UNIVERSITY OF SOUTHAMPTON
Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

A group design project report submitted for the award of
MEng Computer Science

Supervisor: Professor Michael Butler and Colin Snook
Examiner: Dr Mike Poppleton

Extensions to UML-B Notation and Toolset

by James Amor, Tim Lewy and Chris Lovell

December 14, 2006

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

A group design project report submitted for the award of MEng Computer Science

by James Amor, Tim Lewy and Chris Lovell

The UML-B notation has been created as an attempt to combine the success and ease of use of UML, with the verification and rigorous development capabilities of formal methods. However, the notation currently only supports a basic diagram set. To address this we have, in this project, designed and implemented a set of extensions to the UML-B notation that provide a much fuller software engineering experience, critically making UML-B more appealing to industry partners. These extensions comprise five new diagram types, which are aimed at supplying a broader range of design capabilities, such as conceptual Use-Case design and future integration with the ProB animator tool.

Contents

List of Figures

Acknowledgements

The group would like to thank the following people for their help and input throughout this project:

Professor Michael Butler for supervising our project, guiding our projects focus and analysing our designs.
Colin Snook also for supervising our project and giving us an in-depth insight into UML-B.
Ian Johnson from AT Engine Controls Ltd. for his input into the requirements of the project.
Dr Mike Poppleton for his input through the progress seminars.
Elisabeth Ball for demonstrating UML-B and answering our questions early on in the project.

Chapter 1

Introduction

As part of an ongoing effort to increase the uptake of formal methods and make them more accessible to the wider community, a notation known as UML-B [25] is being developed. Discussed in detail in the following sections, the aim of this notation is essentially to combine the rigorous and formal capabilities of the B-notation [22] with the ease of use and low barriers to entry of the widely used UML [18] methodology.

Along with the notation, a set of tools for the creating and application of the notation is required. Some have already been developed by the University of Southampton. The first of these was based upon IBM's Rational Rose package [23, 8], which is a package for the design and modelling of software systems. This provided an effective tool for constructing models. However, being based on a proprietary and closed-source platform, meant the final tool lacked the openness and flexibility ideally required for such a tool to be successful. It also imposed a minimum level of investment upon anyone wishing to make use of it. The tool, nonetheless, provided proof of concept.

To address these issues a new tool was developed, based upon the highly popular and open Eclipse platform [6] (also originally from IBM). Eclipse is an open source, Java-based, platform which is highly extensible. This provided a much friendlier environment for developing the tool. The Eclipse framework also includes several plug-in frameworks such as GMF and EMF (see Section 2.4) that provide an extended level of support for creating model based tools and applications, such as the UML-B environment. These factors, combined with the existing popularity of the platform, made it an ideal platform for UML-B.

Developed as part of the RODIN project (see Section 2.4.1) the UML-B tool provides support for the established range of UML-B notations. Now this is available, it is being brought to the attention of interested companies (such as AT Engine Controls Ltd., the industrial partner to this project). As a result, it has become apparent that there is a need to expand the UML-B notation to include other diagramming types such as the Component, Sequence and Use-Case Diagrams.

The existing notations are very 'B-orientated' and they provide adequate functionality for a software engineer to perform diagrammatically, what could otherwise have been achieved using formal methods. In this respect they achieve their goals. However, there is little else to them; especially considering that UML-B is supposed to be UML extended with B connotations, rather than the other way round [4].

UML-B is sold as both a combination of and an alternative to formal methods and UML design. To fulfil this role, it must support the entire software engineering process, not just those parts requiring rigorous formalisation. As such new, extended notations are being designed that will be aimed at enhancing UML-B as a software engineering process. This is the aim of this project.

1.1 Project Brief

More formally, this project extends the UML-B tools currently available and allows the use of new diagrammatic models, which extend and enhance the use of UML-B. It achieves this by first extending the existing UML-B metamodel and secondly, by producing Eclipse plug-ins to support the editing of new diagram types. The new diagrams are adapted from existing UML notations that were omitted from the original UML-B specification. The new diagrams that have been decided upon and implemented are:

- Activity diagram
- Component diagram
- Object diagram
- Sequence diagram
- Use-Case diagram

These new models are to be integrated with the existing UML-B models and tools, so that the entire extended notation can be seamlessly utilised. The key requirement of these extensions is that they enhance the software engineering process, without raising the learning curve or introducing barriers to entry.

1.2 Project Scope

This project has been somewhat limited in scope due to restrictions in terms of time and man power. The nature of the group project system, under which this work was carried out, was such that the total time allowed for implementation was only ten weeks of part-time work, by three students. Bearing this in mind, it was decided that the scope of this project would extend to fully designing and implementing new model and diagram types for the UML-B notation. Originally, three new diagrams were proposed, but as the project progressed a further two were found to be necessary and the scope was extended.

There is much more work that could be performed to further enhance the UML-B notation and toolset. Some thought has been given to this and several possibilities are outlined in Section 7. In addition, though it was outside of the scope of the project, each new diagram has been evaluated for possible uses within the larger UML-B framework. These are presented together with some possible corresponding semantics for the various diagram elements.

Chapter 2

Background

2.1 Formal Methods

Formal methods are “mathematically based techniques for describing system properties” [33] when designing a computer system. Formal methods are used to design and verify systems to try and ensure completeness of the system being designed. Due to the high cost in terms of time taken to design a system using formal methods and the expertise required from the system designers, formal methods are generally limited to being used in safety critical systems.

2.1.1 B-Method

The B-Method, designed by J.-R. Abrial [1], is a formal methods approach described by Schneider for “specification and development of computer software systems” [22]. The B-Method takes aspects of formal systems design, such as Z, and has been designed as a simplification of other formal method approaches. In this project we discuss two variations of the B-Method: Classic-B and Event-B. Classic-B is the name given to Abrial’s original B concept, where as Event-B [16] is an extension to B to allow for the formal design of distributed systems.

In Event-B, events replace the idea of operations in Classic-B. Events have guard conditions associated to them, similar to pre-conditions in Classic-B. In Event-B an event can be called at any time when the guard condition is true, meaning that events being called is largely a random process.

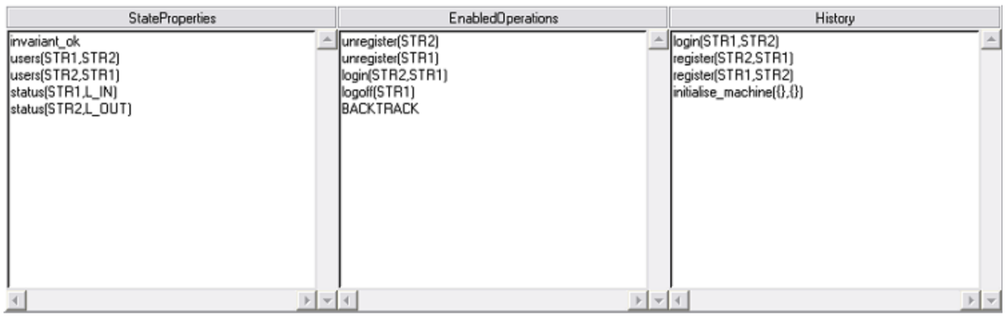
The B-Method has been used to solve several large formal system designs, including the design of the Paris Metro Line 14’s Train Control System [5].

2.1.2 ProB

ProB is an animation and model checking tool for the B-Method [15]. Currently ProB is used in Classic-B model checking, however there is an intention for it to be modified to work with Event-B and then to be added to the RODIN platform.

ProB allows the user to create a Classic-B model and then provides various methods for consistency checking. The consistency checking in ProB is used to check that the invariant is preserved throughout a given model's lifetime. If the checker manages to find a situation where the invariant has been violated, the checker stops and displays in the animator the current state for the model. In this project we are less concerned about the consistency checking element of ProB, other than for it to be the motivation for using the ProB animator.

The ProB animator, as shown in Figure 2.1, shows for a given state, the current states of all variables in the model, the available operations and the list of operations that were called to get to the given state. When an operation is called in the ProB animator, the state properties and the history are updated to reflect this and the invariant is recalculated.



StateProperties	EnabledOperations	History
invariant_ok	unregister(STR2)	login(STR1,STR2)
users(STR1,STR2)	unregister(STR1)	register(STR2,STR1)
users(STR2,STR1)	login(STR2,STR1)	register(STR1,STR2)
status(STR1,L_IN)	logoff(STR1)	initialise_machine({},{})
status(STR2,L_OUT)	BACKTRACK	

Figure 2.1: Output from the ProB animator showing the animation of a system monitoring users being registered to a system and whether they are logged in or logged off.

2.2 UML Design

The Unified Modelling Language or UML [18] is a diagrammatic notation originally developed in the 90's by the Object Modelling Group to support object-oriented software engineers through the entire development process. The Unified name comes from the fact that UML was one of the first notations to bring together both object-oriented analysis and design under a single model. With the help of significant early corporate backing, UML has become a de facto standard, almost universally recognised within the industry. Unlike formal methods, which have been met with some resistance, UML remains consistently popular and remains under development today [13].

The notation itself is very large, its current format (UML 2.0) encompasses 13 separate diagram types, each with well defined semantics. Even so, it is not considered to be strict. The UML philosophy generally aims to provide flexibility, being an enabling technique rather than a constricting one.

An example of its usage: A software engineer might conceptualise his system using Use-Case Diagrams, high level Class Diagrams and Activity Diagrams; perform detailed design it with Package Diagrams, lower level Class Diagrams and Deployment Diagrams; and produce blueprint designs with implementation level Class Diagrams, Timing Diagrams and Sequence Diagrams. This is, of course, just an example usage: there are no regulations compelling you to use certain diagrams only in specific situations.

2.3 UML-B

UML-B [25] is being developed to try and bridge the gap between the UML design model and formal methods techniques. UML is a widely accepted design format and many customers of software systems may expect to see it being used, whereas formal methods is often seen as being unnecessary and costly. In the work carried out by Amey [2] it was reported that customers of software systems often did not like the idea of formal methods being used, with some customers asking "couldn't you use UML?". UML-B brings the two together into a combined standard that provides the diagrams of UML, reinforced with added structure and formalisation. This allows B code to be much more easily constructed, via UML-B diagrams.

Currently UML-B has four UML style diagrams implemented, the Package, Context, Class and State Diagrams. By creating a package with a given context and classes, a system designer can start to design a system as they would in standard UML. By adding state diagrams to classes to show how the events in classes operate, the designer has got to a stage where they are close to defining the total functionality of a system. Finally a system designer must include an amount of formal mathematical logic, known as μB [25], to the class and state diagrams to define how events occur and the guards that have to be true in order for the operation to take place.

When a system designer produces a the required UML-B diagrams along with the events and guard conditions, UML-B can be used to automatically produce Event-B code through the U2B tool [23, 24].

2.4 Eclipse

The Eclipse Framework [6] is a project that started life as a product of IBM, aimed at replacing their VisualAge development environment. In common with VisualAge, the aim of the framework was to provide a platform on which integrated development environments (IDEs) can be based. Eclipse reached mass audience with the popularity [12] of its Java IDE, known as the JDT (Java Development Tool) [10]. However, this is only one of many plug-ins and its real power is as a platform.

Eclipse is now maintained independently from IBM by the Eclipse Foundation [26], a not-for-profit consortium of leading companies. It is one of the most popular development frameworks available today, with companies such as Adobe, Borland and SAP AG choosing it as the basis for their upcoming products [20, 9, 21].

2.4.1 RODIN

The RODIN (Rigorous Open Development Environment for Complex Systems) project [19], is being developed to provide an Eclipse-based platform and open, extensible, environment for formal systems specification. RODIN aims to encapsulate all aspects of formal system design and currently supports several plug-ins and components, such as UML-B and U2B.

With the RODIN UML-B plug-in, a user can define a system in UML-B through the Package, Machine, Class and Sequence diagrams. RODIN then automatically translates the UML-B to Event-B using the U2B plug-in. Details of how the translations are performed can be found in *UML-B: Formal Modelling and Design Aided by UML* [25]. During this process any syntax errors are returned to the user through the problem areas, which are highlighted in red. The completed Event-B code can then be sent to a prover where automatic and semi-automatic proving takes place.

2.4.2 EMF

The Eclipse Modelling Framework (EMF) [28] is a framework for building model-based tools and Eclipse plug-in applications. It includes facilities for both designing models and generating code that can be used as a back-end, upon which more advanced applications can be created. Diagram editors, especially structured ones such as those used in software engineering, are a classic example of a model based application. In this case the model's use is obvious; it is used to store all the semantic information associated with a diagram, abstracted away from the details of presentation or formatting. The framework is not limited to this use though, any Eclipse application that makes use of a model, definable on a class diagram, can take advantage of EMF.

By building an application based upon EMF, the developer enjoys a high level of compatibility with modern CASE tools such as Rational Rose. There is also the advantage of reduced development time, allowing the framework to quickly and reliable generate quite straightforward code, which avoids an otherwise time consuming and laborious process. This code can then be easily altered and maintained from the single model file, allowing potentially significant model changes to be made right the way through the implementation process.

2.4.3 GEF and GMF

The Graphical Editing Framework [30] is a product of the Eclipse Tools Project [27], a major project within Eclipse aimed at developing and promoting the development of significant Eclipse-based tools and support utilities. GEF is a framework for more easily constructing rich EMF-based Graphical editors. It achieves this by implementing an extensible API, based on an MVC architecture, which provides control and view components [3, 17] that utilise an EMF model. A major aspect of the framework is the Draw2D plug-in [14], which provides a diagram-centric layout and rendering toolkit, with an API specialised for creating diagram viewers and editors.

The theory behind GEF is that a programmer can develop a GEF implementation of an editor more rapidly than a bespoke implementation. An advantage of this method, besides time saving, is that by using an established approach the application will be smoother and less susceptible to bugs. However, GEF has a high initial learning curve and as a result one must become an expert with it, in order to develop usable tools.

To address this issue the Graphical Modelling Framework (GMF) [31] was conceived. GMF is an additional plug-in to Eclipse, which provides an API and simple toolset for automatically producing template GEF editors based on a supplied EMF model. Using GMF anyone, even without significant programming skills, can produce a basic diagram editor. The framework is broken down into four scripts, or models, that represent the different aspects of an editor plug-in. These are the: Graphical Model, Tools Model, Mapping Model and the Generator Model, which handles configuration of the plug-in itself. Each of these is composed of specific classes and associations that represent some element of the diagram that, when processed, will be translated into fully implemented GEF code. To ease the process further, a set of Wizards are provided to easily create the models in the correct sequence.

For more information on using GMF, see the official GMF tutorials [29]. Whilst initially useful, these peer-reviewed documents can be confusing and are frequently lacking in key areas. Therefore, to compliment this we have included our own overview of the GMF diagram creation process in Chapter 4.

Chapter 3

Requirements

The requirements of both the new diagram types and the associated development tools are detailed in this section. This reflects how the Project Brief will be realised and which aspects of the notations were assessed to be necessary and suitable for adoption into the UML-B notation. Any specific restrictions, extensions or changes made to the notations are also outlined. These requirements directly drive the structure of the new models and so were highly important in developing the new metamodels and editors.

3.1 Activity Diagram

The Activity Diagram must look like the UML 2.0 Activity Diagram [7], as shown in the example in Figure 3.1. The Activity Diagram must have elements for Initial Node, Activity, Fork, Join, Decision, Merge, Final Node, Flow Final along with a notion of a flow linking the elements. This differs slightly from the UML 2.0 Activity Diagram by the removal of partitions in the UML 2.0 definition of the Activity Diagram. The elements used in this Activity Diagram must look like their equivalents in the UML 2.0 Activity Diagram.

Activities must have labels associated to them, which are linked to the Activities name. The label must be positioned inside the center of the Activity element and must resize itself and the graphical component as the amount of text in the label increases. Flows going into Activities must also have labels that are linked to the Guard Conditions for the Activity the flow is going into.

All elements in the Activity Diagram have different requirements in terms of how many Flows the element can receive and how many Flows the element can transmit. The Initial Node allows no incoming Flows and at most one outgoing Flow. An Activity allows at most one incoming Flow and at most one outgoing Flow. A Fork allows one incoming Flow and many outgoing Flows. A Join allows many incoming Flows but only one outgoing Flow. A Decision allows one incoming Flow and many outgoing Flows. A Merge allows many incoming Flows but only one outgoing Flow. A Final Node allows only one incoming Flow and no outgoing Flow and the same is true for a Flow Final element.

Ideally there should be one tool in the Activity Diagram editor for creating a Flow and the editor should be able to determine the starting and ending elements to determine whether there are any constraints on the Flow.

There may only be at most one Initial Node in an Activity Diagram. Whilst there may be any number of the other elements in an Activity Diagram.

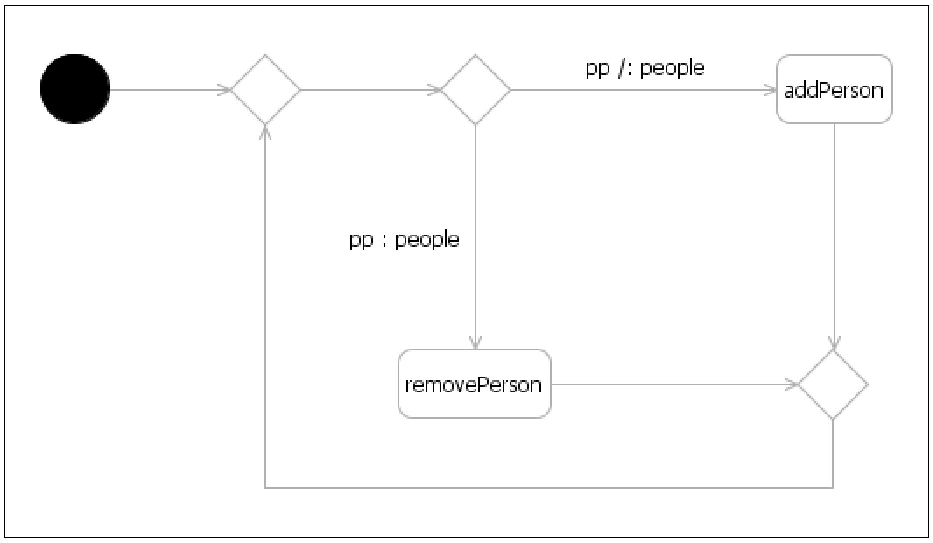


FIGURE 3.1: An example of a Activity Diagram for a database of people with μB added to the guard conditions for the activities. The Activities *addPerson* and *removePerson* are protected by the Guard Conditions $pp \notin people$ and $pp \in people$, which are defined using their ASCII equivalents.

3.1.1 UML-B Specific Requirements

The Activity Diagram will be integrated with the other UML-B diagrams at several points. These include with the UML-B Machine on the Package Diagram and on the Class element of the UML-B Class Diagram.

3.1.2 Uses in UML-B

The Activity Diagram is used more in early system design to determine how the system may operate. The Activity Diagram sits closely with the Use-Case Diagram to develop initial understanding on the problem being solved. In terms of relationships with UML-B, the Activity Diagram has not been given a specific purpose for processes such as U2B translation and is provided solely to help the user understand and model their system.

To allow flexibility of using the model, the Activity Diagram should be able to be defined in the machine and class levels of UML-B. Activity diagrams can also sit outside of a machine or class and be defined as their own entity.

3.2 Component Diagram

The Component Diagram must have a similar look and feel as a UML 2.0 Component Diagram, an example of which is shown in Figure 3.2. This requires that the Component Diagram uses Object, Required Interface and Provided Interface elements and Associations between both Components, and Required and Provided Interface pairs

Components should be represented by a rectangular box with a component icon in the top right corner and a *name* label. Components should also be able to nest within each other.

Provided Interfaces should be represented by an unfilled circle and are attached to a Component with a solid line. Required Interfaces should be represented by an open semicircle and are attached to a Component with a solid line. Both Interface types should have a *name* label.

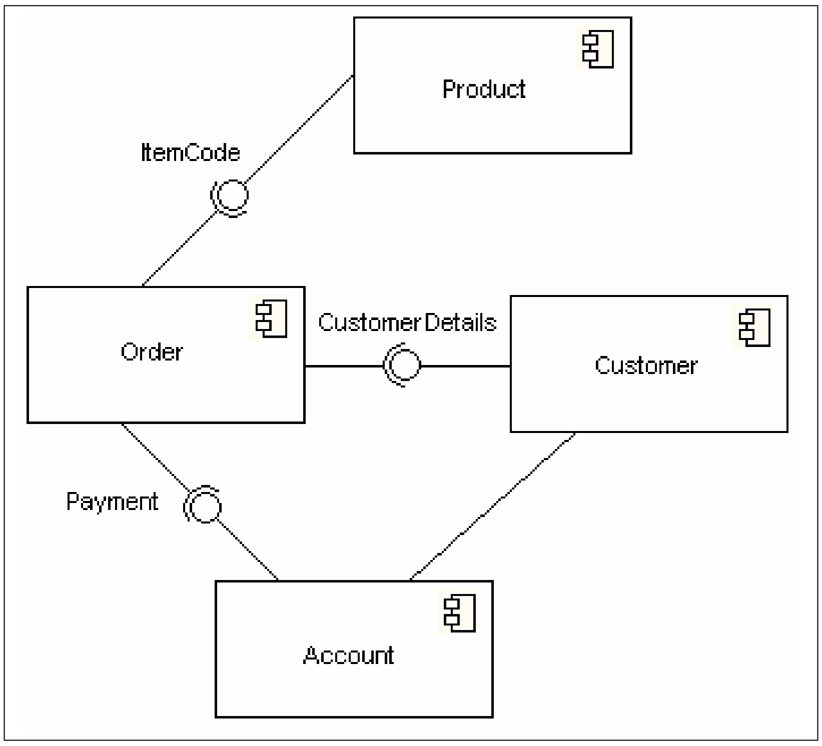


FIGURE 3.2: An example UML 2.0 Component Diagram showing the ‘ball and socket’ join

Associations between Interface pairs should consist of one Provided Interface connected to one Required Interface such that it results in the 'ball and socket' type join. Unconnected interfaces should be connected to their parent Component by a straight, perpendicular line and should be positioned close to their parent. The facility to connect together two unconnected interfaces to produce the 'ball and socket' join should be provided. This action should be able to be undone; such that a pair of connected interfaces revert to their unconnected state.

Associations between Components should be represented by a solid line with no arrow. The Association should have an optional label.

3.2.1 UML-B Specific Requirements

To integrate the Component Diagram with the UML-B tool it needs to be linked to a Package. The user should be able to right-click a package and create a new Component Diagram from the pop-up menu. Component Diagrams should also be creatable without the need for an existing Package.

3.2.2 Uses in UML-B

A Component is representative of a Machine or Group of Machines. The diagram can be used in UML-B to show the way the different parts of a system interact and what Interfaces are be needed. This will be useful when a UML-B system is in the refining process.

3.3 Object Diagram

The Object Diagram must have a similar look and feel as a standard UML 2.0 Object Diagram, as shown in Figure 3.3. This requires that the Object Diagram uses Object elements and Associations between Objects.

The Object element should be represented as a rectangular box with a separate compartment for the Object's attributes. The Object should also have a *Class Type* and a *Name*, which displayed as label with the format *Name: ClassType*. This should allow for Objects without a defined *Name*, but not for Objects without a defined *Class Type*. Attributes for Objects should be displayed as *AttributeName: Value*. This should allow for Attributes without a *Value*.

Associations between Objects should be represented by a solid line with no arrow head and should have an optional label to describe the relationship between the Objects.

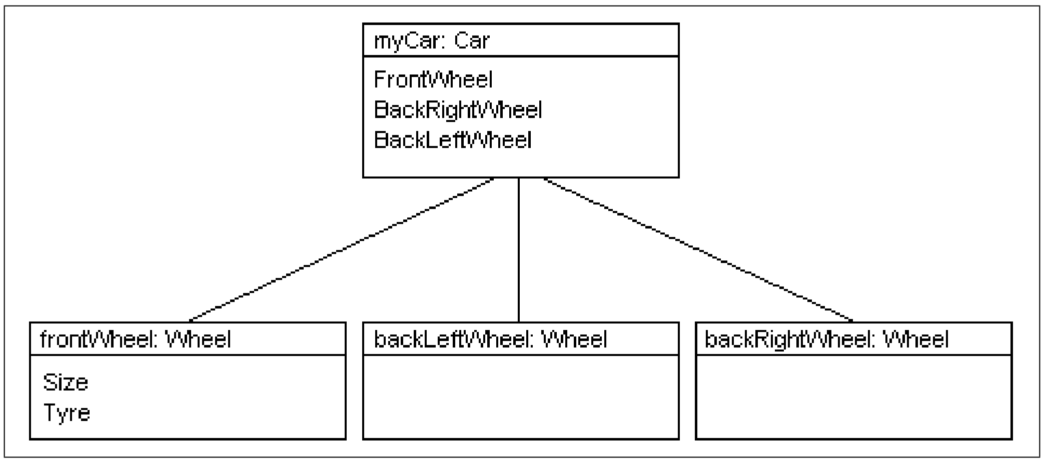


Figure 3.3: An example UML 2.0 Object Diagram showing the make up of a system modelling a car.

3.3.1 UML-B Specific Requirements

To integrate the Object Diagram into the UML-B tool it must sit with the other available diagrams and complement their functionality. This will be achieved by linking the Object Diagram to the existing UML-B tool in the several ways.

The Object Diagram should be attached to Machines and Contexts as it will provide modelling capabilities for both of these UML-B components. It should therefore be possible for the user to create a new Object Diagram through the right-click menu options of a Machine or Context and for the created diagram to be linked to the parent. The XMI produced from the Component Diagram should be contained within the existing .umlb XMI file.

The Object Diagram should allow the user to create a new Object as an instance of an existing UML-B Class. This should be integrated into the right-click pop-up menu such that the option is presented when the user right-clicks on an empty space in the diagram.

3.3.2 Uses in UML-B

The Object Diagram is used to model a static snapshot of a dynamic system. This gives it several possible uses in UML-B. Firstly, the diagram can be used as a model of a complex system of entities to get an idea of ther structure as a first step in designing a system. Secondly, the diagram can be used to model the relationship of Objects inside a UML-B Machine or Context. This could be useful to explain the way in which Classes interact with each other. Thirdly, the Object Diagram can be used as a supplementary diagram when a system is being animated. The diagram can show the state of the system at any point during the animation and can be linked to an Event Call in a Sequence Diagram to show the state of the system up to that point in the diagram.

3.4 Sequence Diagram

The Sequence Diagram is used to depict a sequence of progressions and transitions between various elements on the diagram, which are arranged as swim lanes running from top to bottom (see Figure 3.4). In UML 2.0 the semantics correspond to Object Oriented Design. The diagram shows method calls between Objects in a system. This is useful in modelling the sequence of operations that make up some larger task or process.

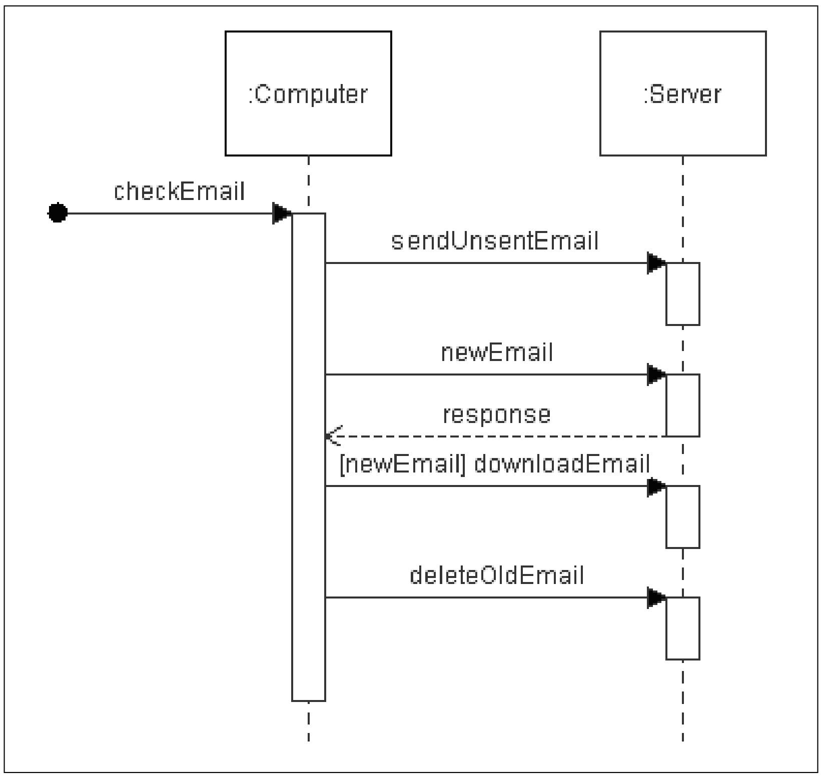


FIGURE 3.4: An example of a UML 2.0 Sequence Diagram showing the sequence of events involved in an email system.

Syntactically, the Sequence Diagram is to look much the same as the UML 2.0 Sequence Diagram. The swim lanes represent Objects in the system (instances of the UML-B Classes). These are represented by a yellow box containing the Class name and Object label, with a dotted line that extends vertically downwards, terminating with an X. The length of the line can be extended to accommodate varying lengths of diagram.

The activity boxes, which in UML 2.0 represent periods of activity within the Objects are included in the diagram. However, in terms of UML-B, these are more to complete the look of the diagram than to add meaning. With the B language not being Object

Oriented, there is no real notion of Object life or activity. The Classes present in the UML-B diagram, when translated, are recreated procedurally using sets. The activity boxes are useful however, for providing an element to anchor links to.

There are two types of link present on the UML-B Sequence Diagram. Standard links, which reflect method calls in UML 2.0, now represent Event Calls (the closest equivalent to methods in Event-B). These are represented by a solid horizontal line, which runs from an area of activity in one object to that in another. The target end of the link is decorated by an open arrow, as in the UML 2.0 version. The other type of link is a dotted line, in UML 2.0 this represents a method return. In UML-B this notation is included, partially to complete the illusion of object orientation and partly as it could correspond to an animator backtrack.

Of all the UML 2.0 diagrams, the Sequence Diagram is probably the most restricted in terms of appearance. The swim lanes, which correspond to Class instances, or Objects, should all be aligned on the same horizontal axis, with their upper boundaries at a fixed distance from the edge of the canvas. The user may arrange the swim lane positions horizontally, with the ability to change the spacing between them, but not move vertically. The activity boxes are locked to the centre of the swim lane's dotted line, they can be resized vertically but not horizontally.

3.4.1 UML-B Specific Requirements

The layout of the links on the diagram are to be more formalised than in UML 2.0. In UML 2.0 there are little rules or restrictions on how or where they can be placed. In UML-B the links are restricted so that they are also horizontal lines and there is a strict notion of sequence, which is the key aspect of the Sequence Diagram to record for output. Only one link may occupy any one vertical level on the diagram, this removes an ambiguity in determining the order of occurrence. The diagram editor will have to enforce this, providing functionality to automatically position and organise the links, depending on the order in which the user wishes to place them. The order will also need to be recorded persistently in the model file.

The diagram will be integrated with the existing notations. This will occur at two points. Firstly there will be a method of linking the diagram to Machines within the UML-B Package Diagram. A Machine effectively represents a collection of Classes, so it makes sense to link sequences of calls between these Classes to the relevant parent Machine. Secondly, within the Sequence Diagram itself there will be the ability to link the Object swim lanes to their corresponding Class elements and the Event Call links to their corresponding Events, both of which will have been defined via the other diagrams.

3.4.2 Uses in UML-B

In UML-B we can use the sequence depicted in the diagram to actually drive a test animation (see Section 7.1), which can then be run to verify the validity of the model. Achieving this would be relatively straightforward; an animation script could be generated simply by reading the order that the Event Calls are recorded in.

3.5 Use-Case Diagram

The Use-Case Diagram must look like the UML 2.0 Use-Case Diagram [7], as shown in the example in Figure 3.5. Therefore the Use-Case Diagram must have elements for Actors, Use-Cases and System Boxes. The elements in the Use-Case Diagram must look like the elements in the UML 2.0 Use-Case diagram.

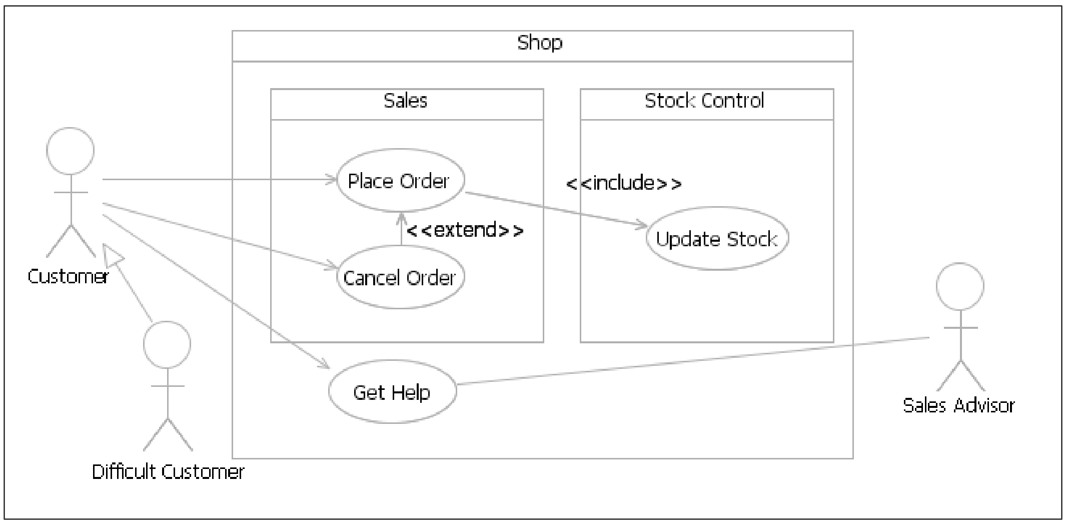


Figure 3.5: An example of a Use-Case diagram for a shop

The Actor, Use-Case and System Box elements must also have labels that are associated to the to a property of the element. The labels must be positioned differently for each element in the Use-Case Diagram. The label in the Actor element is linked to the Actor's *name*. The Actor label must sit below the actor and resize itself automatically if the text in the label requires it to do so, without resizing the actor graphical figure. The Use-Case label is linked to the Use-Case's *description*. The Use-Case label must be centered inside the Use-Case graphical figure and the label must resize itself and the Use-Case figure as the amount of text in the label increases. The System Box label is linked to the System Box's *name*. The System Box labels position is not critical so long as it can be read and distinguished as a label for the System Box.

The System Box must allow other System Box and Use-Case elements to be nested inside of them as is allowed in the UML 2.0 Use-Case Diagram. The nesting must not interfere with connections between Use-Cases or Actors. Actors cannot be nested inside of a System Box.

Actors representing the system are not a special element of the Use-Case diagram. To create a System Actor, the system designer must add <<system>> to the actor's name label.

The links between elements in the Use-Case Diagram are defined as being: a link from an Actor to a Use-Case; a link from a Use-Case to an Actor; a link between an Actor and another Actor and three types of links from Use-Case to Use-Case, which are *include*, *extend* and *generalize*. All links in the Use-Case Diagram exist as a many to many relationship. The link from an Actor to a Use-Case show where an Actor is *using* a particular Use-Case, this link is a continuous line with an open ended arrow head pointing at the Use-Case. The link from a Use-Case to an Actor shows when a Use-Case invokes or requires another Actor, this link is a continuous line with no arrow head. The link between two Actor elements is used to show one Actor *generalizing* another Actor, the link is a continuous line with a closed arrow head at the end. The *include* link between two Use-Case elements shows where one Use-Case must invoke another Use-Case for it to complete its operation, this link is a continuous line with an open arrow head pointing at the Use-Case being included. The *include* link also requires a pre-set uneditable label that contains the text "<<include>>". The *extend* link between two Use-Case elements shows where one Use-Case is an extension to the behaviours described in the Use-Case being pointed to. The *extend* link is a continuous line with an open arrow head pointing at the Use-Case being extended. The *extend* link also requires a pre-set uneditable label that contains the text "<<extend>>". The *generalize* link between two Use-Case elements allows the generalization of the Use-Case being pointed to. The *generalize* link requires a continuous line with a closed arrow head pointing at the Use-Case being generalized.

3.5.1 UML-B Specific Requirements

The Use-Case diagram will have no semantic meaning within UML-B in terms of U2B translation, it is primarily a support notation (see Chapter 7). Use-Case diagrams should be able to be created from both the package and machine elements.

3.5.2 Uses in UML-B

The Use-Case diagram is used as a high level system design tool and as such has no formal role in UML-B in terms of U2B translation. The purpose of the Use-Case Diagram is to help the system designer in the design process and is included to help UML-B be a complete solution in system design.

The Use-Case Diagram should be able to be created from the package and machine level in UML-B. The Use-Case Diagram should also work as a standalone diagram type within a UML-B project.

Chapter 4

Editor Construction

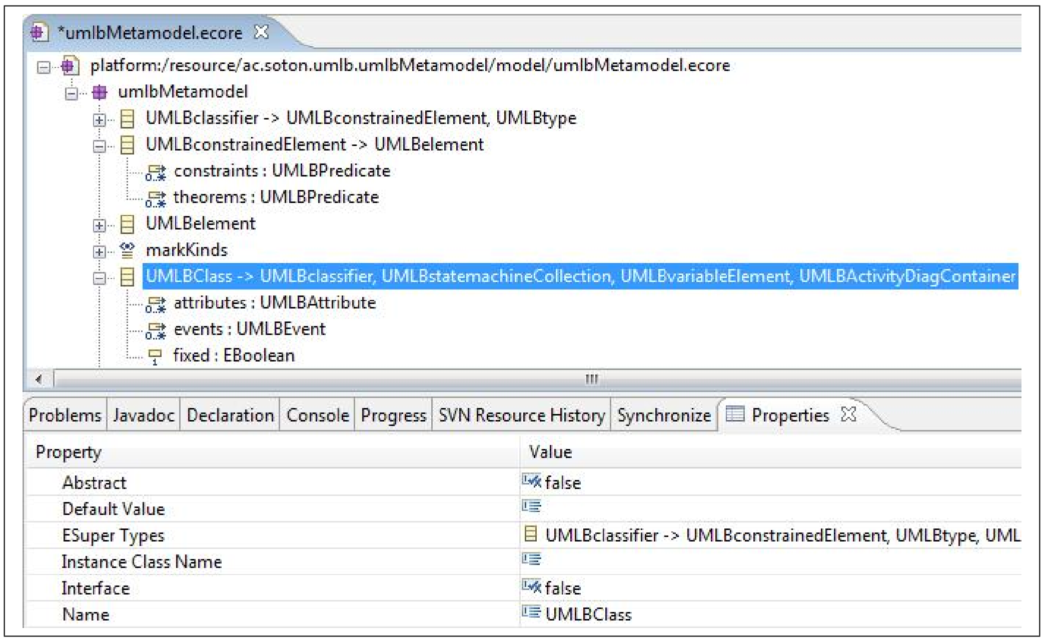
There are several steps that need to be taken to construct a diagram editor in the GMF framework. This process starts with the definition of a metamodel for the diagram and then moves into the definition of the Graphics, Tooling and Mapping models. A Genmodel is created from the Mapping model and is used to generate the diagram code. A short overview of diagram editor construction in GMF is given in the following section.

4.1 Metamodel

The EMF Metamodel for a diagram defines the way in which the diagram nodes can interact with each other. The metamodel XMI format is known as an Ecore file, which is a standard that exists outside of Eclipse; as such it is possible to define the metamodel externally, using a tool such as Rational RSA. Within Eclipse there are three options for editing the model: directly editing the XMI, which is useful for copying large blocks of the model between files; editing the model via the EMF tree editor, which provides a methodical and unambiguous approach (shown in Figure 4.1); and lastly using the GMF editor, this allows the model to be created using a class diagram-like viewer, which is extremely useful in aiding visualisation and conception (it is this viewer that the metamodel screenshots have been taken from). The nodes are specified as EClasses in the metamodel, which can have a number of EAttributes, EReferences and EOperations.

EAttributes are used to define the attributes that are desired in the diagram class and have to be given both a name and a type. Attribute types can be any of the standard Java types, such as String, Long or Boolean. The name of the attribute is not restricted but it is often good practice to name it in such a way that it can be differentiated from similar attribute names in different classes.

EReferences are used to define the relationships between one class and another, or between one class and itself. Setting a reference between two classes or back to the same



```
*umlbMetamodel.ecore
platform:/resource/ac.soton.umlb.umlbMetamodel/model/umlbMetamodel.ecore
  umlbMetamodel
    UMLBclassifier -> UMLBconstrainedElement, UMLBtype
    UMLBconstrainedElement -> UMLBelement
      constraints : UMLBPredicate
      theorems : UMLBPredicate
    UMLBelement
    markKinds
    UMLBClass -> UMLBclassifier, UMLBstatemachineCollection, UMLBvariableElement, UMLBActivityDiagContainer
      attributes : UMLBAttribute
      events : UMLBEvent
      fixed : EBoolean
```

Property	Value
Abstract	false
Default Value	
ESuper Types	UMLBclassifier -> UMLBconstrainedElement, UMLBtype, UML
Instance Class Name	
Interface	false
Name	UMLBClass

FIGURE 4.1: An extract from the EMF tree editor

class allows the classes to be linked with connectors in the diagram editor. Links can also be modelled by having a canonical EClass representing the link, this would then need to have two EReferences, one to the target node and one to the source node. EReferences have several properties that need to be set in order for them to work properly.

The *name* property must be set and it is good practice to give the reference a uniquely identifiable name within the metamodel to make it easier to identify at later stages, leaving this blank can cause errors during code generation. The *eType* of the reference must be set to the target class, which will also set the *eReference type* property. The *upper bound* and *lower bound* can both be set to restrict the cardinality of the reference. Setting the *upper bound* to -1 allows for an unlimited upper bound. The *containment* property can be set to true or false. Setting it to true allows the target of the reference to be included as a child node in the parent of the reference. This is useful if a recursive nesting behaviour is required; the required class can have a self referencing reference with the containment set to true and the resulting node in the diagram will be capable of recursive nesting.

EOperations are used to define some behaviour for a class in the metamodel that is not determined solely in the metamodel design. There are various ways in which EOperations can be used, such as creating a method in the Java output for the class that the EOperation is defined in. EMF automatically adds methods for any EOperations that exist in a class, into the implementation code for the class. This method is then updated by changing the *@generated* tag to *@generated NOT*, to stop EMF overwriting the method at a later date, and by adding the desired code to the body of the method. Any EOperations defined in the metamodel must then be explicitly called from elsewhere.

For example it is possible to define link constraints in GMF that are represented in Java code that are used to call the classes defined using EOperations.

Once the metamodel has been constructed, an EMF GenModel must be created from the metamodel. This is available through the right-click menu on the metamodel file (.ecore extension), in the Eclipse Package Explorer window. Generally the GenModel does not require alteration. The Model Code and Edit Code need to be generated from the EMF GenModel before continuing, this can be performed using operations provided by the right-click menu on the root node of the EMF Model.

4.2 Graphical Model

The Graphical Model is used within the GMF framework to define the graphical elements of the diagram. This includes nodes, labels, connections and decorations for connection ends. The Graphical Model contains a Figure Gallery that contains the figures that are used to define the shapes; the elements that define the nodes, labels and connections are under the root of the file.

GMF provides a simple wizard for the creation of Graphical Models, which is available through the right-click menu on the metamodel file. The wizard allows the desired nodes, labels and connections to be selected, and will automatically choose a sensible set of defaults when provided with a root class. The output from the wizard is a simple Graphical Model that contains figures and elements for the selected classes, labels and connections. The generated Graphical Model can be heavily modified to produce the required behaviour.

There are three major modifications that can be made to the Graphical Mapping. The first is to change the default graphical representation of the nodes. This is accomplished by defining a new figure in the Figure Gallery and setting the appropriate node to use the new figure. New figures can either be one of those available by default, such as Rounded Rectangles, Ellipses and Polygons, or they can be Custom Figures. To use a Custom Figure a Java class needs to be defined that extends the *org.eclipse.draw2d.Graphics.Shape* class and overrides the *fillShape()* and *outlineShape()* methods. The custom class then needs to be referenced by the Custom Figure.

The second major modification is to add a Compartment that can be used in the Mapping Model. This is a two stage process; the Compartment must be added below the root of the Graphical Model and a Figure needs to be defined for it. Compartments need to be named, and can optionally be forced to display this name in the diagram editor.

The third major modification is to modify the style of the Connections. There are two options for this: modifying the style of the line or modifying the style of the Decorations that appear at the source and target ends of the Connection. The line is easily modifiable through the properties that define it in the Graphical Model. Several properties can be changed, such as *line kind*, *line width* and the graphical elements used for source and target Decorations. The Decoration is more difficult to modify and often involves the creation of a Custom Decoration in the same way as a Custom Figure is created.

4.3 Tooling Model

The Tooling Model is used within GMF to define the toolbar and menus to be used with the diagram editor. The Tooling Model allows several types of menu to be included, such as Context, Pop-up and Main Menus. The toolbar, which is the main focus of the Tooling Model, is defined within a Palette and contains Tool Groups, which contain the Tools.

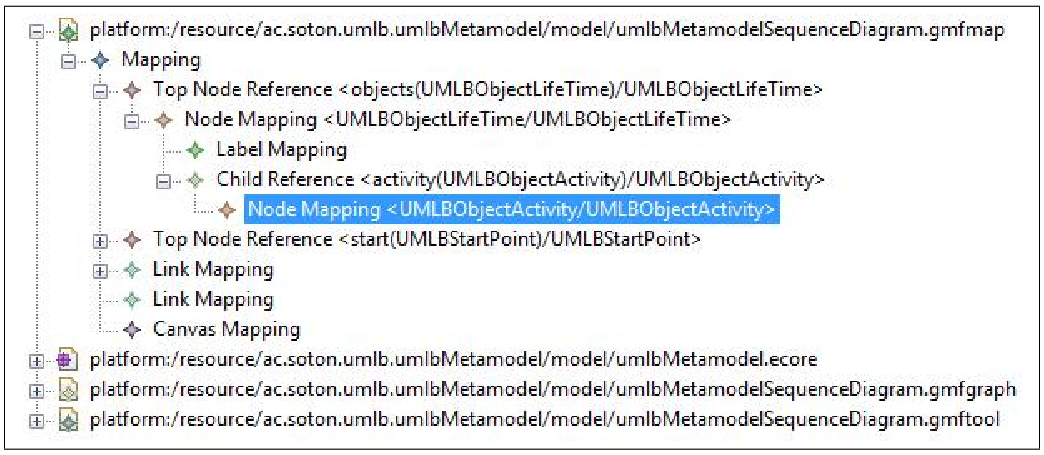
GMF again provides a wizard to generate a Tooling model, which is available through the right-click menu on the metamodel file. The wizard produces a reasonable set of defaults that can be modified to provide the required behaviour. The most common modification of the Tooling model is to split the Tool Groups into different categories and to change the names on the Tools, this is easily performed.

4.4 Mapping Model

The Mapping Model is used within GMF to map the Graphical and Tooling Models to the metamodel. The Mapping Model can be automatically generated through the wizard, which GMF makes available, but this is slightly unreliable and has a tendency to associate a creation tool for one element with the graphical definition for a different element, so the resulting Mapping Model must be carefully checked.

The Mapping Model consists of several Top Node References, which each contain one Node Mapping and are used to map nodes in the diagram to both the element in the metamodel and to the creation tool. It is within the Node Mapping that Label Mappings, Child References and Compartment Mappings are defined.

Label Mappings map a Diagram Label in the Graphical Model to one or more attributes in the metamodel class that is referenced by the enclosing Node Mapping. Compartment Mappings map a Compartment from the Graphical Model into the metamodel class referenced by the enclosing Node Mapping. Compartments can also have children, which are specified as Child References of the Node Mapping. Child References allow metamodel classes to have children and as such contain an inner Node Mapping, or a reference to an already defined Node Mapping. An example of this is shown in Figure 4.2, the UMLBObjectActivity node is nested within the UMLBObjectLifeTime node.



```
platform:/resource/ac.soton.umlb.umlbMetamodel/model/umlbMetamodelSequenceDiagram.gmfmap
  Mapping
    Top Node Reference <objects(UMLBObjectLifeTime)/UMLBObjectLifeTime>
      Node Mapping <UMLBObjectLifeTime/UMLBObjectLifeTime>
        Label Mapping
        Child Reference <activity(UMLBObjectActivity)/UMLBObjectActivity>
          Node Mapping <UMLBObjectActivity/UMLBObjectActivity>
    Top Node Reference <start(UMLBStartPoint)/UMLBStartPoint>
    Link Mapping
    Link Mapping
    Canvas Mapping
platform:/resource/ac.soton.umlb.umlbMetamodel/model/umlbMetamodel.ecore
platform:/resource/ac.soton.umlb.umlbMetamodel/model/umlbMetamodelSequenceDiagram.gmfgraph
platform:/resource/ac.soton.umlb.umlbMetamodel/model/umlbMetamodelSequenceDiagram.gmftool
```

FIGURE 4.2: An extract from Sequence Diagram GMF Map, showing a nested node.

Once the Mapping Model has been completed it is used to create the Generation Model. This is done through the right-click menu of the Mapping Model.

4.5 Generation Model

The Generation Model is used within GMF to produce the diagram code. As such, it contains many features that can be modified to produce specific features in the diagram code. Figures defined the Graphical Model can be given a *default size* property at this stage for example.

The Generation Model is not normally modified as most of the functionality should be included in the prior diagrams. One common modification however is to change the package location that the diagram code is generated into. This becomes necessary if there are several diagrams being produced from one metamodel, to avoid one diagram overwriting another; it also helps to have the digram code in separate packages to make the addition and maintenance of custom code easier. The generation package is modified by changing the Gen Editor's *package name prefix* and the Gen Plugin's *ID* to the desired package name.

4.6 Extensions and Code Customization

The use of GMF with EMF allows the rapid construction of a wide range of different diagrams. However, there is a limit to the level of complexity that the framework can provide. For instance, if one wanted to create a diagram with some specialised graphics, formatting or behaviour, there would be very little support in GMF.

The solution to creating more complex diagrams is to extend what GMF produces. There are a number of ways of achieving this. GMF and EMF do themselves provide some support for extensions. In EMF it is possible to add custom constraints to the nodes. An example of this is seen in the Activity Diagram Metamodel (Section 5.1.1). Adding custom constraints creates place-holder methods when the model code is generated, by adding code to these methods constraint behaviour can be customised.

GMF provides inbuilt extension mechanisms within the GMF Graph. Here it is possible to provide custom implementations of key features at certain points. For instance, it is possible to create a custom graphical representation for any of the elements on the diagram. This is done by creating a Java class that draws the shape of the element exactly as desired (see Section 5.4.2.1 for an example). It is also possible to specify a custom layout for a container, so that elements that appear within that container can be laid out in a specific manner (again, the sequence diagram shows an example of this).

Beyond these two mechanisms, it is not possible to obtain additional complexity through standard routes. However, as GMF is an API on top of GEF, it is possible to manipulate the GEF code and Eclipse plug-in configuration directly. These are the files that are generated by the GMF Generator Model and at this level it is possible to widely alter the behaviour of the editor, within the constraints of GEF. Performing this requires an intimate knowledge of the GEF framework and how it works. There is not space within this report to provide this, however some example modifications and explanations can be found within the sequence diagram (Section 5.4).

Chapter 5

Design

Each editor was developed using the process outlined in Chapter 4. The design details of each, together with any associated extensions or technical challenges are presented in this chapter. The metamodels of each diagram are presented individually (see Chapter 5) but are in fact unified under a single hierarchy. This allows an entire UML-B project to be carried out using a single output model file, increasing the level of cohesion and portability.

5.1 Activity Diagram

The Activity Diagram consists of many element types: Initial Node, Activity, Decision, Merge, Join, Fork, Final Node and Flow Final. The Activity Diagram also consists of a single Flow link type that is constrained in different ways for each element type.

5.1.1 Metamodel

The base class in the Activity Diagram Metamodel, shown in Figure 5.1, is the UML-BActivityDiagram. The diagram type is defined as being made up of multiple Merges, Joins, activities, Final Nodes, Flow Finals, Forks, Decisions and Flows. The diagram also allows at most one Initial Node.

The nodes directly aggregated from UMLBActivityDiagram are the components of the diagram, for example UMLB_AD_InitialNode represents the Initial Node component in the Activity Diagram. The Merge, Join, Activity, Final Node, Flow Final, Fork, Decision and Flow elements have no maximum limitation on the number of times they can appear in the diagram. The Initial Node element is restricted through the *initial* aggregation to at most one Initial Node element per Activity Diagram.

The Activity element, UMLB_AD_Activity contains a *name* property of type EString. The name property is designed to allow the Activity to have a name, that then links to the label inside of an Activity in the diagram. The Flow element, UMLB_AD_Flow contains a *guard* property also of type EString. This allows guard conditions to be added to the Flows, which are used when a Flow enters an Activity.

To solve the problem of keeping track of the Flows allowed into and out of a diagram element, the metamodel designed has four restrictive classes based on incoming and outgoing properties for Flows. Flows in the metamodel are defined as the UMLB_AD_Flow class and are associated to the UMLB_AD_Element class, which is the base class for all components that can be added in the Activity Diagram. The associations *incoming* and *outgoing* are used to define the set of Flows that an Activity Diagram element has incoming and outgoing. The associations *target* and *source* are the opposite associations to *incoming* and *outgoing* respectively. In the metamodel, the *incoming* and *outgoing* associations have no upper limit set, meaning that for any element in the Activity Diagram, the element can have multiple incoming Flows and multiple outgoing Flows. Obviously we do not want all elements of the Activity Diagram to have multiple incoming and outgoing Flows, therefore we must restrict this. In the metamodel, Flow restrictions are defined by the classes UMLB_AD_Element_NoIncoming, UMLB_AD_Element_OneIncoming, UMLB_AD_Element_NoOutgoing and UMLB_AD_Element_OneOutgoing, where the restrictions are no incoming Flows, one incoming Flow, no outgoing Flows and one outgoing Flow respectively. Elements are restricted in the metamodel by generalizing one or more of these restrictive classes.

The UMLB_AD_Element base class for all diagram components provides two methods which return a boolean depending on whether a node is legal to start from, or to end at. The method *isLegalToStartFrom* checks to see if the given element is of type UMLB_AD_Element_NoOutgoing, if it is then false is returned as the element cannot accept any Flows. Next the *isLegalToStartFrom* method will check to see if the element is of type UMLB_AD_Element_OneOutgoing, if it is and the number of outgoing Flows is zero then the element is legal to start a Flow from. The method *isLegalToEnd* checks to see if the given element can have another incoming Flow assigned to it. Firstly *isLegalToEnd* checks to see if the element is of type UMLB_AD_Element_NoIncoming, if it is false is returned as the element cannot accept incoming Flows. Next *isLegalToEnd* checks to see if the element is of type UMLB_AD_Element_OneIncoming, if it is the *incoming* Flows is checked to see if it is empty and so able to accept this incoming Flow, the result of this is returned.

5.1.2 Activity

An Activity is a rounded rectangle with a label corresponding to the *name* attribute defined in UMLB_AD_Activty in the metamodel. An Activity is only allowed to have

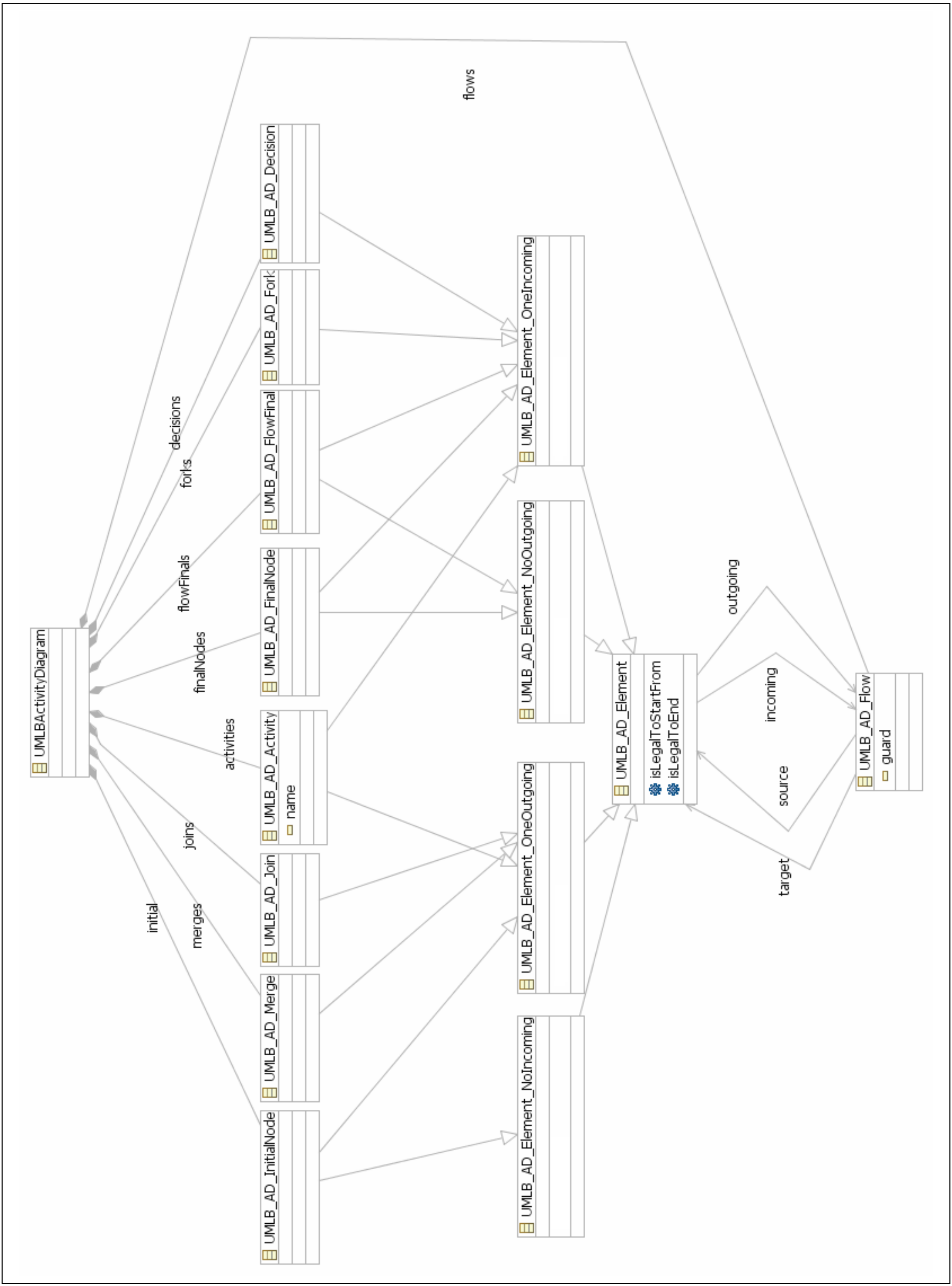


FIGURE 5.1: The Activity Diagram Metamodel

a single incoming Flow and a single outgoing Flow. The incoming Flow should contain the guard conditions for the Activity and this ability to do this is provided by the Flow object in the metamodel. An Activity is only allowed one incoming Flow and one outgoing Flow, which is restricted in the metamodel.

5.1.3 Initial Node

The Initial Node is a filled circle and is defined in the metamodel as UMLB_AD_InitialNode. The Initial Node is restricted in the metamodel so that it has no incoming and only one outgoing Flow.

5.1.4 Decision

The Decision node is a diamond shaped element and is defined in the metamodel as UMLB_AD_Decision. Decisions are restricted in the metamodel so that they have one incoming Flow, the outgoing Flows are not restricted as multiple outgoing Flows are allowed.

5.1.5 Merge

The Merge node defined in the metamodel as UMLB_AD_Merge, is graphically equivalent to the Decision node and so uses the same graphical component in the GMF graph. The Merge element is not restricted in its incoming Flows so as to allow multiple incoming Flows, whilst the outgoing Flows are restricted so that only one outgoing Flow is allowed.

5.1.6 Fork

The Fork node is a thin, filled rectangular bar that can be positioned horizontally or vertically in a standard UML 2.0 Activity Diagram. In this Activity Diagram, the Fork graphical component is by default a vertical bar however the user can resize and reshape the bar to allow them to produce a horizontal bar. The Fork element has its incoming Flows restricted so that only one incoming Flow is allowed, whilst the outgoing Flows are not restricted thus allowing multiple outgoing Flows.

5.1.7 Join

The Join element, defined in the metamodel as UMLB_AD_Join, is graphically equivalent to Fork and so uses the same graphical component. The Join element has no restrictions on the incoming Flows so as to allow many incoming Flows, whilst the outgoing Flows are restricted so as to allow only one outgoing Flow.

5.1.8 Flow Final

The Flow Final element is graphically defined as a circle with a cross through through it, in the metamodel Flow Final is defined as UMLB_AD_FlowFinal. The size of the Flow

Final graphical component cannot be changed by the user, as GMF does not resize the cross in relation to the size of the circle meaning that resizing the figure would produce an unrecognisable diagram element. The Flow Final element is restricted so that it accepts only on incoming Flow and no outgoing Flows.

5.1.9 Final Node

The Final Node element is defined graphically as a circle with a second filled inner circle, in the metamodel Final Node is defined as UMLB_AD_FinalNode. The Final Node element is restricted so that only one incoming Flow is accepted and no outgoing Flows are allowed.

5.1.10 Flow

The Flow element is the most complex part of the Activity Diagram to define, due to there being many restrictions on the Flows that Activity Diagrams can have incoming and outgoing. In the metamodel, Flows are defined as UMLB_AD_Flow and have one property, the *guard*, which should be used to show the guard conditions going into an Activity. Graphically, a Flow is a line with an open arrow head pointing in the direction of the Flow, but it also contains a label element that is linked to the guard attribute.

In GMF, Flows are assigned two link constraints, one for incoming and one for outgoing nodes. These constraints are defined in Java in the diagram code and when the constraints are called by the editor, the methods *isLegalToStartFrom* or *isLegalToEnd* are called depending on whether the Flow is trying to start from or connect to a graphical component. These constraints determine whether the editor allows a Flow to be placed in the model.

5.1.11 Technical Challenge

The Activity Diagram presented a technical challenge in terms of defining the restrictions on Flows between Activity Diagram elements in the metamodel. Activity Diagrams have various different restrictions on the number of Flows that a diagram element can have incoming and outgoing, which is difficult to define in the metamodel.

Initially the Activity Diagram Metamodel was produced so that every diagram element type generalized the same parent base class that had a single Flow association referencing itself, see Figure 5.2. The problem with this approach was that it meant any element could have any number of incoming outgoing Flows, which is not the behaviour required.

Another way that was looked at to try and control Flows was to try and define all of the links that are possible between all pairs of elements. By doing this you would be able

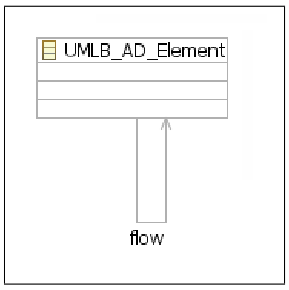


FIGURE 5.2: Initial implementation of defining a Flow in the Activity Diagram. All Activity Diagram elements generalize the UMLB_AD_Element.

to define the relationship constraints, such as one to one, one to many. However this method has three disadvantages to it, the first disadvantage is that the metamodel would become extremely complicated with such a high number of associations between classes. The second disadvantage is that GMF would create a separate link in the diagrams editor for each and every association that defines a link in the metamodel, meaning it would make the editor extremely unintuitive for the user. Finally, this way of defining the metamodel would have the problem that you would still be able to have multiple Flows coming out of an element, such as an Initial Node, where you only want at most one Flow to come out of. Multiple Flows coming out of an Initial Node would be legal in this type of model as there would be a link from the Initial Node to multiple other nodes.

Looking at the original UML-B diagrams found that the State Diagram had a similar requirement on counting the number of links diagram elements had going in and coming out. The portion of the Activity Diagram metamodel shown in Figure 5.3, shows how the metamodel has to be defined in order to count the number of incoming and outgoing Flows. In the metamodel the *incoming* and *outgoing* associations will contain all of the Flows that are going into and coming out of the given diagram element.

The final problem was to use the information about the number of incoming and outgoing Flows to try and constrain whether or not any given diagram element can have a Flow assigned to them. In Figure 5.3 there are the operations *isLegalToStartFrom* and *isLegalToEnd*, which are used to determine whether or not Flows can leave or enter a given element. The code in these operations could have been defined for each and every diagram element, however this would have created a large amount of code, with little indication in the metamodel of what was happening. This is why the final metamodel has the constraint classes talked about in Section 5.1.1, so as to show in the metamodel what the Flow constraints for each element are.

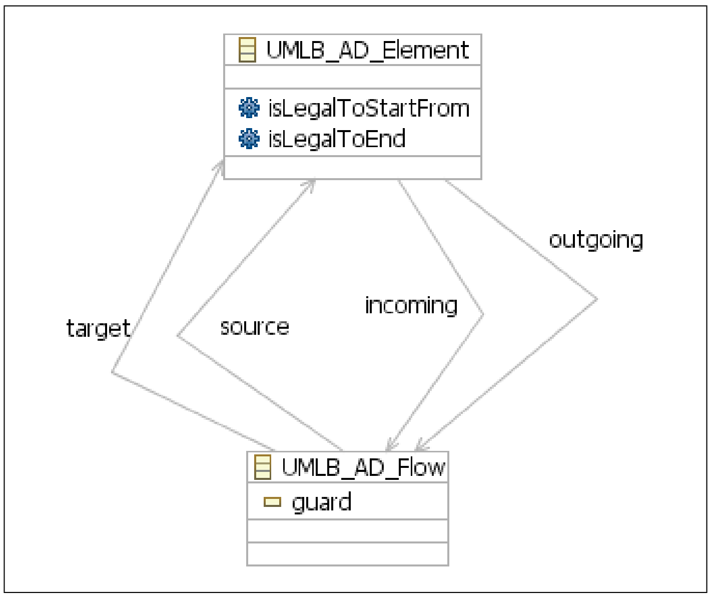


Figure 5.3: Part of the Activity Diagram Metamodel that demonstrates how incoming and outgoing Flows can be differentiated. The associations *source* and *target* are define whether a given Flow is coming from the element of going into the element. The associations *incoming* and *outgoing* define the set of incoming and the set of outgoing Flows an element has.

5.2 Component Diagram

The Component Diagram consists of Components, Required Interfaces, Provided Interfaces, associations between Required and Provided Interfaces, and associations between Objects. The Component Diagram is used to show how a system is built up and how the different parts interact. In UML-B this is at the Package level, with a Component representing a Machine or Group of Machines. The design of the Metamodel and implementation of the Editor are explained in the following section.

5.2.1 Metamodel

To get the required graphical behaviour for the Component Diagram the Metamodel shown in Figure 5.4 was created. This metamodel is stand alone but has been included in the UML-B Metamodel for the sake of completeness.

The UMLBComponentDiagram class is the main root of the diagram and is aggregated to the UMLBComponent class. The UMLBComponent class is the main class for the diagram as it is the representation of a Component. Both interfaces are aggregated to the UMLBComponent to allow them to be used as child nodes when the Component Diagram is being used. The Compartment has two self referencing links; the aggregation CD_Subcomponent, which allows for the nesting of Components, and the association

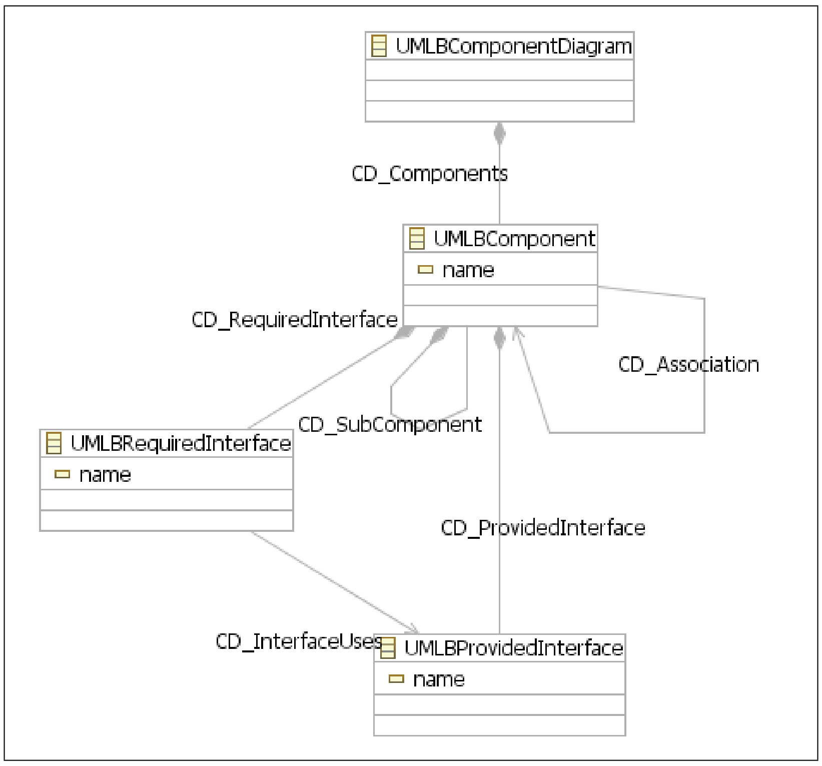


FIGURE 5.4: The Component Diagram Metamodel

CD_Association, which allows Components to have associations that do not involve interfaces.

Having Interfaces as children of a Component will make using the XMI output of the diagram easier in translation to B, because the process of identifying the interfaces that belong to a Component is straight forward. The child Interfaces are part of the parent Component's XMI node, which would not be the case if the Interfaces were not children. In this case, the links between Components and Interfaces would have to be looked at, which would be more difficult and time consuming.

Required Interfaces are associated to Provided Interfaces through the CD_InterfaceUses association. This allows for Required Interfaces to link to Provided Interfaces and visualize interface usage in the Component Diagram.

5.2.2 Component

The Component is implemented using a custom Java class to define the shape that the node uses. The custom class draws the Component as a square edged rectangle with the UML 2.0 Component icon in the top right-hand corner. The rectangle is drawn a set distance inside the bounds of the node to create a margin in which Required Interfaces and Provided Interfaces can sit. This allows the Interfaces to sit next to their parent

Component and appear to be joined to the outside. This would not have been possible if the rectangle was drawn on the bounds of the node as the Interfaces would lie inside the rectangle.

Interfaces are positioned inside the Component through use of the Layout Manager. GMF provides a Layout Manager that, by default, allows child nodes to be repositioned anywhere within the bounds of the parent node. The required behaviour for the Component is that its children are snapped to the edge of the Component so that Interfaces appear to be joined to the edge of the drawn rectangle. Interfaces are also able to be dragged and dropped into a new location and snap to the closest edge from that point. To achieve this behaviour a custom Layout Manager was written.

The custom Layout Manager calculates which edge of the Component each Interface is closest to and moves the Interface to be sitting on the outside of that edge. This is done by calculating the centre points of both parent and child. These are used to get the normalized vectors from the centre point of the parent to the four corners of its bounds and to the centre of the child. These vectors are used to calculate which sector (top, bottom, left or right) the child lies in, and therefore which edge it is closest to on the parent. Calculating the closest edge by using vectors allows the Interfaces to be dropped outside the bounds of the Component and still snap back correctly.

5.2.3 Interfaces

Both the Required Interface and Provided Interface are implemented through the use of a custom Java class to define their shape. The Required Interface is drawn as a semicircular arc, connected to the Component through a perpendicular line extending from the base of the arc. The Provided Interface is drawn as a closed, unfilled circle, joined to the Component through a short, perpendicular line.

As both Interfaces need to be attached to all four sides of the Component, their drawn image must be rotated such that they are always connected to the Component. To do this, the Interface classes make use of the custom Layout Manager to calculate their closest edge. The image is then drawn to align with the closest edge.

5.2.4 Interface Association

The association between a Required Interface and a Provided Interface is visualized in the diagram by a dashed line with an open arrowhead pointing from the Required Interface to the Provided Interface.

5.2.5 Component Association

The association between Components in the diagram is visualized by a solid line with no arrowhead between the two Components in the association.

5.2.6 Technical Challenges

The technical challenge in developing the Component Diagram lay in the custom Layout Manager, which was challenging for two reasons. Firstly, when defining a custom Layout Manager in GMF, layout edit policies are used to define which child nodes in the parent node can be moved and edited. The edit policy must be specified within the generated diagram code, rather than in the GMFGraph file where the Layout Manager is specified, and will use the default policy otherwise. The default edit policy is not to allow anything to be moved within the parent and this initially caused problems. Since there is no mention in the available documentation about setting an edit policy, the default edit policy was being used. Using the default edit policy meant that the drag and drop functionality for the Interfaces did not function. As the only visible thing to be changed in the GMFGraph file was the Layout Manager, the assumption was that it was the custom Layout Manager that was causing the drag and drop to be non-functional. This assumption led to a lot of time being spent with the custom Layout Manager before it being realized that the layout edit policy needed to be changed. The requisite changes were made to use the XYLayoutEditPloicy, which provided the functionality that was needed.

Secondly, the algorithm to snap Interfaces to the edge of the Component took several iterations to obtain the correct functionality. The initial algorithm calculated the closest edge by calculating the distances between each edge of the Interface and each corresponding edge of the Component using the bounds of both nodes. The Interface was snapped to the edge with the shortest distance. This produced the desired result provided that the Interface was not moved outside the bounds of the Component. If this were the case, the Interface would occasionally snap to an incorrect edge, especially if it had been moved outside a corner. Figure 5.5 shows an example of bad snapping in the Component Diagram Layout Manager. This problem was rectified by implementing the vector system described above.

5.3 Object Diagram

The Object Diagram consists of Objects, Attributes and associations between Objects and is used to model a static instance of a dynamic system. Within UML-B this could be showing how a system looks during animation, showing how a system looks at a point in

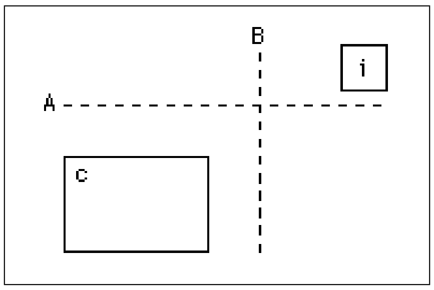


FIGURE 5.5: An example of a Component Diagram showing a configuration where component i would snap down to the boundary line A, when it should snap to B.

a Sequence Diagram or showing the relationships between real-world objects before the system is formally designed. The metamodel and editor design for the Object Diagram is explained in the following section.

5.3.1 Metamodel

To get the required graphical behaviour for the Object Diagram the metamodel shown in Figure 5.6 was created. This metamodel is not stand-alone but is integrated with the UML-B metamodel. This is to allow the diagrams functionality to be integrated with the rest of the UML-B diagrams and to allow the diagram to make use of existing UML-B classes.

The UMLBObjectDiagram class is the main root class for the diagram and is linked through an aggregation to the UMLBObject class. The UMLBObject class is the representation of an Object in the metamodel and has a *name* attribute and an association back to itself. This association provides for the ability to link Objects to other Objects in the diagram. The UMLBObject class is associated to two additional classes that were already present in the UML-B Metamodel; UMLBAttribute and UMLBClass. The link to UMLBAttribute allows Objects to have optional Attributes with the same functionality as UMLBAttributes, which are used in the already existing Class Diagram. The *instanceOf* association to UMLBClass is included for completeness but should allow for enhancements to the functionality of the diagram.

The aggregation from UMLBSequenceElement to the UMLBObjectDiagram class is to allow the Object Diagram to be linked to the Sequence Diagram through the Event Calls. Having the Object Diagram as a child of the UMLBSequenceElement allows for a link between an Event Call and an Object Diagram, which can be used when providing the right-click pop-up menu functionality.

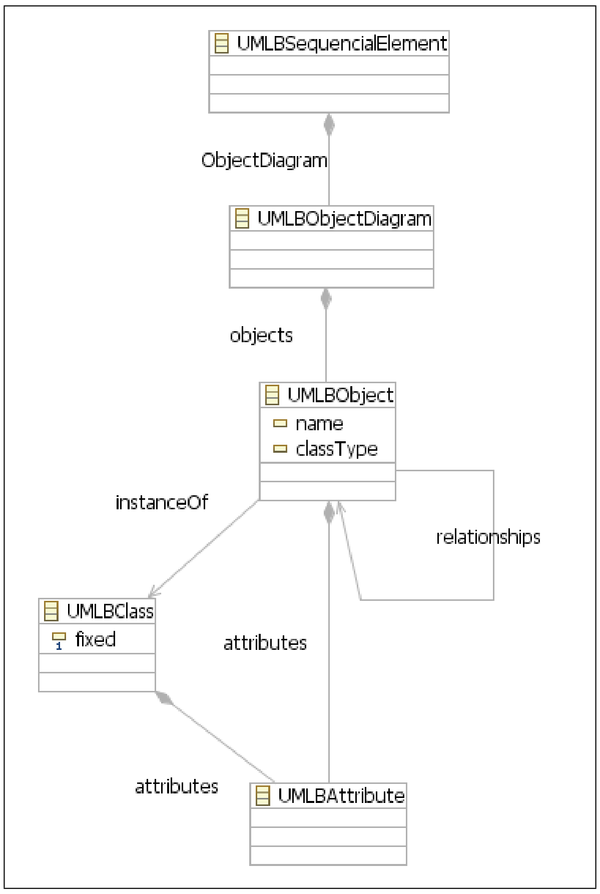


Figure 5.6: The Object Diagram Metamodel

5.3.2 Object

The Object node is represented graphically with a rounded rectangle containing a label for the Object's name, and a separate labelled compartment that stores the Object's Attributes. Attributes have a label that is used to display both the name and value of the attribute. The rounded rectangle and labelled compartment were used to maintain a visual similarity between the Object Diagram and Class Diagram, which uses the same design for Class nodes. The Object is resizeable to any dimension, again, matching the functionality of the Class Diagram.

The mapping of two data items to one label, for both the Object and Attribute labels, is defined in the GMF mapping. Labels can take a variable number of features, which correspond to Class attributes in the metamodel. Labels have a view pattern, which defines how the features are viewed in the diagram, and an edit pattern, which defines how the features are edited in the label. The feature used for the Object label are *name*. The features used for the Attribute label are *name* and *initialValue*. For the Attribute label the view and edit patterns used are *0:1*, which produces the '*name: value*' style view for Attributes.

It is possible to have null values for any label feature in the Object Diagram, which provides for having attributes without a specified value. However, to get a null value, the value for the non-null part of the label must be entered via the properties window. This is due to the direct editing on the label not handling null values very well. If one feature is left blank when the label is being edited no value will be recorded for either feature.

5.3.3 Object Association

The Object association for the diagram is visualized as a solid line with no arrow. Associations are constrained using an OCL constraint, specified within the GMF mapping, so that the two Objects joined by an association are different. This prevents associations from one Object back to itself.

5.3.4 Toolbar

The toolbar is defined in the GMF tooling definition and contains the buttons necessary to create Objects, Attributes and associations. The toolbar is split into two tool groupings, one for nodes and one for links, both of which are collapsible. This has been done to keep the toolbar looking similar to the toolbars in the other UML-B diagram editors.

5.4 Sequence Diagram

The Sequence Diagram represents the sequence of operations to be carried out by a Machine in some example procedure, possibly with the aim of driving an animation script. The diagram displays UML-B Objects as swimlanes, with calls to events as arrows that take the state of the Machine from one point to another. Visually on the diagram, the arrows move from a calling Object to a receiving Object, denoting the sequence in which the units of code are executed.

5.4.1 Metamodel

The UML-B Sequence Diagram meta model (see Figure 5.7) is relatively straightforward in terms of complexity, at least compared to some of the other diagrams. The root node is the sequence itself, which is then comprised of Object Lifetimes, a start point and sequential elements (links).

The Object lifetime elements have three types of children: A *name* property that acts as a label to distinguish one instance of a class from another. A reference to a Class object that records what Class the object is an instance of (this will have been defined in another diagram, such as a Class Diagram). And finally the Activity Boxes, which are the yellow vertical boxes that provide a region for the links to anchor to.

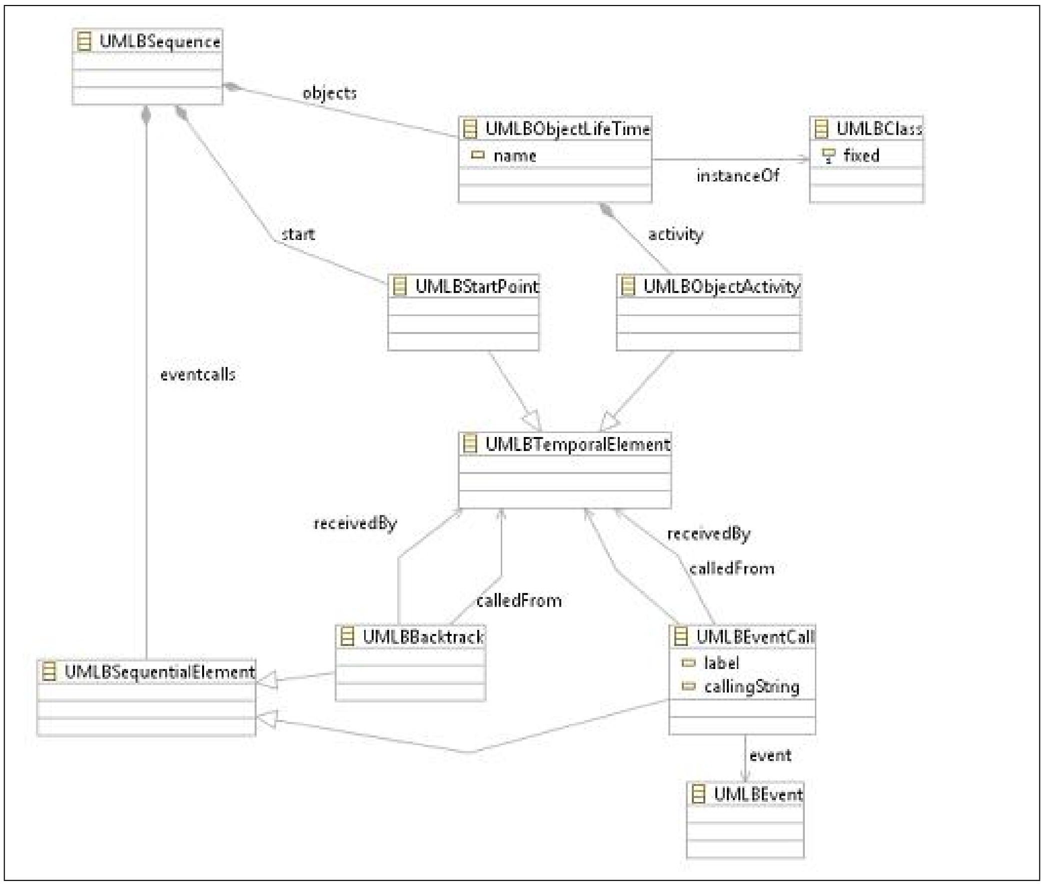


FIGURE 5.7: The Sequence Diagram Metamodel

The start point is a leaf node on the diagram that does not have any children or attributes. This is due to the fact that the start point is merely a graphical widget that provides a starting position for the sequence. It is not a mandatory element on the diagram and it does not carry any semantic meaning with it. However, an Event Call linking the start point to an object can be associated with an event just like any other, this is useful if there is an initialisation method that is called by the environment, perhaps when a program is first started up (it can be used to show an Event Call coming from a region of the system that falls outside of the scope of the Sequence Diagram). It shares a common superclass with the object lifetime, UMLBTemporal element, this is so that the elements which are to be arranged horizontally on the page can be handled uniformly by the Layout Manager. It also means that, in theory, one could easily extend the notation at this level to include another type of element, without having to reimplement the layout.

The two types of links, or sequential elements, on the diagram are the Event Calls and the backtrack link. They are linked by a common superclass as they are handled very similarly by the canvas. It again, allows the same code for positioning, linking and layout to be applied universally to all links. Links are represented by their own elements on the diagram, rather than simply allowing object lifetimes to directly reference other object lifetimes. This provides a more distinct representation of the Event Calls, which are arguably the most important element on the diagram, within the model file. Rather than simply having a reference under the lifetime node, there is a child link node that holds all the information in a readily available and explicit place. It also allows us to record the order of the nodes on the diagram, which could not have been done if there was not an actual node for each link.

The backtrack link is very simple, only recording where it comes from and goes to. There is no need to record anything else, as the link merely represents a transition back to a previous state and provides no new data. The Event Call is slightly more complicated, here there is a reference to the event that is being called, a label attribute to aid it's identification on the diagram and a string that represents the parameters being passed to the event.

5.4.2 Technical Challenges

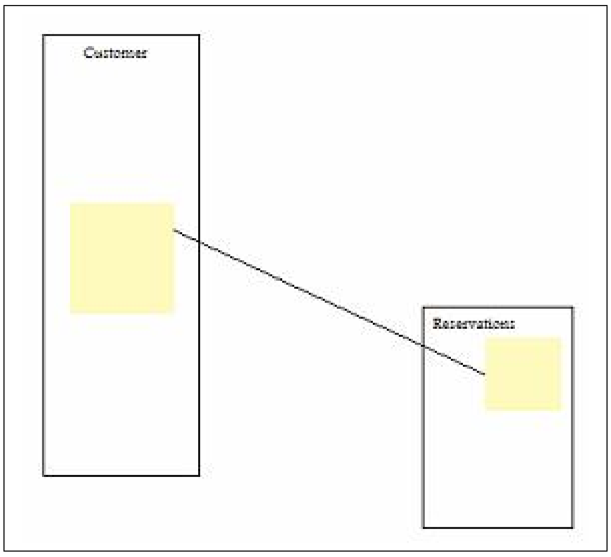


FIGURE 5.8: The Sequence Proto Diagram

Whilst the metamodel is simple, creating the Sequence Diagram editor so that it looks and behaves in a way similar to the UML 2.0 Sequence Diagram is another matter. Of all the UML 2.0 diagrams, this is the one that is mostly frequently poorly implemented and awkward to use. The main reason for this, is that it is not simple type of diagram where the user can drop elements onto the canvas anywhere and then link them up as they wish. As stated in the requirements, the layout is quite strict, with elements being aligned in swim lanes and links always being horizontal. Unfortunately GMF only natively supports the former and so the complex functionality has to be programmed in manually.

To demonstrate the magnitude of this, Figure 5.8 shows the proto-Sequence Diagram that GMF was able to produce. This was the basis of the editor, all other functionality has been added by hand.

5.4.2.1 Custom Shapes

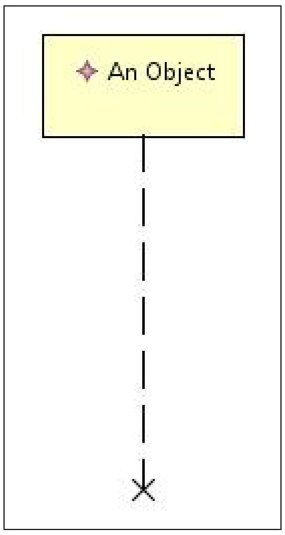


FIGURE 5.9: A Custom Shape used in the Sequence Diagram

The shapes displayed on the diagram, which represent the Object Life Time elements proved too complex to create in GMF. Instead they were created by utilising a custom shape object that extends the Draw2d.Shape class and is registered as the graphical representation of the element at the GMF level. This class can be found in the Package *ac.soton.umlb.umlbMetamodel.diagram.custom*. By overriding the class' draw and fill methods it was possible to recreate the desired shape exactly. First an outline is drawn; the box at the top of the shape (shown in Figure 5.9) is of a set relative size and shape, this is multiplied by the zoom level of the diagram to produce an absolute size. The dotted line that travels down the centre of the shape, starts from where the box at the top finished and ends at a position that is half the absolute size of the cross at the bottom away from the lower limit of the area. The cross is drawn in the same way as the box, only using two *drawLine()* calls, rather than *drawRectangle()*. Secondly the shape is filled, in reality only the box needs colouring in and so a fill rectangle, slightly smaller than the border is used.

5.4.2.2 Layout

There are two aspects of the Sequence Diagram that required specialised forms of layouts. These were hinted at during the requirements phase and are now shown in Figure 5.10. Firstly the elements that are placed directly onto the canvas have to be laid out as shown; that is the Object Life Time elements are all aligned to the top of the page and the start point is kept at the left hand side. Other than the start point, the elements may be arranged horizontally, with the ability to reorder them, though they must not overlap. The second layout behaviour required is the positioning of the Activity Boxes within the life times. Here the box must be aligned to the dotted line and must start at a given distance below the Object Box.

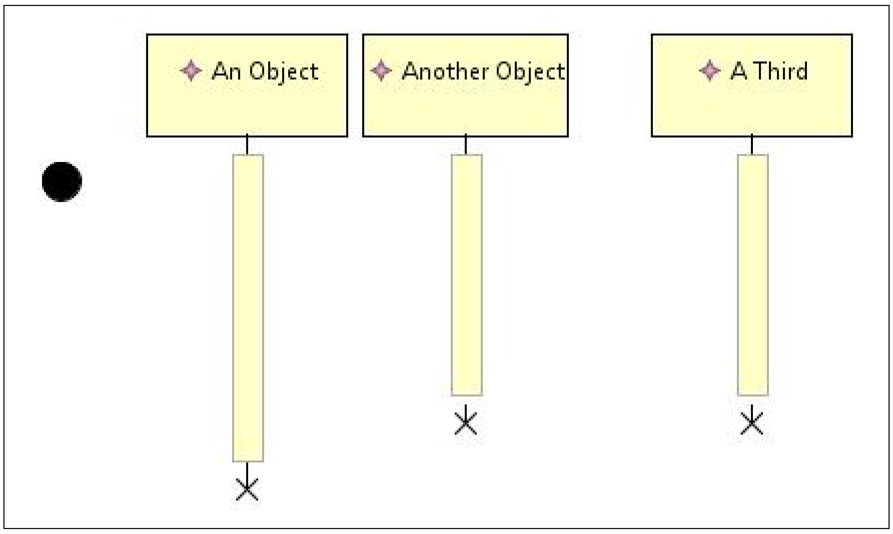


FIGURE 5.10: Layout Manager behaviour in the Sequence Diagram.

Adding a custom layout to a container requires attaching a special class that implements *Draw2D.AbstractHintLayout*. Within this interface the key method is *Layout(IFigure Parent)*, which handles the laying out of all the children of the supplied figure (which will correspond to the container being manipulated). The layout code iterates through each of the figure's children and alters their bounds and positioning depending upon the desired location (acquired using *getConstraints()*). Beyond this it is a simple matter of crunching numbers and calculating the correct positions. It is also important to take into account the zoom and scroll functions of Eclipse, this is done by translating the calculated coordinates from relative points to absolute points (there is a method provided for this, *IFigure.translateToAbsolute()*).

Layouts are generally registered with the plug-in at the GMF level. In the GMFGraph model it is possible to specify a custom layout class for any container. When specifying a layout for the canvas itself it is slightly harder, this has to be done using GEF. The layout class has to be added in to the GEF code responsible for creating the canvas figure (in the Sequence Diagram this is *ac.soton.umlb.umlbMetamodel.diagram.edit.parts .UMLBSequenceEditPart*).

5.4.2.3 Connection Anchors

The behaviours of the links on the Sequence Diagram proved very challenging to implement: how they were laid out, where they were connected and how they could be moved. Link behaviour in GEF (there are no options whatsoever for changing link behaviour in GMF) is managed by two classes, the *EditPart* class that controls the figure to which the link is connected to and the *ConnectionAnchor* class. In essence the behaviour is controlled by having two connection anchors, one at either end of the link. By controlling the positioning of these anchors on the border of the connected elements, the position of the link can be changed. This is done by implementing the *ConnectionAnchor.getLocation(Point reference)* method.

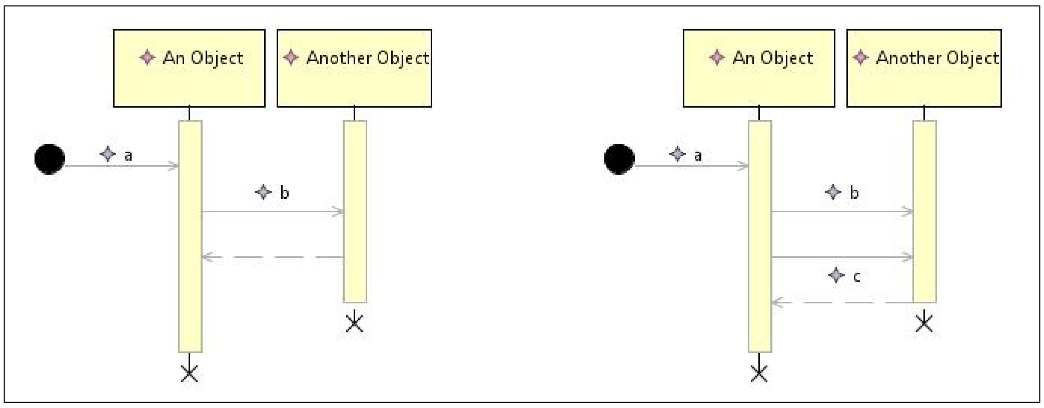


Figure 5.11: Diagram showing connection behaviour in the Sequence Diagram.

The EditPart class, in this case *ac.soton.umlb.umlbMetamodel.diagram.edit.parts .UMLBObjectActivityEditPart*, contains four methods that handle creation of connection anchors. The *getSourceConnectionAnchor(Request request)* and *getTargetConnectionAnchor(Request request)* methods are called whilst the links are being created and dragged around, but before the mouse is let go and the position becomes permanent. These methods are passed a *Request* object that contains the position of the connection request. The *getTargetConnectionAnchor(org.eclipse.gef.ConnectionEditPart arg0)* and *getTargetConnectionAnchor(org.eclipse.gef.ConnectionEditPart arg0)* methods handle permanently setting the link location and recreating the anchors when the diagram is opened from a saved file. These methods are only passed a *ConnectionEditPart* object that does not contain any information regarding the desired position of the link. As a result the final position of the link should either not depend upon the position at which the mouse was clicked on the diagram, or the location has to be somehow transferred from the first two methods to the second two. The Sequence Diagram does the later. When a connection anchor is created from a request object, the mouse location is recorded in a static variable, this is then used to remember the correct y coordinate of the connection. The x coordinate is always equal to the x bound of the ObjectActivity element border that is closest to the opposing anchor (the position of which is passed via the reference point), ensuring that the connection always runs between the closest two sides of the connected elements. A convenient knock on effect of using a static variable is that the y coordinate for the source and target anchors is shared, causing the two to move in sync vertically, always perpendicular to the ObjectActivity box.

The actual y positions are not exactly equal to the position at which the link was initially placed, the order of the link on the diagram is taken into account and used to space the links nicely. This can be seen in Figure 5.11, where adding a new link to the middle of the diagram causes the other links to space out.

5.4.2.4 Ordered Links

It was important that the links were not only arranged neatly on the diagram, but also that the order in which they appeared, running from top to bottom, was recorded in the meta model. This was achieved by ensuring the the collection of Event Calls recorded in the meta model was orderable (an option present when configuring the EMF) and by including code to update the order at relevant points.

The code was added to the Connection Anchors so that whenever a link was either added or moved on the canvas, the position in the order was recalculated. The key aspect here is that the existing links are always spaced at a specific distance apart (taking into account the level of zoom on the diagram). By taking the desired position of either the source or target anchor, belonging to the new or relocated link, relative to the top of the Activity Box, and dividing it by the spacing; one could (with some rounding) obtain the position of the link in the order. This is then used to a) work out the correct position

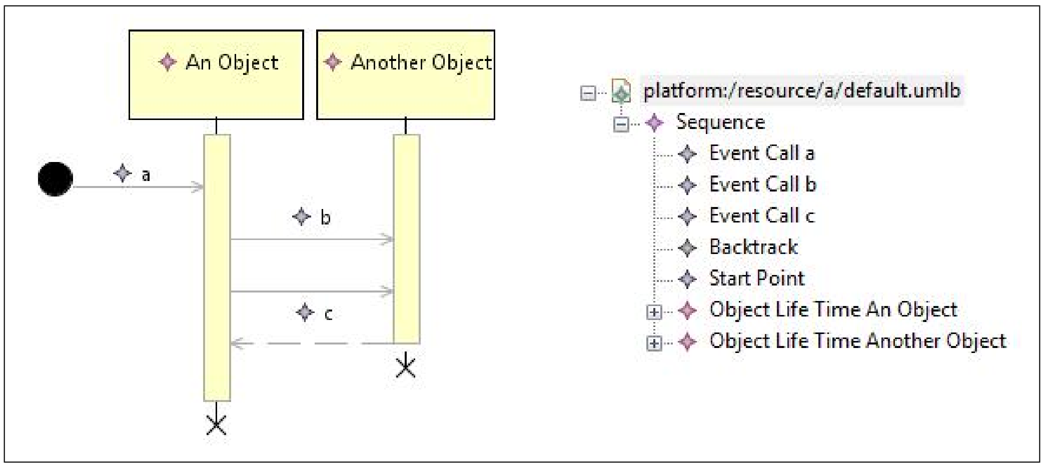

An Object
Another Object
a
b
c
platform:/resource/a/default.umlb
Sequence
Event Call a
Event Call b
Event Call c
Backtrack
Start Point
Object Life Time An Object
Object Life Time Another Object

FIGURE 5.12: Diagram showing model link ordering in the Sequence Diagram.

for the link on the canvas, and b) update the order of the link in the meta model. This behaviour can be seen on Figure 5.12, even though link C was added to the diagram after all the other links, it appears in the correct order in the model hierarchy

Updating the meta model from the GEF layer in the framework is not just a matter of calling the relevant method. Changes made to the model have to pass through a command system, which manages concurrent changes being made to the model from the various different views upon it. This allows changes to be done and undone transactionally, without violating the consistency of the data. In order to change the order of the links, first the command stack has to be obtained from the diagram, then a new *MOVE* command is created. This is supplied with details of: the model feature that is being altered, the element that is being moved, the editing context that is performing the operation and the new position in the index that the element is to be moved to. The object can then be pushed onto the stack, which will execute it in due course.

5.5 Use-Case Diagram

The Use-Case Diagrams consist of three element types: Actor; Use-Case and System Box, along with the various links that join them all together.

5.5.1 Metamodel

The base class in the Use-Case Diagram Metamodel, shown in Figure 5.13, is the Use-CaseDiagramParent that represents the overall Use-Case Diagram. The UseCaseDiagramParent is made up of many Actor, UseCaseItem and SystemBox elements. Actor, UseCaseItem and SystemBox generalize UCDElement, where UCDElement is designed to allow shared properties between all of the Use-Case Diagram elements. UCDElement creates the *name* property so that Actor, UseCaseItem and SystemBox all have a *name* attribute.

As System Boxes are required to allow other System Boxes and Use-Cases to be nested inside of them, there are aggregates from SystemBox to SystemBox and SystemBox to UseCaseItem. These aggregates allow multiple SystemBox and UseCaseItem items to be created inside of any SystemBox.

Associations between classes in the Use-Case Metamodel represent the types of links that are allowed in a Use-Case Diagram. The links *uses* and *calls* represent the situations where an Actor can use a Use-Case and where a Use-Case requires an Actor to complete its execution. The UseCaseItem is associated to itself through the *includeUseCase* and *extendUseCase* links that represent the situations where a Use-Case has to include or extend another Use-Case.

The generalization of Actors or Use-Cases also requires a self-referencing association from both the Actor and UseCaseItem classes. Doing this however complicates the model definition as the Use-Case Diagram would have two different *generalize* links. To solve this problem, the Actor and UseCaseItem classes generalize UCDGeneralizable in the metamodel. There is then a single *generalize* self-referencing association from the UCDGeneralizable class.

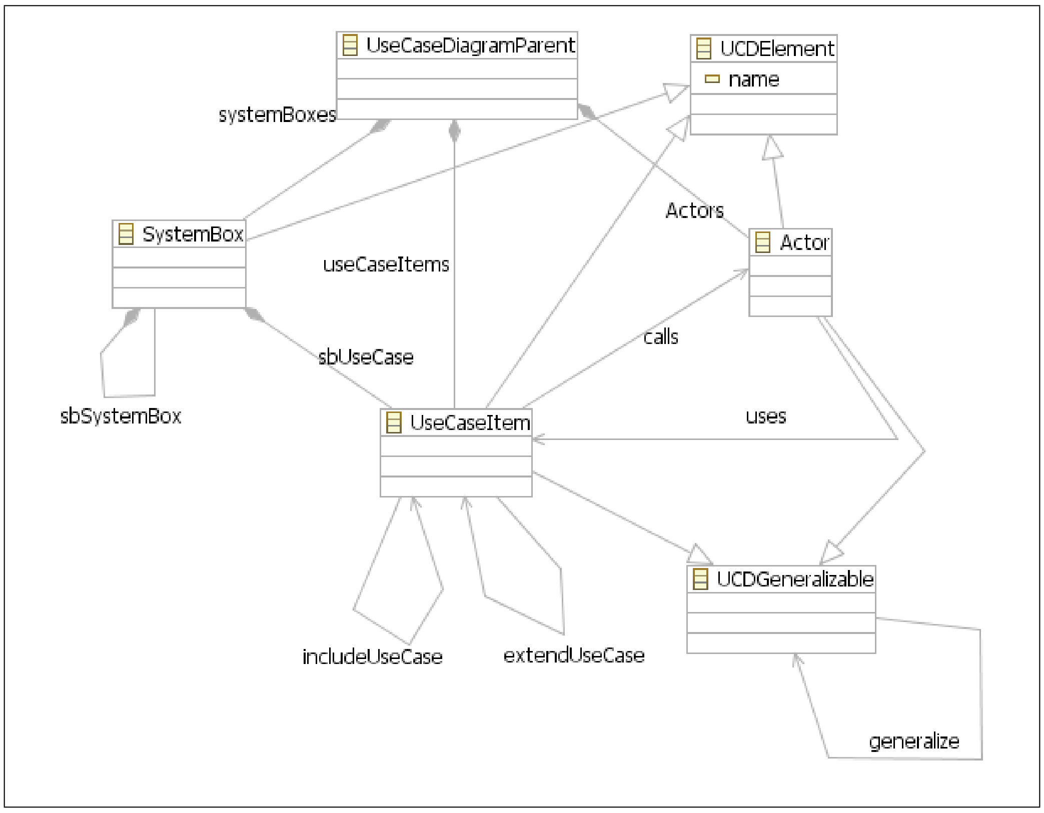


FIGURE 5.13: The Use-Case Diagram Metamodel

5.5.2 Actor

The Actor is comprised of a graphical component, the 'stick man', along with a label that is linked to the Actor's *name* attribute. The 'stick man' graphical component is defined in GMF as an ellipse for the head and a custom figure for the body, positioned so that the body and head match. The graphical component cannot be resized by the user when they are creating a Use-Case Diagram, because this caused problems of the head and body no longer joining each other. This is not a problem when the diagram is viewed at a different zoom level inside of Eclipse, as GMF maintains the structure of the graphical component by scaling it.

The label for the Actor is positioned outside of the main Actor graphical object, so that the label can resize itself without trying to resize the 'stick man' graphical object.

5.5.3 Use-Case

The Use-Case is an ellipse containing a centered label that is linked to the *name* of the Use-Case. In GMF the label is centered by applying a border layout to the ellipse shape and then setting the label to be a center component. The Use-Case ellipse will also expand automatically as the length of the text in the label increases so that all of the text in the label is shown.

5.5.4 System Box

The System Box is a rectangle container with a labelled bar at the top that shows the name of the System Box. As in the UML 2.0 Use-Case Diagram, System Boxes are able to nest other System Boxes and Use-Cases. Providing this nesting capability is a two stage problem, the first part of defining a metamodel to allow this has been discussed in Section 5.5.1, the second part is setting up GMF to allow this.

To allow nesting in GMF, child references for System Box and Use-Case have to be added to Node Mapping for the System Box Top Node Reference in the GMF mapping file. The System Box node mapping must also be set as a compartment by adding a Compartment Mapping. By doing this, GMF will now manage the graphics and interface when any System Boxes or Use-Cases that are placed inside of another System Box. The graphical and interface components that GMF will control here include adding and removing components from a System Box, moving a System Box and adding scroll bars to a System Box to show its entire contents if the System Box is not large enough to show its entire contents.

5.5.5 Links

The links defined in the Use-Case Diagram are designed to look like the links in a UML 2.0 Use-Case Diagram. Therefore the *use*, *include* and *extend* links are all lines with an open arrow head pointing in the direction of the link. The *generalize* link has a closed arrow head and the *call* link from a Use-Case to an Actor has no arrow head.

The *include* and *extend* links automatically generate a label depicting whether they are an *include* or *extend* link as they should do in a UML 2.0 Use-Case Diagram. The include and extend labels have had the editable property set to false so as not to allow the label to be modified.

The links designed in the metamodel allow multiple links to be made per element in the Use-Case Diagram. There are some properties for links that cannot be defined in the metamodel, such as preventing Use-Cases including, extending or generalizing themselves. In these cases the links can have link constraints set in the GMF mapping defined using OCL as shown in OCL 1, which ensures that the node at the start of the link is not the same as the node at the end of the link.

```
self <> oppositeEnd
```

OCL 1: OCL to ensure no self-referencing

In the Use-Case Metamodel, Actor and UseCaseItem have been generalized to UCDGeneralizable, so in its default behaviour any generalize link will allow Actors to generalize Use-Cases legally. As this behaviour is not legal in a Use-Case Diagram, there must be some constraints made about the links that are made, which again can be made in the GMF mapping. The OCL shown in OCL 2 ensures that the start and end nodes in a link are of the same type, either Actor or UseCaseItem.

```
    (self.oclIsTypeOf(Actor) and oppositeEnd.oclIsTypeOf(Actor)) or
(self.oclIsTypeOf(UseCaseItem) and oppositeEnd.oclIsTypeOf(UseCaseItem))
```

OCL 2: OCL for generalize constraint

5.6 Diagram Integration

Part of the requirements for each diagram was that they must be integrated into the existing UML-B notations and, where appropriate, integrated with each other. As already discussed, the first stage of this is to unify the metamodels under a single file. Beyond this, integration takes the form of being able to create or open specific diagrams via a right-click option on relevant diagram elements. It has also been made possible to navigate from a nested diagram, back up to the parent. For instance, where a Sequence Diagram is created as a child of a Machine element, it is possible to open the Package Diagram in which that Machine appears, by right-clicking on the Sequence Diagram canvas and clicking 'Open Parent'.

This functionality is obtained by extending the editors at the GEF level. A custom class is created within the Package of each editor that is to have a right-click action added. This class is registered with Eclipse in the plugin.xml file of that editor, as an object contributor to the *popupMenu* plug-in extension point. The class then has to handle calls to the new action, providing the relevant prompts and either opening the relevant file or displaying a new diagram wizard. When creating a new diagram the class has to provide the correct root element, in some cases (where the root node is not an element on the parent diagram) this has to be created using an Add Command to the model.

Chapter 6

Testing

The development of new diagram editors for the UML-B system has produced many things than need to be tested for correctness and completeness. Elements that need testing fit broadly into one of four categories; requirements, Metamodels, diagram behaviour and custom code. The testing procedure used for each of these categories is discussed in this chapter.

6.1 Requirements Testing

As most of the requirements were to define UML 2.0 style models that already existed, requirements testing was a relatively simple process. Once the requirements were defined they were presented to our supervisors who then agreed each of our requirements documents.

6.2 Testing Metamodel Design

The design of the Metamodel for each diagram type was critical to ensure that the diagram worked correctly when it was implemented. Testing the Metamodel design was through peer review, which occurred in design meetings. Each Metamodel was presented to the group along with the reasons as to how it was designed and how it is intended to work. Once the group agreed that the Metamodel design was reasonable, the Metamodel was implemented as a diagram type.

6.3 Testing Diagram Behaviours

Testing diagram behaviours was again a relatively simple process as we were creating refinements of standard UML 2.0 models with well known behaviours. Each diagram type was tested against its requirements to ensure that correct linking between diagram elements took place, that diagram elements could only be used in certain legal situations and that any data handling not visible to the user took place. The outcome of this testing process is shown in Appendix A, where each diagram is shown in a complex configuration demonstrating all possible link types and diagram elements.

In some circumstances more in-depth testing of diagram behaviour had to be carried out because there were large amounts of coded elements that provided behaviours and functionality that GMF cannot provide on its own. In particular in-depth testing involved the Sequence Diagram's layout management and the Component Diagram's Component Layout Manager.

6.3.1 Testing the Component Layout Manager

The Component Layout Manager is responsible for ensuring that both Provided and Required Interfaces snap to the edge of a Component in the diagram editor. This behaviour was thoroughly tested by attempting to position Interfaces at several points around the diagram. Positioning points were chosen so that they would test both the snapping when the Interface was inside the drawn area and when the Interface was outside the drawn area. In the cases where the Interface was outside, more points were chosen to test the correctness of snapping near corners as described in Section 5.2.6.

6.3.2 Testing the Sequence Layout Manager

The Sequence Diagram Layout Managers controlled two aspects of the diagrams behaviour: the positioning and management of the top level elements on the canvas and the positioning and sizing of the Object Activity Boxes within the Object Life Time boxes. The later was tested by ensuring that the Activity Box was always correctly positioned and sized no matter what the size or position of the parent element. It was also tested by resizing the parent element after the box had been added, ensuring that size moved relative to the parent and that the spacing between the top of the box and the top of the dotted line remained constant.

The Canvas Layout Manager has several elements that had to be tested. Firstly, we ensured that start points were always on the left and could not be moved and that Object Life Times elements were always a set distance from the canvas top and could not be moved vertically. Secondly, we ensured that the horizontal positioning of the

Object Life Times was correct. The elements were allowed to be placed at any point along the x-axis, but were not permitted to overlap. We ensured moving one of the elements to the opposite side of another correctly caused them to be reordered. Each of these areas was rigorously tested by carrying out repeated tests of every possible combination of moving, resizing and reordering. Attempts were also made to induce incorrect behaviour by trying to overlap elements.

6.3.3 Testing Link Management

The important aspects of the link behaviour to be tested are that the links must always remain horizontal, must by movable and reorderable and must be evenly spaced in the order they appear in the Metamodel. Each of these factors was verified through repeated testing of the linking; repeatedly recreating links in every possible location with every different combination of parent nodes.

6.4 User Acceptance Testing

Throughout the project we demonstrated the diagrams as they were implemented with our project supervisors. This ensured that any changes to the diagrams or their requirements were determined as early on in the project as possible.

In the penultimate week of the project we were able to demonstrate all of the diagrams with most of their features working to our supervisors and our customer. In this session we were able to demonstrate how all of the diagrams would sit together inside a UML-B project and it gave our supervisors and customer chance to provide us with feedback and a small number of extensions to implement. In this meeting our supervisors requested that the Activity and Component Diagrams should be linked back to the original UML-B diagram types.

Chapter 7

Future Work

Updating and extending UML-B has a very large scope for continued work. In this project we have worked on designing and specifying the requirements for the various models we have implemented. Throughout this process we have developed some further ideas of how UML-B can interact with other components in RODIN and in this next section we will outline some key ideas about updates to RODIN through UML-B. These are discussed in this chapter.

7.1 Integration with a Model Animator

In the B animator ProB, the system is held in terms of the state the system is in, the operations that are enabled and the history of operations that took place in an animation. Sequence Diagrams are used to show the order and flow of actions in a given system, whilst Object Diagrams can be used to show the state of a system at a given time. Here we can see the beginning of the link between the two diagrams and the animator, where Sequence Diagrams can be linked to the animator's history of operations and the Object Diagram can be linked to the animator's state properties.

UML-B interacting with model animators will be of use in two areas of model checking. Firstly a user may wish to define a set of sequences to perform on a model to determine that a particular behaviour maintains the invariant. To do this we will need a method of translating UML-B to an animation script. Alternatively, a user may wish to investigate an animation sequence that had taken place, especially if a given animation violated the invariant. In this scenario we should provide a system that takes an animation output and produces a UML-B model showing how the animation progressed.

7.1.1 UML-B to Animation Script

The UML-B Sequence Diagram can be used to develop a sequence of events to apply to the system being designed. Starting from the Initial Node, the Sequence Diagram allows the user to define the events being called along with any parameters that need to be passed to the event. This is achieved through the Event Call link. The terminating end of the Event Call link determines the Class to which the event belongs. Finally the order in which the Event Calls are placed can determine the order in which the events are called.

Eclipse stores the Sequence Diagram data in an XMI file that is ordered in the event order defined in the diagram. This means it should be possible for the XMI file to be taken and converted into a format that an animator can take as input, defining the sequence of events to call in a given model.

7.1.2 Animation to UML-B

A user may also wish to have a more in-depth look at how an animation sequence ran, especially in terms of debugging animations that lead to an invariant being violated. In this situation both the sequence and Object Diagrams come into use, by using the Sequence Diagram to show the sequence of events and the Object Diagram to show the system state.

Firstly we must assume that the animator stores the state of the animation after each animation step. Secondly, the animator must store all Event Calls and all backtracks that occur during an animation sequence. This means that after any event or backtrack we can recall the state of the system. The animator should be able to provide this data in a file that can be subsequently opened by RODIN and used to build a Sequence Diagram.

In loading the output from an animation, a UML-B Sequence Diagram should be built and the state information stored in a format that can be used to link from the Event Call in the Sequence Diagram to the state at that time. In an inverse operation to the UML-B to animation sequence method previously described, the Sequence Diagrams can have events added to them in the order in which they happened in the animation. The target for an event and the parameters passed to the event should be a part of the information passed back from the animator about the event called. What cannot be determined is where an event originated from, except for the very first event that must have come from the Initial Node. The UML-B Sequence Diagram in this situation should not attempt to work out where events originate from as this is an unsolvable problem with many different solutions but all with different meaning to the user, see Figure 7.1. In this case all events should originate from a System Class and the user can then decide to arrange where events originate from if they so wish. With the Sequence Diagram built, the user should then be given the option to select any event in the Sequence Diagrams operation and build an Object Diagram showing the system state at that time.

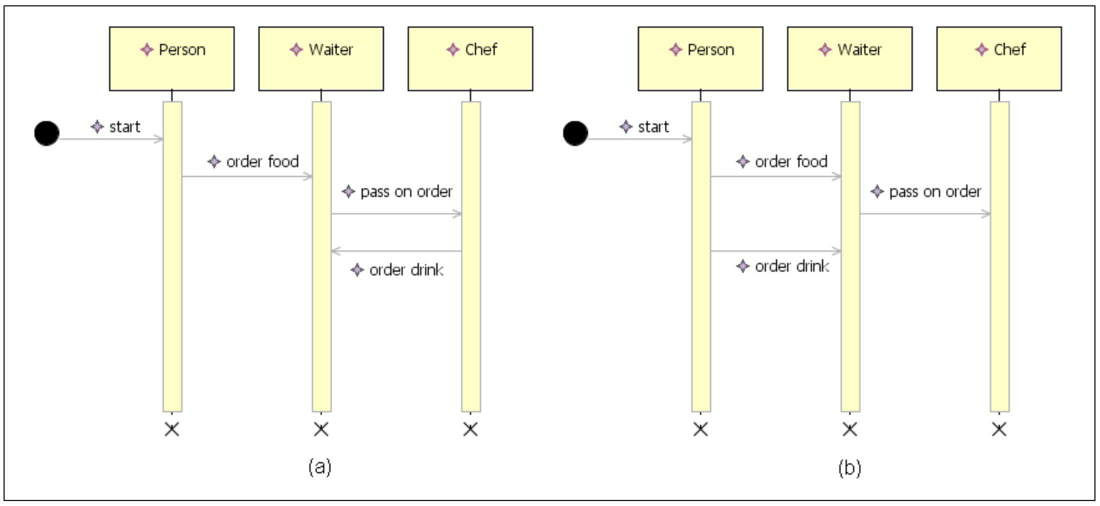


Figure 7.1: Two Sequence Diagrams showing a restaurant scenario with a Customer, Waiter and Chef. In a UML-B Sequence Diagram these two diagrams would produce the same animation script, but are different in meaning. Diagram (a) shows a model where the originating Class of an event is determined by the last Class to accept an event, which produces the meaning that the Chef is ordering a drink. Diagram (b) shows the real meaning that was required in this diagram, where it is a Customer who orders a drink.

7.2 Model Refinement

Refinement in B is a key component of formal system design and one that is currently missing from UML-B. With the addition of the Component Diagram, it could now be possible to implement model refinement in UML-B.

The Component Diagram lets us unplug elements of the system and replace them with modifications whilst still maintaining the same system behaviour. To create a system for refinement, the Component Diagram would need to be further integrated into UML-B such that a Component represented an element of the UML-B model or a section of Event-B code where refinement would be required.

An example situation may be that a UML-B model is defined that includes a set containing all the users of a particular system. A Component of this model would be the Class containing all of the users that was modelled as a set. This Component could then be replaced with a refined Component that contained all of the users of a system modelled in a list with the various checks to ensure a user only occurs at most once in that list.

7.3 Error Notification

The U2B process may in some cases come across errors in the models defined. The diagrams that were part of UML-B before this project are able to report these errors to the user, this is achieved by highlighting in red sections of the model that have caused errors [32]. Error notification may not be appropriate in all of the diagram types; here we present the diagrams where error notification could be used.

7.3.1 Activity Diagram

At this time the Activity Diagram does not have a specific role inside UML-B in terms of U2B translation and determining how error notification could be provided is limited. One error checking method that could be included would be to check the guard conditions that lead into activities to ensure that they are in a legal syntax. In a situation where the guard condition is not legal syntax, the guard label could be coloured red to show where the error has occurred.

7.3.2 Sequence Diagram

As Sequence Diagrams will be used to run and display model checking animation sequences, error notification in Sequence Diagrams could be used to show where a model checker violates an invariant. This form of error notification could have two distinct circumstances.

The first error reporting situation is when a model checker is used and no Sequence Diagram was used to provide a test script. In this situation we have no expected list of events to be called and if the model checker violates the invariant the Sequence Diagram could display all of the events leading up to the violation.

The second scenario is if a Sequence Diagram has been used to provide an animation test sequence that subsequently led to the invariant being violated. In this situation the model checker should return back the original Sequence Diagram but clearly show the events that were called and the events that did not occur because of the invariant violation. To do this, the event arrow that caused the violation could be coloured red and all of the events after be coloured in a different colour, perhaps orange, to signify they were never called.

7.4 Custom Properties Windows

The UML-B diagrams implemented before this project all have custom properties windows [11] that are used to hide the unrequired EMF and GMF properties that would otherwise be shown in the diagram [32]. Implementing custom properties windows for each diagram would have the advantage of removing the complexity of the EMF and GMF default properties and just focusing the user's attention on the UML-B specific properties, as well as keeping the same look and feel throughout UML-B.

7.5 Mathematical Notation

In some of the diagrams there is a requirement to include some formal notation by using μB. In the diagrams we have created, this is achieved through ASCII entry and display only. A future improvement to the diagrams we have created would be to implement a system similar to that produced by Andrew Tillman [32] in entering μB through the RODIN UML-B keyboard and to display any μB in true mathematical notation.

Chapter 8

Project Management

Developing the new diagram editors for UML-B has been an involved and complex process, requiring careful management of time and personnel throughout the course of the project. The roles of each individual, group dynamics, project planning and communication are discussed in this chapter.

8.1 Individual Roles

The group members had little prior knowledge about the technologies being used in this project. As the project progressed, members were able to specialise their skills in various aspects of the technologies used.

While a large amount of the project was group based especially during the specification of requirements and design, the group was able to split the work into individual tasks when it came to implementing the models designed.

8.1.1 James Amor

James brought a good knowledge of Java to the project, along with a solid understanding of the UML 2.0 diagram set. As a result James took on a major part in the group's design meetings, those held with the group's supervisors and in drawing up the requirements specifications for each diagram. James also took a secondary coding role, providing the additional functionality needed on the Component Diagram and acquiring a good understanding of the GMF framework through the implementation process. James also took on the role of editor for the group's final report, becoming responsible for ensuring that the report was consistent and well proof read.

During the implementation process, James worked on the Object Diagram and the Component diagram. The Object Diagram was a relatively simple model to implement, whilst the Component diagram was far more complex, with a hard to define graphical editor, requiring large amounts of custom code. This involved the definition of new graphical elements and the creation of a Layout Manager to snap Interfaces to the side of Components.

In writing the report, James wrote about the diagrams he implemented in the Requirements and Design chapters and wrote, with contributions from other members of the group, the Evaluation and Editor Construction chapters. James also contributed to the Project Management and Testing chapters.

8.1.2 Tim Lewy

Tim came to the project from a strong Java background, with a keen interest in software engineering, application development and implementing technologies. As a result he took on the major programming aspects of the project, in particular the Sequence Diagram, becoming an expert in extending and manipulating the supplied frameworks to provide complex functionality. In meetings he gave many demonstrations of the work in progress, answering questions on implementation specifics and endeavouring to ensure that the software engineering aspects of the project were always well looked after, and did not become overlooked.

During the implementation process, Tim worked the Sequence Diagram. With the Sequence Diagram being extremely challenging in terms of developing an entirely new Layout Manager and linking system that was not supported by GMF.

In writing the report, Tim wrote about the diagram he implemented in the Requirements and Design chapters, along with the vast technical challenges of implementing the Sequence Diagram. Tim also researched and wrote background on UML 2.0, Eclipse and its various plug-ins as well as contributing to the Abstract, Introduction, Editor Construction, Future Work, Testing, Project Management and Conclusion chapters.

8.1.3 Chris Lovell

Chris was the only member of the group to take the required course, Comp3011: Critical Systems and as such provided insight into B for the group. The knowledge of B was of use to the group during the specification of requirements, so as to help develop links between UML 2.0 and B. His knowledge of B was also used to answer questions in progress seminars that revolved around the ideas of translating the new diagrams produced to B or animation sequences. Chris was also the only member of the group to have experience in using LaTeX to produce a large technical document and so made sure this process went smoothly by supporting the other group members.

During the implementation process, Chris worked on Use-Case Diagram and the Activity Diagram. With the Use-Case Diagram being a relatively easy diagram to implement and the Activity Diagram being a far more complex diagram in terms of defining a metamodel and controlling flows between diagram elements.

In writing the report, Chris wrote about the diagrams he implemented in the Requirements and Design chapters. Chris also wrote the formal methods sections of the Background chapter, along with sections in the Project Management, Future Work, Testing, Conclusions chapters and compiling the Appendices.

8.2 Group Dynamics

Even though none of the group members had worked on projects with each other previously, the group quickly gelled and worked solidly together throughout the entire project. This group project was also unusual in that no one member neither acted nor needed to act as a group leader to guide the project. The reason for there being no requirement for a group leader was that we divided the tasks to be carried out as a group and individual members chose the tasks to do themselves. Each member then took full responsibility for ensuring that the task was completed too a very high standard.

Throughout the project, all group members listened to each others ideas and thoughts about the direction the project should take, along with determining how to resolve the technical challenges faced. This was especially important in working out the requirements for the diagrams developed and in writing the final technical report. Equally important was group members questioning each others ideas to try and determine if a particular line of thinking may lead to undesirable consequences elsewhere in the project. For example, when defining how an animation sequence could be converted back into a Sequence Diagram, the idea was originally presented that each Event Call in the sequence would start from the Class that the previous Event Call ended. After analysing this further we realised that this could create diagrams with the wrong meaning, as discussed in Section 7.1.2. Next the idea that Event Calls would not need to have a starting Class when being converted from an animation sequence was introduced by a group member. This idea again seemed reasonable, but when another group member questioned the technical feasibility of defining a link in GMF with no starting node, the idea was refined further to end with the method defined in Section 7.1.2.

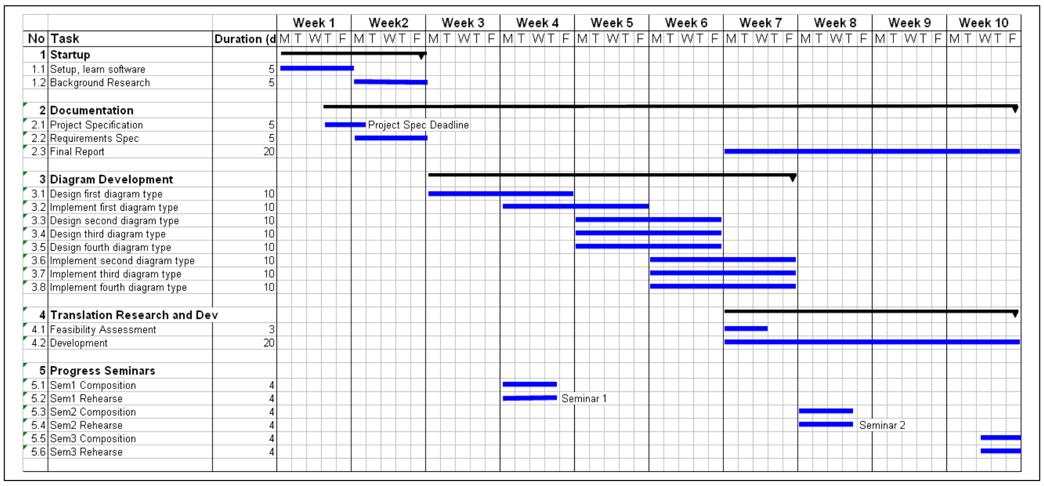
There was a level or respect in the group in ensuring that work was carried out as soon as possible. For example, most of the design meetings took place on a Thursday or Friday and usually ended with a couple of new diagram types that could be implemented. The usual etiquette was for a proof of concept prototype diagram to be implemented by the following Monday morning were any implementation issues were discussed, the level of completeness of the diagram only depended on the technical aspects of the model.

When a group member became stuck with an aspect of the project, be it trying to solve a particular problem, understanding how to create the graphs in Eclipse or creating the final document in LaTeX, another group member was quick to offer guidance or help. For example, in the beginning of the project we struggled to understand how to use GMF to create the diagrams as this process had minimal documentation. One group member was able to put in more time in trying to understand how to develop a diagrams using GMF and so was then able to help the rest of the group when they became stuck with various issues in the implementation process. Another example came later on in the project when the report writing started. We decided to write the document in LaTeX, so as to help version control and multiple accesses to the document. However, only one group member had any real experience in writing a large report in LaTeX and so it became their responsibility to support the other two group members.

Overall, the group formed a very strong bond during the project, meaning the whole project was completed as a group and that any decisions, problems and achievements belonged to the group and not any individual.

8.3 Work Plan Overview

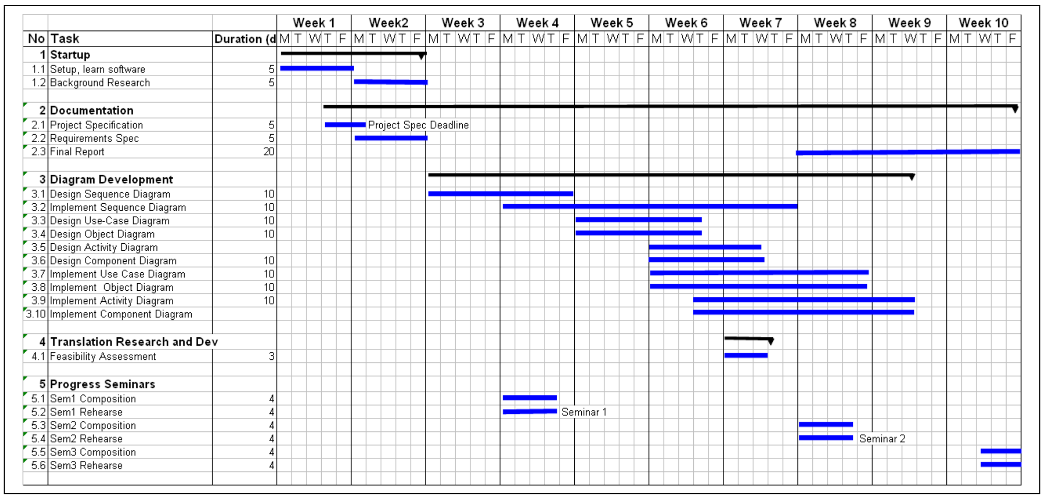
The planning of the project was undertaken by the entire group at the beginning of the project. A meeting was held to discuss the work that needed to be done and the individual task breakdowns that this would entail. Estimates of the time it would take to complete each task were drawn up, resulting in the Gantt chart shown in Figure 8.1.



No	Task	Duration (d
1	**Startup**	
1.1	Setup, learn software	5
1.2	Background Research	5
2	**Documentation**	
2.1	Project Specification	5
2.2	Requirements Spec	5
2.3	Final Report	20
3	**Diagram Development**	
3.1	Design first diagram type	10
3.2	Implement first diagram type	10
3.3	Design second diagram type	10
3.4	Design third diagram type	10
3.5	Design fourth diagram type	10
3.6	Implement second diagram type	10
3.7	Implement third diagram type	10
3.8	Implement fourth diagram type	10
4	**Translation Research and Dev**	
4.1	Feasibility Assessment	3
4.2	Development	20
5	**Progress Seminars**	
5.1	Sem1 Composition	4
5.2	Sem1 Rehearse	4
5.3	Sem2 Composition	4
5.4	Sem2 Rehearse	4
5.5	Sem3 Composition	4
5.6	Sem3 Rehearse	4

Figure 8.1: The Initial Gantt Chart, produced during Week One

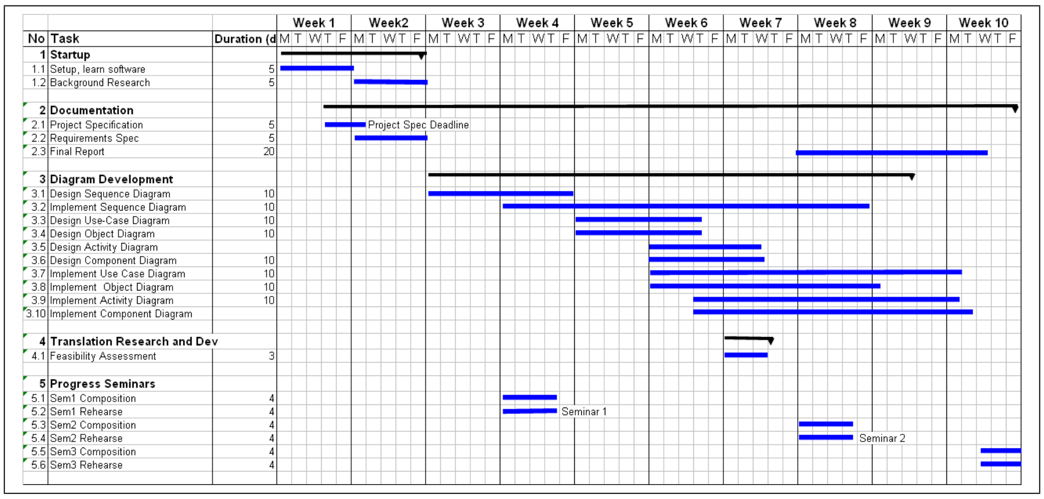
This Gantt chart shows that the group intended to have all the diagram editors complete by the end of week 7 and then spend some time on translating one of the diagrams into UML-B. This was a very optimistic estimate of, which it became clear was not achievable as development of the diagram editors was far more complex and time consuming than had been initially thought. This led the group to have another project planning meeting to discuss and revise the goals of the project. The outcome of that meeting was that the group would concentrate on the diagram editors and drop the implementation of a translation element of the project. The Gantt chart shown in Figure 8.2 was drawn up as a revised project schedule.



No	Task	Duration (d
1	**Startup**	
1.1	Setup, learn software	5
1.2	Background Research	5
2	**Documentation**	
2.1	Project Specification	5
2.2	Requirements Spec	5
2.3	Final Report	20
3	**Diagram Development**	
3.1	Design Sequence Diagram	10
3.2	Implement Sequence Diagram	10
3.3	Design Use-Case Diagram	10
3.4	Design Object Diagram	10
3.5	Design Activity Diagram	
3.6	Design Component Diagram	10
3.7	Implement Use Case Diagram	10
3.8	Implement Object Diagram	10
3.9	Implement Activity Diagram	10
3.10	Implement Component Diagram	
4	**Translation Research and Dev**	
4.1	Feasibility Assessment	3
5	**Progress Seminars**	
5.1	Sem1 Composition	4
5.2	Sem1 Rehearse	4
5.3	Sem2 Composition	4
5.4	Sem2 Rehearse	4
5.5	Sem3 Composition	4
5.6	Sem3 Rehearse	4

Figure 8.2: The Revised Gantt Chart, produced in Week Six

The revised Gantt chart shows that the estimates of the length of time required to complete the diagram editors has increased to compensate for their increased complexity and time drain. There is also a fifth diagram on the revised diagram (the Component Diagram), which was added after a meeting with the project supervisors. The revised Gantt chart has dropped the translation element, aside from the feasibility assessment that had been carried out.



No	Task	Duration (d
1	**Startup**	
1.1	Setup, learn software	5
1.2	Background Research	5
2	**Documentation**	
2.1	Project Specification	5
2.2	Requirements Spec	5
2.3	Final Report	20
3	**Diagram Development**	
3.1	Design Sequence Diagram	10
3.2	Implement Sequence Diagram	10
3.3	Design Use-Case Diagram	10
3.4	Design Object Diagram	10
3.5	Design Activity Diagram	
3.6	Design Component Diagram	10
3.7	Implement Use Case Diagram	10
3.8	Implement Object Diagram	10
3.9	Implement Activity Diagram	10
3.10	Implement Component Diagram	
4	**Translation Research and Dev**	
4.1	Feasibility Assessment	3
5	**Progress Seminars**	
5.1	Sem1 Composition	4
5.2	Sem1 Rehearse	4
5.3	Sem2 Composition	4
5.4	Sem2 Rehearse	4
5.5	Sem3 Composition	4
5.6	Sem3 Rehearse	4

Figure 8.3: The Final Gantt chart

The revised project plan was followed up to the end of the project, with some development tasks over-running. This was due to some last minute changes and integrating the editors into the rest of the UML-B environment. The final Gantt chart, shown in Figure 8.3, charts the group's actual progress through the project and is very similar to the revised Gantt chart, shown in Figure 8.2.

8.4 Communication

Communication between the group and the supervisors took place through email, discussions and meetings.

8.4.1 Communication with Supervisors and Customer

Throughout the project we had weekly meetings with our supervisors, Michael Butler and Colin Snook. These meetings were a time to show the progress we had made, address any problems and guidance on the direction the project should take. We first met our customer, Ian Johnson from AT Engine Controls Ltd. at our first progress seminar and we met again at the end of the project where we demonstrated a near complete project.

As the project required us to look at how the models we were designing could fit in with UML-B and RODIN, the weekly meetings gave us opportunity to discuss the various ideas that we had. Most of the time we presented our ideas and our supervisors looked at whether they were feasible ideas. In some situations we were unable to determine how a particular model might be used in UML-B and our supervisors were able to advise us on how they thought it might be used.

8.4.2 Communication between the Group

The group communicated through informal meetings in the undergraduate labs, more formal meetings in library study rooms and through email.

During a typical week the group met for a minimum of four hours to discuss project progress, problems and technical issues that we needed to resolve. In the implementation process this usually involved us demonstrating what we had achieved so far, how we had gone about achieving it and discussing any problems that were found when implementing it. As we found that we all met similar problems at various stages of the implementation process, these informal meetings were valuable to help each other.

The group also met roughly once a fortnight for a structured technical meeting lasting two to three hours where system requirements and designs were developed. In these design meetings, each diagram type was presented to the group by the person who had taken the responsibility to implement the diagram. This presentation normally consisted of a reminder of what the standard UML 2.0 equivalent is required to have in terms of its components along with how the model should be used. The presentation also had the initial ideas for how the diagram could be used in UML-B. The meeting then set out about defining the requirements as well as trying to see what the likely technical difficulties would be and these difficulties could be overcome.

Chapter 9

Evaluation

Updating and extending the UML-B tool set to include five new diagram types has produced a large number of requirements that cover the form and function of these diagrams. The new diagrams all require evaluation against their original requirements and this chapter discusses that evaluation.

9.1 Activity Diagram

The Activity Diagram meets all the requirements set out in Section 3.1. The requirement for the diagram to use the same graphical elements as the UML 2.0 Activity Diagram has been met, resulting in the same graphical appearance. The restrictions on the number of Flows and outgoing Flows from each diagram element have been met to give the same functionality in the diagram as the UML 2.0 equivalent.

The integration of the diagram to the UML-B tool has been partially accomplished. Activity Diagrams can be liked to Machines, and opened through the right-click menu from a Machine in a Package Diagram.

There is a very minor issue with Flows not joining directly to the edges of some nodes but ending in blank space a small distance away. This is not a major issue and is purely graphical. It does not detract from the functionality of the diagram, which will be very useful in the early stages of designing a UML-B system.

9.2 Component Diagram

The Component Diagram has met most of the points set out in Section 3.2. The graphical requirements for the Component, Required Interface and Provided Interface nodes have mostly been met, as have the requirement for the graphical appearance of Component associations.

The requirement for Interface pairs to join using the 'ball and socket' style has not been met due to time and complexity reasons. Instead, a dashed arrow with an open arrowhead has been used to join Required Interfaces to Provided Interfaces. The dashed arrow is legal notation in UML 2.0 and has not adversely affected the diagram's functionality but has had the effect of allowing a Required Interface to connect to multiple Provided Interfaces. This was not specified but is advantageous as it allows the diagram to be used to model more complex interactions between Components.

The Component association link does not appear to link up properly to the Components it is linking. This is because the bounds of the Component element are a set distance outside the drawn rectangle of the Component. The result is that the anchor points for the link attach to the bounds, not the drawn rectangle. As this does not affect the functionality of the diagram it has not been fixed due to time restraints and the complexity of the fix. This is not a major issue for the diagram

The integration of the Component Diagram with the rest of the UML-B tool has been implemented at the Package level. A Component Diagram can be opened through the right-click popup menu from blank space in the Package Diagram and the owning Package Diagram can be opened from a Component Diagram.

Overall the Component Diagram has met most of its requirements and not suffered in functionality by not meeting any. The diagram has in fact gained some functionality by allowing Required Interfaces to link to many Provided Interfaces, thus allowing for the modelling of more complicated systems. The Component Diagram will be a very useful tool to aid in the UML-B refinement process.

9.3 Object Diagram

The Object Diagram has met most of the points specified in Section 3.3. All the graphical requirements have been met, resulting in the required UML 2.0 style being achieved. The look of the Object has been modified slightly (rounding the corners and adding a label to the Attributes compartment) to better fit the look of the other UML-B diagrams but this only serves to enhance the diagram. The associations in the diagram also meet the requirements.

The functionality for the Object *name* label to show both the Object's name and its class type has not been met as this is not possible in GMF at this time. The requirement for the Attribute *name: value* label to require that all Objects have a class type and that all Attributes have a name has not been implemented due to time restrictions. This does not affect the core functionality of the diagram.

The integration of the Object Diagram into the rest of UML-B has been largely accomplished. The Object Diagram is creatable from both UML-B Machines and Contexts, through the right-click popup menu. It is also possible to open the containing Machine or Context Diagram from the Object Diagram, again, through the popup menu. The ability to create an Object as an instance of an existing Class has not been implemented due to time constraints but this does not adversely affect the core functionality of the diagram.

Overall, the Object Diagram has met enough of the requirements for it to be useful in the UML-B environment. The features that are unimplemented were mostly only required as additional touches to make the diagram easier to use. Their exclusion has not affected the core functionality to any great extent.

9.4 Sequence Diagram

The Sequence Diagram has met all of the points specified in Section 3.4. All of the graphical requirements for the layout and appearance of the diagram have been adhered to and the required similarity to UML 2.0 has been achieved.

The requirement for the diagram to record Event Calls in the vertical order in which they appear in the diagram has been met by re-ordering the XMI produced from the diagram. This is a very important achievement for the diagram as it will make the transition of the diagram into an animation script significantly easier.

The Sequence Diagram is also fully integrated to the UML-B Package Diagram; Sequence Diagrams are able to be linked to Machines and opened from the right-click menu on the Machine. The Sequence Diagram has also integrated the Object Diagram, which can be linked to an Event Call in the diagram and opened accordingly.

Overall the Sequence Diagram has met all of the requirements and is a very useful addition to the UML-B tool.

9.5 Use-Case Diagram

The Use-Case Diagram meets all of the points specified in Section 3.5. The graphical elements used are the same as those in the UML 2.0 Use-Case Diagram and the requirement to have the same look as UML 2.0 has been met. The requirement for the behaviour of the diagrams has also been met. Restrictions on the elements that can be placed into System Boxes have been observed and System Boxes are nestable. The links between the elements in the diagram all conform to the specification in both graphical and behavioural terms.

The integration of Use-Case Diagrams into UML-B has been fully completed. Use-Case Diagrams are integrated to the UML-B Package Diagram and Machines within the Package Diagram. New Use-Case Diagrams can be created through the right-click menus from both of the existing UML-B elements. The parent element can be opened from the child Use-Case Diagram.

Overall, the Use-Case Diagram has met all of its requirements and provides all of the correct functionality to make it a useful tool in the design of UML-B systems.

Chapter 10

Conclusions

The existing UML-B notations provide diagrams that can be used by a developer to graphically create B-method specifications. We have taken these notations and built upon them, creating new diagram types and editors that support not just the formal methods behind UML-B, but the entire software engineering process as well. This project has:

- Proposed five new diagram notations.
- Created supporting metamodels for these notations through EMF.
- Developed graphical editors for each notation through GEF and GMF.
- Defined possible semantics for the elements of each new diagram.
- Advised on possible future uses and work to take advantage of the extentions.

10.1 Project Outcome

This project has been extremely successful, we have produced a wide variety of diagram types that can now be used within UML-B. Implementing these diagrams has meant overcoming a very large learning curve in building the skills to effectively use EMF and GMF to define diagram editors.

Our biggest achievement has been to define possible new ways for the UML-B notation to be used, through utilising Sequence Diagrams to define animation sequences that can be used in a model checker. It is also possible that the Sequence and Object Diagrams could be used to show the system state when a model checker violates the invariant. This is an important achievement as previously UML-B diagrams were only being thought of as being used in U2B translation. Combining Sequence and Object diagrams with the model checking process would open up a new area of development in RODIN and UML-B, with many possibilities for further projects.

10.2 Project Management

The group has performed well, forming an effective and cohesive team in a very short space of time. We have succeeded, especially in terms of work load management, sharing responsibilities and using each others expertise. Our time and efforts have been handled well and we have managed to bring the project to a natural conclusion within the ten week time span.

Bibliography

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[2] P. Amey. Dear sir, Yours faithfully: an everyday story of formality. In F. Redmill and T. Anderson, editors, *Proc. 12th Safety-Critical Systems Symposium*, pages 3–15. Springer, 2004.

[3] C. Aniszczyk. Using GEF with EMF. Technical report, IBM, June 2005.

[4] M. Butler, C. Snook, and I. Oliver. *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[5] D. Craigen, S. Gerhart, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Trans. Softw. Eng.*, 21(2):90–98, 1995.

[6] J. des Rivieres and W. Beaton. Eclipse platform technical overview. Technical report, The Eclipse Foundation, 2006.

[7] M. Fowler. *UML Distilled, 3rd Edition.* Pearson, 2003.

[8] IBM. Rational rose. `http://www-306.ibm.com/software/rational/`, last accessed December 2006.

[9] Borland IDE: JBuilder 2007. `http://www.borland.com/jbuilder/`, last accessed December 2006.

[10] Java Development Toolkit (JDT). `http://www.eclipse.org/jdt/`, last accessed December 2006.

[11] D. Johan. Take control of your properties. *Eclipse.org*, May 2003.

[12] S. M. Kerner. Eclipse adoption on the rise. *internetnews.com*, September 2006.

[13] C. Kobryn. UML 3.0 and the future of modeling. *Software Systems and Modelling*, 3(1), 2004.

[14] D. Lee. Display a UML diagram using Draw2D. Technical report, IBM, August 2003.

[15] M. Leuschel and M. Butler. ProB: A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[16] C. Métayer, J.-R. Abrial, and L. Voisin. Event-b language. Technical report, RODIN, 2005.

[17] B. Moore, D. Dean, A. Gerber, G. Wagenknecht, and P. Vanderheyden. Eclipse Development using the Graphical Editing Framework and the Eclipse Modelling Framework. Technical report, IBM, last accessed December 2006.

[18] OMG. Unified modeling language 1.5 specification. Technical report, Object Management Group, March 2003.

[19] RODIN - Rigorous Open Development Environment for Complex Systems. `http://rodin.cs.ncl.ac.uk/`, last accessed December 2006.

[20] Adobe - Flex 2 - IDE, Eclipse based IDE, Integrated Development Environment. `http://www.adobe.com/products/flex/flexbuilder/`, last accessed December 2006.

[21] SAP - Components & Tools of SAP NetWeaver: SAP NetWeaver Developer Studio. `http://www.sap.com/platform/netweaver/components/developerstudio/index.epx`, last accessed December 2006.

[22] S. Schneider. *The B-Method: An Introduction.* Palgrave, 2002.

[23] C. Snook. UML-B rose script. `http://www.ecs.soton.ac.uk/~cfs/`, last accessed December 2006.

[24] C. Snook and M. Butler. Using a Graphical Design Tool for Formal Specification. In G. Kadoda, editor, *Proc. 13th Anuual Workshop of the Psychology of Programming Interest Group*, pages 311–321, 2001.

[25] C. Snook and M. Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

[26] The Eclipse Foundation. Eclipse forms independent organization. *Eclipse.org*, February 2004.

[27] The Eclipse Foundation. The Eclipse Tools Project, Top Level Project Charter. `http://www.eclipse.org/tools/eclipsetools-charter.php`, 2006.

[28] The Eclipse Foundation. Eclipse Modelling Framework. `http://www.eclipse.org/emf/`, last accessed December 2006.

[29] The Eclipse Foundation. GMF Tutorial. `http://wiki.eclipse.org/index.php/GMF_Tutorial`, last accessed December 2006.

[30] The Eclipse Foundation. Graphical Editing Framework. `http://www.eclipse.org/gef/overview.html`, last accessed December 2006.

[31] The Eclipse Foundation. Graphical Modelling Framework. `http://www.eclipse.org/gmf/`, last accessed December 2006.

[32] A. Tillman. A UML-B Drawing Tool for Eclipse. Master's thesis, University of Southampton, School of Electronics and Computer Science, September 2006.

[33] J. M. Wing. A Specifiers Introduction to Formal Methods. *Computer*, 23(9):8–23, 1990.

Appendix A

Testing Diagrams

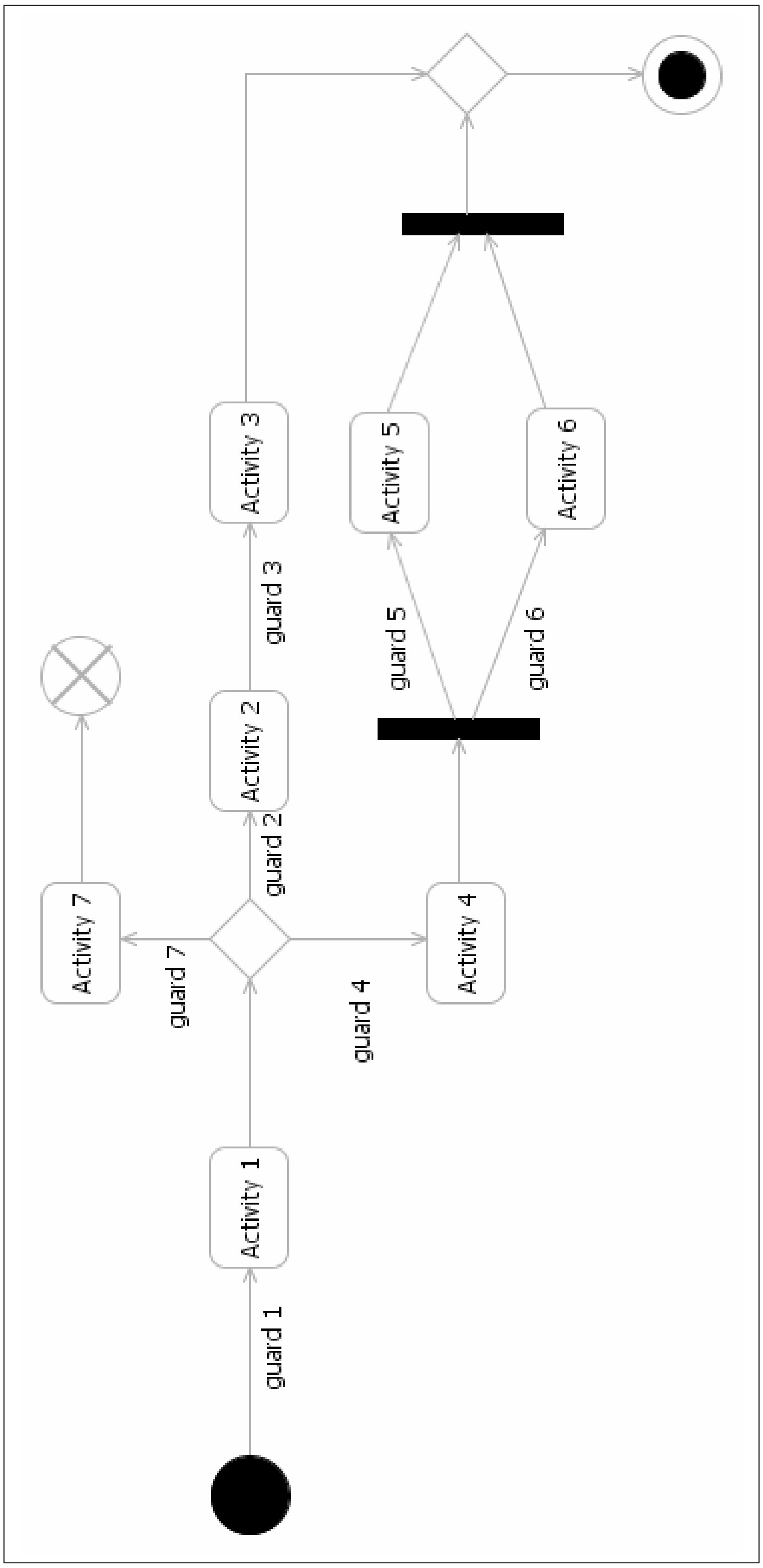


FIGURE A.1: Example of the Activity Diagram with all link types implemented.

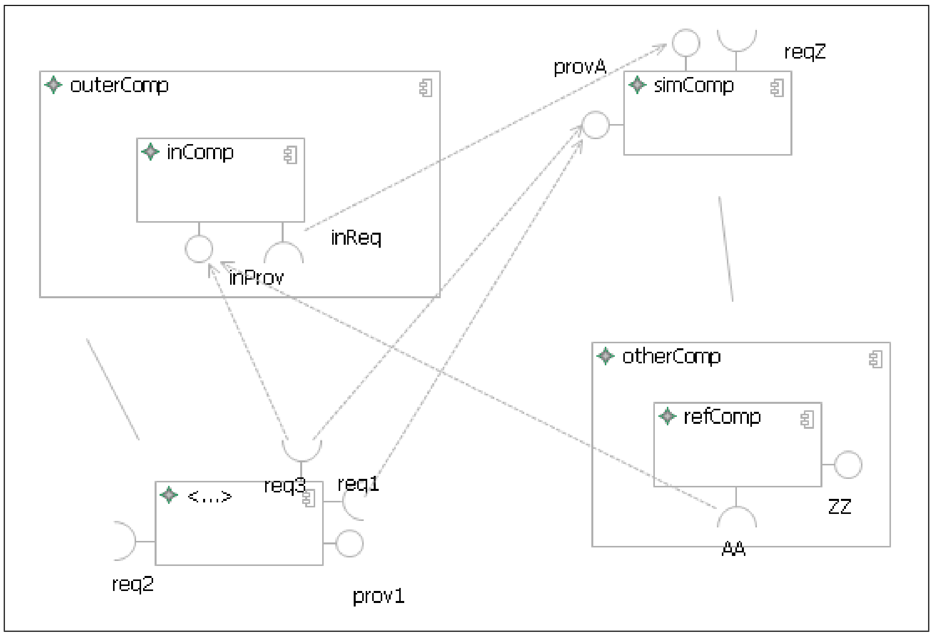


FIGURE A.2: Example of the Component Diagram with all links, elements and nesting features shown.

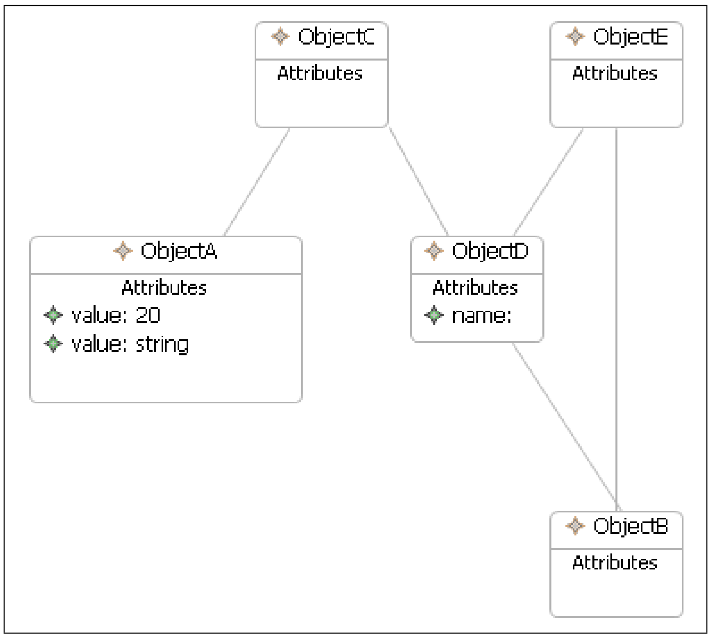


FIGURE A.3: Example of the Object Diagram with all elements and links shown.

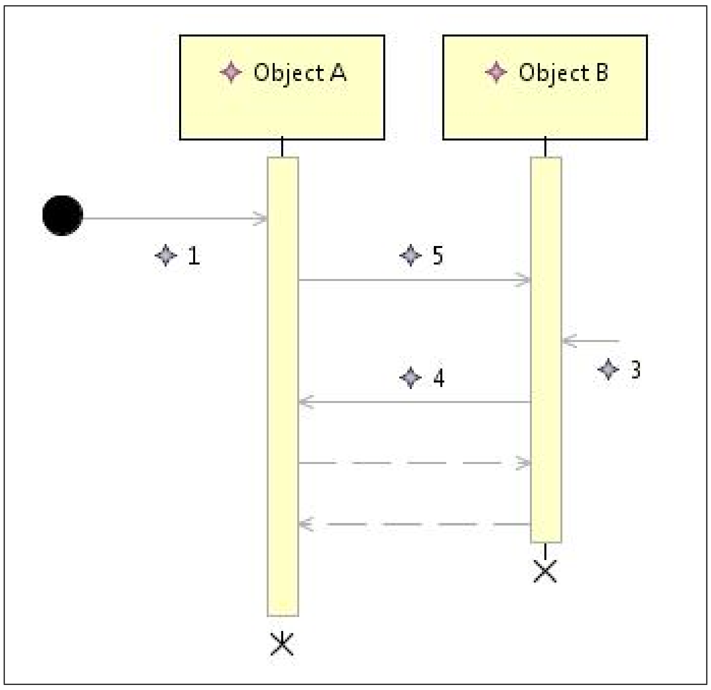


FIGURE A.4: Example of the Sequence Diagram with all elements and links shown.

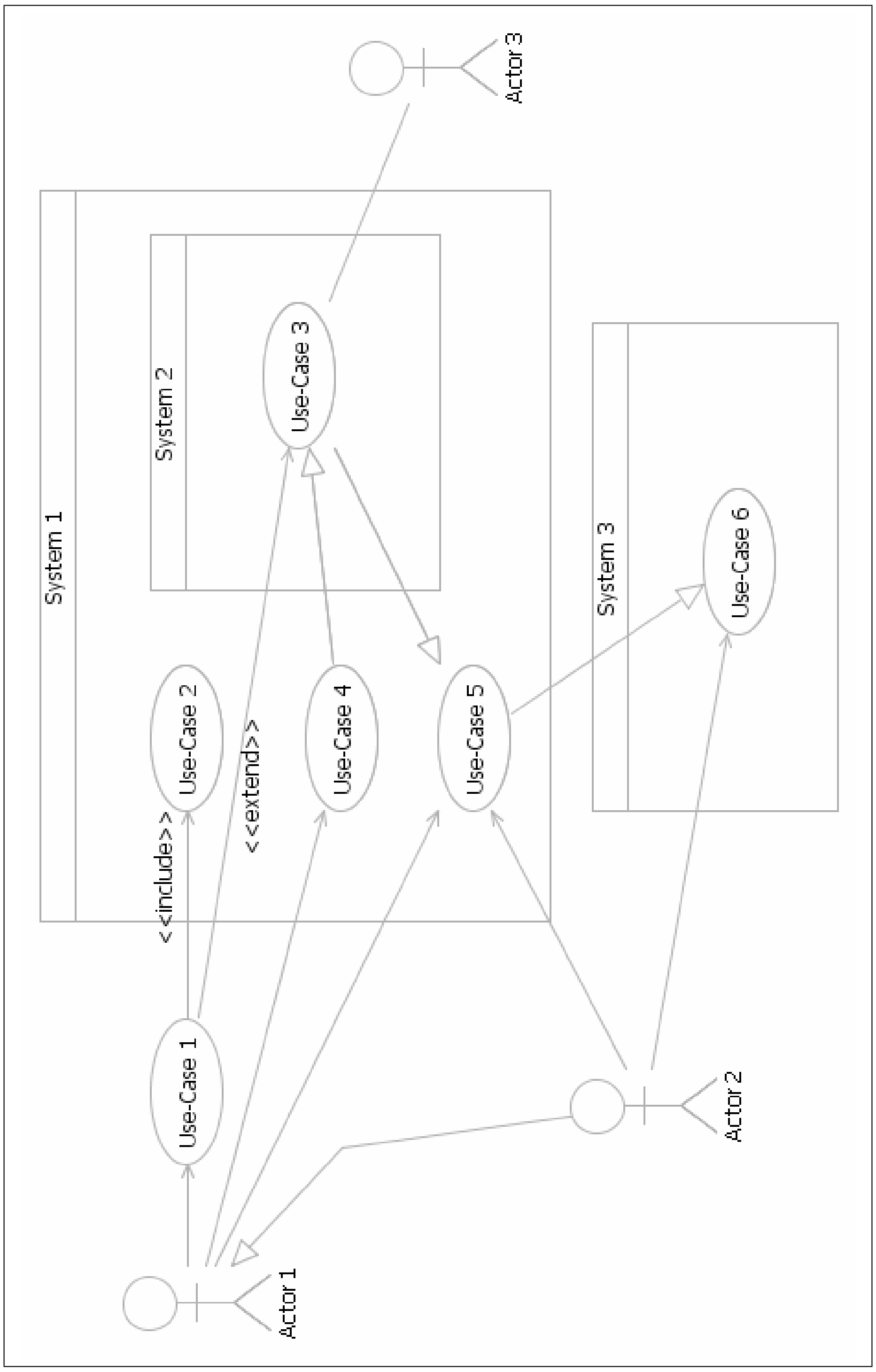


FIGURE A.5: Example of the Use-Case Diagram with all link types implemented. Also shown is the nesting ability of the System Boxes. Note that the links between Use-Cases are not affected by whether the Use-Cases is inside a System Box or not, or whether two Use-Cases are inside the same System Box.

Appendix B

CD Listing

Presented here is a summarised listing of all of the files that have been submitted.

B.1 Basic GMF Required Files

All five of the diagrams implemented have their own set of files that GMF requires to create the diagram type. These files are all stored in *ac.soton.umlb.umlbMetamodel/model/* and have the following file types and meanings as shown in Table B.1.

File Extension	**Description**
.gmfgraph	Contains all of the image data that GMF uses in a diagram. Special cases are for the Use-Case and Activity diagrams, as their .gmfgraph file calls in another .gmfgraph file that contains the drawing elements, this is covered later on.
.gmftool	Defines the order and layout of the toolbar in the graphical editor.
.gmfmap	Combines all elements of the metamodel, .gmfgraph and .gmftool that make up the given diagram.
.gmfgen	Produced by GMF, this is the file that is used to generate the diagram code.

Table B.1: Table showing the files GMF expects and their location in our file system. All files can be found under *ac.soton.umlb.umlbMetamodel/model/*.

The prefixes for each diagram type are:

- *umlbMetamodelActivity* - for the Activity Diagram
- *umlbComponentDiag* - for the Component Diagram
- *umlbObjectDiagram* - for the Object Diagram
- *umlbMetamodelStateDiag* - for the State Diagram
- *umlbMetamodelUseCaseDiag* - for the Use Case Diagram

B.1.1 Activity and Use-Case Diagram Special Case

Both the Activity and Use-Case Diagrams use two .gmfgraph files to define the graphical components. The .gmfgraph files mentioned previously define the top level graphical components, but as both the Activity and Use-Case Diagrams require complicated graphical components these have been defined in secondary .gmfgraph files.

File	**Description**
mygraphs/activityDiagramGraph.gmfgraph	Contains all of the shapes that make up the graphical components found in the Activity Diagram.
useCaseGraphs/useCaseModels.gmfgraph	Contains all of the shapes that make up the graphical components found in the Use-Case Diagram.

Table B.2: Table showing extended GMF graph files we have implemented and their location in our file system. All files can be found under *ac.soton.umlb.umlbMetamodel/model/*.

B.2 Integration Code

All of the diagrams have had code added to them to assist in integrating the diagrams with the rest of UML-B. The integration code for each and its location is summarised in Table B.3.

File	**Description**
Activity Diagram	activitydiagram.custom.OpenParentDiagram packageDiagram.custom.OpenActivityDiagram classDiagram.custom.OpenActivityDiagram
Component Diagram	componentDiagram.custom.OpenParentDiagram packageDiagram.custom.OpenComponentDiagram
Object Diagram	objectdiagram.custom.OpenParentDiagram packageDiagram.custom.OpenObjectDiagram
Sequence Diagram	diagram.custom.OpenParentDiagram diagram.custom.OpenObjectDiagram packageDiagram.custom.OpenSequenceDiagram
Use-Case Diagram	usecasediagram.custom.OpenParentDiagram packageDiagram.custom.OpenUseCaseDiagram

Table B.3: Table of classes used for integrating diagrams into the rest of UML-B. All packages listed are local to the ac.soton.umlb.umlbMetamodel package

B.3 Diagram Specific Code

Presented here are files that contain code that is specifically used by a particular diagram to achieve some behaviour.

B.3.1 Activity Diagram

File	**Description**
UMLB_AD_ElementImpl.java	Contains the code *isLegalToEnd()* and *isLegalToStartFrom()* defined in the Activity Diagram Metamodel.

Table B.4: Table showing files in *ac.soton.umlb.umlbMetamodel.componentDiagram/src/ac/soton/umlb/umlbMetamodel/umlbMetamodel/impl* that are used by the Activity Diagram to implement code defined in the Activity Diagram Metamodel.

File	**Description**
UmlbMetamodelBaseItemSemanticEditPolicy.java	Contains the Java constraints *isLegalToStartFrom()* and *isLegalToEnd()* defined in *umlbMetamodelActivity.gmfmap.*

Table B.5: Table showing files in *ac.soton.umlb.umlbMetamodel.activitydiagram/src/ac/soton/umlb/umlbMetamodel/activitydiagram/edit/policies* that are used by the Activity Diagram to implement code defined as Activity Diagram Link Constraints.

B.3.2 Component Diagram

File	**Description**
Component.java	Code that defines the custom shape of the Component node
ComponentLayout.java	Code that defines behaviour of the custom layout manager
ProvidedInterface.java	Code that defines the shape of the Provided Interface
RequiredInterface.java	Code that defines the shape of the Required Interface

Table B.6: Table showing files in *ac.soton.umlb.umlbMetamodel.componentDiagram/src/ac/soton/umlb/umlbMetamodel/componentDiagram/custom/* that are used by the Component Diagram.

B.3.3 Sequence Diagram

File	**Description**
edit/parts/FixedConnectionAnchor.java	Code for controlling Connection Anchor behaviour
custom/ActivityLayout.java	Layout code for the activity box
custom/SequenceLayout.java	Layout code for the diagram
custom/ObjectShape.java	Code for generating a custom lifeline shape

Table B.7: Table showing files in *ac.soton.umlb.umlbMetamodel.diagram/src/ac/soton/umlb/umlbMetamodel/diagram/* that are used by the Sequence Diagram.