal sequence of operation. Note that this sequence diagram contains a message that is passed from, and then back to, a single life line. Although these are not strictly necessary, they sometimes help the person reading the model to understand more about what is going on. The message in question here is the 'print notes' message. Although this message does not describe an interaction between life lines, it was deemed as being useful to include it to make explicit the fact that there is part of the scenario where printing takes place.

7.6.3 Wrapping up scenarios

Let us summarize.

- Scenarios are defined as instances of use cases and, as use cases represent stakeholder requirements, the scenarios relate directly back to the original requirements for the system.
- Scenarios are realized by sequence diagrams that stress the logical timing relationships between life lines.
- Several scenarios may be modelled for each use case. In fact, enough scenarios would be modelled in order to show a good representation of the overall system operation.

One question that is often asked about scenarios is, 'How many should be produced?' As with most of the questions that people tend to ask most often, there is a simple answer, but it is one that most people do not seem too happy with. The simple answer is 'just enough', which is very difficult to quantify. It is important to consider enough scenarios so that you have covered all typical operations of the system and several atypical operations. A good analogy here is to think about testing a system. It is impossible to test fully any complex system, but this does not mean that there is no point in testing it. In fact, it is important to test the system to establish a level of confidence on the final system. Indeed, this analogy is appropriate in another way, as scenarios form a very good basis for acceptance and system testing for just this reason.

7.7 Documenting requirements

7.7.1 Overview

This section looks at the practical issue of documenting requirements. The way in which requirements are documented will depend upon the process that is being adopted by the organization.

Some processes do not rely on documentation as such – for example, the RUP. In the RUP, all of the models that are created and in many cases some of the components that make up the models, are stored as artefacts. These artefacts are roughly equivalent to the deliverables in a more traditional approach and form the core of the project repository. In such an approach, each use case is blocked as an artefact and the models themselves are also treated as artefacts. Therefore, when it comes to documenting

such use cases or models, all information is stored in the model, which can then be automatically drawn out into a report as and when necessary. This approach relies heavily on tools that are compatible with the approach and that are intended to make life simple and such documentation a matter of pressing a single key, rather than assembling documents by hand. There are advantages and disadvantages with such an approach, but these are beyond the scope of this book.

For the sake of consistency, the example that will be used here will follow on from the ISO-type process that has been defined previously.

The actual structure of the requirements documentation that will be used for this example will be based on a model of the user requirements specification (URS) that was derived from existing standards and best practices using the approach introduced in Chapter 6. Let us suppose that, in order to define the contents of a URS, a source of information was looked for. In this case, the content of the URS will be based on the one defined by the European Space Agency [7]. This was modelled according to the guidelines laid out in Chapter 6 and resulted in the model shown in Figure 7.34.

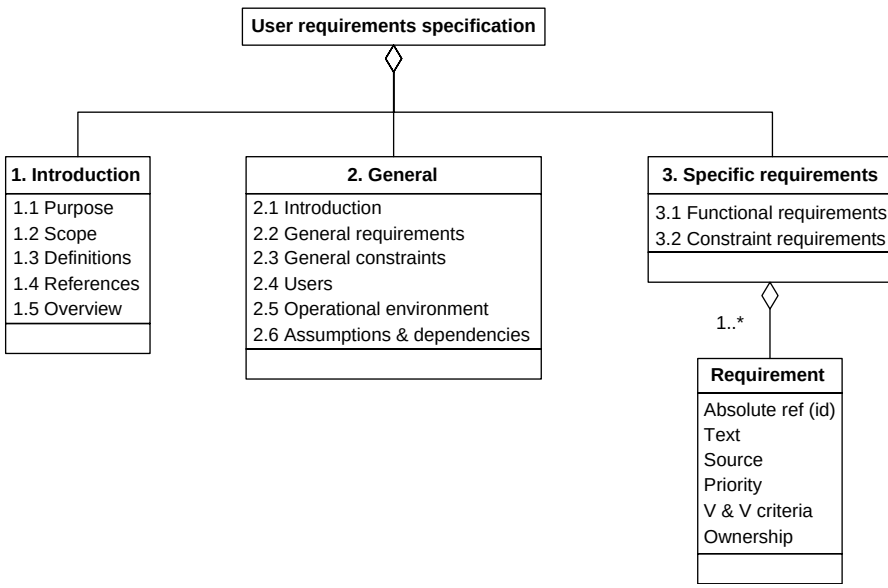

Figure 7.34 Typical contents of a user requirements document

Figure 7.34 shows the structure for the URS, as derived from the process modelling exercise that has been discussed previously. The model shows the various sections of the document as blocks, and the various subheadings are shown as properties on each block. Of course, depending on how many levels of nested heading exist in the document, there would be an associated hierarchy of blocks in the document model.

It can be seen that the 'User requirements specification' is made up of '1. Introduction', '2. General' and '3. Specific requirements'. '1. Introduction' has the subsections '1.1 Purpose', '1.2 Scope', '1.3 Definitions', '1.4 References' and '1.5 Overview'. '2. General' has six subsections and '3. Specific requirements' has two subsections. In addition, '3. Specific requirements' is made up of one or more 'Requirement'. These requirements are described according to the model defined in Figure 7.5, which is actually included in this model.

This document may then be populated by making use of all the models that have been created so far, such as the contexts, stakeholder models, requirements models and scenarios.

7.7.2 Populating the document

Each part of the document that has been identified in Figure 7.34 will now be discussed and appropriate diagrams will be suggested for each section.

7.7.2.1 Section 1 – introduction

The introductory section is mainly concerned with setting the scene for the project and perhaps justifying why it is being carried out.

- The business context and parts of the system context may be used to illustrate the first two subsections '1.1 Purpose' and '1.2 Scope'.
- Definitions in '1.3 Definitions' may be defined according to the terms used as SysML elements.
- The information in '1.4 References' must be generated manually.
- Finally, '1.5 Overview' may again make use of the parts of the business context and system context information.

As can be seen, most of the information for the first part of the document can be derived directly from the SysML models that have been created so far.

7.7.2.2 Section 2 – general

The second section of the document may be populated as follows.

- Section '2.2 General requirements' and section '2.3 General constraints' may be obtained from high-level use cases. This will include the use case models for the high-level requirements and the business context use case model. This is because, as stated previously, many stakeholder requirements and constraints will be derived from business requirements.
- Section '2.4 Users' may be defined directly from the user models and stakeholder models, which were represented previously using block definition diagrams.
- Section '2.5 Operational environment' may be described by the information in the system context model, which was realized by using a use case diagram.

Again, the SysML diagrams can be used to populate most of this section of the document.

7.7.2.3 Section 3 – requirements

The third section of the document may be populated as follows.

- Section '3.1 Functional requirements' and section '3.2 Constraint requirements' will form the bulk of the document. This information may be extracted from low-level requirement use case diagrams, requirements descriptions that were realized using text descriptions and state machines, and, finally, scenarios. Scenarios were realized by interaction diagrams: sequence diagrams and communication diagrams.

The third section can be populated almost entirely using the SysML models. This is not surprising, as the idea behind this chapter was to model requirements and this section is concerned purely with requirements.

7.7.3 Finishing the document

In order to finish off the document, the following points should be borne in mind.

- Include the actual models, where appropriate, in the document itself. This is not always possible but can prove to be very useful. As an interesting aside, non-SysML experts will be able to understand most simple SysML models with a minimum of explanation, provided that the readers do not realize that they are SysML diagrams. This may sound strange, but many people will mentally 'switch off' if they think that the diagrams are in a language that they do not understand and will make no attempt to understand them.
- Add text to help the reader understand the diagrams. By basing the text directly on the diagrams, there is less room for ambiguity through misinterpretation and it also ensures that the terms used in the document and the models are consistent. It is important to remember that the knowledge concerning the project exists in the models, rather than the documents. Think of the documents as a window into the model, not the other way around. Always remember to change the model first, rather than the document, to retain consistent project knowledge. Many SysML CASE tools allow documentation to be generated directly from the model.
- Structure the actual requirement descriptions according to the structure of the requirements that was shown in the original requirement model, showing properties associated with each requirement. Again, this leaves less room for ambiguity and, if the modelling has been performed well, should be a good way to structure things anyway.
- The document will then drive the rest of the project and be a formal view of the models. Any changes that are to be made to the system should be made in the model and then reflected in the user requirements document.

Above all, remember to make as much use as possible of the SysML models. Every model will have been created for a reason, so bear them all in mind when writing the document. Modelling should be an essential, useful and time-saving part of the project, not just an academic exercise to produce nice pictures.

7.8 Summary and conclusions

In summary, therefore, this chapter has introduced and discussed the points covered by the following model.

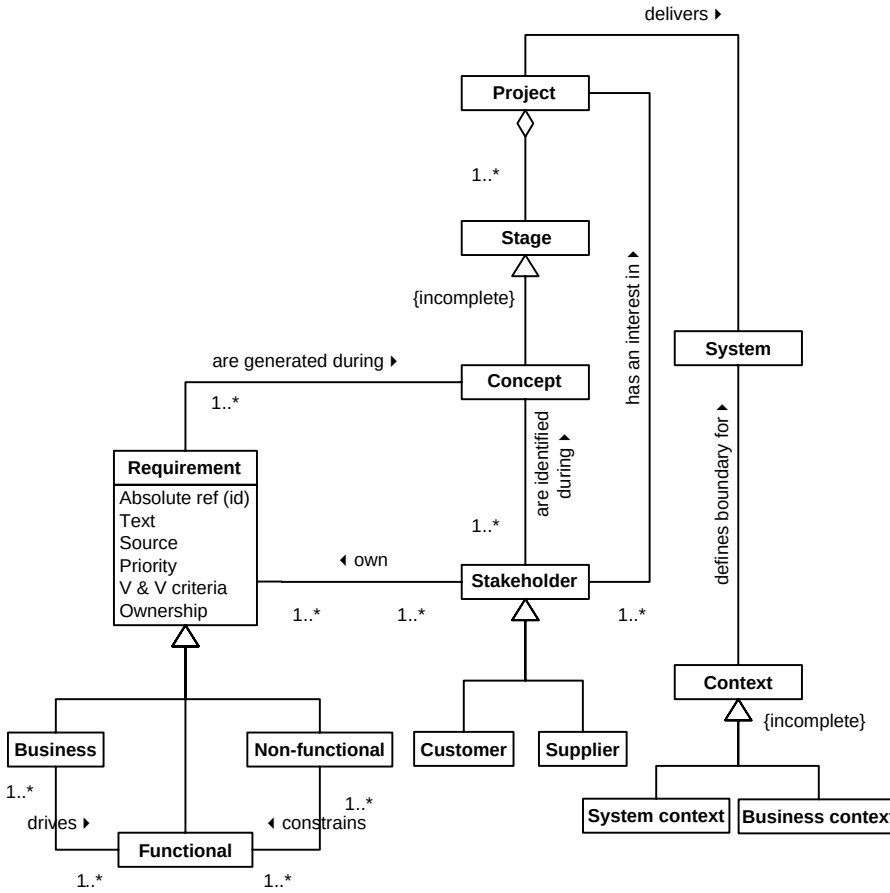

Figure 7.35 Concepts introduced in this chapter – meta-model

Figure 7.35 shows the following points.

- Each ‘Project’ is made up of a number of one or more ‘Stage’. One type of ‘Stage’ is ‘Concept’. No other stages were shown at this point, as it was beyond the scope of the chapter.
- One or more ‘Requirement’ is generated during the ‘Concept’ stage and each ‘Requirement’ must be one of three types: ‘Business’, ‘Functional’ or ‘Non-functional’.

- One or more ‘Business’ requirement drives one or more ‘Functional’ requirement, and one or more ‘Non-functional’ requirement constrains one or more ‘Functional’ requirement.
- One or more ‘Stakeholder’ is identified during the ‘Concept’ stage and each owns one or more ‘Requirement’. There were two main types of ‘Stakeholder’ identified: the ‘Customer’ and the ‘Supplier’.
- There are many different types of ‘Context’. Two are shown: the ‘Business context’ and the ‘System context’. The types of ‘Context’ define the boundaries for the ‘System’, which is delivered by the ‘Project’.

Figure 7.35 summarizes the essential concepts of requirements modelling, but this chapter related all these concepts to SysML modelling, which may be summarized by the following diagram.

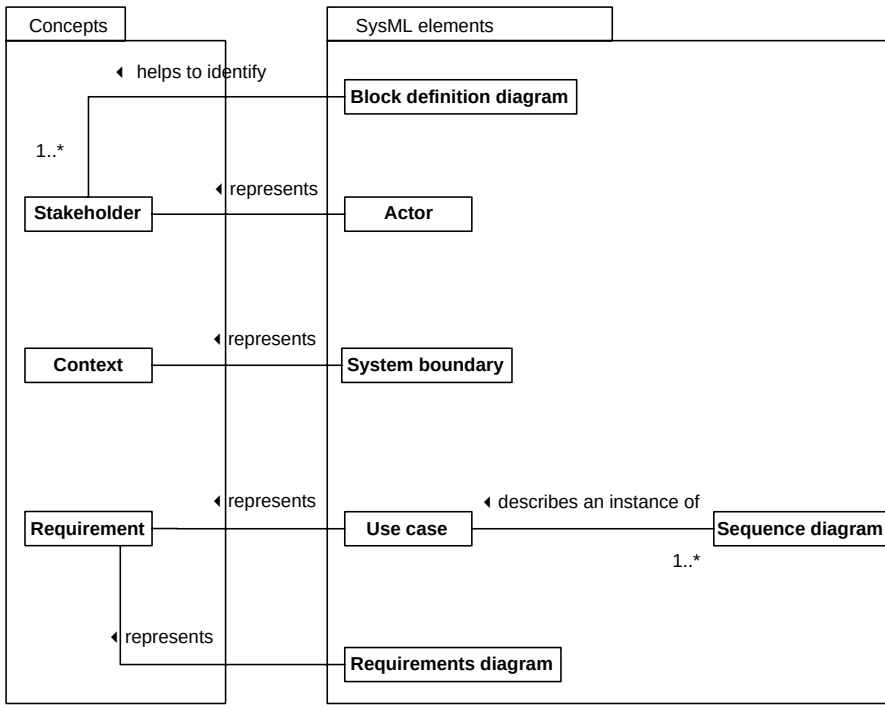

Figure 7.36 Summary of relationships between concepts and SysML elements

The relationships between requirements engineering concepts and SysML elements are summarized in Figure 7.36. From the diagram, we can see the following.

- Block definition diagrams were used to create a stakeholder model, which was useful when trying to identify stakeholders, which is an essential part of the concepts phase.

- Each stakeholder, once identified, could be represented as a SysML actor on a use case diagram. This becomes particularly important when it comes to modelling contexts for a particular project.
- The context for the project is indicated by using a SysML system boundary element. This element is underused by most modellers.
- Use cases were used to represent the three different types of requirement. Each use case was used to represent either a single requirement or a grouping of like requirements, which could then be decomposed into further requirements.
- Sequence diagrams were used to model specific scenarios, in order to understand the operation of the system. Each scenario was an instance of a use case that gave a good, strong traceability path back to the original requirements model.
- Use cases and thus requirements could be further described using a text description or a state machine or activity diagram. It is important that these be differentiated from interaction diagrams that show scenarios, or instances, of a use case.
- Requirements diagrams could also be used, unsurprisingly, to represent requirements. However, the lack of context and stakeholders raises questions on the usefulness of this diagram. Possible uses of the requirements diagram are to formalize a textual description of a use case or to decompose use cases further.

The best way to learn these concepts in more detail, like all aspects of the SysML, is to take an example and model it for yourself. Section 7.9 gives some ideas about how these ideas may be taken further.

7.9 Further discussion

1. Consider a project that you are familiar with and try to identify a stakeholder model, using a block definition diagram. Does the generic model shown in Figure 7.7 work for your example, or does it need to be changed considerably? In either case, why is this? Which context will each stakeholder appear in: the system context, the business context, or both?
2. Take another look at the model shown in Figure 7.18 and select a different requirement. Try to decompose this requirement into lower-level requirements so that it still makes sense. Also, bear in mind that whatever requirements are generated must be kept in line with the original business requirements that were shown in Figure 7.4.
3. Look at the requirements shown in Figure 7.22 and select a different use case from the one chosen for the example. This time, try to create some scenarios that would represent typical system operation.
4. Choose any requirement that has been discussed in this chapter (whether functional, non-functional or business requirement) and document it as if it were part of a user requirements specification. Which part of the document would it be most applicable to?
5. Model the business requirements for another organization (maybe your own) and see if they make sense to someone else. Choose a few existing projects or other

pieces of work and see if they fit in with your business requirements model. Is the organization's mission statement included as part of this model? Is it possible to demonstrate to a third party that the organization is meeting its mission statement, or not?

6. Consider the system context shown in Figure 7.18. How would this be different if it were the system context from the customer's point of view, rather than the supplier's? Are there any new requirements that need to be added, or any existing ones that need to go? Are the stakeholders the same for both contexts and, if not, then why not? Does the new system context still meet the business requirements in the business context? Does it need to meet these business requirements?
7. Take one of the business requirements shown in the business context in Figure 7.4 and try to think of the type of project that may be associated with some of the other business requirements. Can you come up with a quick system context for such a model? Which stakeholders, if any, are relevant to the new system context and the business context? Are there any new stakeholders to be identified?
8. Complete the model in Figure 7.16 using the three basic and one stereotyped relationship introduced in this chapter.

7.10 References

1 Jackson M. *Software Requirements and Specifications*. Boston, MA: Addison-Wesley; 1995
2 Davis A.M. *Software Requirements: Analysis and Specification*. Prentice Hall International; 1990
3 Schach S.R. *Software Engineering with Java*. Maidenhead: McGraw-Hill International Editions; 1997
4 Stevens R., Brook P., Jackson K. and Arnold S. *Systems Engineering: Coping with Complexity*. Harlow: Pearson Education; 1998
5 O'Connor J. and McDermott I. *The Art of Systems Thinking: Essential Skills for Creativity and Problem Solving*. London: Thorsons; 1997
6 Skidmore S. *Introducing Systems Design*. 2nd edn. Oxford: Blackwell Publishing, 1996
7 Mazza C., Fairclough J., Melton B., De Pablo D., Scheffer A. and Stevens R. *Software Engineering Standards*. Hemel Hempstead: Prentice Hall Europe; 1994

7.11 Further reading

Jacobson I. *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Boston, MA: Addison-Wesley; 1995

Jacobson I., Booch G. and Rumbaugh, J. *The Unified Software Development Process*. Boston, MA: Addison-Wesley; 1999

Schneider G. and Winters J.P. *Applying Use Cases: A Practical Guide*. Boston, MA: Addison-Wesley; 1998

Appendix A
Summary of SysML notation

> 'By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race.'
>
> Alfred North Whitehead (1861–1947), *An Introduction to Mathematics*, 1911

A.1 Introduction

This appendix provides a summary of the meta-model and notation diagrams for SysML that are used in Chapter 4. For each of the nine SysML diagram types, grouped into structural and behavioural diagrams, three diagrams are given:

- a partial meta-model for that diagram type;
- the notation used on that diagram type; and
- an example of that diagram type.

The same information is also given for the SysML auxiliary constructs. The appendix concludes with two diagrams that illustrate some of the main relationships between the SysML diagram, which also show how the concepts involved in parametric definition and usage are related.

This appendix does *not* add further information to that found in Chapter 4, but is intended to provide a single summary section of the noted diagrams. See Chapter 4 for a discussion of each diagram.

A.2 Structural diagrams

This section contains diagrams for each of the five SysML *structural* diagrams:

- block definition diagrams (Figures A.2–A.4)
- internal block diagrams (Figures A.5–A.7)
- package diagrams (Figures A.8–A.10)
- parametric diagrams (Figures A.11–A.14)
- requirement diagrams (Figures A.15–A.17).

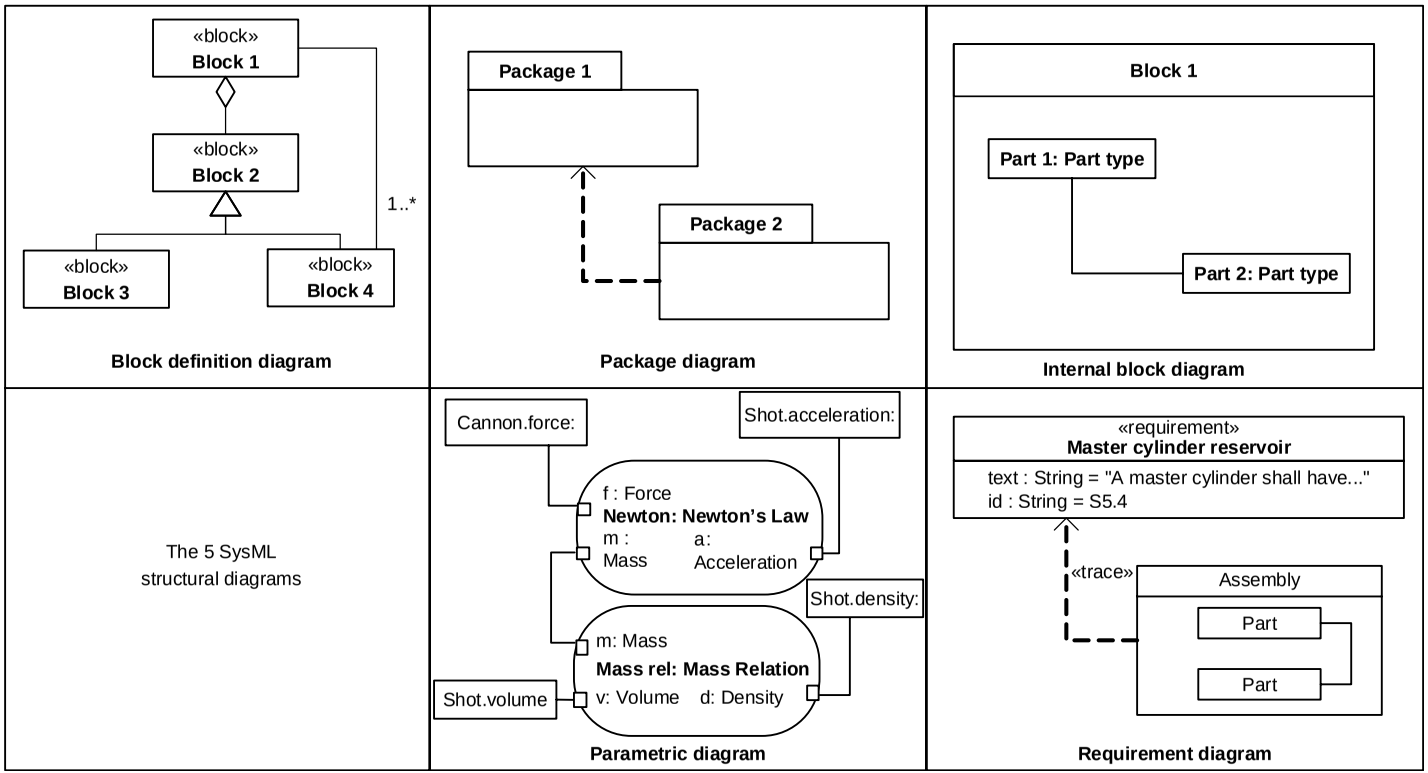

Figure A.1 Summary of structural diagrams

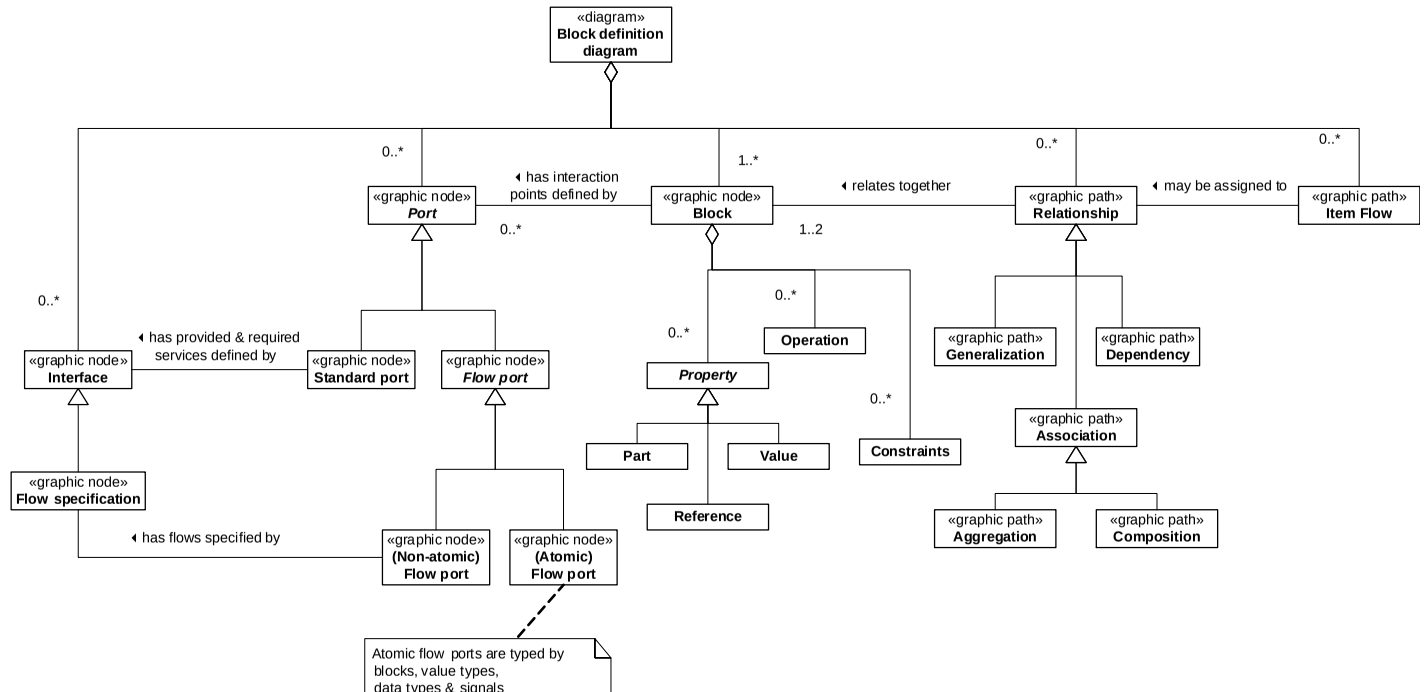

Figure A.2 Partial meta-model for block definition diagrams

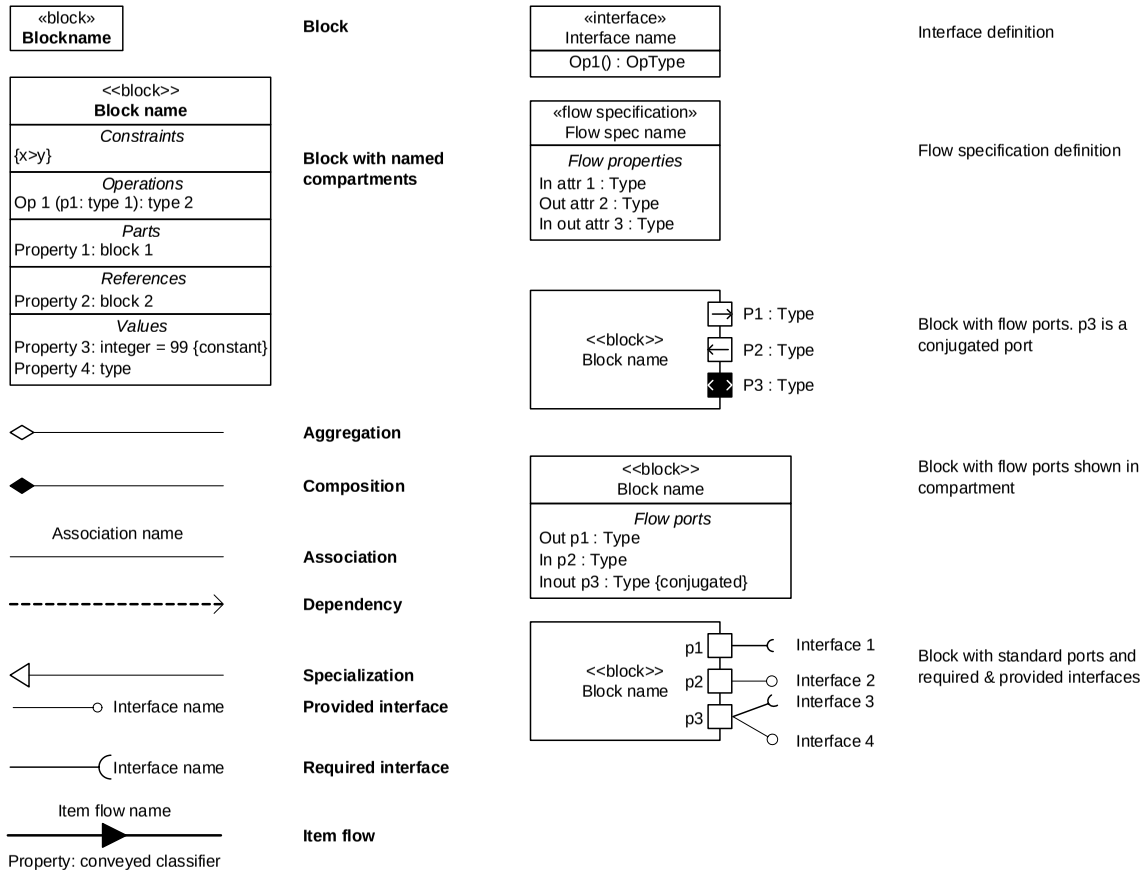

Figure A.3 Block definition diagram notation

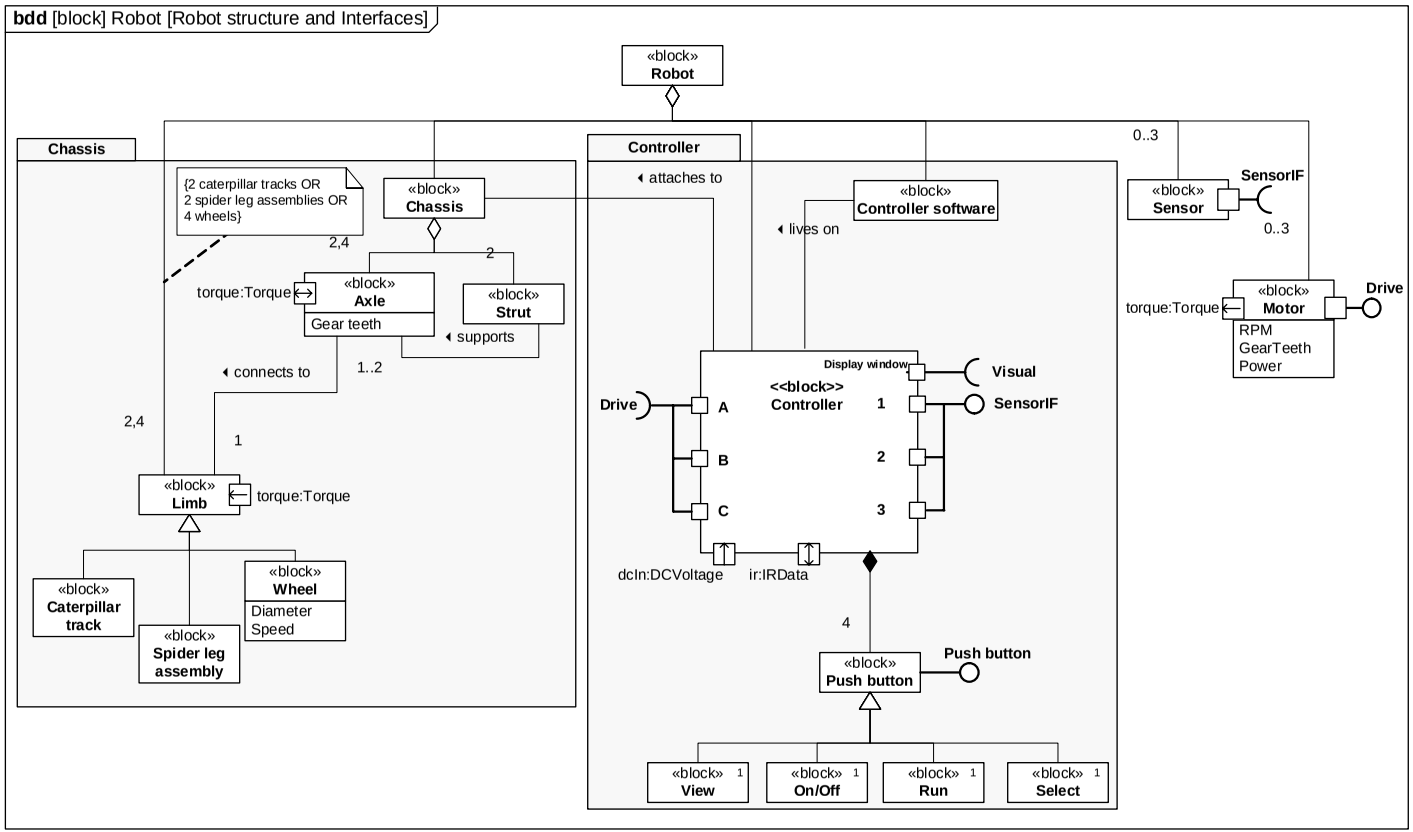

Figure A.4 Example block definition diagram

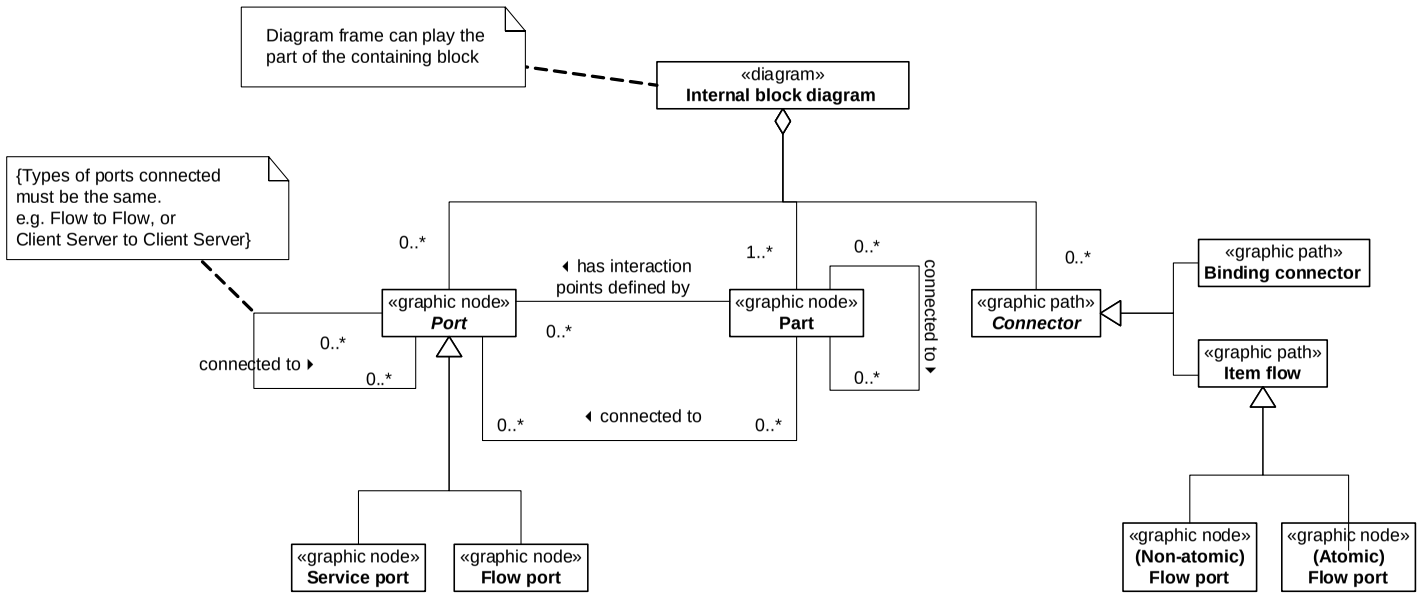

Figure A.5 Partial meta-model for internal block diagrams

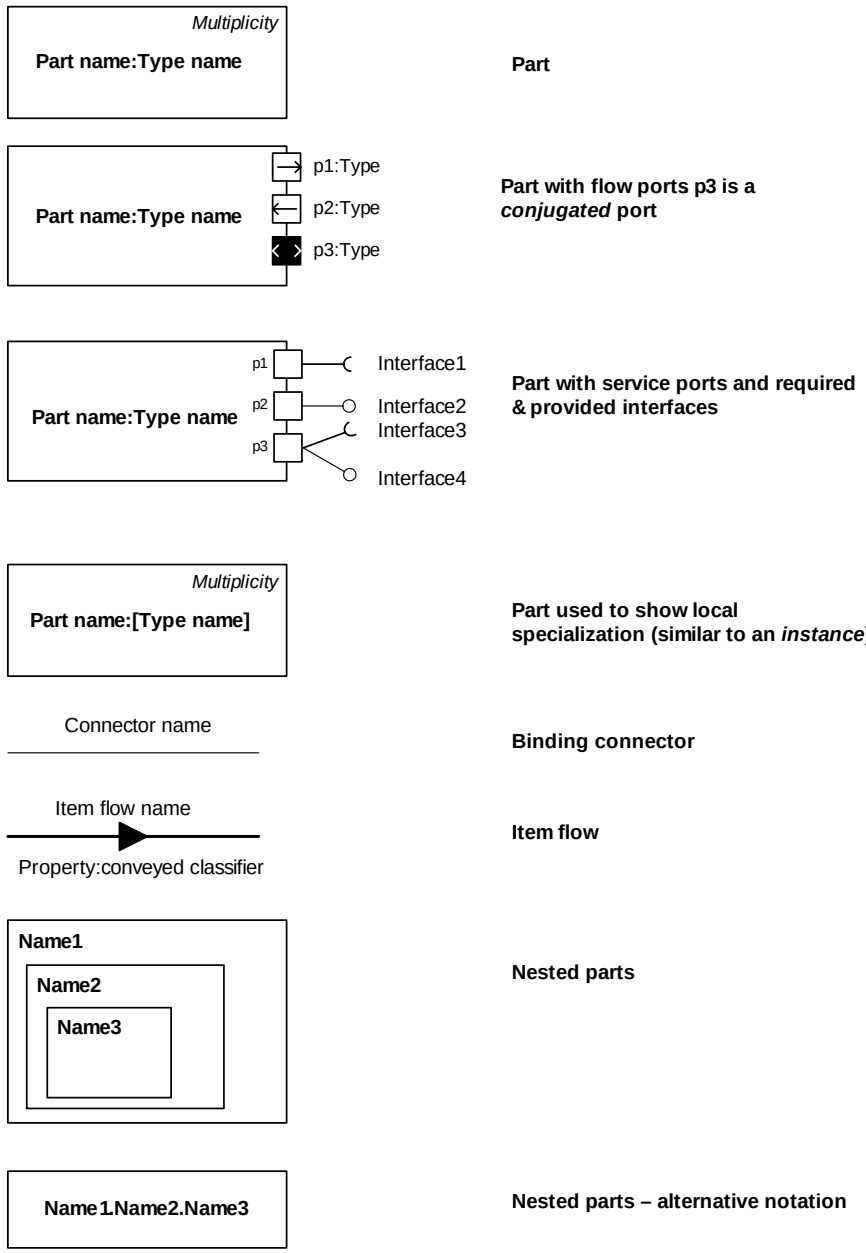

Figure A.6 Internal block diagram notation

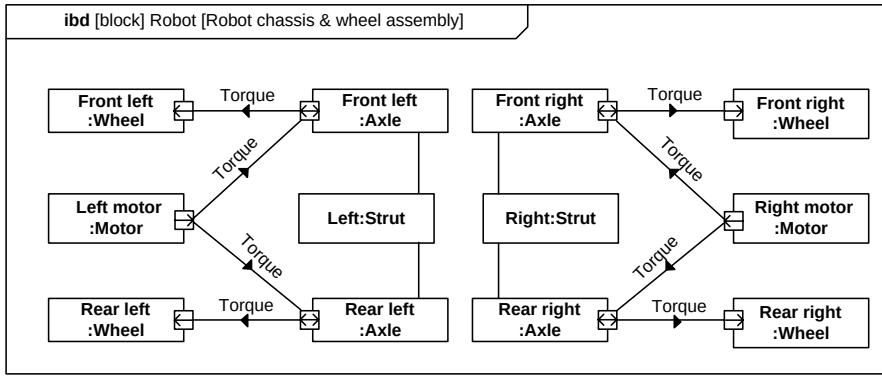

Figure A.7 Example internal block diagram

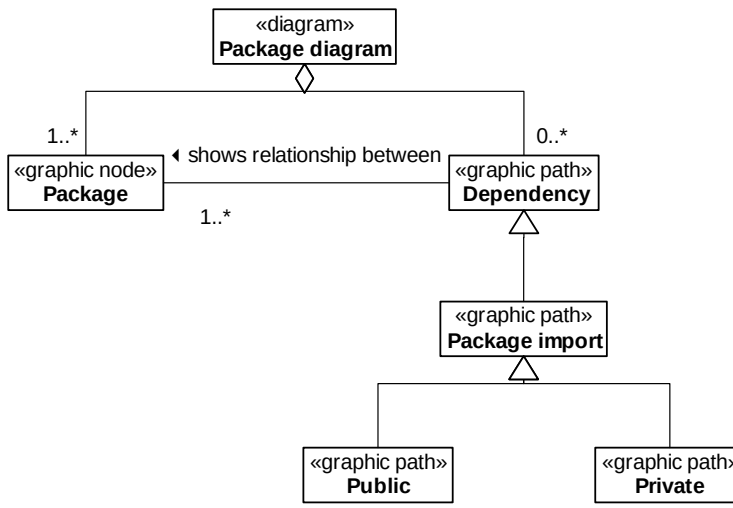

Figure A.8 Partial meta-model for package diagrams

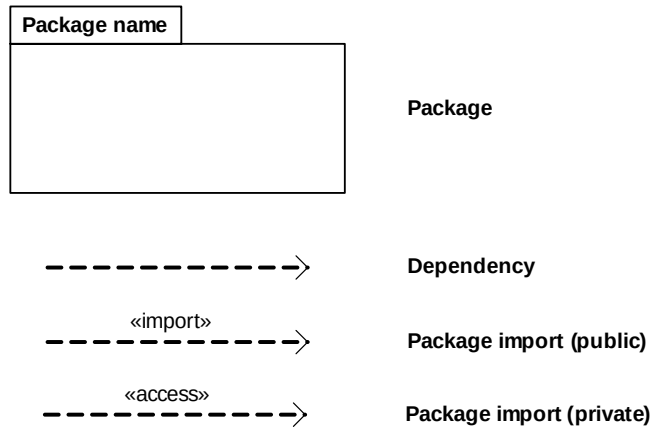

Figure A.9 Package diagram notation

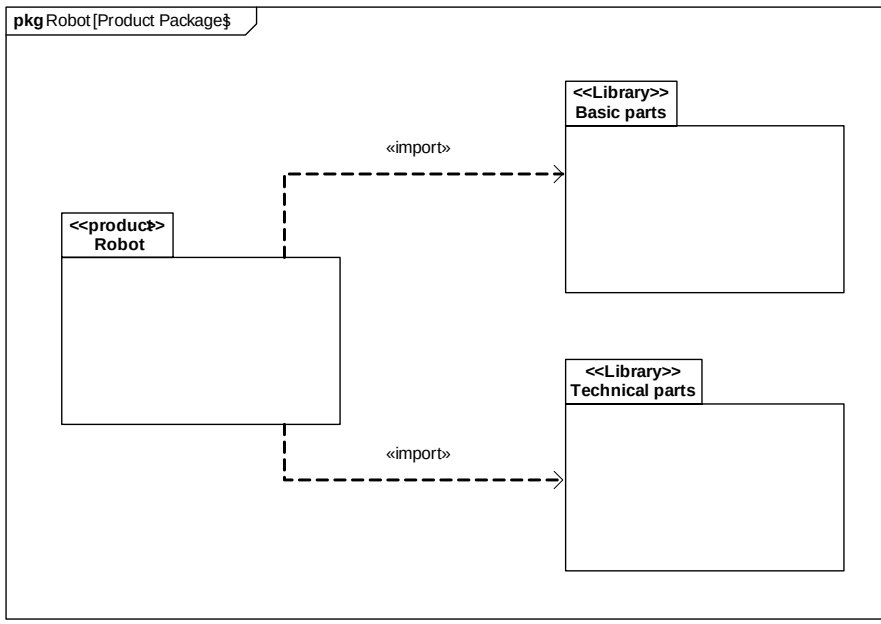

Figure A.10 Example package diagram

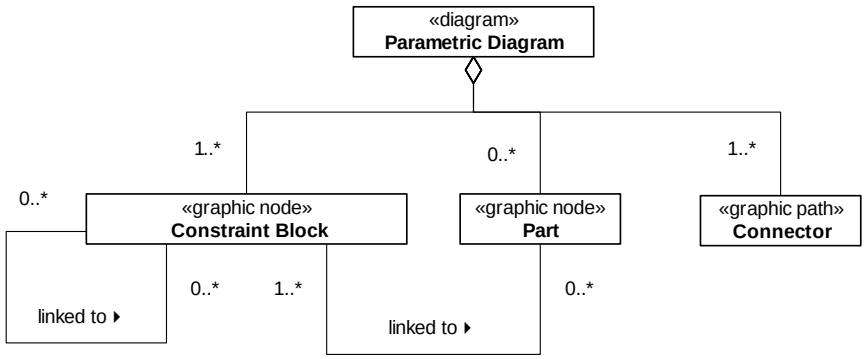

Figure A.11 Partial meta-model for parametric diagrams

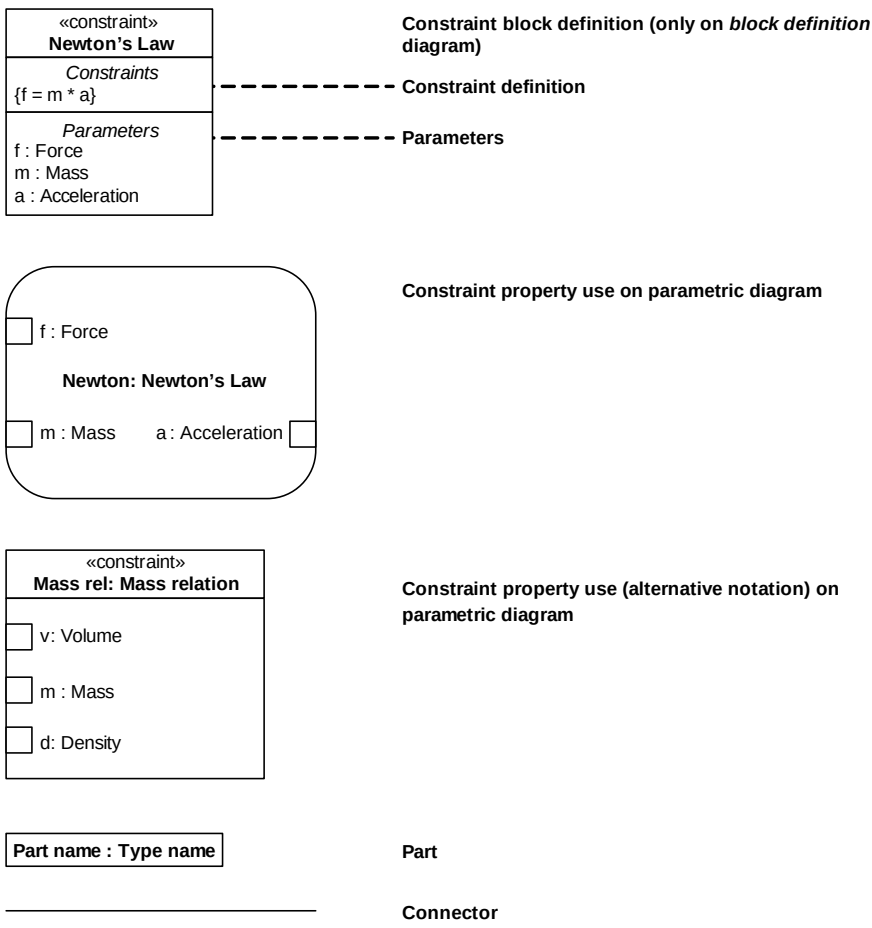

Figure A.12 Parametric diagram notation

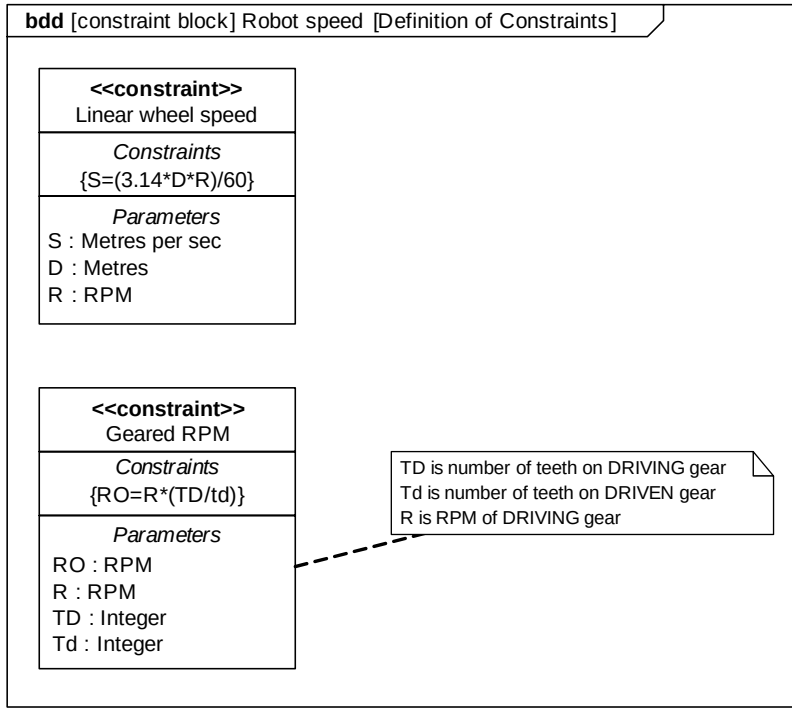

Figure A.13 Example parametric diagram – definition

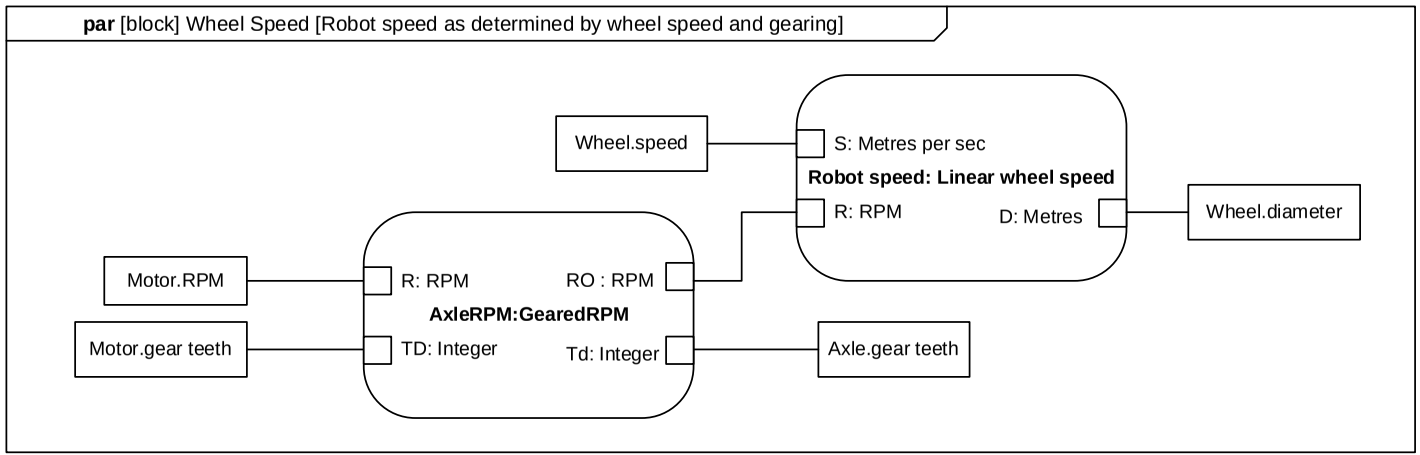

Figure A.14 Example parametric diagram – usage

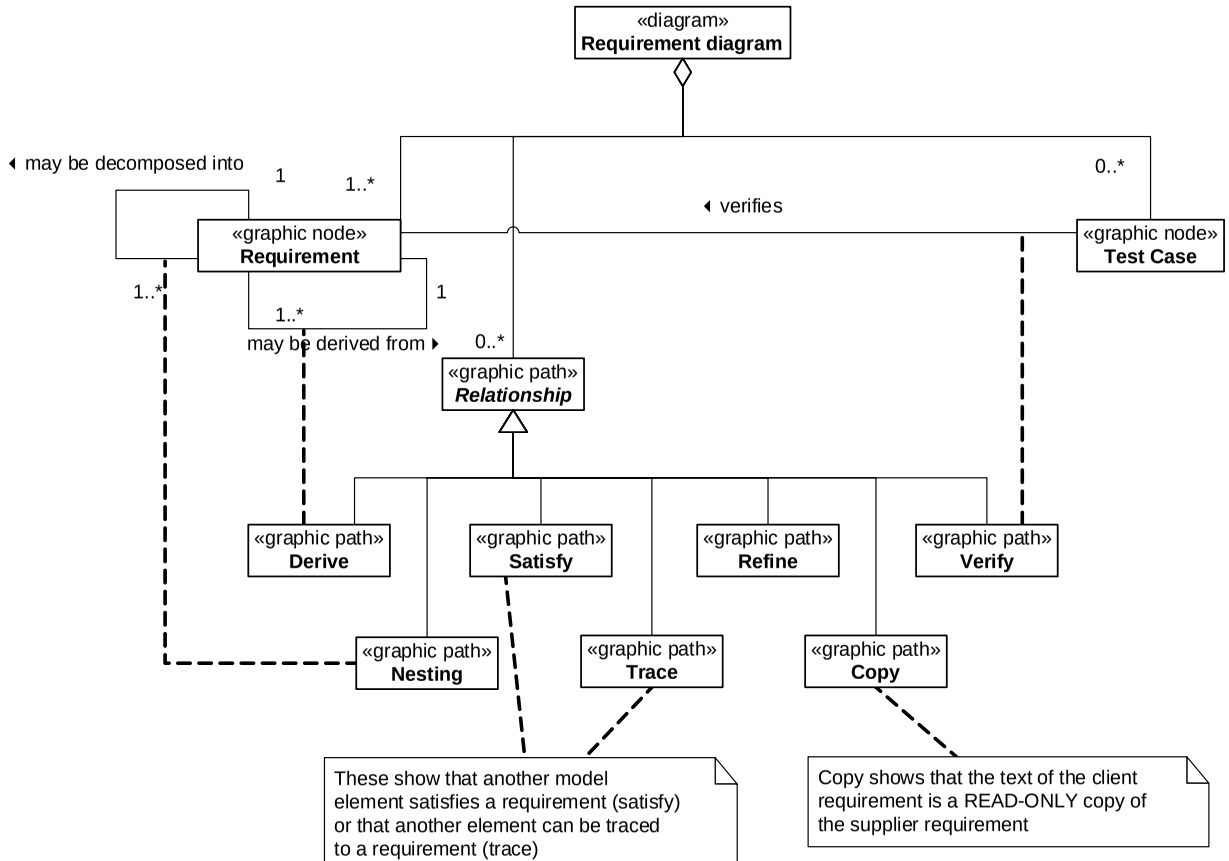

Figure A.15 Partial meta-model for requirement diagrams

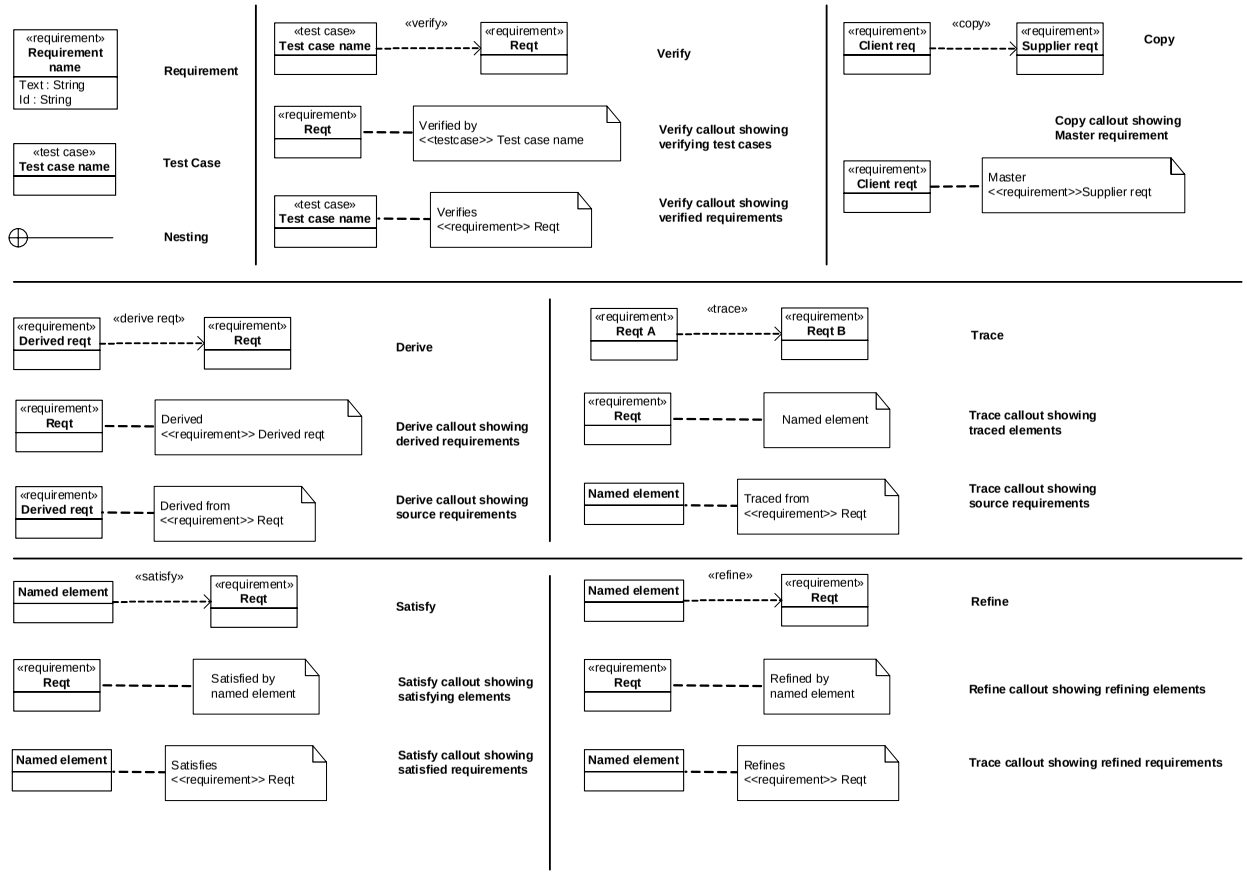

Figure A.16 Requirement diagram notation

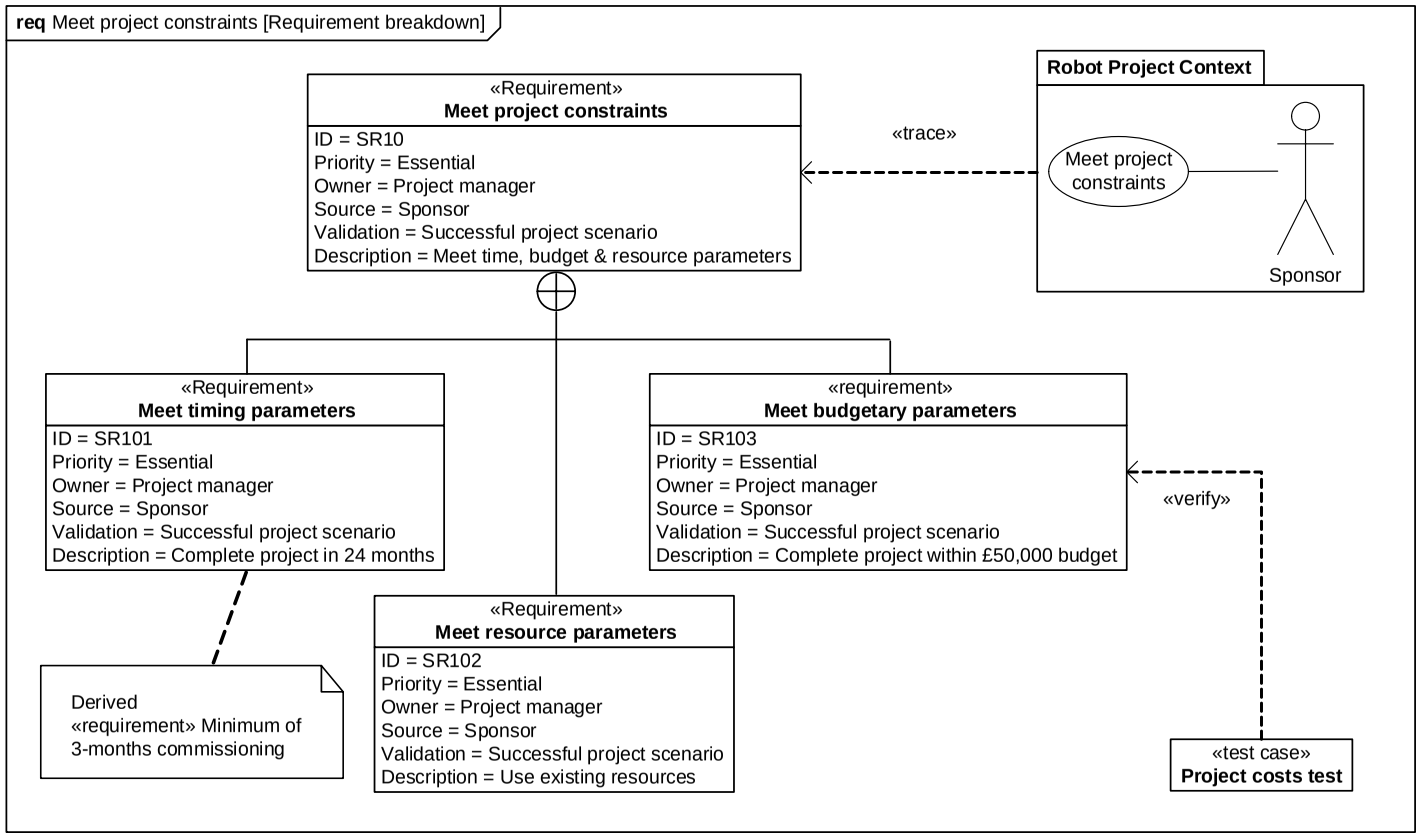

Figure A.17 Example requirement diagram

A.3 Behavioural diagrams

This section contains diagrams for each of the four SysML *behavioural* diagrams:

- state machine diagrams (Figures A.19–A.21)
- sequence diagrams (Figures A.22–A.24)
- activity diagrams (Figures A.25–A.28)
- use case diagrams (Figures A.29–A.31).

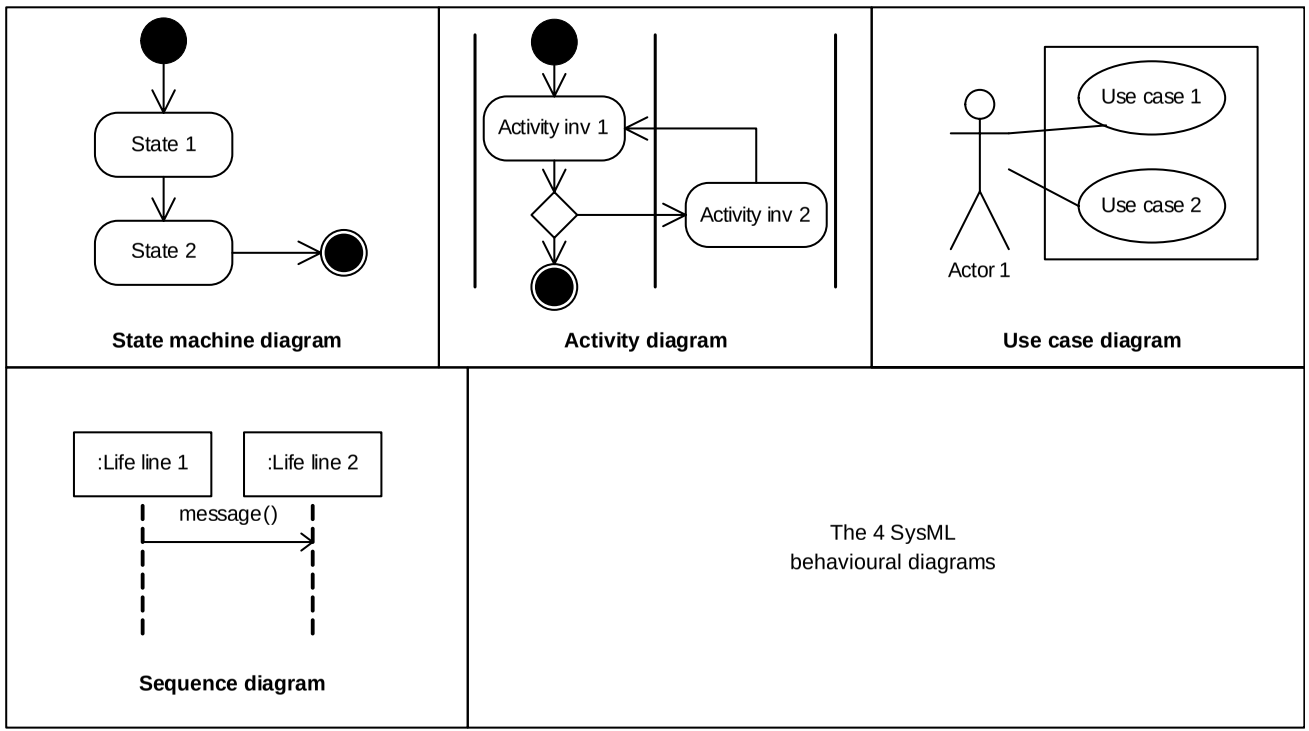

Figure A.18 Summary of behavioural diagrams

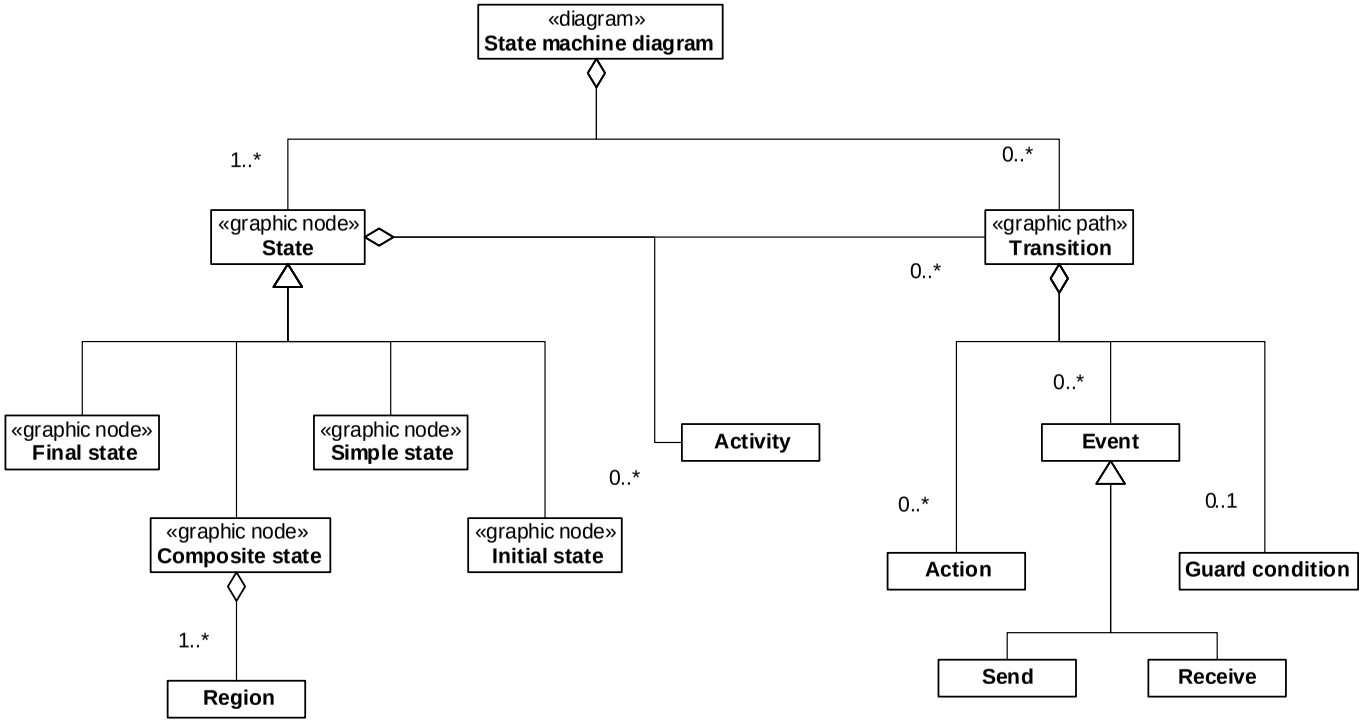

Figure A.19 Partial meta-model for state machine diagrams

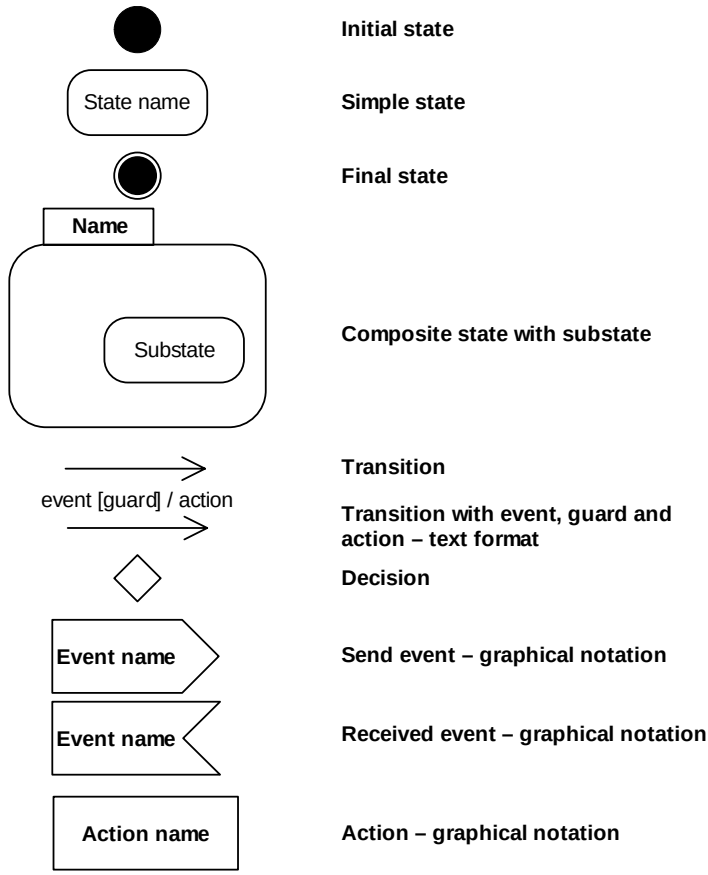

Figure A.20 State machine diagram notation

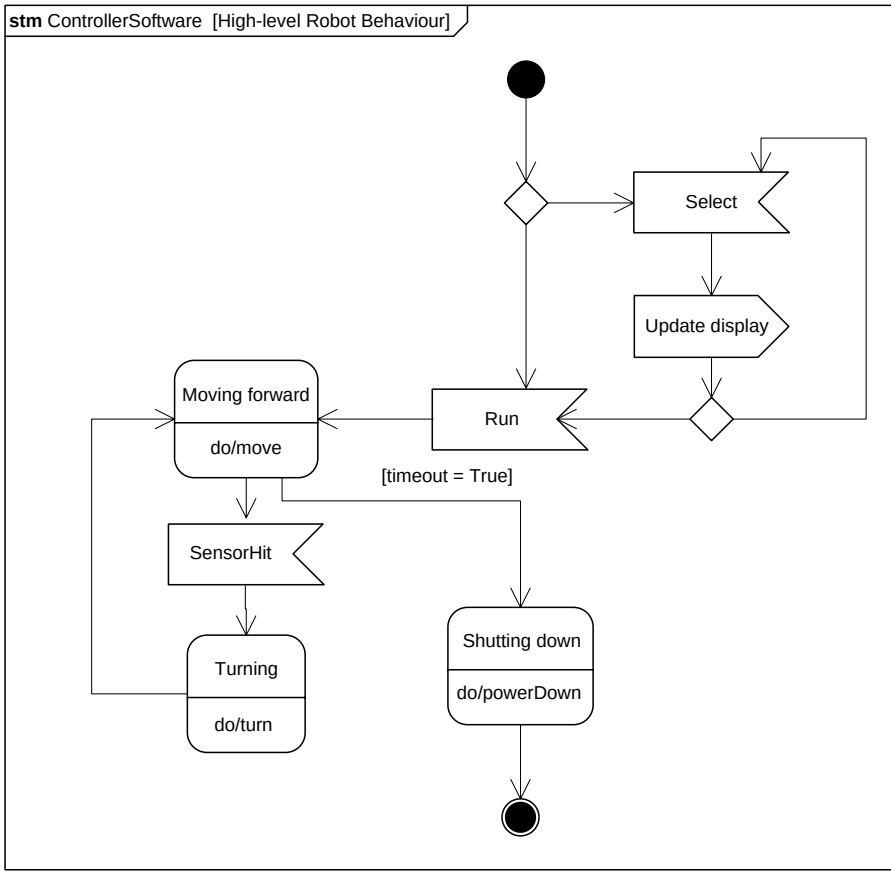

Figure A.21 Example state machine diagram

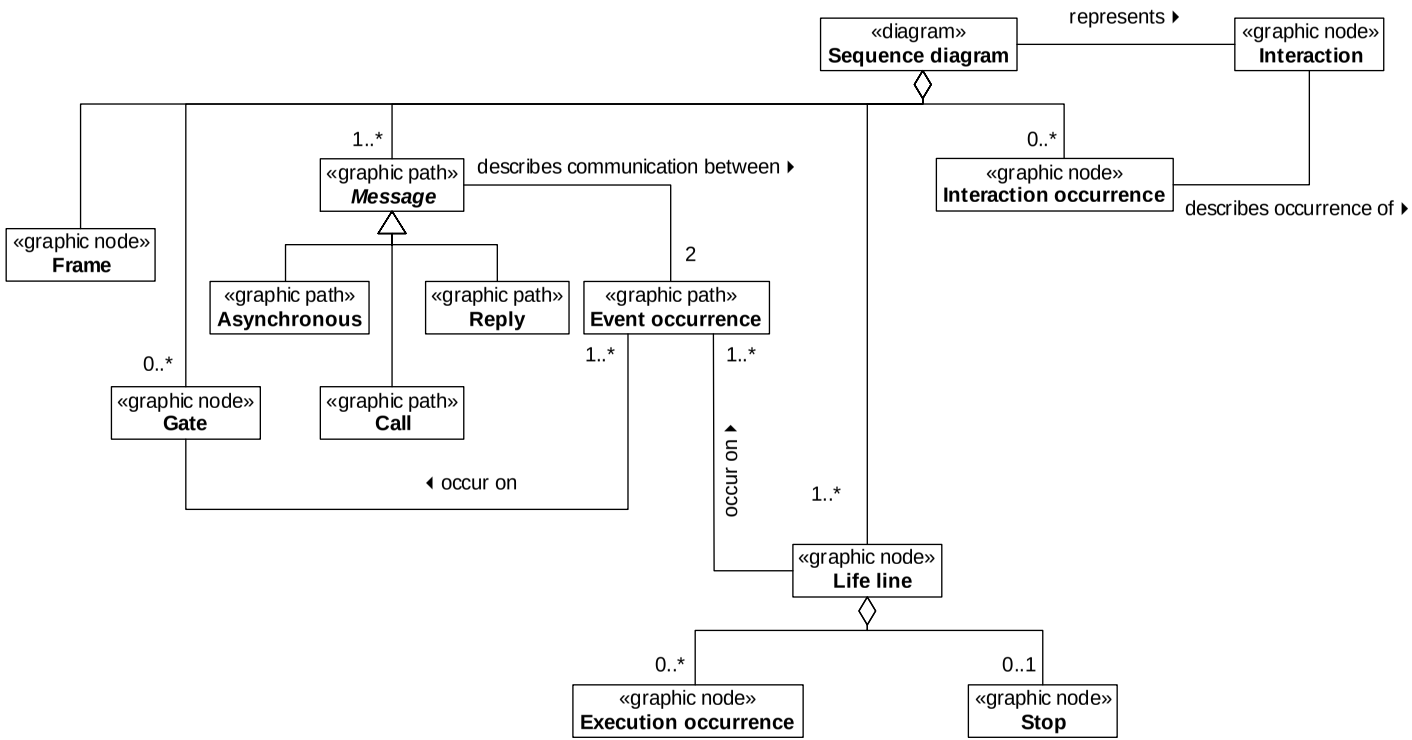

Figure A.22 Partial meta-model for sequence diagrams

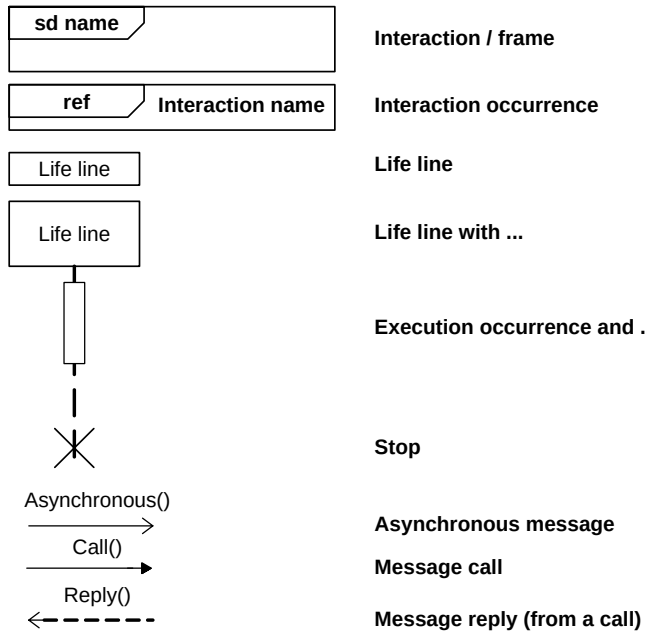

Figure A.23 Sequence diagram notation

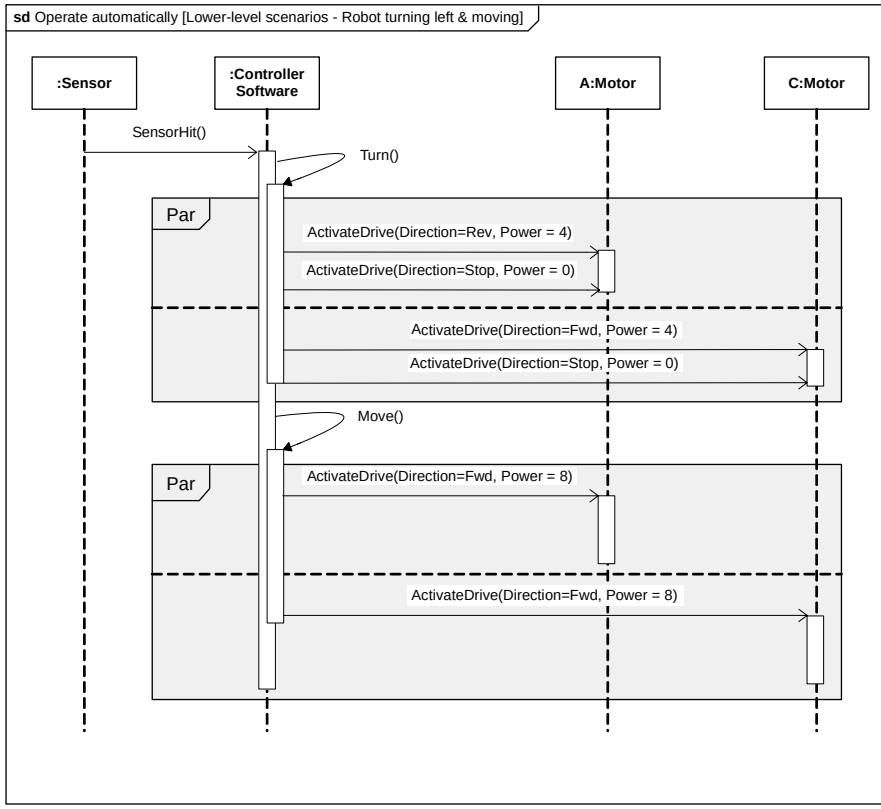

Figure A.24 Example sequence diagram

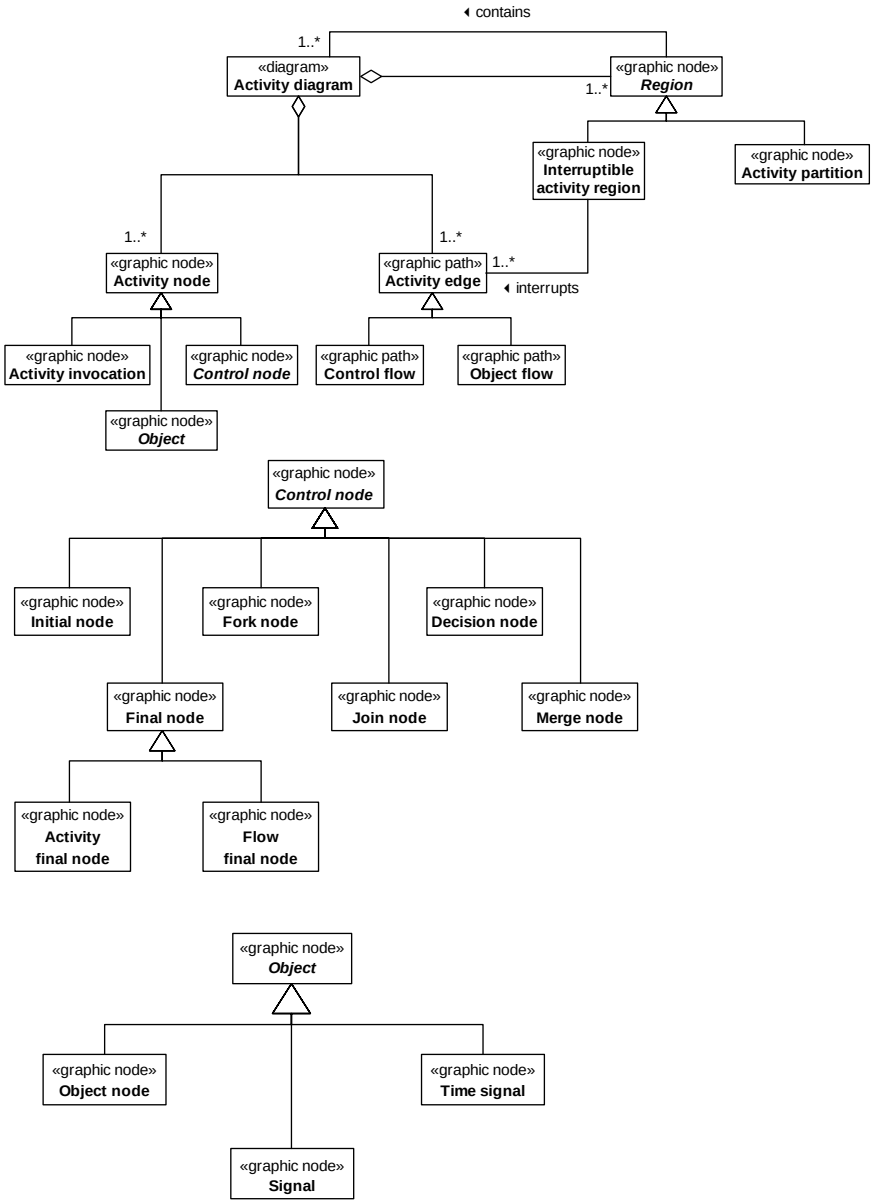

Figure A.25 Partial meta-model for activity diagrams

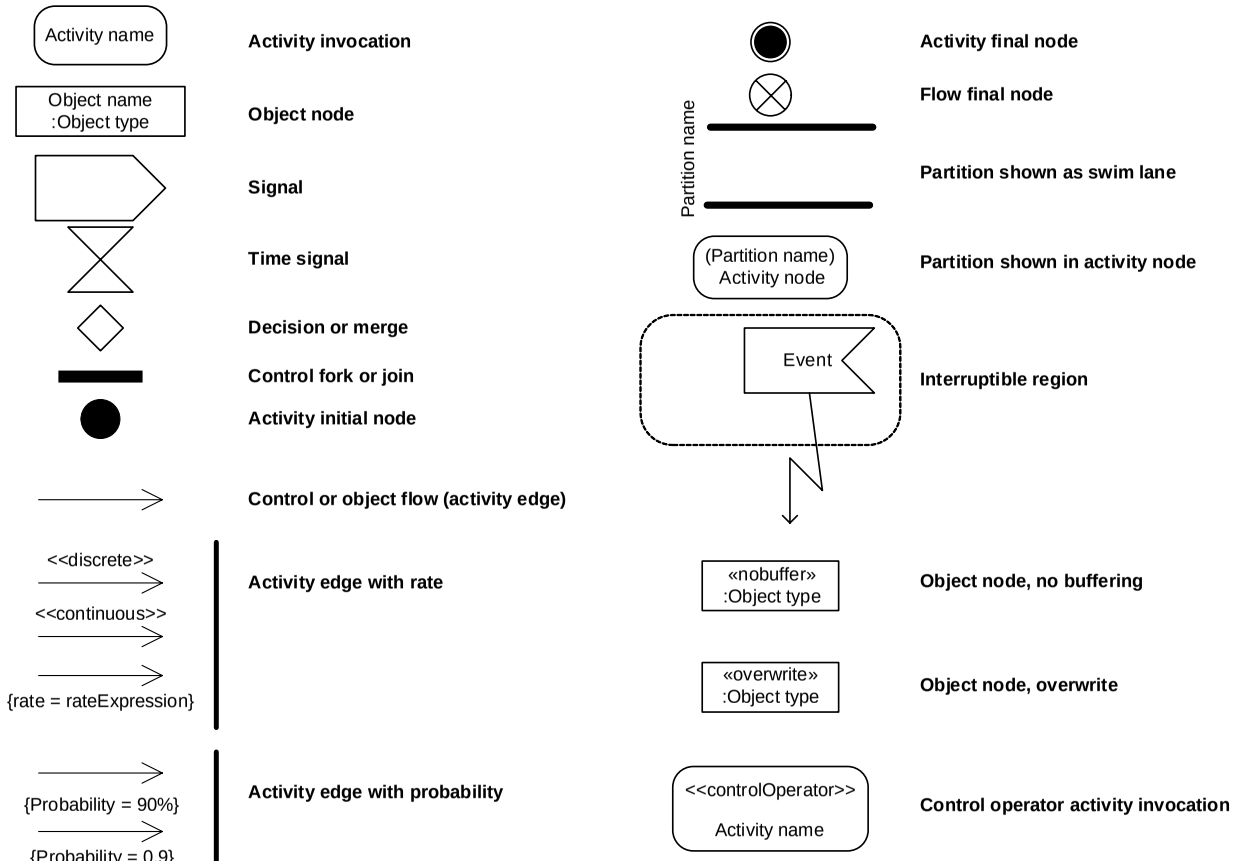

Figure A.26 Activity diagram notation

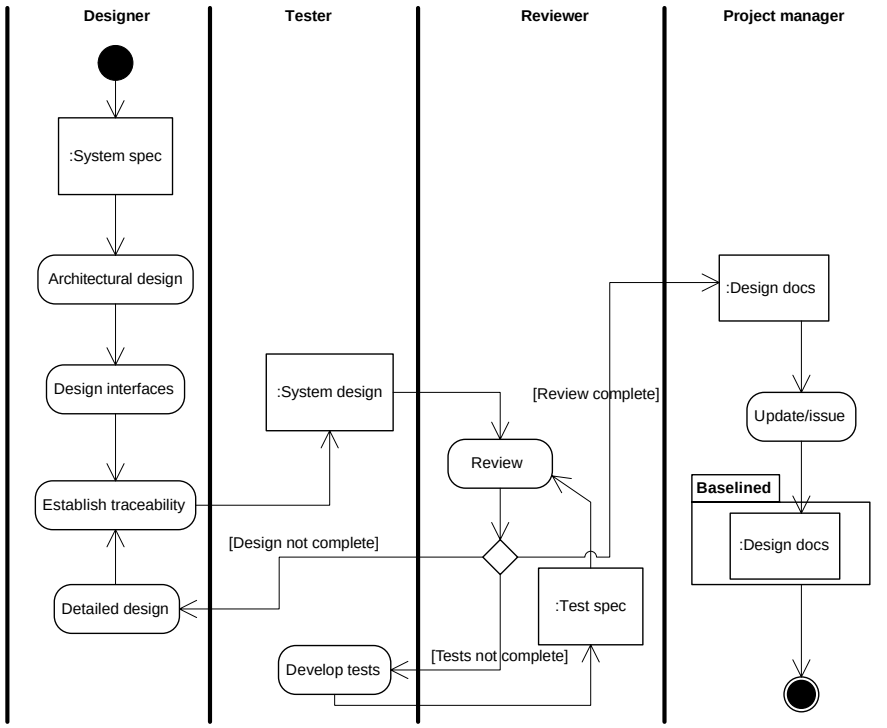

Figure A.27 Example activity diagram

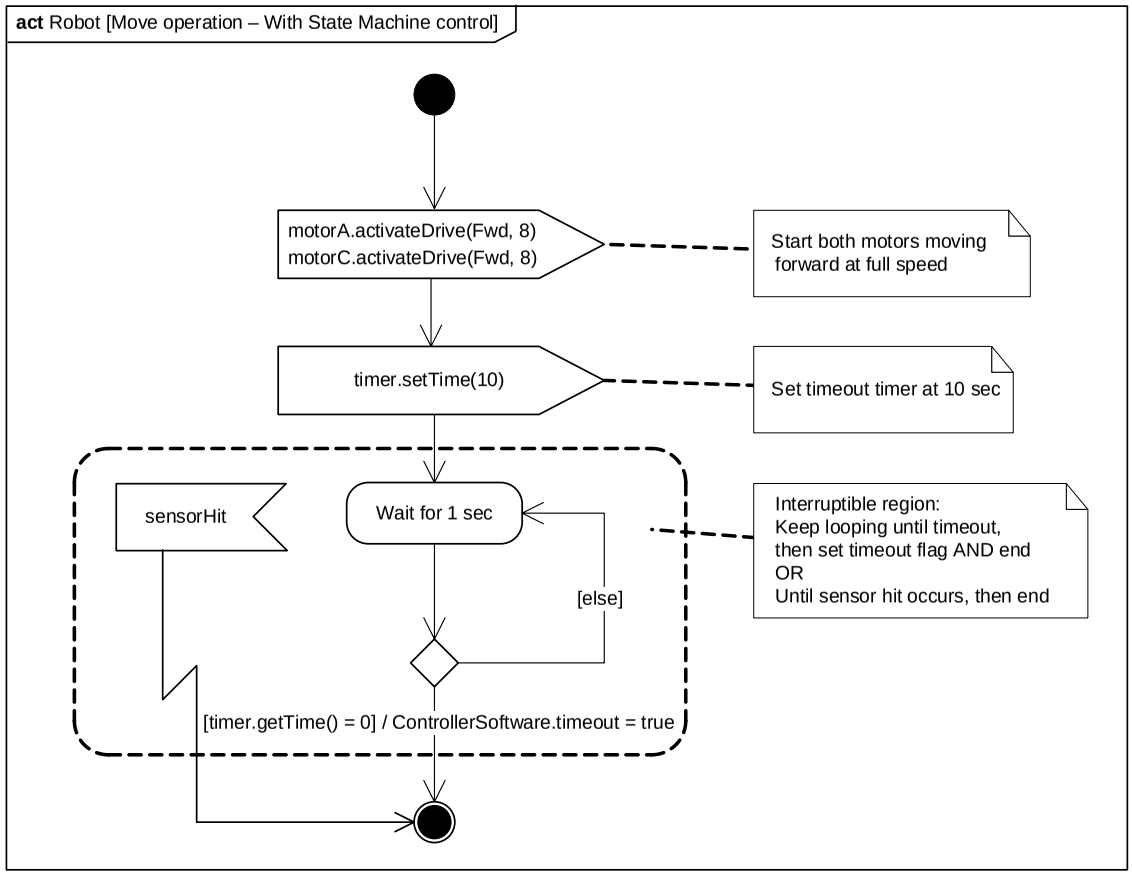

Figure A.28 Example activity diagram

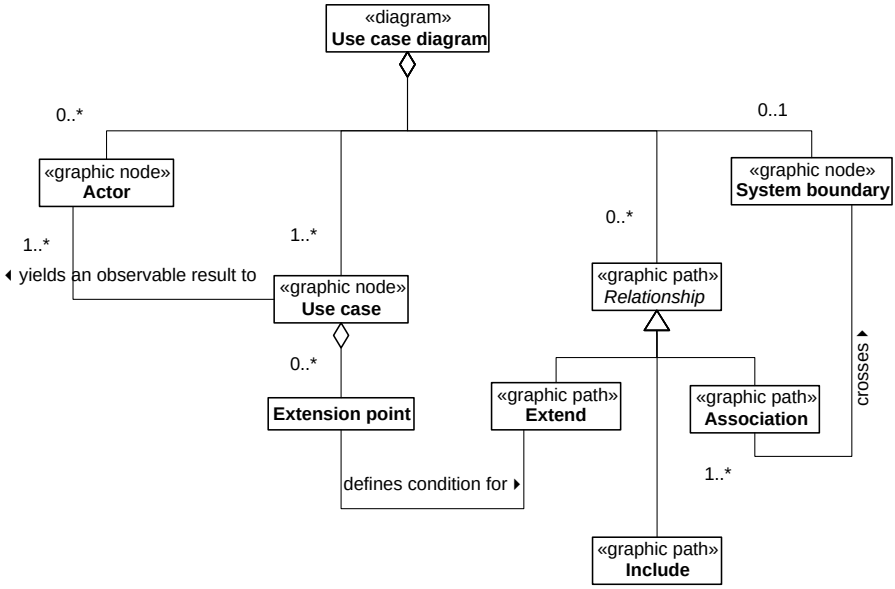

Figure A.29 Partial meta-model for use case diagrams

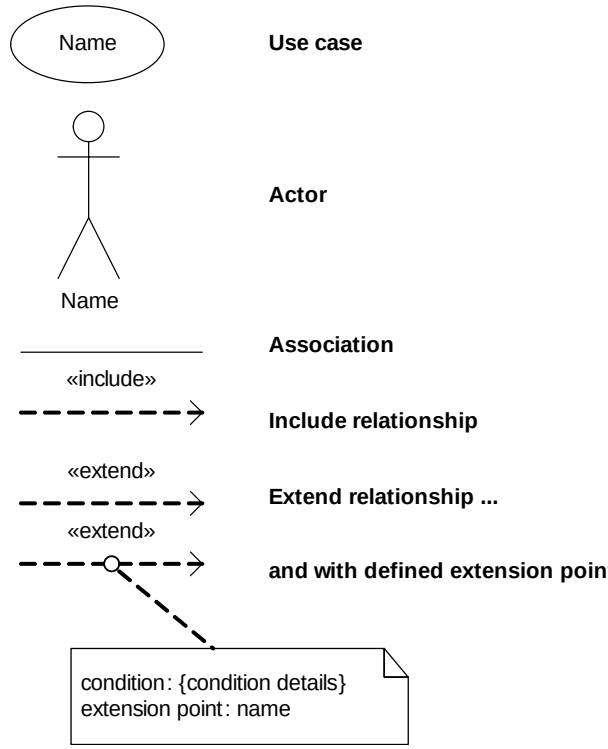

Figure A.30 Use case diagram notation

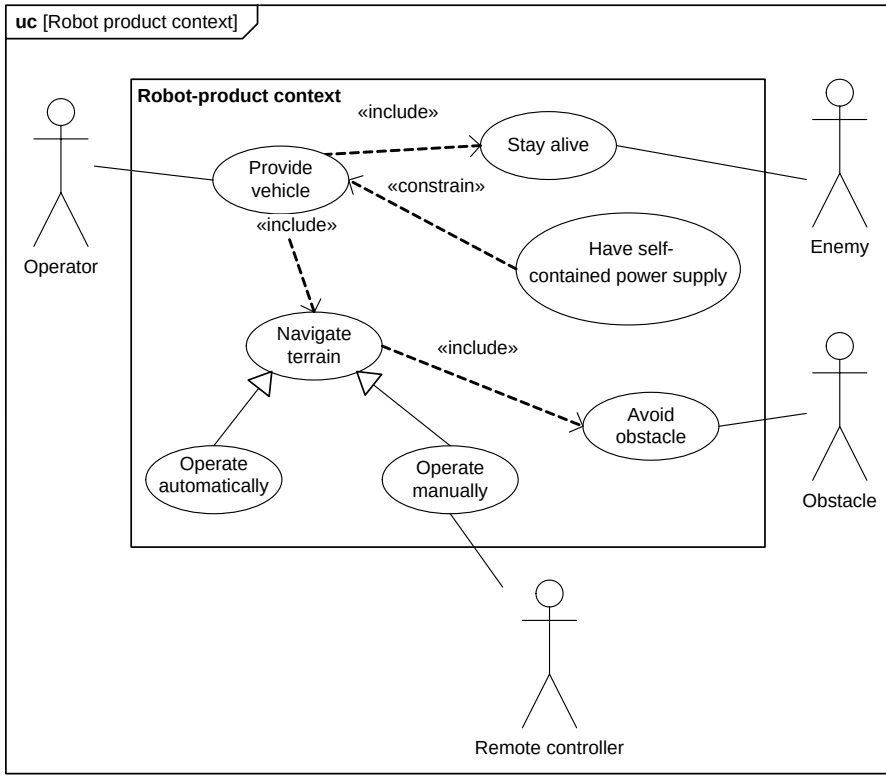

Figure A.31 Example use case diagram (product context)

A.4 Cross-cutting concepts

This section contains diagrams for the following *cross-cutting concepts* that can be applied to any diagram:

- allocations (Figures A.32–A.34)
- auxiliary constructs (Figures A.35–A.37).

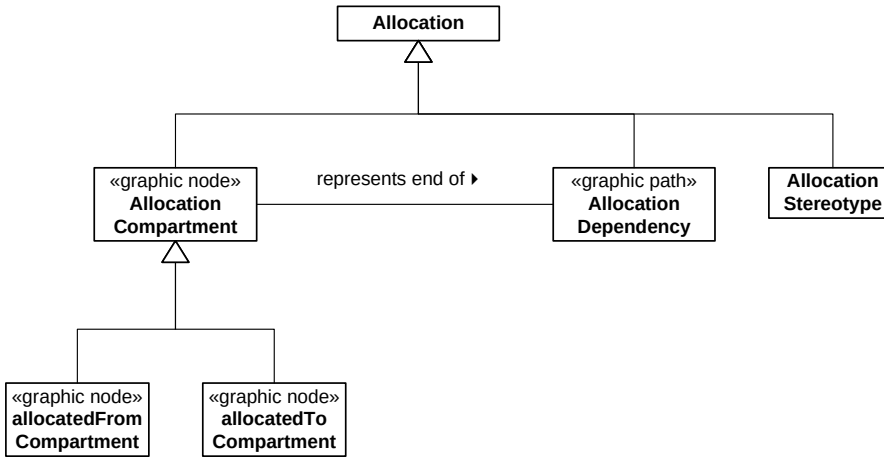

Figure A.32 Partial meta-model for allocations

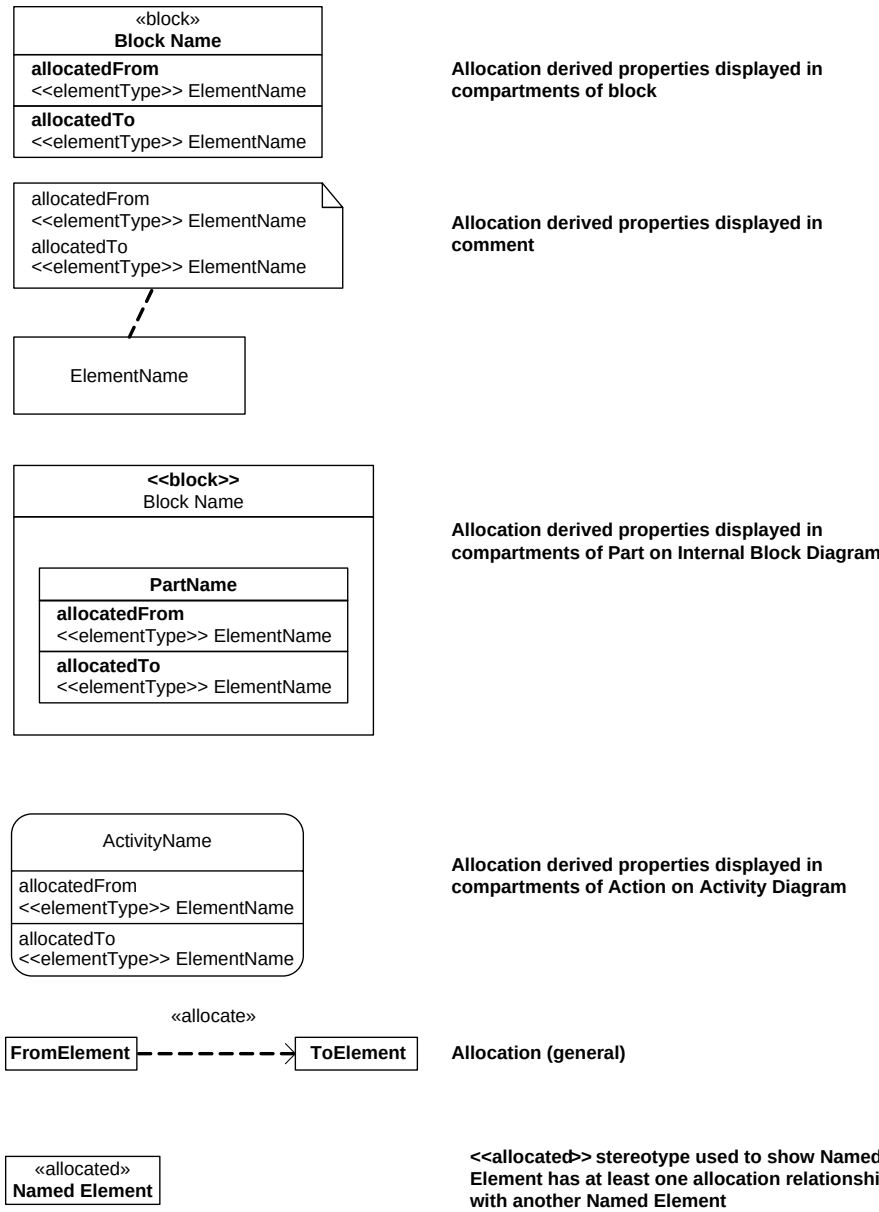

Figure A.33 Allocation notation

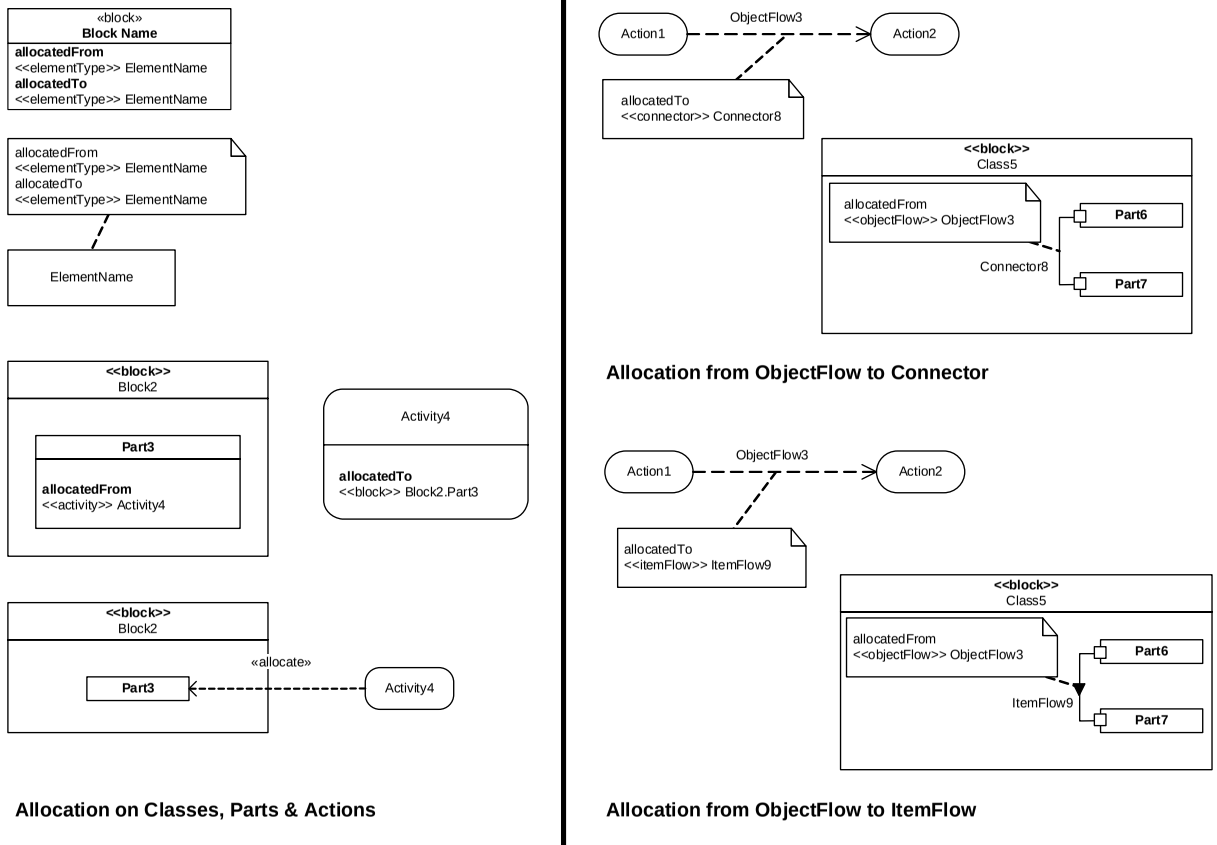

Figure A.34 Example of allocation usage

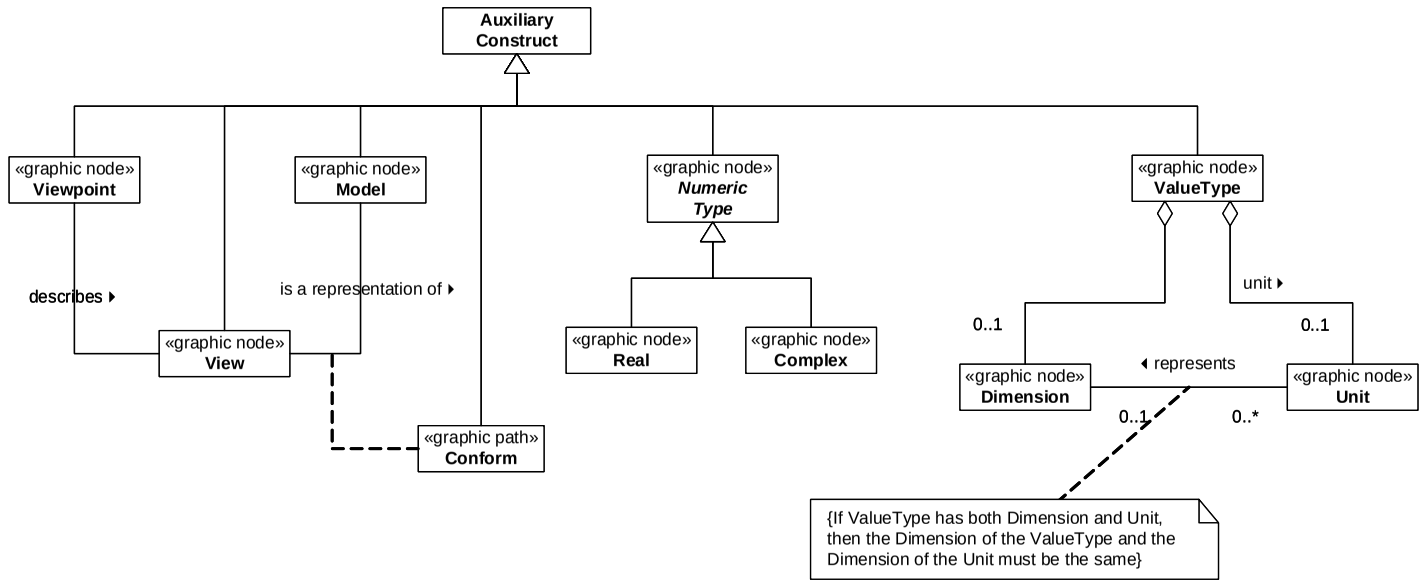

Figure A.35 Partial meta-model for auxiliary constructs

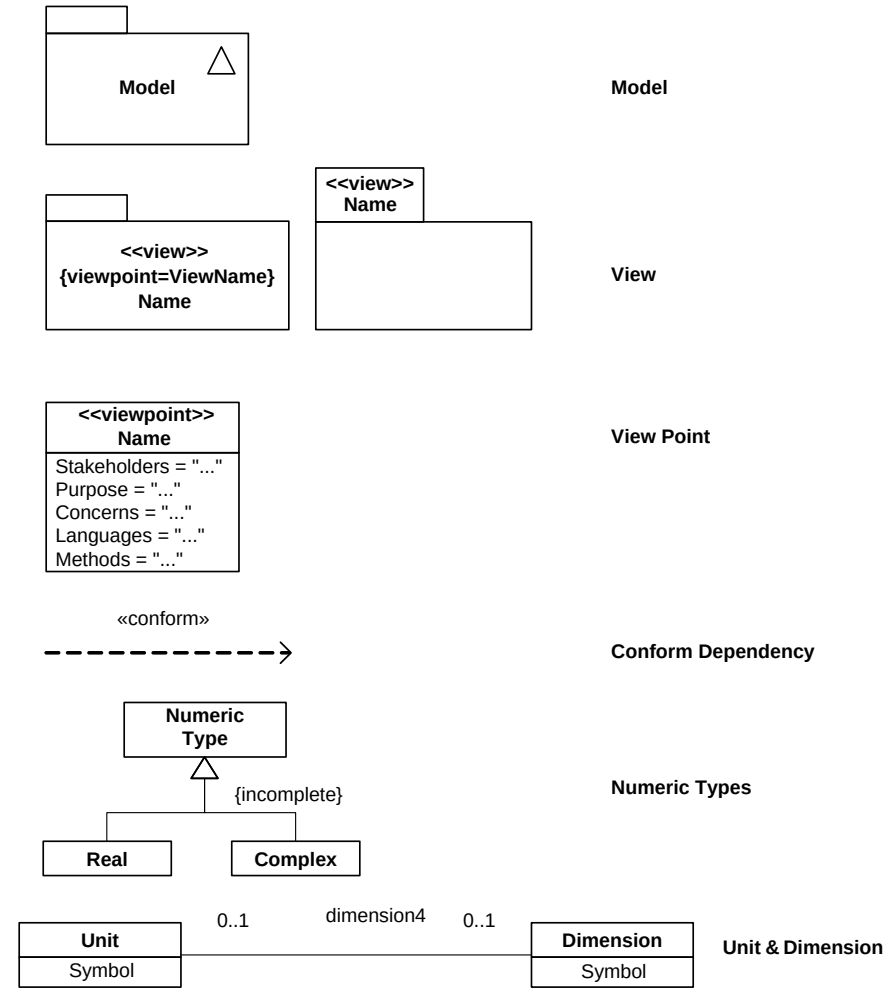

Figure A.36 Auxiliary construct notation

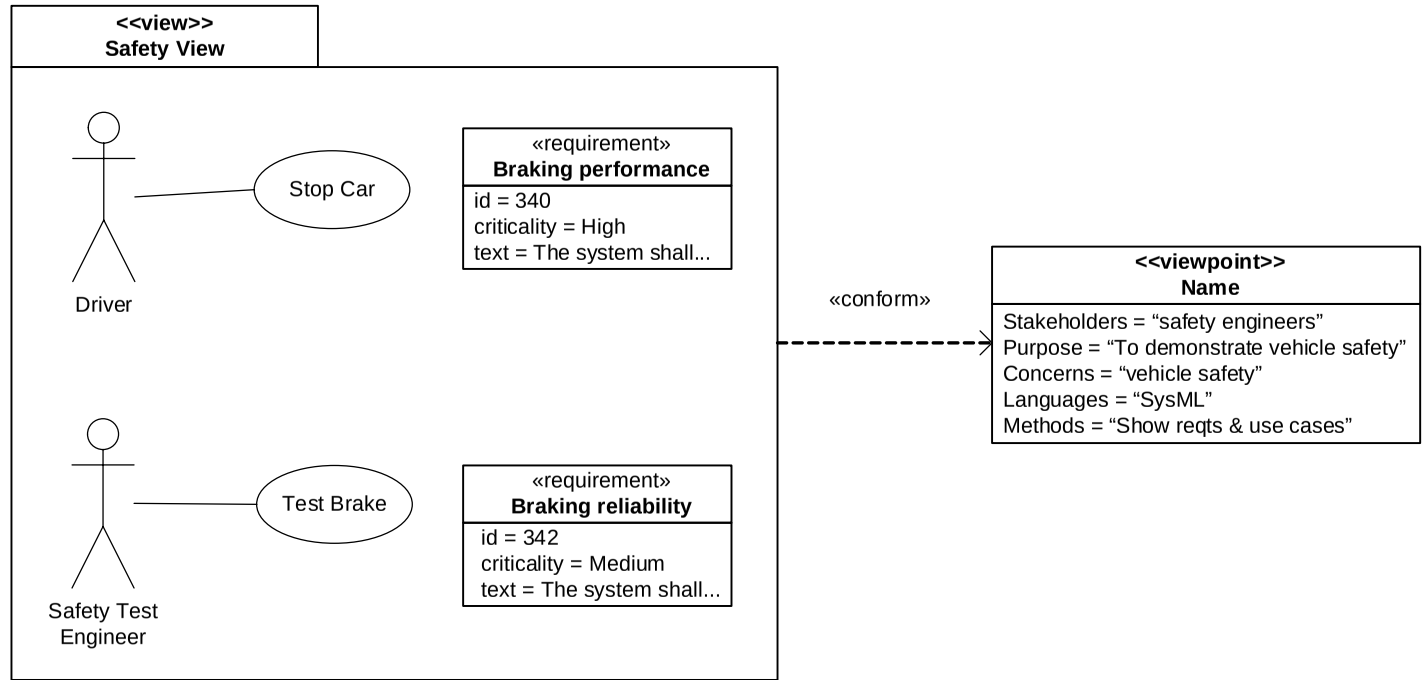

Figure A.37 Examples of auxiliary notation

A.5 Relationships between diagrams

SysML consists of nine diagrams that are used to capture and model different aspects of a system. Figure A.38 illustrates the main relationships between the diagrams (with the exception of the package diagram).

For parametric constraints, Figure A.39 illustrates some of the relationships between their definition and usage.

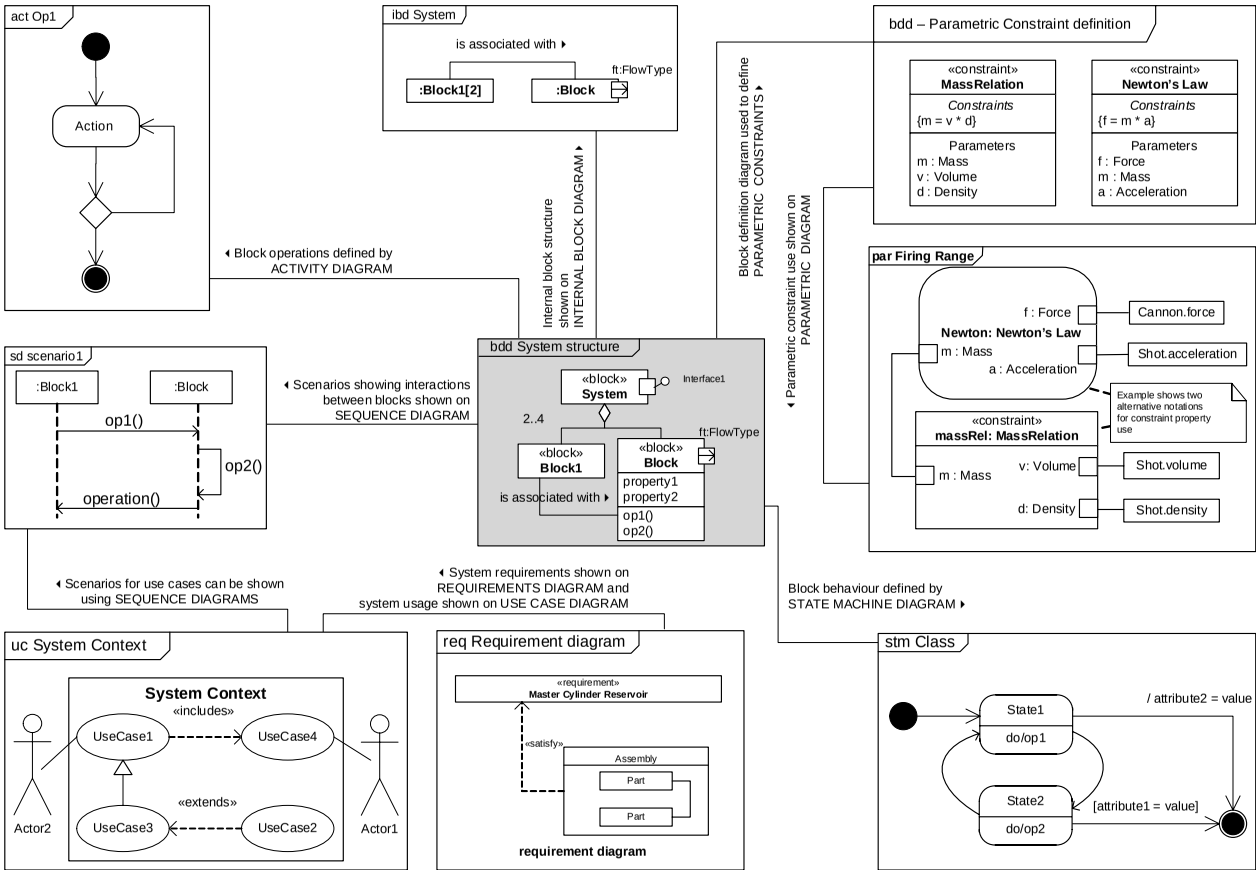

Figure A.38 Relationships between SysML diagrams

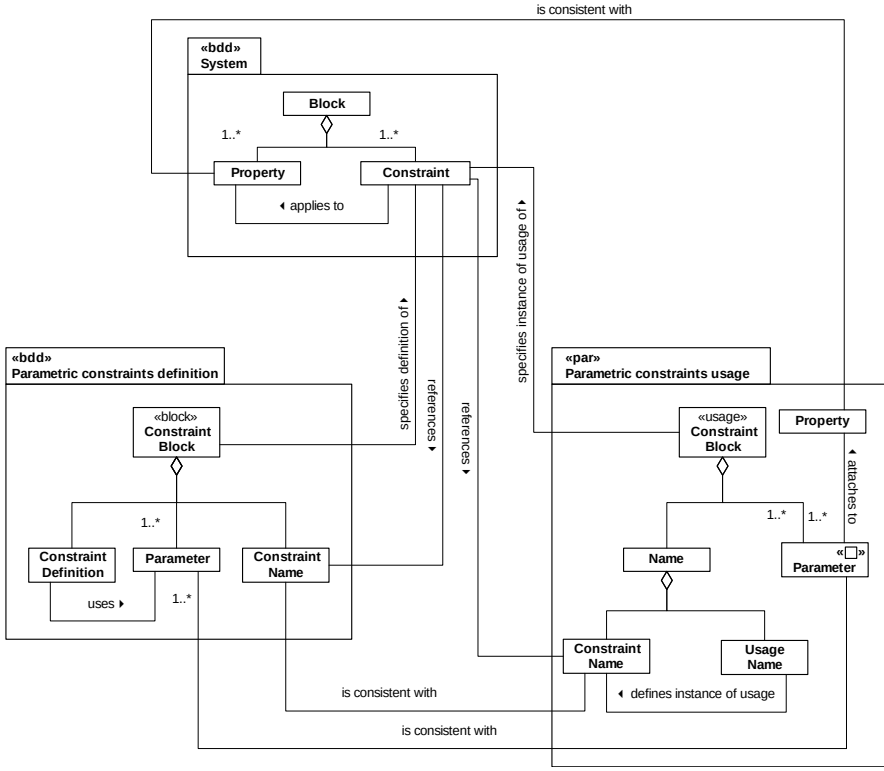

Figure A.39 SysML – relationships between blocks, parametric constraint definitions and parametric constraint usage

Appendix B

Using SysML concepts in UML

> ‘Oh, how fine are the Emperor’s new clothes! Don’t they fit him to perfection?’
> Nobody would confess that he couldn’t see anything, for that would prove him either unfit for his position, or a fool.
> ‘But he hasn’t got anything on,’ a little child said.
>
> Hans Christian Andersen (1805–75), ‘The Emperor’s New Clothes’, 1837

B.1 Introduction

While SysML has undoubtedly introduced some useful new notation and concepts – in particular flow ports, flow specifications and parametric constraints – it is *not* a new language but rather a subset of UML with some additions. The diagrams omitted from SysML may prove problematical for systems engineers wishing fully to model a system, particularly those working with software engineers who may be modelling with the UML and therefore using the omitted diagrams.

One solution for systems engineers is to use UML rather than SysML, but to *add* SysML constructs to UML. This appendix illustrates how this may be done.

B.2 Flow ports and flow specifications

SysML introduces the concepts of *flow ports*, *flow specifications* and *item flows*, which extend the concept of ports and interfaces found in UML to enable interfaces to be defined that pass and receive data, material and energy.

It is relatively easy to add these concepts to UML. The starting point is to define some new stereotypes, as shown in Figure B.1.

The diagram in Figure B.1 would also need to be coupled with descriptive text and constraints that describe the purpose and use of these stereotypes.

Figure B.2 shows a SysML block definition diagram that uses flow ports, flow specifications and item flows.

Figure B.3 shows the same diagram in UML and uses the stereotypes defined in Figure B.1.

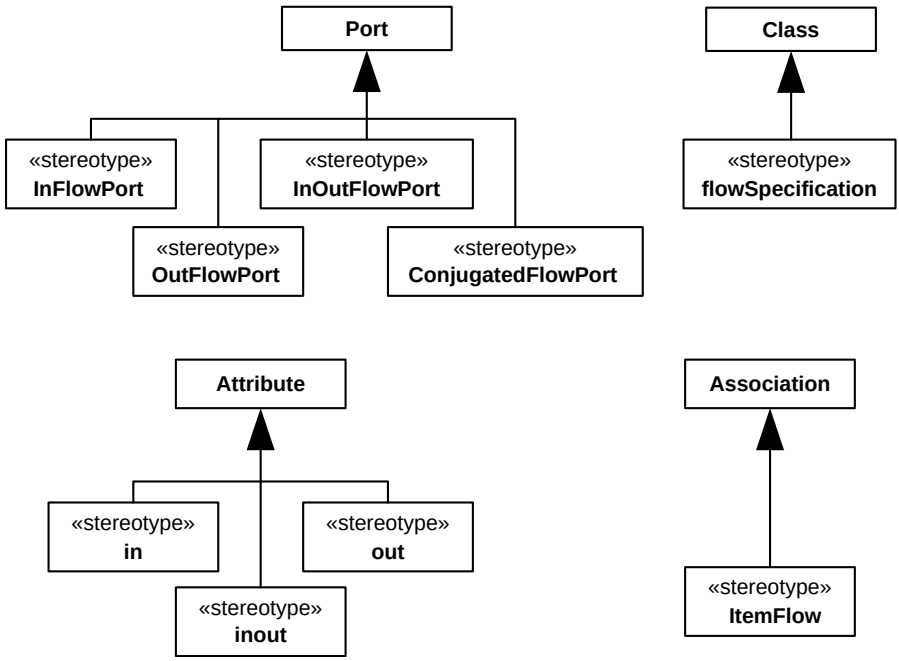

Figure B.1 Stereotypes to add flow port etc. concepts to UML

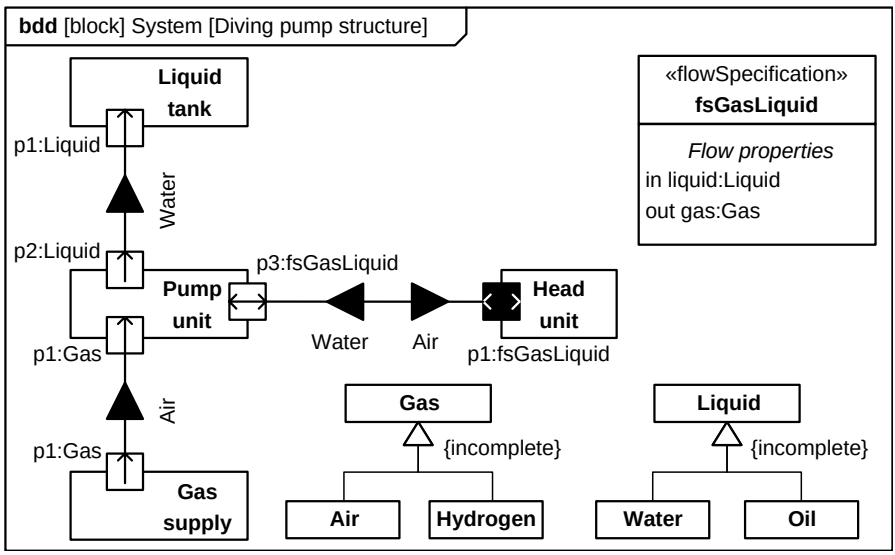

Figure B.2 SysML diagram showing the use of flow ports, flow specifications and item flows

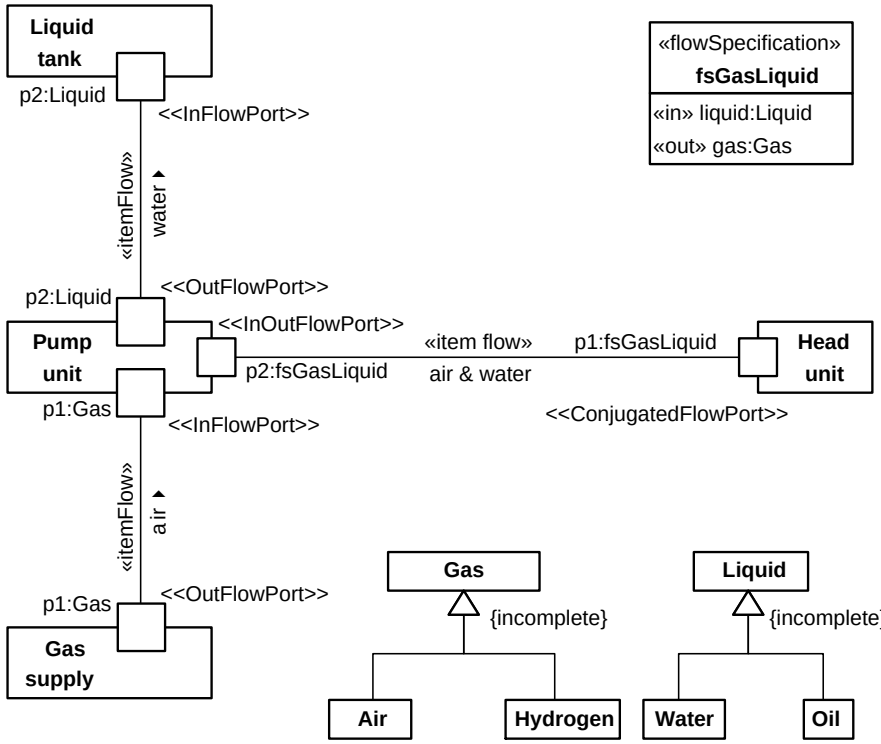

Figure B.3 UML diagram showing the use of flow ports, flow specifications and item flows

UML does allow alternative images to be associated with stereotypes, with stereotyped elements displayed using the alternative image rather than the textual stereotype notation. The various flow port stereotype definitions could be defined with alternative images in order to make the UML diagram much more similar to the SysML, but care is always needed when defining new graphical symbols, as their use may make reading a diagram difficult for those not familiar with the new symbols. A standard UML element with a stereotype makes it clear to the reader that the basic UML notation has been extended.

B.3 Parametric constraints

SysML also introduces the concepts of *parametric constraints*. These are defined on a SysML block definition diagram and used on a parametric diagram. Examples are shown in Figures B.4 and B.5.

Again, UML can be extended through stereotypes to include these concepts. Figure B.6 shows an example of the definition of the needed stereotypes. Again, this

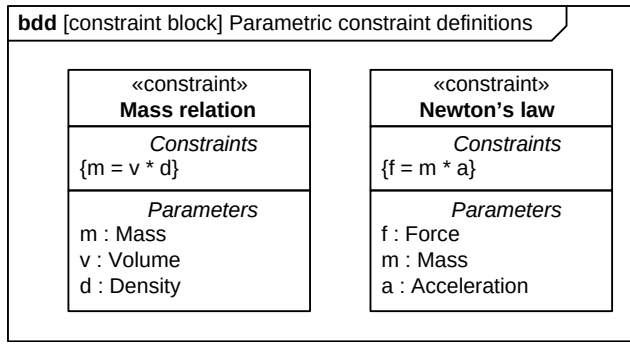

Figure B.4 SysML parametric constraint definition

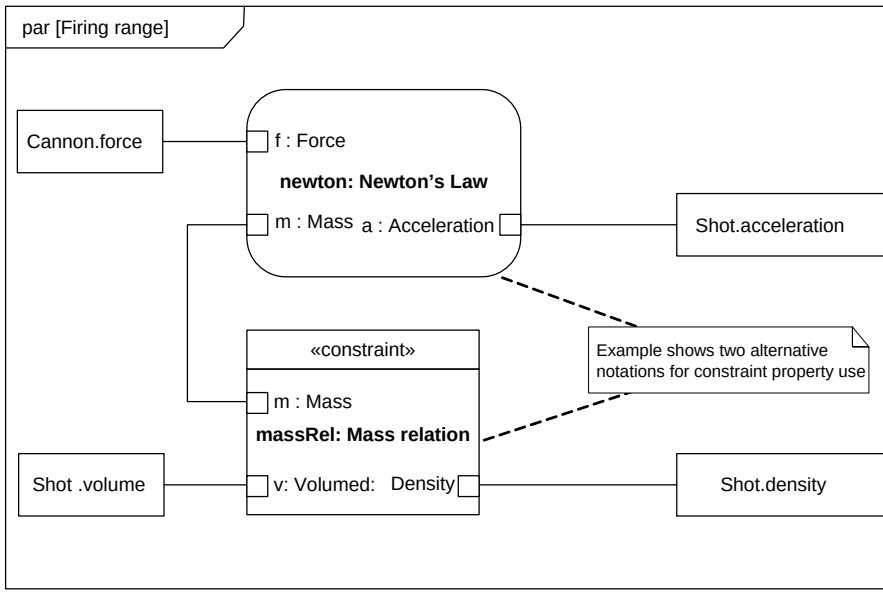

Figure B.5 SysML parametric usage

diagram needs to be accompanied by text and constraints describing the use of the stereotypes.

Figure B.7 shows a UML-class diagram defining two parametric constraints using the stereotypes defined in Figure B.6. Note the use of an additional *constraints* compartment on these classes. UML allows for any number of additional named compartments to be added to a class.

Figure B.8 shows the use of UML parts and the stereotype from Figure B.6 to describe the usage of parametric constraints.

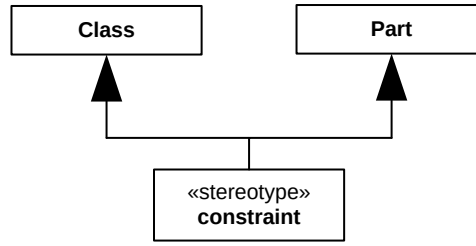

Figure B.6 UML stereotype definitions for parametrics

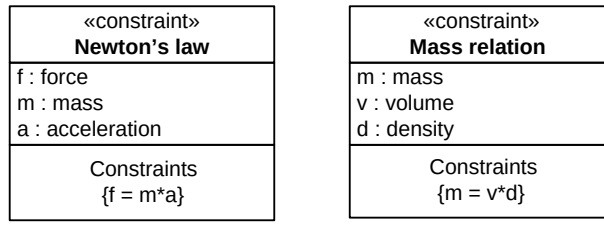

Figure B.7 UML parametric definitions

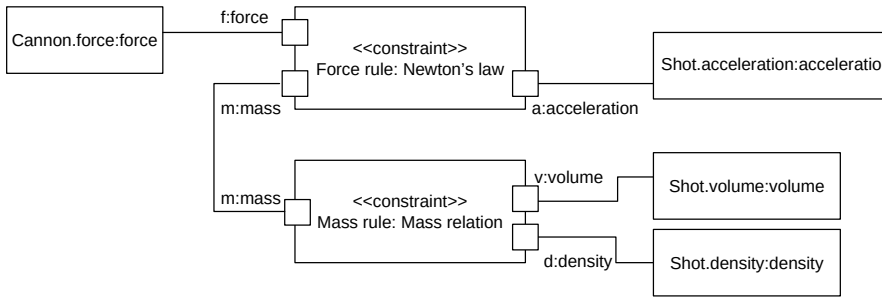

Figure B.8 UML diagram showing use of parametric constraints

B.4 Activity diagrams

SysML introduced some small notational additions to activity diagrams that allow rates and probabilities to be added to activity edges, and the concepts of buffering or no overwrite to be added to object nodes.

All of these concepts can be added to UML using either the existing constraints notation (for probabilities and defined rates) or stereotypes (for discrete and continuous rates and for the object node notations).

The stereotypes needed are shown in Figure B.9.

UML diagrams produced using these stereotypes will look identical to their SysML counterparts.

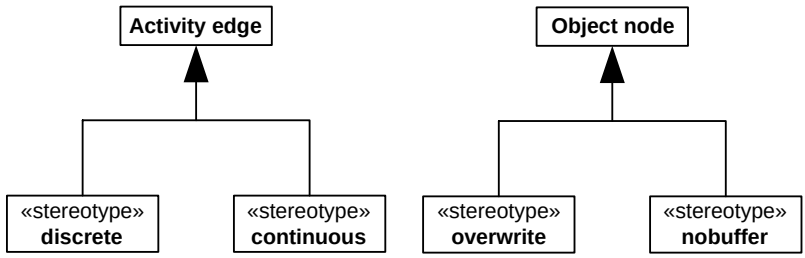

Figure B.9 Activity diagram stereotypes

B.5 Requirement diagrams

SysML requirement diagrams make use of a number of stereotyped dependencies, together with stereotyped blocks and the nesting symbol.

By stereotyping dependencies and classes, and using the composition symbol to replace the nesting symbol, requirement diagrams can easily be drawn in UML. Note that the nesting symbol *is* part of the UML, but is usually used to show structures of nested packages. The *meaning* of the nesting symbol in its use on SysML requirement diagrams is one of *composition* to show that a requirement is composed of a number of sub-requirements. For this reason the composition symbol, rather than the nesting symbol, is recommended. In fact, earlier versions of the SysML specification used the composition symbol rather than the nesting symbol.

The definition of the required stereotypes is left as an exercise for the reader.

Index

SysML for Systems Engineering

Systems modelling is an essential enabling technique for any systems engineering enterprise. These modelling techniques, in particular the unified modelling language (UML), have been employed widely in the world of software engineering and very successfully in systems engineering for many years. However, in recent years there has been a perceived need for a tailored version of the UML that meets the needs of today's systems engineering professional. This book provides a pragmatic introduction to the systems engineering modelling language, the SysML, aimed at systems engineering practitioners at any level of ability, ranging from students to experts. The theoretical aspects and syntax of SysML are covered and each concept is explained through a number of example applications. The book also discusses the history of the SysML and shows how it has evolved over a number of years. All aspects of the language are covered and are discussed in an independent and frank manner, based on practical experience of applying the SysML in the real world.

Dr Jon Holt is the founder director of Brass Bullet Ltd, a systems engineering training and consultancy company. He is a Fellow of both the IET and the BCS and is recognised as a thought leader in the world of systems modelling. He is also the author of several books and is an award-winning public speaker.

Simon Perry is a director of Brass Bullet Ltd. He has over 20 years experience working as a software engineer, systems architect and systems modeller in a wide range of business areas including defence, finance, building construction, nuclear installations, utilities and transport. He is a Member of the IET, the BCS and the IAP.

The Institution of Engineering and Technology
www.theiet.org
978-0-86341-825-9