Advanced tool support for requirements engineering

Jakob Kragelund
S062399

Kongens Lyngby 2012

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Abstract

The goal of this thesis is (1) to create a set of visual editors for capturing UML model fragments that occur in requirements engineering; and (2) to create functions to extract, merge, and export these fragments into a draft analysis model. This tool is intended to be used in course 02264 (Requirements Engineering) at the Technical University of Denmark, and will focus on the elements taught in the course.

The model fragment editors will provide very basic functionality only, and the project will extensively use Eclipse/EMFs features for creating these editors automatically. The main focus will lie with the higher level functions of weaving model fragments, but some effort will also go into integrating these features into a parallel project providing basic tool support for requirements engineering.

Preface

This is the Master's Thesis for Jakob Kragelund presented to the Technical University of Denmark as partial fulfillment of the requirements for obtaining the Master's Degree in Digital Media Engineering at the Department of Informatics and Mathematical Modelling. This thesis has been written during the period of September 1st 2011 to Feburary 27th 2012, under the supervision of Prof. Dr. Harald Störrle, and is worth 30 ECTS credits.

I would like to thank **Anders Friis** for having been a great partner these past six months. You have been responsible for getting me motivated, cheering me up when everything has been difficult and painful, and celebrating the many victories we have shared. Without you, this thesis would not been anywhere the state it is in today.

I extend my thanks to my friends and family, especially **Lotte Hansen**, who have been overbearing and supportive when my head and time has been filled with nothing but work.

Finally, I thank **Harald Störrle** for being a great supervisor. You have always had time for me, with both conceptual, technical and motivational support. You have given me much more than I expected, and I hope that my work will be of benefit for you in the future.

IMM, DTU, February 27th 2012.

Jakob Kragelund, s062399

Contents

Abstract **i**

Preface **iii**

1 Introduction **1**
1.1 Motivation 2
1.2 Goals 3

2 Background **5**
2.1 Requirements Tracing 6
2.1.1 Requirements Tracing in relation to RED 6
2.2 Model Weaving 8
2.2.1 UML Package Merge 9

3 Analysis **13**
3.1 Context 14
3.1.1 University course 14
3.1.2 Users 15
3.1.3 Other systems 15
3.2 Creating model fragments 16
3.2.1 Restriction of model types 17
3.2.2 Reducing notational complexity 22
3.2.3 Choice of UML elements 23
3.2.4 Sketchiness 26
3.3 Weaving UML fragments 27
3.3.1 Weave specification 28
3.3.2 Manual weave specification 35
3.4 Improvements 36
3.4.1 Locking 36

3.4.2 Visual Folder Editor 38
3.4.3 Comment export 39
3.4.4 Navigation 39
3.5 Domain model 40
3.5.1 RED Core meta-model 40
3.5.2 Model elements domain model 41
3.5.3 Weave data model 41

4 Design 45
4.1 Technology 45
4.1.1 GMF – Graphical Modeling Framework 45
4.2 Model Fragment Editor 48
4.2.1 Editor functionality 49
4.3 Weave editor 51
4.4 UML weaving 51
4.4.1 Weave specification 53
4.4.2 Conversion to Prolog 57
4.4.3 Weaving process 59
4.4.4 Convert result to Java 62
4.4.5 Reviewing the Weave Result 63
4.5 Weave Result editor 64
4.6 Improvements 65
4.6.1 Locking 65
4.6.2 Reporting 68

5 Implementation 71
5.1 Structure 71
5.1.1 Plugins 72
5.2 External libraries 73
5.2.1 JPL – Java Prolog Bridge 73
5.3 GMF 74
5.3.1 Fragment Editor 74
5.3.2 Weave Editor 78
5.3.3 Sketchiness 81
5.4 Installation procedure 85
5.4.1 Users 86
5.4.2 Developers 86

6 Evaluation 89
6.1 Evaluation criteria 89
6.2 Usability 90
6.2.1 Performance 91
6.3 Functionality 92
6.3.1 Completeness 92

6.3.2 Correctness 93

7 Conclusion 97
7.1 Limitations and Future Work 98

A Performance comparison of reflective versus non-reflective visitor implementation 101

B Weaving user guide 105
B.1 Creating model fragments 105
B.2 Creating a weave model 106
B.3 Specifying weave details 108
B.4 Evaluate the weave result 108

Chapter 1

Introduction

This thesis describes the work of designing and implementing tool support for the requirements engineering process, with the purpose of alleviating some of the communication issues which might typically arise between clients and developers during a software development project.

The background for the development of this tool, called the Requirements Engineering eDitor (RED), is the course 02264 Requirements Engineering at the Technical University of Denmark, which lacks decent tools to support the students' work in creating requirements specifications. The existing tools on the market have flaws such as cost, stability, usability or lack of specific features to support the course syllabus. This thesis, together with Friis (2012), tries to solve this problem, by building a requirements specification tool from scratch, with focus on the course syllabus.

While Friis focuses on the basic functionality of such a tool such as editors for various textual artifacts, reports, glossaries etc., this thesis will focus on creating visual editors for capturing UML fragments which may occur during requirements engineering. These UML fragments are used to model small parts of the system, and accompanies the textual representation of a requirement, in order to clarify the intention and meaning. In addition, some basic functionality is created which allows the semi-automatic extraction and merge of these fragments into a draft analysis model.

1.1 Motivation

The requirements engineering process is a set of activities related to gathering, documenting, maintaining and communicating the requirements, goals, context and purpose of a system in which software is a major component. Note that requirements are not only specified during the initial stages of a project's lifecycle, but should be maintained and revisited continuously during the life of the project. Requirements engineering activities include (but are not limited to)

- Requirements elicitation
- Requirements analysis and negotiation
- Requirements specification
- System modelling
- Requirements validation
- Requirements management

(Sommerville and Sawyer, 1997)

These activities will naturally involve a subset of the stakeholders of the system, such as analysts, developers, system architects, the client, the end users, domain experts, etc. Stakeholders communicate their requirements to the analysts, who may specify them and communicate them back to the origin for validation, and to other stakeholders such as developers, who need the requirements.

Al-Rawas and Easterbrook (1996) describe a number of communication problems, which are introduced during this process. Of relevance is the issue of notation, which means that end users prefer to talk about their requirements and the system in terms of its general behaviour, functionality and applications, and they prefer to do so in a natural language. On the other hand, analysts, architects and developers prefer a more technical notation, and talk about a system in terms of procedures, objects, data structures using a formalised language, such as UML. An example of how this is is manifested is given by Al-Rawas and Easterbrook (1996, p. 5):

> ,,*Regardless of the chosen notations, most users express their requirements in natural language. Then it is the job of the analyst to translate requirements statements into some kind of representational objects in a domain model. Once the requirements are modelled, they*

> *are presented to endusers for validation. At this stage the analysts are faced with another communication problem when endusers are not familiar with the notations used to model their requirements."*

This means that requirements validation becomes difficult or impossible to accomplish, especially with large and/or complex domain models, as the people who are supposed to validate that the given requirements have been adequately covered by the design, are unable to understand or comprehend the design itself.

Another issue raised by Al-Rawas and Easterbrook (1996, p. 5) is that requirements specified by the end users typically do not suit the developers very well, as described by this next quote.

> *,,On the other hand, when analysts, under pressure to keep up with the project schedule, pass raw natural language requirements to programmers, then time is wasted in trying to interpret them. A programmer we interviewed during our empirical study complained that he often has to read large amounts of text in order to understand a single requirement, which could have been represented very concisely using a diagram or a formal notation."*

1.2 Goals

The main goals of this thesis is to build upon the requirements engineering tool described in Friis (2012), to advance its usefulness for classroom usage for the 02264 Requirements Engineering course at the Technical University of Denmark.

This is primarily done by extending the tool with UML modelling capabilities, allowing users to create and attach UML model fragments which are produced during a requirements engineering process, to requirements in the tool. In addition to using these fragments for documentation purposes, there will also be methods for extracting them, and weaving together sets of fragments into a more comprehensive model.

The purpose of this primary functionality is twofold: improve communication between the analyst and client on one side, and the developer on the other, by adding a more formal notation in addition to the existing textual specification; and lay the foundation for improved requirements traceability in the tool, by enabling linking between requirements and the following analysis model.

In addition to this primary functionality, a number of minor features will be added to the tool as well. These features were discovered by Friis when evaluating the first version of the tool, and will provide a number of functionality and usability improvements.

To summarise, this thesis will

- Add UML modelling capabilities to the requirements engineering tool created by Friis
- Provide functionality for extracting and weaving UML model fragments
- Improve the existing tool with functionality requests identified during the evaluation by Friis

In the following chapters, these goals will be converted to a set of concrete problems. In Chapter 3, the problems and solutions are discussed at a high level of abstraction, outlaying the direction taken to achieve the stated goals. In Chapter 4, the various design decisions taken are covered, which in a higher level of detail states how the problems are met. In Chapter 5, various low-level implementation details are discussed, before evaluating the solution in Chapter 6.

Chapter 2

Background

This chapter provides some background knowledge for the areas of requirements tracing and model weaving. This is done by looking at the current literature for definitions and usage of the terms, which gives an insigh into what the purpose of tracing and weaving is and how it is used.

Requirements tracing underlies one of the primary goals of this thesis, that of providing some support for adding vertical traces from requirements to design. The RED tool already supports some tracing mechanisms, and this thesis seeks to expand the tool's capabilities in this area.

Model weaving is used to implement the tracing mechanisms, in manners which are covered in later chapters. A short introduction to model weaving, with focus on the application of weaving UML models, is given.

Other areas which are covered by this thesis, such as requirements engineering and UML, are not covered in this chapter, and it is assumed that the reader has some general knowledge of these areas.

2.1 Requirements Tracing

Requirements Tracing refers to the ability to *,,describe and follow the life of a requirement, in both a forward and backward direction"*, ideally through the whole system's life cycle (Gotel and Finkelstein, 1994).

Jarke (1998) lists four kinds of traceability links, which distinguishes how the trace relates to requirements:

- *Forward from requirements* assigns responsibility for implementing a given requirement, and makes it possible to analyse the impact of changing a requirement
- *Backward to requirements* shows that the system maps back to the requirements
- *Forward to requirements* establishes a link from client needs and technical assumptions to requirements, and enables analysis of changes to these underlying factors
- *Backward from requirements* are used to validate the requirements based on the underlying factors

The first two types are commonly called post-traceability, while the latter two are called pre-traceability, and together, they serve an important role in both validating requirements, and estimating the impact of changes. Post-traceability is usually considered an easier problem than pre-traceability, since the first deals with traces of various manual and automatic transformation steps from requirements to the completed system, which are more or less formal, while the latter has to cope with communication problems and informality (Winkler and von Pilgrim, 2010).

In addition to these categorisations, a distinction is also made between *horizontal* and *vertical* traceability. The first describes tracing links between artifacts belonging to the same level or phase of a software development project, while the second covers links across such phase boundaries. An illustration of the various concepts are shown in Figure 2.1.

2.1.1 Requirements Tracing in relation to RED

Requirements tracing is an important tool for managing the lifecycle of requirements, and analysing the impact which proposed changes will have on the

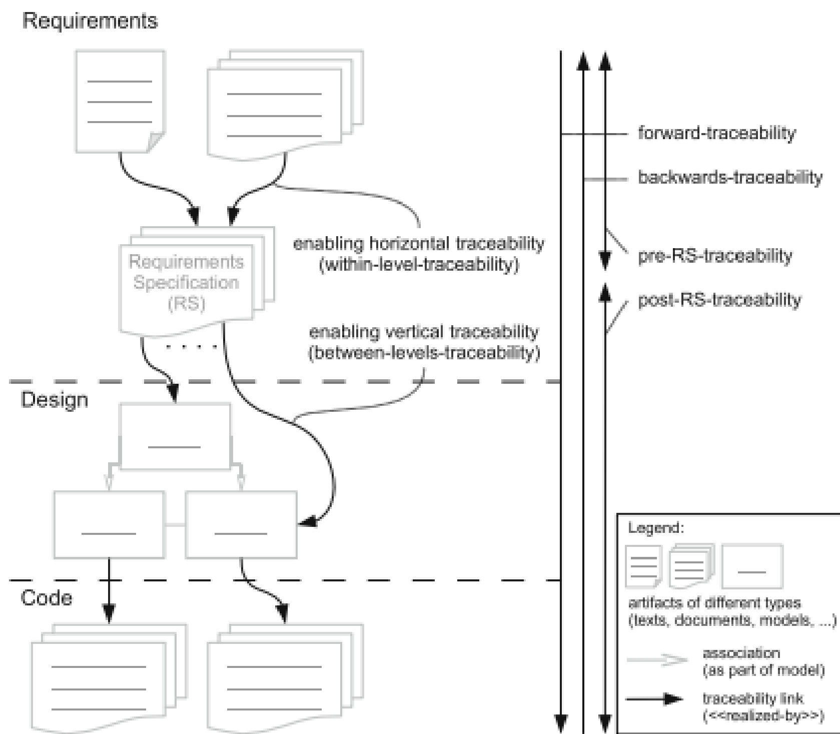


Figure 2.1: Traceability concepts, courtesy of Winkler and von Pilgrim (2010)

complete system. Tracing, however, can be an arduous process, requiring a lot of manual interaction. Complete traceability of a system is often close to impossible which means that automating the tracing as much as possible can provide great benefits by easing the task of the analyst.

The core functionality of the RED tool allows some basic tracing. Winkler and von Pilgrim (2010) lists six core questions about artifacts, which can be answered by traces:

1. What information is recorded in the artifact?
2. Who has created or updated the artifact?
3. Where has the information come from?
4. How is the information represented?
5. When has the artifact been created, modified or evolved?
6. Why has the artifact been created, modified or evolved?

The core functionality in RED automatically keeps track of the latest changes to artifacts, though it does not keep a complete history. It also allows the assigning of artifacts to specific people, which keeps track of responsibility for various requirements. An important feature of RED is that it enables linking between different, arbitrary artifacts in the tool, both as references, or as dependencies, exclusions etc. This allows some horizontal trace links, though they are to a large degree created manually. Tracing backwards from requirements is also supported to some degree, by the ability to document stakeholders and goals, and create links between these and requirements.

As will be described in later chapters, this thesis looks at adding vertical trace links to the tool as well, by making it possible to connect individual requirements to UML models, which in turn can be converted into code. This opens the prospect of having a trace link automatically created from the requirements documented in the tool, through design models, to the final code.

2.2 Model Weaving

In model driven software development, models are first class entities rather than simply documentation artifacts. In order to utilise models in this way, a lot of

work has gone into developing methods for working with models, transforming them and using them as a replacement for source code.

Model transformations can be thought of as functions – they take one or more models as input, perform some operations and produce an output model. Weaving is a special case of model transformations. Fabro et al. (2005) notes that weaving differs from the traditional understanding of model transformations on three accounts, which can be used to get a better understanding of the definition of weaving:

- A model transformation usually takes one model as input and produces another as output, though extensions may allow multiple inputs or outputs. Model weaving basically takes a number of models as input in addition to a weaving metamodel, and produces a single output model.
- Transformations are usually fully automatic, while weaving may need heuristics or user guidance.
- There is no standardised weaving metamodel, as the application of weaving is application specific. On the other hand, model transformations may conform to a fixed metamodel, which is the metamodel of the transformation language.

The benefit of model weaving comes from the ability to use the principle of *separation of concerns*. The ability to weave models together allows the construction of small models, each describing individual concerns, instead of having large monolithic models which are difficult to maintain and reuse, and close to impossible to extract meaningful information in an easy manner.

The area of aspect-oriented software development has many of the same goals as model weaving, in terms of separation of concerns, which shows in the fact that a lot of work is going into researching how model weaving can be used to implement aspect-oriented modelling.

2.2.1 UML Package Merge

The UML Infrastructure defines the concept of a *package merge*, which is *,, … a directed relationship between two packages, that indicates that the contents of the two packages are to be combined."*(OMG, 2011).

The package merge allows one to model different aspects of a single concept in different packages, so as to divide the modelled concepts into smaller and

simpler models for clarity, readability, etc. This resembles the model weaving described above, and can serve as inspiration for the model weaving which is done in this thesis.

An example of how the package merge works is shown in Figure 2.2. In this example, we have four packages P, Q, R and S, each modelling different aspects of elements A, B and C (top). The syntax for the merge is shown as a directed dotted connection labeled ≪*merge*≫, between two packages. The package from which the connection originates is termed the *receiving package*, while the package which is the target of the connection is termed the *merged package*. The result of the merge is termed the *resulting package*.

The connection implies a set of transformations, in which the contents of the two packages are combined. This combination basically does two things:

1. if an element appears in either package, it appears in the resulting package.
2. if two elements in the two packages represent the same element, the two elements are merged together into a single resulting element according to some formal rules.

The result of these transformations can be seen in the bottom part of Figure 2.2. Note that the merge connection implies the actual transformations, which means that the top model is semantically equivalent to the bottom model.

In the bottom part of the figure, which shows the result of the package merge, each element is shown with an annotation which shows the origin of the element. For example, the annotation `S::D` in package S means that D originally came from S, while `Q::C` in S shows that C came from Q. The notation `Q::A@S::A` means that the element A in package Q were merged with A in package S to form the resulting element.

The specification of the UML Package Merge limits the definition of transformations to a few metatypes, such as Packages, Classes, Associations, etc., since *„the semantics of merging other kinds of metatypes (e.g. state machines, interations) are complex and domain specific"* (OMG, 2011). The work on model weaving in this thesis goes beyond these basic metatypes, and looks at methods for weaving a larger set of UML metatypes together. The approach to model weaving in this thesis differs from the UML package merge by relying on user input, guiding the weaving process to a certain extent. This is covered in further detail in Section 3.3.

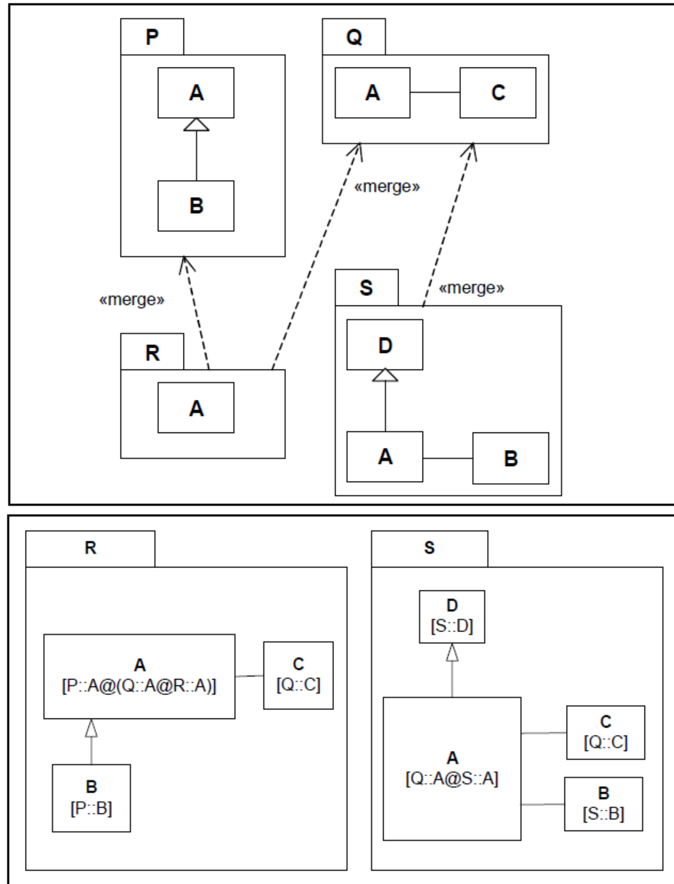


Figure 2.2: UML Package Merge

Chapter 3

Analysis

This chapter analyses the goals, requirements and problems stated in the previous chapters.

The chapter starts by providing the context for the thesis and the tool which is to be implemented, by giving an overview of the environment of users and neighbouring systems in which the tool will exist. Then, a detailed description of this thesis' major parts are covered. These parts can be grouped in three categories: creating model fragments, weaving UML fragments, and general functionality and usability improvements.

The first two categories cover the primary goals described in Section 1.2, and are described in detail in Sections 3.2 and 3.3 while the third is covered in Section 3.4.

As pointed out in Section 3.1.2, one can choose to view the users of the tool in two ways: as professional analysts, working with requirements engineering in the industry; or as students, working with cases in a classroom setting. For the purpose of the following analysis, many of the decisions will be argued from the point of users being professional analysts, since this is the role the students will be taking on as well.

3.1 Context

This section describes the context in which this thesis is implemented, by looking at the users and systems which constitutes the environment of the tool developed in this thesis.

The purpose of this thesis is to help bridging the gap between domain experts and technical experts. This is done by identifying areas where communication is likely to break down or cause misunderstandings during a software engineering process.

Besides the general purpose, a more specific goal is to deliver a tool to the students of the course *02264 Requirements Engineering* at the Technical University of Denmark (DTU).

3.1.1 University course

The course *02264 Requirements Engineering* at DTU is a course for graduate and advanced undergraduate students aiming at *,,[providing] a hands-on introduction to requirements engineering (RE) based on case study work"* [1]. The focus on case study work is what drives the need for tool support, as the students are required to create substantial requirements specifications.

Friis (2012) argues why a custom built tool is the best solution for the course, rather than using an existing tool. His most important points are:

- Price: the tool needs to be provided to students free of charge
- Syllabus coverage: all important elements from the course syllabus should be readily supported by the tool, so it is easy for students to apply what they learn.
- Usability: students should spent time learning the course material, so the tool should have a gentle learning curve.
- Extendability: future course elements will need to be incorporated in the tool.

[1] http://www.kurser.dtu.dk/02264.aspx?menulanguage=en-GB

Based on these requirements, a custom built tool is deemed to be the best solution, especially with the prospect of having future students extend the tool with new components.

When in doubt concering a certain feature or design decision, the solution that better fits the course is preferred.

3.1.2 Users

Given that this tool is designed and built specially for a specific course at DTU, a number of things can readily be said about the expected users of the tool, based on experiences from previous years.

- The users come from multiple nationalities, meaning that all material regarding the tool must be written in English. Localisation can be considered as a possible future improvement.
- The majority of the users likely has some programming experience, but this is not a requirement for the course, so it should not be a requirement for the tool.
- The users are, however, proficient at working with computers and learning new tools, meaning that a steeper learning curve can be accepted.
- Users will also have a number of expectations from their experience with similar types of application in terms of usability and functionality, which will have to be covered.

With a wider perspective, one can also look at the tool as being used in the software industry for requirements engineering. Here, the target users are software analysts, architects and their clients. Other users, such as developers, project leaders and management may be the target of the various documents produced by the tool, such as reports or diagrams.

3.1.3 Other systems

In addition to RED, a few other systems are being developed for use in the course at DTU. Most notably is *FIT* (Formal Inspection Tool), which has previously been used during teaching. FIT is, as the name implies, a tool for managing formal inspections, and provides an online inspection support system, which

allows formal inspections of a wide range of development artifacts (Petrolyte, 2011). It is of interest since the requirements specifications created in RED will likely at some point be the subject of a formal inspection. Because both tools are to be used in the same context at DTU, some measure will be taken to integrate them, in order to support the students when tasked with inspecting their requirements specifications.

3.2 Creating model fragments

A main goal of this thesis is to improve the communication between developers and clients of software development projects. Al-Rawas and Easterbrook (1996) point out that while the customer has a deep understanding of the domain, and is able to articulate their needs in the form of natural language, the developer has a need for a more formal notation with less room for ambiguity.

There are two issues which need to be resolved. First, if a software specification is solely text-based using natural language, it tends to become very verbose in order to cover every aspect thoroughly and unambiguously. Such a long specification is difficult to maintain, it is difficult to verify that it is complete, and from the developers viewpoint, it is difficult and cumbersome to read and transform into code.

Second, when the specification has been transformed into a design, analysts must be able to convince the client that the requirements are adequately covered by the design, keeping in mind that the client may not be familiar with the design notation.

In order to solve the first issue, a concise notation, such as a graphical modelling language, should be incorporated in the specification process very early in the software development phase, when the requirements specification is still being created. When the analysts and domain experts are analysing the requirements of the proposed system, and documenting these in a requirements specification, they can benefit from creating a number of small and simple graphical models at the same time, to clarify the meaning of and remove ambiguities in textual specification artifacts. Graphical models are proposed, since visual sketches provide a high *perceptual* and *conceptual bandwith* (Forbus et al., 2001), meaning that they convey a lot of information in a concise manner. They also involve the powerful visual apparatus of the reader, which provides excellent support for recognising shapes and patterns. This leads to the hypothesis that a number of simple sketches can improve the understanding of an accompanying text-based specification.

Creating these graphical models requires some knowledge of the syntax of the modelling language used, and some experience in order to comprehend the diagrams. However, if the diagrams are kept small and simple, even almost trivial, the barrier of entry is lowered significantly. Siau and Lee (2004) concludes that use of Use Case diagrams and Class diagrams are valuable in the requirements analysis process, even though Class diagrams may be more difficult to interpret. This is mostly due to them having a higher structural complexity. Manso et al. (2009) concludes that structural complexity and size of Class diagrams correlates with cognitive complexity, meaning that larger Class diagrams using complex notation are more difficult to comprehend. Therefore, one could hypothesise that small and simple Class diagrams (and other, typical structurally complex diagram types), which use just a subset of the notation and are limited to a small number of elements, are valuable in the requirements analysis process, while larger and more complex diagrams are less so.

Given that simple diagrams are valuable for requirements analysis, this thesis seeks to support this idea by implementing a diagram editor, and integrating it in the RED tool. This diagram editor should support a visual modelling language which offers both a relatively simple notation, which with some introduction can be understood and interpreted by, for example, a domain expert with little technical background, and also enough expressiveness to enable the developer, who will be reading the diagrams, to effectively transform the diagrams and requirements into code.

UML is chosen as the modelling language, since it offers a relatively simple notation, and because it is a de facto industry standard, making it more useful for developers.

3.2.1 Restriction of model types

Since UML is a very big language, and not all parts are equally used for requirements engineering, we restrict ourselves to the most important notations of UML.

Dobing and Parsons (2005) analyses the current use of UML through a survey among software industry professionals, and performing a statistical analysis of the results. Some of their results are summarised in Figures 3.1 and 3.2. These figures show that all of the surveyed components are used to some degree, with varying levels of perceived effect.

Of interest is especially Figure 3.2, as it deals with the client's role with regards to the use of UML. The high level of usage, perceived value and client

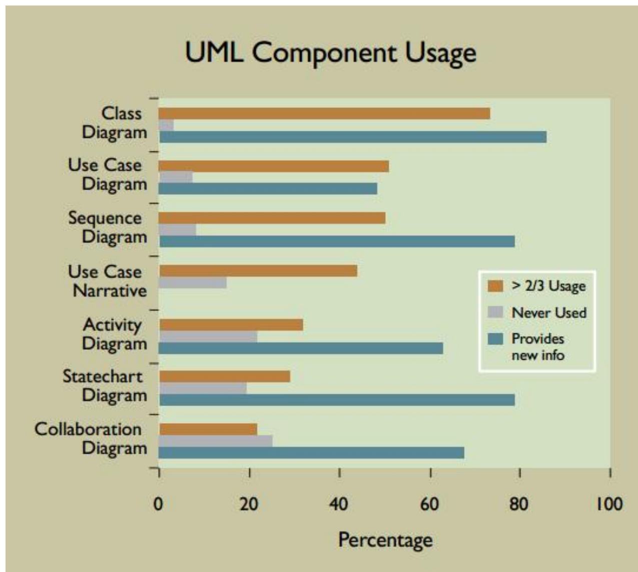


Figure 3.1: The usage of UML components in the industry. Source: Dobing and Parsons (2000)

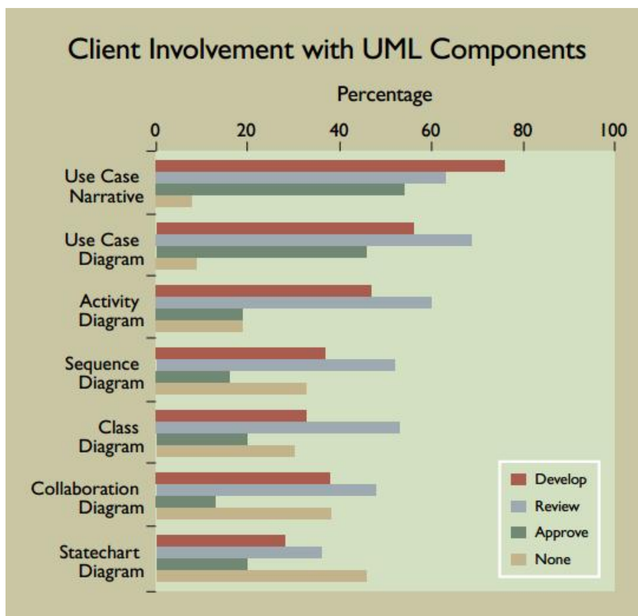


Figure 3.2: Client involvement with UML components. Source: Dobing and Parsons (2000)

involvement in all types of components makes every component a valid candidate for further work in this thesis. In order to decide which, if any, components should be deferred, an analysis of each component follows, along with an evaluation of the difficulty of including each component. As described in Section 4.1, the architectural decisions taken in the beginning of the development of RED means that GMF (Eclipse Graphical Modelling Framework) is the most obvious choice for implementing graphical UML editors, which places some technical limitations and difficulties influencing the choice of which UML components to include.

Use Case Narrative

The Use Case Narrative is the 4th most used component, according to the study, and the one with the most client involvement. Narratives require an editor with text-based widgets, and is typically set up in a tabular format. The creation of a Use Case Narrative editor should be straight-forward, however integrating it with the graphical editors might be more difficult. One can imagine that the user would want to click on a Use Case in a Use Case diagram, and a Use Case Narrative editor would open.

The study indicates that the Use Case Narrative should have a high priority. However, the component is not immediately required for the course for which this tool is built, which means that it can be deferred, if necessary.

Use Case diagram

Use Case diagrams often serve to give an overview of the system under development which, in terms of client involvement, appears to be very valuable, as it is the component with the second highest client involvement right after Use Case Narratives.

This level of client involvement, and the fact that Use Case diagrams are highly used in the course, makes them a prime candidate for inclusion in this thesis. The technical difficulty of implementing this component seems very low, since Use Case diagrams to a large degree consist of just Actors, Use Cases, and connections between these.

Activity diagram

While not being used as much as some other components according to the study, this diagram type seems to have a high factor of client involvement when it is used, and it is also used to some extent in the course.

Activity diagrams are relatively simple to implement, consisting of a number of nodes and connections between these, which means that no custom functionality is required. This makes Activity diagrams a low-effort, medium-benefit diagram type.

Sequence diagrams

Sequence diagrams seem to be used relatively often, and provide much new information when used.

An issue with Sequence diagrams is that they differ from the other components in that the graphical layout of the diagram, i.e. the placement of graphical elements in relation to each other, has a semantic meaning. Sequence diagrams are used to model how objects interact with each other, and specifically, the timing of these interactions. GMF is not suited to let the diagram layout have a semantic meaning, which means that this diagram component requires a lot of customisation. This is countered by the apparent popularity, but the question remains whether the benefit outweights the work required to get the diagram type implemented properly.

Class diagrams

The Class diagram is by far the most widely used UML component of the ones surveyed, which matches the general opinion and experience of the author.

The complexity of Class diagrams are relatively low, consisting of a set of nodes (Classes, Interfaces, Enumerations) and connections (Associations and Generalizations), making them a low effort, high value component and an obvious candidate for inclusion in the tool.

Collaboration diagrams

Collaboration diagrams seem to be the least used types of diagram, and one which is rarely developed with or reviewed by clients. It is rarely used in the course, if used at all. Collaboration diagrams often fill the same role as Sequence diagrams, and the two diagrams are indeed *isomorphic*, meaning that they express the same information, and can be converted without loss of information (Booch et al., 2004). While Sequence diagrams focuses on the timing of interactions between objects in the system, Collaboration diagrams focuses on the organisation of objects and how they interact.

Given the low usage and client involvement, Collaboration diagrams are given a low priority, despite them being seemingly easy to implement.

Statechart diagrams

Statechart diagrams is the component which is used least in collaboration with the client. It is not used very much overall, however it seems to provide a high amount of information when used. This indicates, that in the cases where it is used, it is very valuable.

Statechart diagrams are, like Activity diagrams, comparatively easy to implement. They consist of States of various kinds, Composite States, State Transitions, and State Machines which can contain a set of States.

Given the estimated low amount of work involved in implementing them, and the value they seem to bring when used, they are given a medium priority.

Choice of UML components

Based on the analysis of the various UML components above, the diagram types shown in Figure 3.3 have been chosen. These are deemed the most important, except for Sequence diagrams, which are excluded for technical reasons.

The Class Diagram is chosen due to its widespread use and popularity, and the documented value it provides. Use Case Diagrams are included because they are extensively used in collaboration with clients. Statechart and Activity Diagrams are included instead of Sequence and Collaboration Diagrams due to their simplicity and use. The graphical notation for Statechart and Activity

diagrams is very similar, which implies that a high amount of reuse can be utilised to speed up the development time.

Sequence and Collaboration Diagrams will not be added in the first version developed during this thesis, due to the graphical complexity of Sequence Diagrams, and the low usage of Collaboration Diagrams in the course.

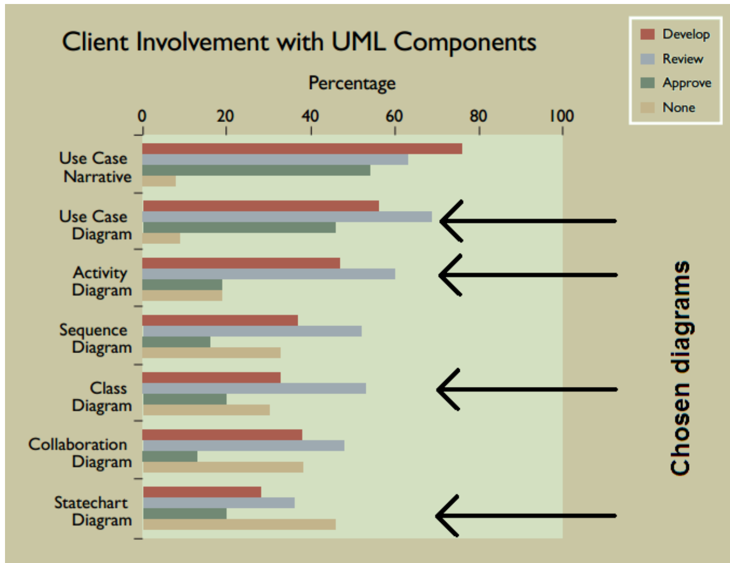


Figure 3.3: Chosen UML diagram types

3.2.2 Reducing notational complexity

In order to maintain a gentle learning curve for the users, and avoid alienation of those unfamiliar with UML, the creation of easily understood diagrams should be kept as simple as possible. This is done in two ways:

- Removing constraints from the UML specification. This implies allowing the user to create diagrams which are not strictly valid UML. This has several advantages – it increases the expressiveness of the diagrams, allowing the user to express things in a simple manner which would not be possible according to the UML specification; it also reduces the need for the user, or anyone else consuming the diagrams created in the tool, to learn all the details of the UML notation. On the other hand, it means

that models produced may not be valid UML, which may result in other issues.

- Removing all but the most basic, intuitive, and most commonly used elements. This also reduces the amount of notation which needs to be learned by users not familiar with UML. In addition, removing anything but a set of rather simple elements promotes simplicity, which is important in the fragment modelling strategy used in the tool.

An issue with the first point is that the models produced by a user may not be strictly valid UML. This poses no apparent issues when the models are confined to the tool, or exported in read-only formats for use in documentation. However, if the models are to be exported, and then imported into another tool for further work, then issues may arise. A point to note, though, is that simply removing various constraints from the UML specification still makes it possible to export the models created in a valid XMI format, which is the standard interchange format for UML models. Thus, whether or not this approach is a good idea or not is an open question, which remains to be resolved.

3.2.3 Choice of UML elements

Only a subset of the UML notation will be added to the modelling part of the tool. This section describes which elements are included, and which are not.

The choice of which elements should be included, is not based on any thorough research. It is decided based on an initial evaluation of how the modelling component will be used. Work could be done to investigate a more optimal selection of model elements, based on the needs of the users.

A summary of the chosen UML elements are shown in Figure 3.4. The decision for which elements are included, and which are not, are given below.

Class Diagram

Class diagrams are kept relatively simple. This is done to promote simple diagrams focusing on concepts, rather than various design details, which should not be added at such an early stage. Class diagrams which contain too many details tend to become large and complex rather quickly, which should be avoided for the intended model fragments.

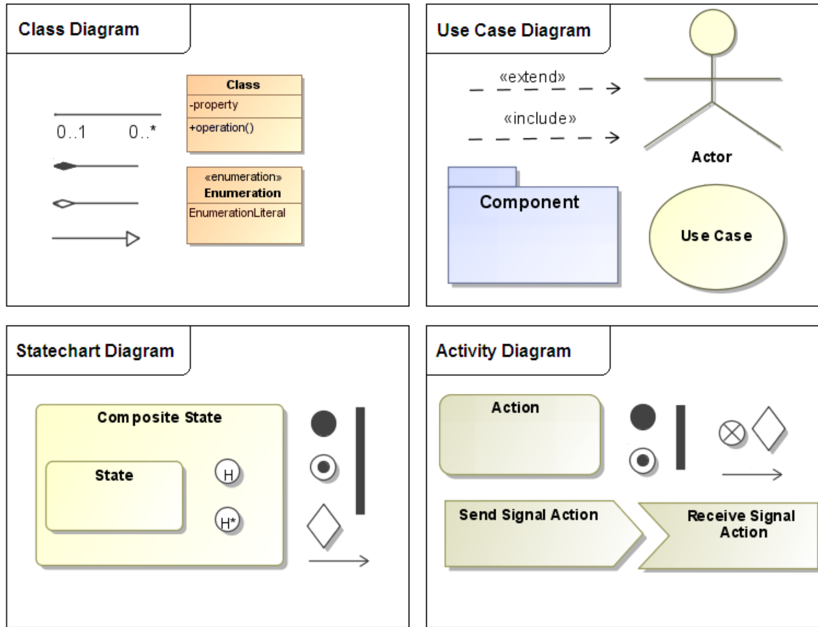


Figure 3.4: Included UML elements

Therefore, class diagrams will support the metatypes Classes, Operations, Properties, Enumerations, Enumeration Literals, Associations (including Aggregations and Compositions) and Generalisations. These elements cover most basic use cases for class diagrams, and are judged to be sufficient for most uses.

This most notably excludes interfaces and dependencies. The concept of interfaces are useful in later stages, when the design of the system is fleshed out, but they may be difficult to understand without knowledge of what they mean conceptually.

Use Case Diagram

Use case diagrams have a quite simple notation by default, which makes it unnecessary to simplify much further. Included elements are actors, components (classifiers representing subjects), use cases, and extend- and include relationships.

This excludes extension points. These are normally used to identify the point where a use case can be extended. This is most useful when the use case di-

agram is accompanied by a textual representation of the use cases, where the extension point can point to a specific location which can be extended. In a purely graphical representation, the extension points are often not needed, and so have been removed for simplicity's sake. They may be added in the future, if evaluations show that they are needed.

Statechart Diagram

Statechart diagrams are simplified by removing the concept of *regions*. Removing regions provide a simpler view of states, although it removes the ability to decompose states into orthogonal regions. Since the emphasis is on creating small and simple model fragments, however, it is estimated that regions are not required, and only adds complexity.

A state is only classified as a composite or simple state based on whether it contains other states or not. There is made no distinction between composite and submachine states, since these are almost semantically and notationally equivalent. This means that simple states can be considered composite states with no substates. This provides a simple notation, where one can easily drag a state into any other state, thereby creating a composite state. Besides this, states are labelled with a name, as well as entry/exit/do activities, which can be used to specify behaviours which should be executed when entering, exiting or being in a state, respectively.

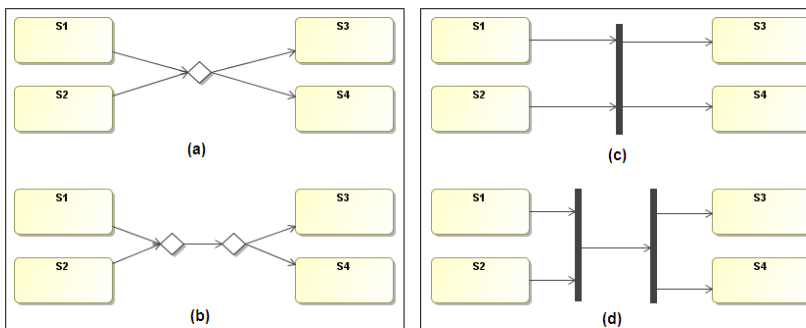


Figure 3.5: Decision/Merge and Fork/Join nodes merged. (a) is short-hand notation for (b), and (c) is shorthand for (d)

Besides states, statechart diagrams contain state transitions, optionally with a guard, as well as various pseudo-states: initial, final, fork, join, decision, merge, deep and shallow histories as well as send/receive signal actions. Of these, the fork and join nodes, and the merge and decision nodes will be merged

into single fork/join and merge/decision nodes. This is described in the UML superstructure as a valid notation in the following quote: *,,the functionality of join node and fork node can be combined using the same node symbol. [...] This case maps to a model containing a join node with all the incoming edges [...] and one outgoing edge to a fork node that has all the outgoing edges"*. The interpretation of this is shown in Figure 3.5.

Activity Diagram

Activity diagrams have a relatively simple notation, consisting of actions nodes, control flows and control nodes (initial, final, flow final, fork, join, decision and merge). Like with statechart diagrams, fork and join nodes, as well as decision and merge nodes are merged. Action and object nodes are labeled with a name, and control flow edges have a guard.

General

In addition to the above elements, the Package and Port metatypes are added. These are general model elements which are shared between the various diagram types. Packages are used to contain various elements, while the port can be attached to other nodes. Ports can then be the source or target of connections.

3.2.4 Sketchiness

The described UML fragments are to be created while the requirements for a software development project are initially specified, which is a rather early phase.

At this point, there should be no thorougly thought through design, no big architectural considerations, no meta-model or data-model set in stone. The focus is on *what* needs to be done, not yet *how* it should be done.

Allowing the user to embed UML fragments in the requirements specification runs the risk of sending a message to the reader that the design is more mature than what is actually the case. UML diagrams are usually created while the design of an application is fleshed out, starting with sketches on a whiteboard and ending with a final set of documents describing the design.

In user-interface design, this issue is also relevant, since visual elements need to be created in an early phase of the project, and might not resemble the final product.

> *„Some of the most serious problems occur if various parties – managers and/or customers and/or marketing – begin to view the early prototypes [read sketches] they see as the final product."*

(Hix and Hartson, 1993)

The solution which designers use is to create hand-drawn sketches of varying fidelity to distinguish between different levels of completeness. The argument for this is the psychological effect of looking at a hand-drawn sketch versus looking at a high-fidelity illustration. A sketch conveys a message of not being set in stone, and still open for discussion and changes, which is an important issue with the UML fragments.

> *„... a sketch is incomplete, somewhat vague, a low-fidelity representation. The degree of fidelity needs to match its purpose, a sketch should have "just enough" fidelity for the current stage in argument building.... Too little fidelity and the argument is unclear. Too much fidelity and the argument appears to be over-done; decided; completely worked out."*

(Hugh Dubberly of Dubberly Design Office; private communication. Qtd. in Buxton (2007), p. 113)

This thesis proposes to transfer this approach of using sketches to signify to the reader that a thorough design is not yet done, to the subject of UML fragments in the requirements specification.

3.3 Weaving UML fragments

This section describes the considerations for how the UML fragments, which are created using the previously described UML editor, and attached to individual requirements, can be woven together into a draft analysis model.

The intention is that this draft analysis model can serve as a starting point for creating the actual analysis model, by importing it into a fully-fledged modelling

tool. In the long run, it should be possible to keep the trace between elements in the analysis model and the individual requirements from the requirements specification, but how this integration with other tools will happen will only be proposed, as it is not part of this thesis.

As described in Section 2.2, model weaving is a special case of model transformations, in which a set of input models are woven together according to a weaving specification, in order to produce a model output.

In this thesis, model weaving is used to weave together a set of model fragments, producing a more complete model, which can be used for further analysis. The weave specification, which is defined in more detail below, is responsible for weaving together the model fragments in a way that makes sense in relation to UML.

3.3.1 Weave specification

The UML package merge, defined in OMG (2011), specifies the semantics of the package merge by defining a set of constraints and transformations. The constraints specify the preconditions, which must be in place before the merge operation, while the transformations specify the semantic effects of the merge.

The package merge only defines constraints and transformations for a subset of the metatypes in UML, most notably Packages, Classes, Associations, Properties, Operations and Enumerations, since *,,the semantics of merging other kinds of metatypes (e.g., state machines, interactions) are complex and domain specific".*

Given that the applications of model weaving in this tool calls for support for more than just these basic metatypes, some basic merge rules are given for the other meta types supported by the UML modelling components of the tool, described previously. These rules constitute a proposal for a simple package merge of these additional meta types, though as noted in the UML Infrastructure, it is difficult to give a perfect specification due to the complex nature of these meta types.

Some of the difficulties of specifying how the additional metatypes should be woven together are covered below, before detailing in Section 3.3.2 how these issues can be solved by allowing the user to manually specify weave details.

The basic UML package merge semantics are given in OMG (2011), while the semantics specific to this application are given below. Since the rules governing

Classes, Enumerations, Properties, Operations and Associations are covered by the basic rules, the Class Diagram will not be covered below. Instead, the basic rules will be used.

Use Case Diagrams

As described in Section 3.2.2, only Actors, Use cases, Classifiers (representing the subject), Extension and Inclusion relationships are supported initially, while extension points are excluded.

Actors, Use Cases and Classifiers all have names, so we can match instances of these metatypes simply by their type and name, the same way as, e.g., classes.

The Include and Extend relationships are matched by their type, source and target. When deciding whether to match two Include or Extend relationships, their source elements and target elements are compared. If they match, then the relationships match and should be merged.

Activity Diagrams

As noted in Section 3.2.2, the following elements are included in the Activity Diagram: Action nodes (including send and receive signal nodes), control nodes (fork, join, decision, merge, final, flow final, initial) and Control Flow relationships.

Action nodes are matched by their type and name. In the tool, it is possible to specify the name of an action, so it is possible to match action nodes in the same manner as Actors and Use Cases.

Control nodes are more difficult to merge. They serve the role of coordinating flows in an activity, and are not identifiable by name, but rather by their role in managing the control flow.

Initial nodes are the starting points for activities. When an activity is started, control tokens are placed at every initial node, which means that multiple flows are created initially, if an activity contains more than one initial node. When merging two activity diagrams contained in packages, each containing a number of initial nodes, it should be considered whether the initial nodes from the merged package should simply be copied to the receiving package, thereby creating new flows, or whether some of the initial nodes can be merged.

To keep matters simple, and not assume too much about the activities at hand, two initial nodes nodes, I_1 and I_2, should only be matched as follows:

1. Take the set of outgoing edges, $Outgoing(I_1)$ and $Outgoing(I_2)$. Let two transitions be considered equal if they match according to the matching rules for edges. If $Outgoing(I_1) \in Outgoing(I_2)$ or $Outgoing(I_2) \in Outgoing(I_1)$, then the initial nodes are matched.
2. If (1) is not satisfied, the initial node from the merged package is copied to the receiving package.

These matching rules are exemplified in Figure 3.6. Here, we have a number of cases:

1. If P is merged with Q, the initial node from P is simply copied to Q since the match condition is not fulfilled. There will then be two initial nodes, each having a transition to A, one of them a transition to B and the other a transition to C.
2. If P is merged with R, the resulting package will be similar to R, since the initial nodes are merged.
3. If P is merged with S, the resulting package will be similar to P, since the initial nodes are merged.

Details for how the edges are matched are described below.

Final and flow final nodes are similar to each other, and different from initial nodes, in that additional final or flow final nodes in a diagram is of no semantic importance. Figure 3.7 shows two different diagrams, which are syntactically identical – we see that we can merge together two flow final nodes without affecting the flow of the activity. Because of this, matching rules for final and flow final nodes can be stated as simply as:

- Flow nodes and flow final nodes can be matched by metatype, but is not required to be so.

This allows the actual implementation to merge these types of nodes in a way which produce the best diagrams, since adding final or flow final nodes typically improves the layout of larger activity diagrams.

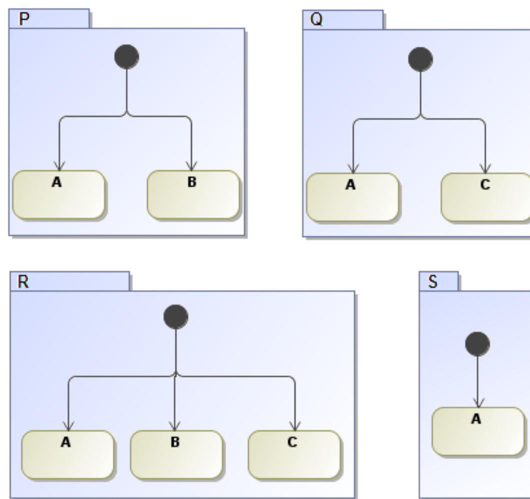


Figure 3.6: When comparing two initial nodes for a possible match, if the outgoing edges from one of them is a subset of the outgoing edges from the other, the nodes match.

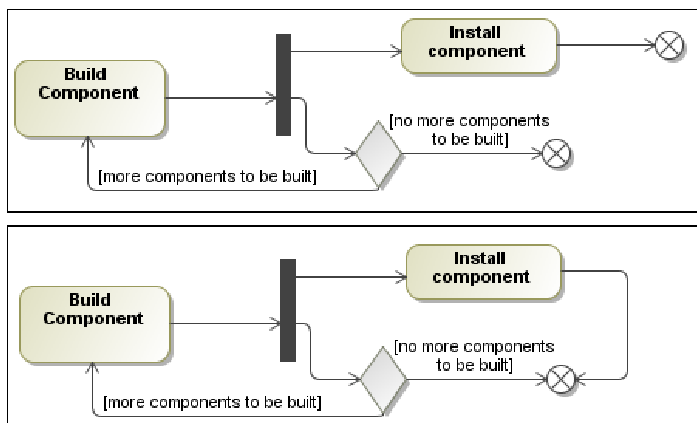


Figure 3.7: Two syntactically identical activity diagrams

Fork and join nodes are treated as the same node, combining the characteristics of the two nodes. A diagram containing a fork/join node will therefore have a set of edges incoming to the fork/join node, the node itself, and a set of outgoing edges. When comparing two such subdiagrams S_1 and S_2, consisting of a fork/join node and attached edges, for a possible match, the following six situations exists (for a given definition of equality between edges, which is given below):

1. The set of incoming edges in S_1 and S_2 are equal
2. The set of outgoing edges in S_1 and S_2 are equal
3. The sets of incoming edges in S_1 and S_2 are disjoint
4. The sets of outgoing edges in S_1 and S_2 are disjoint
5. The sets of incoming edges in S_1 and S_2 have a nonempty intersection, but not equal
6. The sets of outgoing edges in S_1 and S_2 have a nonempty intersection, but not equal

Simple rules for weaving together fork/join nodes can be set up when certain combinations of the above situations occur. Consider the weaving of two fork/join nodes, where

1. (1) and (2) hold. In this case, the incoming and outgoing edges are the same in the two fork/join nodes. These nodes can be matched.
2. (3) and (4) hold. In this case, none of the incoming edges to one fork/join node occurs in the set of incoming edges of the other fork/join node, and likewise with outgoing edges. In this case, the nodes should not be matched.
3. Any other combination of situations hold. In this case, further analysis is required, as noted below.

Fork/join nodes are quite complex to weave, since it is difficult, or even impossible, to interpret the intended meaning of the weave of two diagrams. An example of this is shown in Figure 3.8 – the weave of packages P and Q can either be what is shown in package R or S, depending on the interpretation of the meaning of the fork. One interpretation is that either B and C should run concurrently, or B and D should run concurrently, as shown in R, while the other interpretation is the one shown in S, which is that B, C and D should all

be run concurrently. The intended interpretation can not easily be deduced, so the best approach is to stick to one approach, and make it possible for the user to switch to the other approach.

As described below in Section 3.3.2, an easy way for the user to specify weave details is to draw connections between various elements, including packages. A sensible solution is then to not match fork nodes per default, only matching them when the user explicitly states that they should be matched.

Decision and merge nodes are, like fork and join nodes, combined and treated as a single decision/merge node, which contains the combined properties of the decision and merge nodes.

Decision/merge nodes do not present the same problems as fork/join nodes, due to the fact that no additional tokens are created or destroyed in such a node. Looking at Figure 3.9, which is the same diagrams as in Figure 3.8, but with decision nodes instead of fork nodes, it is apparent that the two possible results, depending on whether the decision nodes in P and Q are woven together or not, are nearly semantically equivalent, which for the purposes of these model fragments may be adequate.

In order to keep a consistent strategy, the same approach as described above with fork/join nodes is taken. This means that decision/merge nodes are only matched, if the sets of incoming and outgoing edges in the two nodes are equal.

Control flows raises some issues which makes it difficult to specify an automatic weaving approach, which does not require user interaction.

An easy approach, which can cover a lot of common cases, is to specify that two edges match and can be woven together, if their respective source and target elements also match. This covers the basic cases of having two actions, a_1 and a_2, with an edge between them, modelled the same way in two different model fragments to be correctly woven together. This approach breaks down, however, when the source or target elements can not be matched, because their own matching rules depend on the matching of the edge itself. This leads to a circular matching attempt, in which two edges check whether their respective source and target elements match, who in turn check whether their outgoing and incoming edges match, including the edge which is trying to be matched in the first place.

When weaving two flow edges, f_1 and f_2, a match is calculated as

$$\mathit{match}(\mathit{source}(f_1),\ \mathit{source}(f_2)) \wedge$$

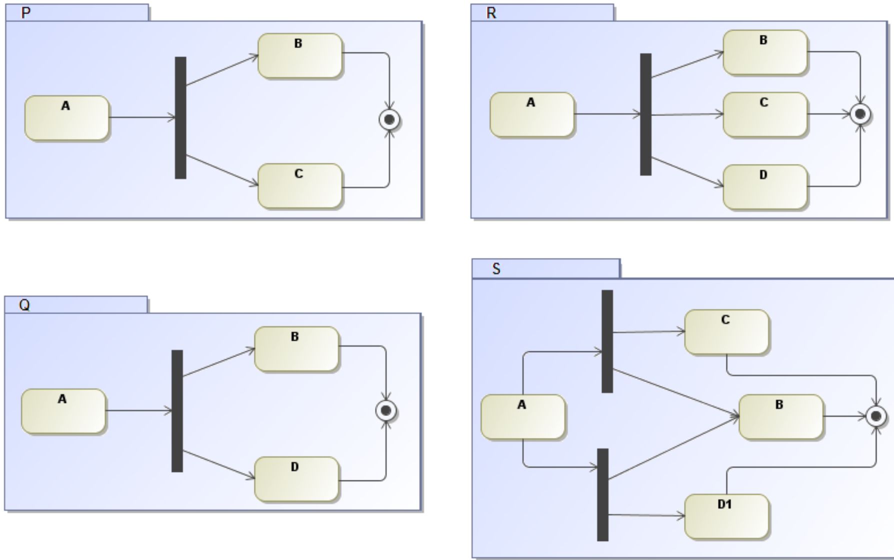


Figure 3.8: Should the result of weaving P and Q be R or S?

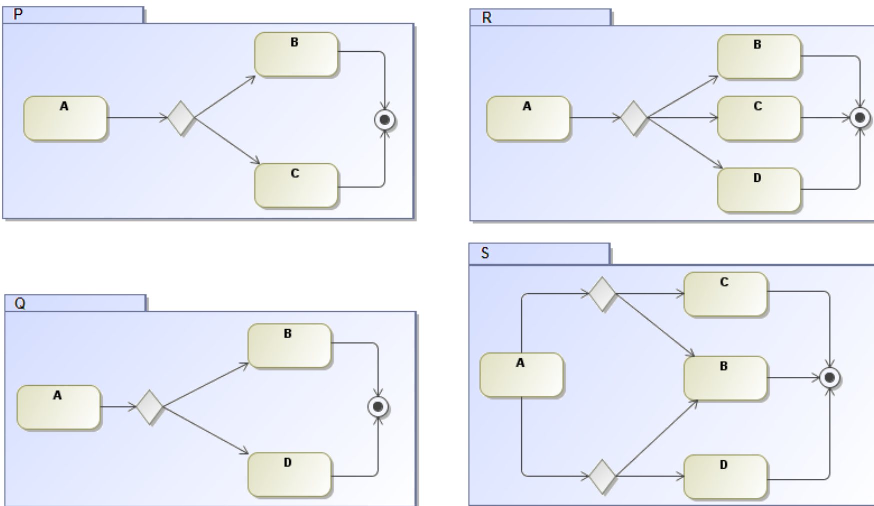


Figure 3.9: Merging or not merging two decision nodes connected by the same outgoing transition produces almost semantically equivalent results

$$\textit{match}(\textit{target}(f_1),\ \textit{target}(f_2)) \wedge$$
$$\textit{match}(\textit{label}(f_1),\ \textit{label}(f_2))$$

The label includes the guard condition, as well as the name of the edge.

State Machine Diagrams

The weaving rules for state machines are closely related to the activity diagram rules. States are matched by their type, name. If two states match, their entry-, exit and do-behaviours should be merged, e.g. concatenated.

Pseudo-states, such as initial, final, decision/merge and fork/join states are matched in the same manner as the control nodes in activity diagrams, described above. The same is true for state transitions, which are matched in the same manner as flow edges.

3.3.2 Manual weave specification

As described above, it is difficult, if not impossible, to define a complete, fully automatic weaving process for all UML metatypes. A solution to this problem is to involve the user in manually specifying weave details.

As noted above, many UML elements, such as pseudo-nodes, control nodes, state transitions and control flows are difficult to weave, since the intended meaning of the weave result is difficult to deduce from the model fragments which are input to the weaving process.

The previous sections have listed a few conditions which can be used to guide weaving in cases when there is little doubt about the outcome. An example of such a case is when two pseudonodes, such as a decision node, in two input model fragments has the exact same incoming and outgoing edges, or when two edges have the same source and target.

When none of these conditions are fulfilled for two given elements, the default behaviour should be to not weave the elements together, and instead depend on the user to specify how exactly the weaving should be done. If nothing is specified by the user, the default behaviour should be used.

How the user specifies the weave detail is covered in Section 4.4.1.

3.4 Improvements

In addition to the major features described above, which are the main goals of this thesis, some work has been done on designing and implementing a number of smaller features, which aim at providing additional functionality and improving usability of the tool. Some of these features are a result of the evaluation done by Friis (2012), and some are requests by the project's stakeholders, mainly the course lecturer.

Table 3.1 lists the implemented improvements, along with their justification.

Improvement	Justification	Section
Locking	Collaboration, reviewing, integration	3.4.1
Visual folder editor	Usability	3.4.2
Comment export	Integration, reviewing	3.4.3
Navigation improvements	Usability	3.4.4

Table 3.1: Required improvements

3.4.1 Locking

A feature requested by the stakeholders of the tool is the ability to *lock* certain elements, in order to disable editing. The purpose of this is to facilitate the reviewing process, where the artifacts created in the tool are passed on to others for commenting and evaluation. This is especially useful in the course setting, where students have to share their work internally in a group for comments and review, and in the end with the lecturer for evaluation.

Distributing the artifacts created in the tool can be done in two ways: as a generated report, or as a file exported from the tool, which can be re-imported. These two methods each have their purpose and benefits. The tool is not necessary for the reviewer to review a generated report, and it is possible to abstract from anything not directly related to the content, such as learning how the tool works. On the other hand, generated reports lose many of the benefits of the tool, such as navigation, linking between elements and the ability to comment directly on individual elements.

When distributing an exported file from the tool for review, the review process will be close to the one illustrated in Figure 3.10:

1. Author exports the items to be reviewed

2. Copy of the exported file is distributed to reviewers
3. Reviewers import the file into their own copy of the tool
4. The contents are reviewed, and comments are made inside the tool, attached to specific elements
5. Comments are possibly exported to be used in some other tools for inspection
6. The commented elements are exported again, and sent back to the author
7. The author imports the commented file next to the original copy, and incorporates any changes resulting from the comments.

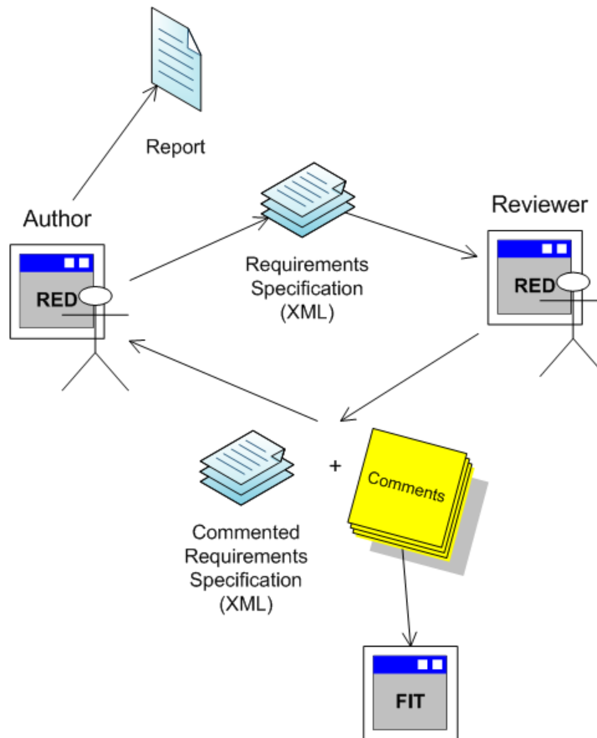


Figure 3.10: Typical review process

An important point is that the role of the reviewer is not to make changes in the distributed document, since this means that the author has to merge these changes into his existing work, possibly along with changes from other reviewers, which may conflict with each other. Without an advanced merge functionality

built into the tool, which can handle these cases, there needs to be some way of preventing reviewers from making changes to the actual tool artifacts, and limiting their editing powers to being able to comment on elements. This is done by having the author locking the elements before they are distributed to the reviewers, disabling the editing.

Details for how the locking solution is designed is described in Section 4.6.1.

3.4.2 Visual Folder Editor

During the course of developing a large and complex software system, the requirements specification may end up growing very large, containing thousands of elements structured in a complex hierarchy of folders. The element explorer view serves to give an overview and easy navigation of all the elements in a given project.

In addition to this existing view, a more graphical overview of the folder structure may be beneficial. While the element explorer displays every element in the project, a dedicated editor which graphically shows the folder structure provides another dimension, which may be especially useful when the author wants to change the project structure.

This graphical editor should allow the user to:

- Get an overview of the folders in a given project, and how they are structured
- Create new folders and insert them into the existing structure
- Delete existing folders
- Rearrange folders, by dragging folders out of their original parent folders and into new folders

A benefit of this graphical editor, aside from the new functionality, is that it provides some experience with the underlying technology used to create graphical diagram editors, which is used extensively to create the model fragment editor, as well as the weave editor.

3.4.3 Comment export

In order to integrate with the Formal Inspection Tool, as described in Section 3.1.3, a method for exporting inspection data from RED is required.

A software inspection typically follows 6 stages: planning, overview meeting, preparation, inspection meeting, rework and follow-up (Petrolyte, 2011). The reviewers perform their examination of the inspection subject during the preparation stage, and any defects are pointed out during the inspection meeting.

The review of requirements specifications developed using RED can be done in two ways: either as a review of a report generated by the tool, or by reviewing the various artifacts directly in the tool. The advantage of the second option is that the reviewer maintains all the benefits which RED provides, such as navigation, linking, and most importantly: commenting directly on elements.

The reviewer can note down any and all issues found as comments attached to elements. For the inspection meeting, it should be possible to extract all these comments, so they can be imported into another tool, such as a spreadsheet or FIT, for an easy overview of the issues found.

Since no finalised import format for FIT is defined at the time of writing, a common, standardised export format, such as CSV (comma-separated values), should be used. The exported data should contain the relevant information which might be needed by FIT. The information about a comment, which is useful in such a tool, is:

- The unique ID identifying the element to which the comment is attached
- The comment details: creation time, author, status and comment text
- The path, in RED, to the element containing the comment. This is useful for displaying the origin of the comment in FIT.

3.4.4 Navigation

A usability improvement, which was suggested during the initial evaluation of the tool, was improvements to navigation when working with a large number of elements. In this situation, a large number of open editors tend to pile up, which can make navigation difficult, since only a small number of open editors are shown in the editor pane of the tool as shown in Figure 3.11, which shows a situation with 17 open editors.

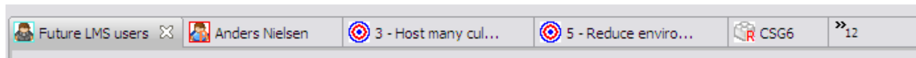


Figure 3.11: List of currently open editors in the tool

When working on multiple, related elements at the same time, one tends to jump back and forth between the various editors, as changes in one editor leads to changes in another. To avoid the user having to switch between editors by finding the desired editor in the list of open editors, shortcuts should be added which lets the user go back and forward to previously opened editors.

These shortcuts can be added as buttons on the toolbar as well as hot-keys, which lets the user navigate back and forwards through the history of previously opened editors, like the *back*- and *forward*-buttons in a browser. A notable distinction from a browser, however, is that while a modern browser lets you have multiple tabs, each with their own back/forward history, the tool should instead only have a single back/forward history common for all editors.

3.5 Domain model

In order to create the domain model for the model editing and weaving functionality, it is necessary first to look at the meta-model underlying the core functionality in the RED tool. As described below, RED has a well thought out, simple conceptual model, which is used as the basis for all added functionality. As a result, the functionality implemented in this thesis should further build upon this model, in order to integrate as smoothly as possible with the rest of the tool.

3.5.1 RED Core meta-model

The basics of the RED Core meta-model, as shown in Figure 3.12, is based on the concept that (almost) everything in the tool is an *Element*. This provides a very unified approach to extending the tool with new functionality, since simply extending the `Element` interface enables a lot of standard functionality, and allows new functionality to fit into existing features, such as navigation, search, referencing etc., with little effort.

Elements which can contain other elements are called *Groups*. Folders, projects and glossaries are examples of groups in the existing tool.

Elements can also have relationships to other elements. These relationships are modelled as elements themselves, which allows elements to have a relation to a specific relationship.

3.5.2 Model elements domain model

The element-based approach taken by the RED Core meta-model guides how the addition of model elements should extend the domain model.

The various UML metatypes can be modelled as *Elements*, inheriting the basic functionality defined in the tool's core. Some metatypes, such as classifiers which can contain other elements, can naturally inherit from *Group*, while relationships between elements can inherit from *ElementRelationship*. This gives a basic domain model as shown in Figure 3.13.

3.5.3 Weave data model

As described in Section 4.1, the use of GMF requires an EMF data model for creating a diagram editor. This section looks at designing this data model, in order to make the creation of the weave diagram editor as simple as possible.

The purpose of the editor is to display a set of model fragments, originating from other editors, side by side, and allow the user to specify various properties in a graphical way, which impacts the behaviour of the following weaving process.

The model fragments shown in the graphical editor presents a technical challenge, arising from GMF's requirement that there should be a containment feature for all nodes added to the graphical mapping model. A feature, in this context, is an attribute or reference defined in the EMF model.

When defining a node mapping, it must be specified where the actual elements, mapped to by the graphical node, come from, relative to the canvas' element. This can be done by specifying a *Children Feature* and a *Containment Feature* of the root element. The *containment feature* is mandatory, as this is used as the feature which should contain any new elements created in the editor, while the *children feature* is optional. The two features are the source of all semantic elements displayed in the diagram.

The requirement that there should be a containment reference raises an issue, since EMF enforces that an object can only have one container. The root ele-

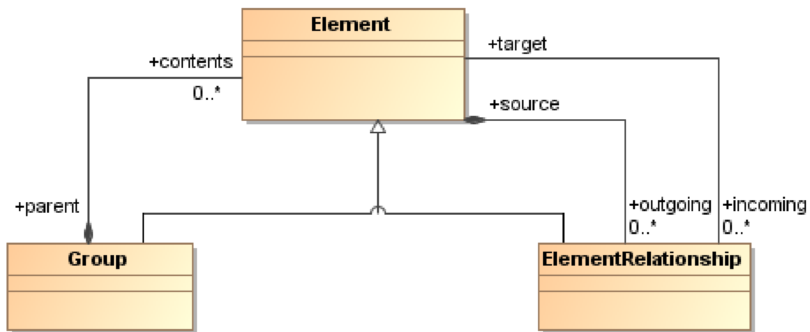


Figure 3.12: RED Core meta-model

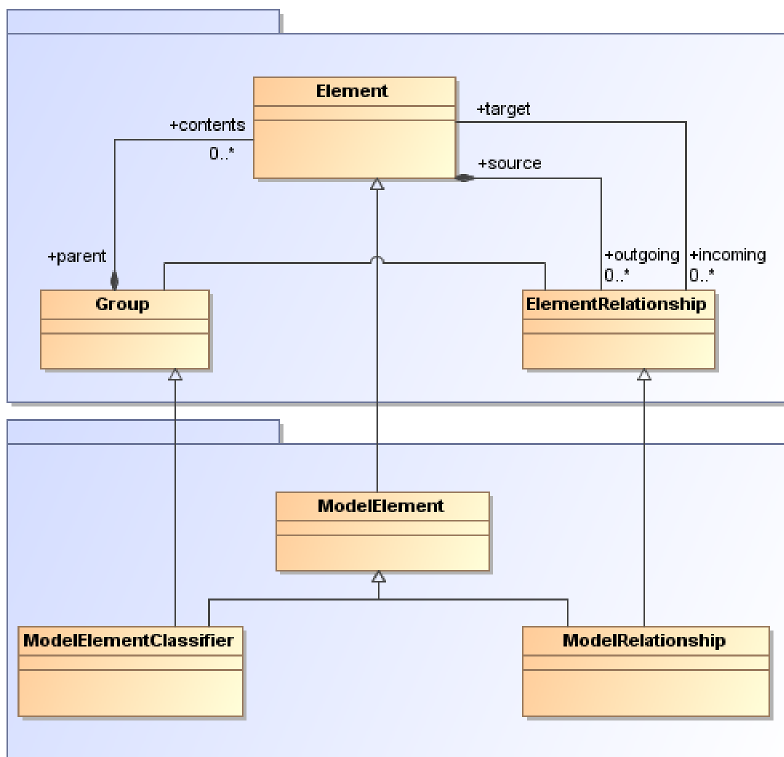


Figure 3.13: Model Element Domain Model

ment for the weave editor should be some *weave model*, which should be used to access the model fragments to be displayed in the editor. However, the fragments should retain their original containers, and not be moved to this *weave model*. In order to alleviate this problem, the *weave model* should reference model fragments with a non-containment feature, while also having containment features which can satisfy GMF's requirement of a mandatory containment feature for new elements.

This leads to the following requirements of the data model for the weave editor:

1. There should be an element which can function as the *root* element for the diagram editor canvas
2. This element should have a non-containment association which can contain any number of model elements
3. It should also have containment associations for containing any new elements

Such a model is displayed in Figure 3.14.

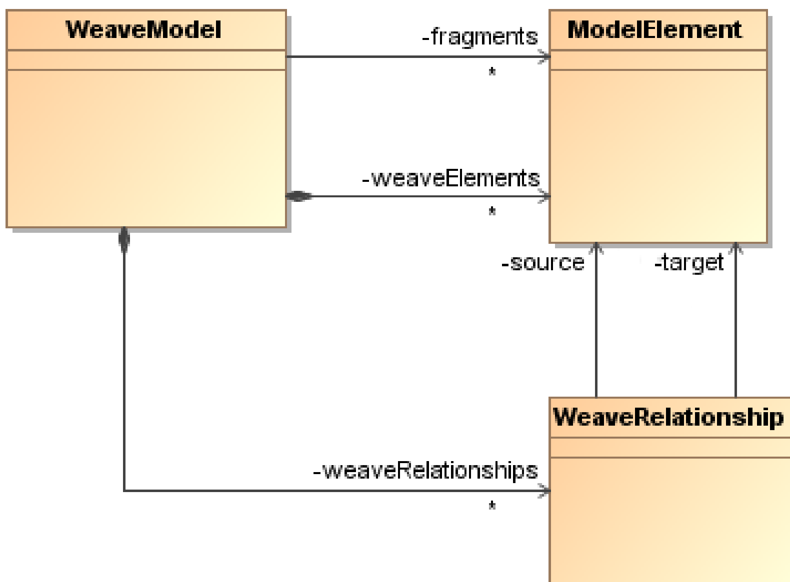


Figure 3.14: Weave Editor Data Model

Chapter 4

Design

In this chapter, the design of the solutions to the problems described in the previous chapter are covered.

4.1 Technology

This section gives a brief introduction to the GMF technology used to create the editors which are central to the modelling and weaving solutions implemented in this thesis. This provides some background knowledge useful for understanding some design decisions, and especially implementation specifics described in Chapter 5.

4.1.1 GMF – Graphical Modeling Framework

The Eclipse framework uses the SWT (Standard Widget Toolkit) for the various widgets and controls, and the Draw2d toolkit on top of this to layout and display graphical components in an SWT environment.

On top of this graphical drawing framework, Eclipse provides the Graphical Editing Framework (GEF), which is a framework for creating rich graphical editors. GEF uses a model-view-control based approach, in which the view is handled by Draw2d and the control-part consists of so-called *EditParts* and *EditPolicies*, which can be installed and uninstalled dynamically per the *Strategy* design pattern. The model can be generated by EMF, or implemented in any other way. As such, GEF is characterised as *model agnostic*.

GMF provides generation and runtime capabilities for graphical editors for Eclipse. It generates GEF editors, and basically automates the generation of a lot of the generic code, which would normally have to be written manually. GMF requires, as opposed to GEF, that the underlying domain model is EMF-based, as the meta-model is used extensively for mapping graphical components to domain objects.

4.1.1.1 GMF models

The creation of GMF diagram editors is model-driven. The GMF Tooling framework specifies a number of meta models, with corresponding editors for the creation of actual models which are used to as input to the code generator, responsible for creating the diagram editor.

The types of models which must be created and maintained in order to create a GMF editor are shown in Figure 4.1, and are as follows:

- Domain Model (Ecore) – This model is used as the semantic model being manipulated by the generated diagram editor. It should be created independently of GMF, and should model the concepts in the domain.
- Graphical Definition Model (gmfgraph) – The definition of all graphical elements which should appear in the generated editor are contained in this model. The graphical definition model is used to define a set of nodes and connections, and the compartments and labels which may be contained therein. The model is independent of the actual usage of the defined figures. As such, it is unaware of the semantic elements which will be represented by the figures defined in the model.
- Tooling Definition Model (gmftool) – This model is used to define the tools which should appear in the tool palette of the generated editor. The model is used to define which tools should appear, as well as their names, descriptions and icons, but the model does not define the behaviour of the tools.

- Mapping Definition Model (gmfmap) – Tying together the three previous models, the mapping definition model is used to map semantic elements from the domain model to graphical figures from the graphical definition model. This model is also responsible for defining how elements are contained in other elements, constraints on compartments and connection source and targets, initial values of various element properties, and what the behaviour of the creation tools defined in the tooling definition model should be.

- Generator Model (gmfgen) – The final model, created by the tooling framework based on the previous models. This model contains a large number of final details on how the diagram editor will be generated, such as names of generated classes.

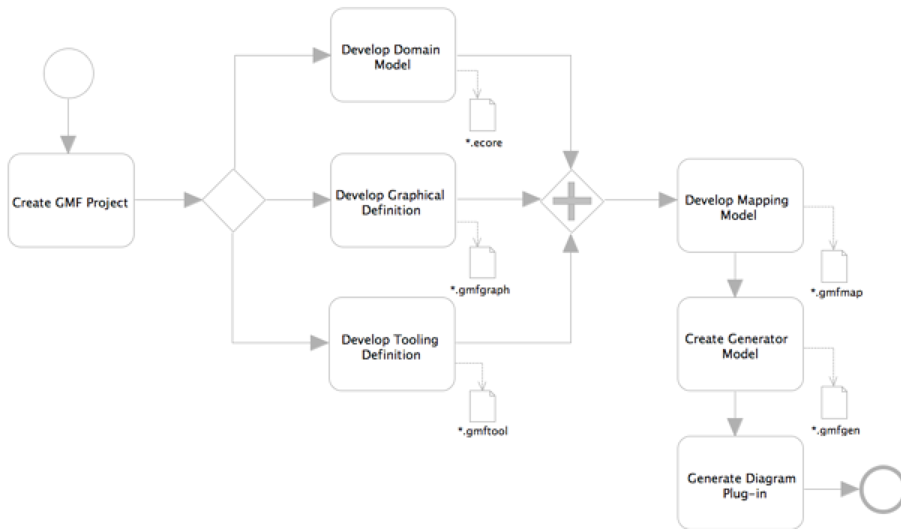


Figure 4.1: GMF Overview (Image courtesy of Eclipse.org)

4.1.1.2 Consequences of using GMF

This section looks at some of the consequences of using GMF, and the impact this has on design decisions.

- GMF contains support for compartments, which are graphical nodes which can contain other nodes. In order to implement these, GMF requires that

the underlying EMF model must have a containment reference relationship to the elements which are contained in the compartment. This means that care must be taken when designing the domain model, to make sure that it is compatible with GMF.

- GMF places, per default, no semantic significance on the layout and placement of diagram nodes. This makes it difficult to implement diagram types, where the position of elements carry semantic information, such as sequence diagrams. An analysis must therefore be made of the difficulty of implementing various features in GMF, and the various diagram types must be prioritised.

- The model driven development approach mandates that changes should be made in the model rather than in the generated code. This slows down development a bit, since editing a model, re-generating and then rebuilding the code takes a lot longer than simply making the change in code and rebuilding. This is especially noticable when testing small changes which might require several iterations of change and test in rapid succession.

4.2 Model Fragment Editor

The purpose of the model fragment editor is to function as an integrated UML editor, allowing the user to easily create small, simple UML models when in the process of describing requirements.

Each individual model fragment should be related to one specific requirement, in order to clarify the purpose and meaning of the requirement, and to serve as direct input to a draft analysis model, in order to gain the option to trace directly from requirement to model to code. This encourages a tight integration between model fragments and requirements.

The RED tool is editor-based, which means that individual elements in the tool, such as requirements, goals, personas etc., are edited using a specialised editor. These editors can contain multiple pages or *tabs*, separating the functionality of a single editor.

Given this, the graphical model editor should be integrated into the requirements editor as a new page, which will allow the user to easily create a model fragment while documenting a requirement. It will also create a strong connection between the requirement and the model fragment, clarifying the purpose of each model fragment.

One disadvantage of this integrated approach, is that it is not immediatly possible to create a model fragment which describes a set of requirements covering a common area of the specified system, as each model can only be attached to a single requirement. This could be useful if a single small model fragment adequately covers a set of requirements, in which case the user would want to indicate this relationship. The solution to this could be to enable the ability to reference a model fragment in one requirement, which is defined in another requirement, effectively sharing the model between requirements.

4.2.1 Editor functionality

The editor should allow the user to easily create a new UML model fragment, or edit an existing one. To this end, it should be easy to add new elements, rearrange or delete existing elements, and edit the properties, such as labels, of elements.

The adding of elements can be done through a toolbar, which contains all the type of elements and connections which is supported by the editor. This allows an easy overview of the options available, and mimicks the functionality of other drawing and modelling tools, where a toolbar or palette is the norm.

Elements can be rearranged and resized by dragging them around the editor canvas, and the target and source of a connection can be changed by dragging the start- or endpoint of the connection to another element. Elements can likewise be moved into the compartment of another element, by dragging the element on top of the containing element. For example, to put a State into a State Machine, the State can simply be dragged on top of the State Machine.

The various elements in the editor may contain labels, such as names and annotations. These labels should be editable by simply clicking them and writing the new text.

Any attributes, which may not be easily represented graphically, or where the ability to change said attribute requires some more advanced functionality, such as a drop-down menu, may be viewed and edited through a *properties*-view. This view should list the various semantic attributes of an element (i.e not layout attributes), and allow the user to change them. If an attribute, such as the name of an element, is shown in the graphical editor, it is also shown in the properties view, and changing the value either place will immediatly be propagated to the other.

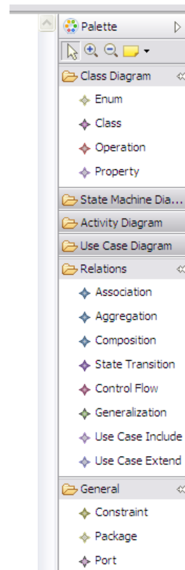


Figure 4.2: Illustration of model fragment editor toolbar

4.2.1.1 Toolbar

The toolbar allows the user to easily create new elements in the editor. The tools which are contained in the toolbar depends on the types of model elements supported by the editor, and should be grouped in an intuitive way, which helps the user find the right tool.

The tools are grouped mainly by UML diagram type: *Class*, *State*, *Activity* and *Use Case*. This makes it easy for the user to find related elements, and allows the user to hide the tools which are not used for the diagram type being worked on, as the groups are collapsible. Common elements, which are not specific to a single diagram type, are collected in a *General* group, and connections are grouped in a *Connections* group. This is done because some connections, such as associations, are shared between diagram types.

An illustration of the toolbar, with the collapsible tool groups, is shown in Figure 4.2. The icons next to the various tools in the toolbar are placeholders, and should be replaced with more appropriate icons.

4.3 Weave editor

The Weave editor serves two purposes. First, giving an overview of the weave about to be performed, by showing the fragments which have been selected by the user side by side. Second, allowing the user to specify details about the weave, such as which model elements represent the same conceptual element.

This should obviously be a graphical editor, in line with the model fragment editor, but it should have some different functionality. The following list points out the key differences between this editor, and the model fragment editor.

- It should not be possible to create new nodes and connections, except for the ones which are relevant for the weave editor. Creating new UML elements should not be allowed in this editor, since the model fragments shown belong conceptually to another editor, and editing them in the weave editor is inconsistent with this separation.
- Likewise, it should not be possible to alter the attributes of the semantic elements, such as the names and other attributes of nodes, start- and endpoints of connections, containment relationships between elements etc.
- It should be possible to specify weave details, as described in Section 4.4.1, by creating annotated weave connections
- The user should be able to initiate the weaving, and inspect the result

4.4 UML weaving

The actual weaving of the model fragments is a separate process from defining the fragments. The common use case is that the user will create a set of model fragments attached to requirements. This will result in a requirements specification, where a subset of the requirements have attached model fragments, which clarify the meaning and intention of the requirement.

One of the goals regarding these model fragments is to be able to take a set of fragments, with each one modelling a small aspect of the same system, and weave them together into a more comprehensive model, which can serve as a starting point for an analysis model.

The actual model weaving is based on work by Störrle, who uses a Prolog-based approach to representing and weaving models. The Prolog representation of

models is based on Störrle (2007), while the weaving is handled by a Prolog module created outside of this thesis.

This section describes the various aspects of weaving model fragments. The process of weaving is shown in Figure 4.3, which displays an activity diagram giving an overview of the actions taken by the user and the system in order to weave together a set of model fragments in the tool. These actions are described in further detail below.

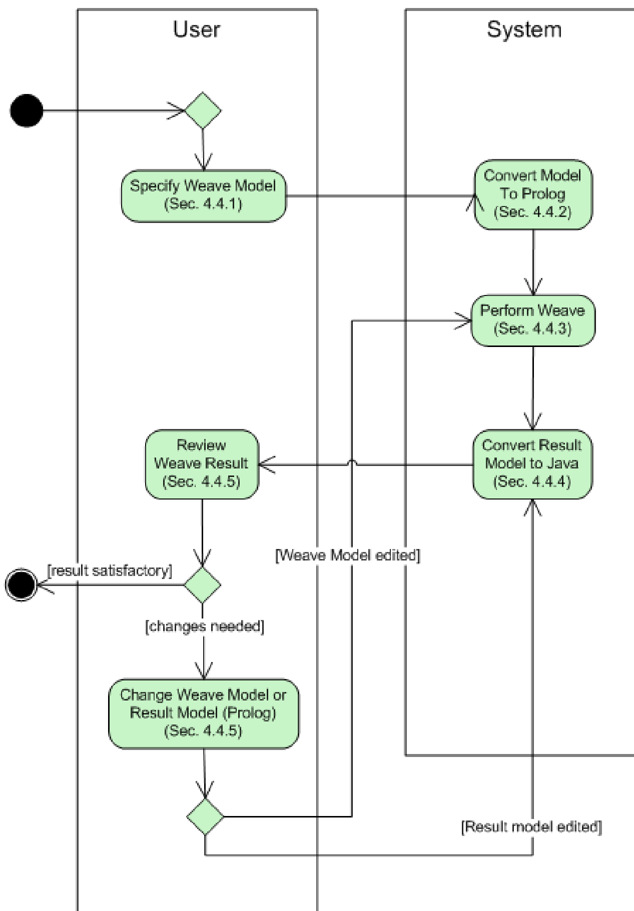


Figure 4.3: Weaving Process

4.4.1 Weave specification

The purpose of the weave process is to weave together a set of model fragments contained in a requirements specification project, into a coherent model. Considering that the number of requirements with attached model fragments may become quite large, it may be necessary, as well as more user friendly, to let the user specify a subset of the set of model fragments contained in the project, to weave together.

This is a sensible approach, when considering the size of the resulting model. One should strive to keep models small and simple, in order to maintain readability and maintainability. As a result, the set of model fragments to weave should all model slightly different aspects of the same subdomain, and not wildly different areas.

In order to do this, it must be possible for the user to specify which model fragments should be woven together in the beginning of a given weave-proces. This can be done through a dialog or wizard, where the user is able to select a set of model fragments in an easy way. Since model fragments are attached to requirements, and identified by these requirements, it is sensible to show the requirements as the selectable element, in the same structure as that portrayed by the element explorer used for navigation in the tool. An illustration of this is shown in Figure 4.4, which shows a dialog used for selecting a set of requirements. The dialog is designed to allow the user to easily select a group of requirements, by simply selecting the containing folder, or by selecting individual requirements through a check box.

After having selected a set of requirements, the model fragments attached to these should be extracted, and the model diagrams should be presented to the user in a single graphical editor, which contains each model fragment in a seperate UML package.

In this editor, the user should be able to finalise the weave specification, by adjusting details for how the weave should be performed. The default weaving behaviour is defined by the rules set forth in Section 3.3.1, which expands the definition of the UML Package Merge to include rules for how Use Case, Activity and State Chart Diagrams are woven together. This requires the user to specify Weave connections (akin to the UML Package Merge connections) between packages, to specify that two packages should be woven together.

However, the user may wish to adjust the details of which elements to weave. Figure 4.5 shows three cases, which are described below:

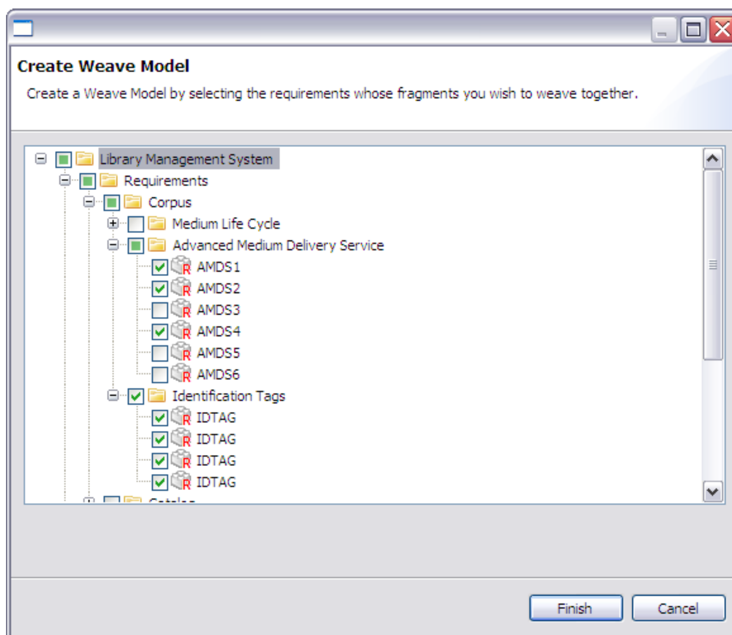


Figure 4.4: Dialog for selecting model fragments

Exception: When a weave connection is created between two packages *P* and *Q*, but the user wants to specify that two elements in these packages, *P::A* and *Q::A*, which would be woven together under normal circumstances, are not the same

Disambiguation: When the user only wishes to weave together a few elements in two packages, without wanting to specify a weave connection explicitly between the two packages, since there may be many other elements which should not be woven together, even though the rules states that they would be

Overriding: When two elements, *P::A* and *Q::A'* should be woven together, even though they do not match according to the weaving rules, but the user wants to specify that they are the same element

To handle these cases, two solutions are proposed:

1. To handle case 2, where the user would want to weave together two specific elements, without extending the weave to their enclosing packages, it should be possible to have an explicit weave connection between elements, in addition to packages

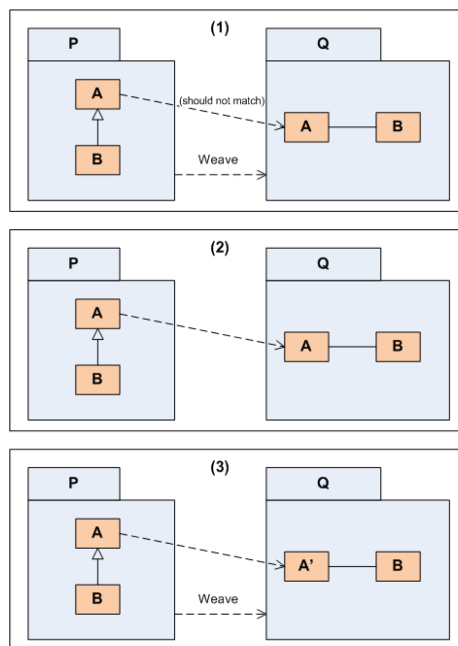


Figure 4.5: Weave specification details

2. To handle cases 1 and 3, where the user wants to include or exclude specific matches, there will be an annotation to the weave connection, where the user can specify these inclusions and exclusions. This annotation will consist of a list of statements, each including or excluding a match in the connected packages. An illustration of how this looks is shown in Figure 4.6.

The syntax for the inclusion and exclusion statements is shown in Listing 4.1.

```
1 + P1::A1 @ P2::A2 [=> A3] (inclusion)
2 - P1::A1 @ P2::A2 (exclusion)
```

Listing 4.1: Weave Annotation syntax

Inclusion is specified by starting the line with a +, while - indicates exclusion. Then follows two parameters, separated by the @-symbol, indicating the elements (A_1 and A_2) from the packages (P_1 and P_2) connected by the weave connection. For inclusion, these parameters state that the two specified elements should be matched (if possible; they should be the same type, though the name may differ). For exclusion, the parameters are used to indicate that the two named elements should not be matched, even though they normally would. Finally, the inclusion-statement allows an optional parameter, which specifies the name of the resulting element, when A_1 and A_2 have been woven together.

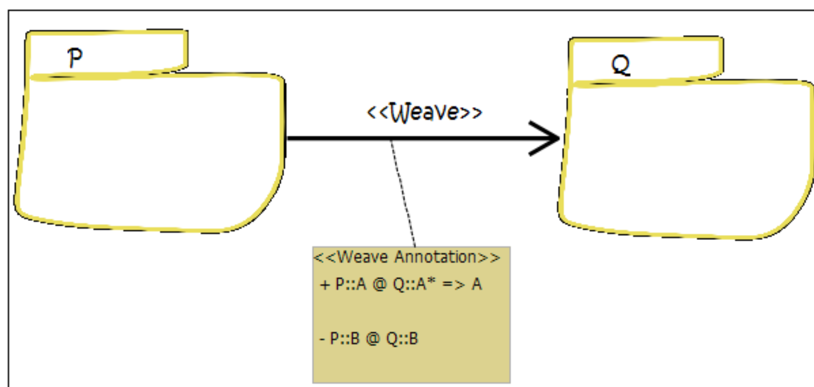


Figure 4.6: Weave connection with annotation

4.4.2 Conversion to Prolog

4.4.2.1 Model representation

The actual weaving of the model fragments is done by a Prolog module that has been created outside of this thesis (cf. Störrle (2007)). The Prolog representation of the model is based on how XMI is used to store UML models in an interchangable XML-format.

Layout is a very important part of making diagrams useful, by making them readable and understandable. We can therefore assume that the user puts some effort into the layout of the model fragments, and as a result expects that this layout is preserved as much as possible during the weaving process.

For simplicity, the layout of the model fragments should be reused in the resulting weave model diagram. This will make it easier for the modelers to validate the weave result.

In order to simply transfer the layout constraints from the model fragment diagrams to the weave result diagram, the layout information is added to the Prolog code.

The layout information can be added in two different places: as part of the model elements, or as a separate Prolog fact, as shown in Listing 4.2. Here, line 1 shows the solution where the layout information is a part of the model element fact, while lines 3 and 4 shows the solution where the layout is separate from the model element.

```
1 me(class-1, [name-'A', x-22, y-43, width-240, height-400]).
2
3 me(class-1, [name-'A']).
4 layout(id(1), [x-22, y-43, width-240, height-400]).
```

Listing 4.2: Location of layout constraints

The benefit of the first solution is that it is more concise, and keeps the information about a model element limited to as few locations as possible. As a result, it is slightly easier to extract the layout information, when converting the Prolog model back to Java. On the other hand, it complicates the model weaving, since the layout information will have to be processed, too.

A disadvantage of the first solution, however, is that the model and diagram information is kept in separate data structures by GMF. Therefore, converting model elements to Prolog facts, and adding layout information, will have to be

done in two separate passes. This complicates the first solution, so the second solution is easier to implement.

Due to the limited benefits of the first solution, and the added ease of implementation of the second solution, we decided for the latter.

4.4.2.2 Conversion Process

The conversion from a Java representation of the model to a Prolog program must happen by visiting each element in the model in some way, and converting it to an equivalent element expressed in Prolog.

The Visitor-pattern is chosen for the task of traversing the model, and doing the conversion. The classic definition states that the visitor pattern is a method for separating an algorithm from the object on which it operates (Gamma et al., 1995). This typically requires a *Visitor*-object, which can traverse an object structure, and perform some operations on it. This visitation, and the execution of the operations, is traditionally done by implementing an `accept()`-method on the objects to be visited, which serves as a callback method to the visitor. The `accept()`-method takes a visitor as input, calls back a `visit()`-method on the visitor, and then passes the visitor along to the children of the object, by calling their `accept()`-methods in turn.

The visitor is typically an interface, which contains a `visit()`-method for each type of object to be visited. Function overloading is thus used to implement separate behaviour depending on the type of the object visited.

To take full advantage of the Visitor pattern, it should be designed in a usage-independent way, so that new implementations of the visitor interface can be added at any time, adding new functionality without the need for changing the existing class structure. For now, the intended use of the pattern is to traverse the elements in a model fragment, and output a Prolog representation, but one could easily image other kind of visitors in the future, either for other kinds of output, or for some wholly different purpose.

This also eliminates the possibility of simply letting the objects know how to transform themselves, by adding a `toProlog()`-method to the `Element`-interface. This may be a simple solution, but it is not easily extendable, and it scatters the conversion logic between a large set of classes and mingles it with other functionality, going against the principle of separation of concerns.

To achieve this, the pattern is implemented at the `Element`-level in the

RED metamodel, instead of the `ModelElement`-level, as seen in Figure 4.7. As seen in the class diagram, an `acceptVisitor()`-method is added to the `Element`, which takes a generic `Visitor` as input. To implement the Java-to-Prolog model converter, a more specific `WeavePrologOutputter` is defined, extending the `Visitor` interface. This specific visitor contains only the specific `visit()`-methods required for visiting the `ModelElement`-classes. Since the `acceptVisitor()`-method only knows about the more general `Visitor`-interface, and thus is only able to call the `visit(Object)`-method, and not the more specific `visit()`-methods. The `visit(Object)`-method is therefore responsible for delegating the call from the `Element` to the relevant `visit()`-method.

This delegation can be done by a small, generic solution using reflection, or by a bit more cumbersome solution with a series of if-then-else statements, checking the type of the element calling the `visit()`-method. The if-then-else solution defeats some of the advantage gained from the visitor pattern, by reintroducing the need to do an explicit type check of the visited object, while on the other hand, the reflective solution is likely several times slower, due to the inherent performance issues with doing reflective method lookups.

For a comparison of the performance of the two solutions, see Appendix A, which concludes that in this concrete case, the method using reflection is up to 5 times slower, although the overall execution time is very low, even for larger models. Given that the test results show that the conversion time of a model containing 400 elements is around 2 ms when not using reflection, and around 9 ms when using reflection, the method using reflection is chosen, as it provides a simpler implementation, which requires less maintenance, and is easier to expand.

4.4.3 Weaving process

The actual weaving process consists of a number of model transformations based on the input fragments, the set of default weaving rules as well as any weave annotations, which further specify how the weaving should be done.

When a set of model fragments, such as the ones shown in Figure 4.8 (a), have been converted to Prolog as described above, the result is a model as shown in Listing 4.3

This model consists of two model fragments, attached to requirements *RequirementA* and *RequirementB*. The RequirementA-fragment contains two classes, *A* and *B*, with an association between them. The RequirementB-fragment contains three classes, *A*, *C* and *H*, where C is an extension of A.

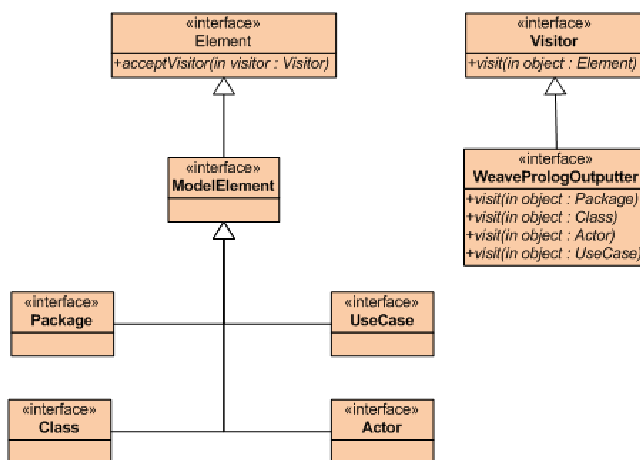
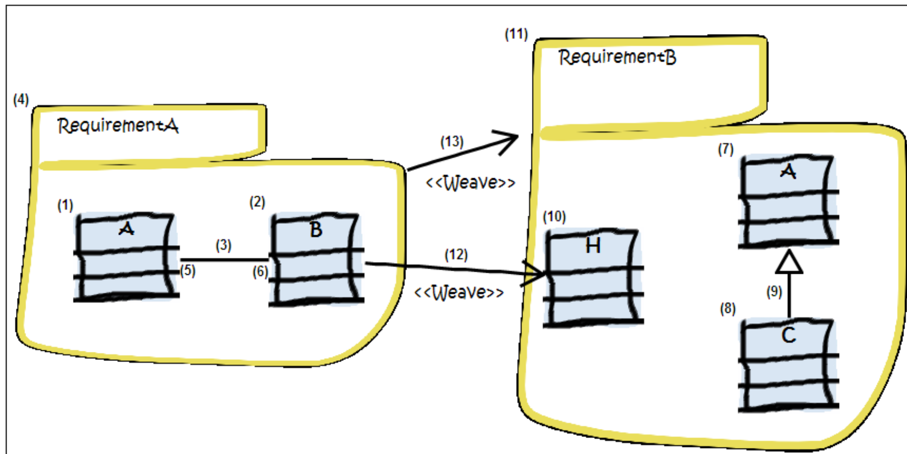


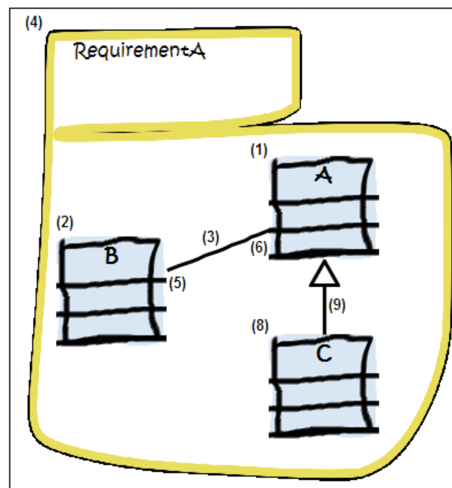
Figure 4.7: Visitor Pattern Use

```
1 :-module('pre-merge').
2 me(package-14, [name-'pre-merge', ownedMember-ids([4, 8, 12, 13])]).
3
4 me(package-4, [name-'RequirementA', ownedMember-ids([1, 2, 3])]).
5 me(class-1, [name-'A', ownedMember-ids([5])]).
6 me(class-2, [name-'B', ownedMember-ids([6])]).
7 me(association-3, [memberEnd-ids([5,6])]).
8 me(property-5, [type-id(2), association-id(3)]).
9 me(property-6, [type-id(1), association-id(3)]).
10
11 me(package-11, [name-'RequirementB', ownedMember-ids([7, 8, 10])]).
12 me(class-7, [name-'A']).
13 me(class-8, [name-'C',ownedMember-ids([9])]).
14 me(generalization-9, [from-id(8), to-id(7)]).
15 me(class-10, [name-'H']).
16 me(weaveAnnotation-12, [from-id(2), to-id(10), body-'default']).
17
18 me(weaveAnnotation-13, [from-id(4), to-id(11), body-'default']).
```

Listing 4.3: Example of weave model, as shown in Figure 4.8a



(a) Two model fragments in a weave model. The annotated identifiers match Listing 4.3



(b) The result of weaving the model in (a)

Figure 4.8: (a) shows two model fragments in a weave model. (b) shows the weave result model after weaving.

These are the fragments, as they have been modelled in the fragment editor. In addition to this, there are two weave annotations, which have been added in the weave editor, as described in Section 4.4.1. The weave annotation with id 13 is used to specify that the RequirementA and RequirementB packages should be woven together. Since the body-attribute is *default*, the default weaving rules are used, which means that the two classes named A in the two packages will be woven together, since they match.

Finally, there is the weave annotation with id 12, which goes from class B in RequirementA, to class H in RequirementB. This weave annotation is added to indicate that these two classes should also be woven together, even though they do not match according to the default rules. This could also be specified using a weave annotation on the weave connection between the packages, using the statement: `+ RequirementA::B @ RequirementB::H`.

The above model is an example of input to the weaving process. On this input, the actual weaving transformations operates, transforming the input into a woven output model. Performing the weaving transformations is a Prolog program written by Harald Störrle, which performs a basic weave transformation. The program, at its current state, simply matches elements by name and type, as well as any matches dictated by weave annotations. The body of weave annotations are not, at the moment, understood, which means that only the default behaviour of the weave connection is supported. This behaviour is to weave together elements connected by these connections.

The result of performing the weaving transformations on the input in Listing 4.3 is shown in Listing 4.4 and Figure 4.8 (b).

```
1 me(package-14, [name-'pre-merge', ownedMember-ids([4])]).
2
3 me(package-4, [name-'RequirementA', ownedMember-ids([3, 1, 2, 8])]).
4 me(class-1, [name-'A', ownedMember-ids([5])]).
5 me(class-2, [name-'B', ownedMember-ids([6])]).
6 me(association-3, [memberEnd-ids([5, 6])]).
7 me(property-5, [type-id(2), association-id(3)]).
8 me(property-6, [type-id(1), association-id(3)]).
9 me(class-8, [name-'C', ownedMember-ids([9])]).
10 me(generalization-9, [from-id(8), to-id(1)]).
```

Listing 4.4: The result on weaving the model in Listing 4.3

4.4.4 Convert result to Java

After the model has undergone a set of weaving transformations, it is converted back to a Java representation, so that the user can continue working on it within

the framework of the tool.

The input to this conversion process is a Prolog program, consisting of a set of facts, each describing either a model element or a layout element.

Given that the program contains both model and layout elements, and GMF keeps the model separate from the diagram, the conversion should be done in two passes. First the model can be extracted, resulting in an EMF model. This can be used to generate an initial GMF diagram. The layout elements can then be extracted, and added to the diagram.

4.4.5 Reviewing the Weave Result

The result of the weaving process is a model, which should be displayed graphically to the user for review. This is done by creating a GMF diagram for the model after it has been converted to Java from Prolog, and displaying this diagram in a graphical editor similar to the one used to create the initial model fragments with.

In this editor, the user is able look at the model, make any number of changes, and save the resulting model. This is intended for the cases where only minor adjustments are required, or where the converted model is satisfactory, and the user wants to develop it further.

In cases where more fundamental changes are required, the user may wish to go back to a previous step of the weaving process, make some changes and redo the process, giving a new result model. This can be done in two places:

1. Before the weaving process is activated by the user, when the weave is being specified. The user can go back and make changes to the model fragments used in the weave, or change the weaving model in the weave editor.
2. After the weave model has been converted to Prolog, but before the weaving transformations have been applied. This requires editing of Prolog code, and allows an expert user to make small changes, without the overhead of changing the model fragments or the weave speficication using the graphical editors.

The first option uses the already described methods for specifying a weave: choosing model fragments, and drawing weave connections between elements.

The second option requires the ability to view and modify the Prolog code as it looks before weave transformations are applied. In order to review the weave result, and do these modifications in an integrated fashion, an editor with the following functionality is proposed:

- A graphical editor for showing the weave result model
- A simple Prolog editor, which contains the Prolog code before weaving transformations are applied

The Prolog editor will contain options for manually editing the Prolog code, and re-apply the weaving transformations to this code. It will also contain options for exporting and re-importing the Prolog code, and transforming the Prolog code directly to a Java representation with a diagram, bypassing the weaving transformations. This is useful for expert users who want to apply their own weaving transformation to the model fragments.

4.5 Weave Result editor

The Weave Result editor presents the result of the weaving process to the user, in the form of a new model diagram. This diagram represents a UML model like the ones created in the model fragment editor, and the user should be able to change, refine and improve this model.

This means that the weave result editor should reuse the tools from the model fragment editor, thus providing the same functionality in terms of modelling.

As described in Section 4.4.5, this editor should also allow the user to view the Prolog code as it looks before the weaving transformations are applied. Expert users will want to make minor changes to this code, and then re-create the weave result based on these changes, as it might prove to be a faster and sometimes more preferrable method for making changes than going back to the model fragments and redoing the whole weaving process.

Also, expert users may want to apply their own model transformations, rather than using the standard implementation. This is done by providing the option for exporting the Prolog model, after which the user can run whatever transformation he desires outside of the tool, and then reimporting the transformed model. This transformed model can then be directly converted to a Java model, without applying the standard transformations.

An illustration of the part of the weave result editor which provides this Prolog editing functionality is shown in Figure 4.9.

4.6 Improvements

4.6.1 Locking

The locking functionality described in Section 3.4.1 is meant to enable the author of a requirements specification to share the specification for review, and gain insightful comments and remarks, without having to go through an arduous process, where a number of small changes from the reviewers must be merged into the author's own work.

The purpose of locking the document may be two-fold: the author wants his work to be reviewed and commented, or he wants it to be only evaluated, without the reviewer being able to comment. This is useful in the course setting, where the work of students must be evaluated by the lecturer at the end of the year.

This leads to three different locking levels: unlocked, partially locked where commenting is enabled, and completely locked where all editing is disabled. These locking levels are handled as follows:

- Unlocked: no restrictions are in place
- Partial lock: Editing of element data is disabled. This is done by disabling the *save*-function in the editors for the locked elements. The user should be warned that an element is locked, so he is aware that changes are disabled.
- Complete lock: As with the partial lock, saving of editors is disabled, in addition to adding new comments, and deleting or editing existing comments.

The locking is done that the level of individual elements, which gives the author a fine-grained control over which elements should be locked, and which should be unlocked. This allows the user to specify that all the contents of a folder, except for a specific subset, should be locked. This is done by the following recursive algorithm, which determines the lock status of an element:

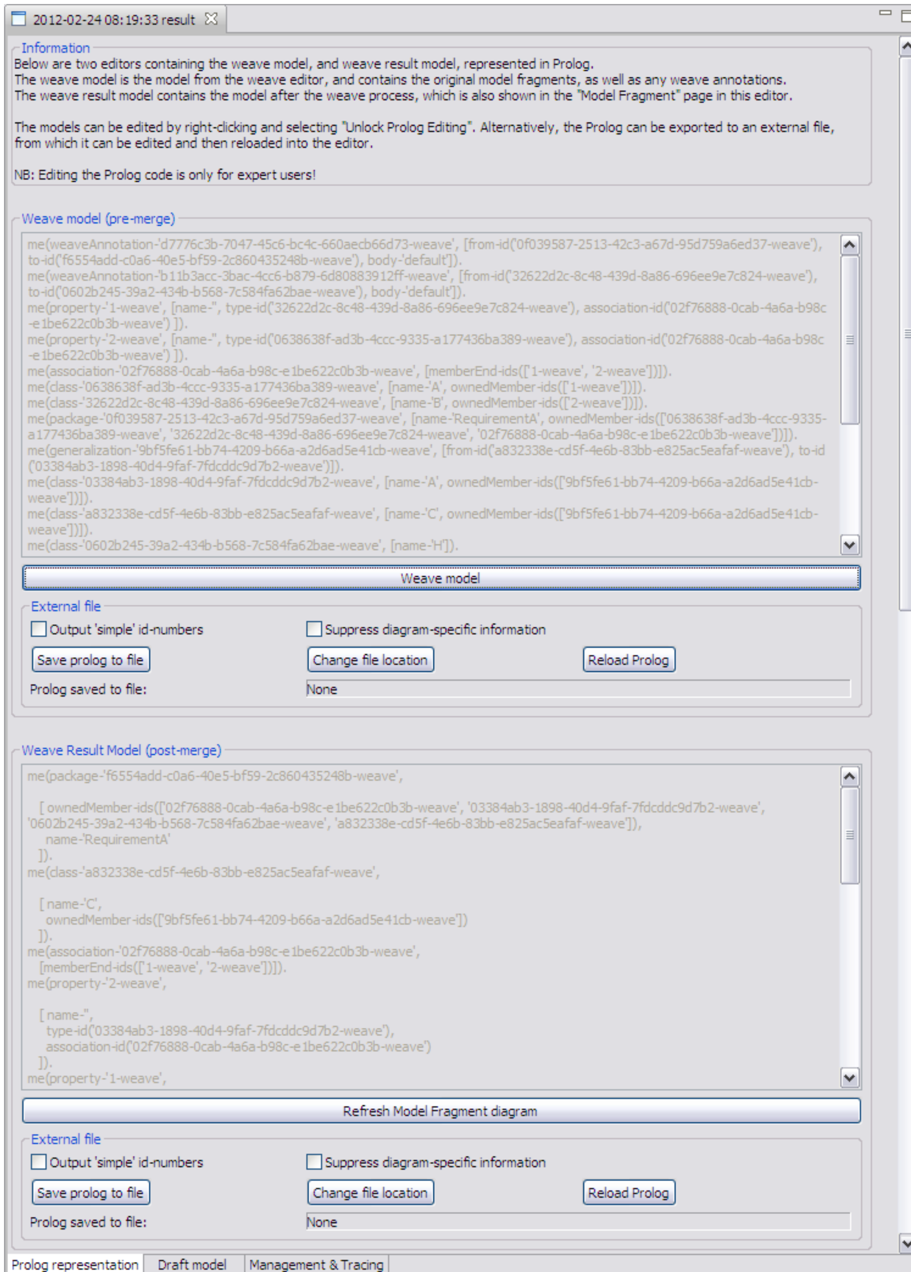


Figure 4.9: Illustration of the Weave Result Editor

```
1 proc getLockStatus() : LockType
2   if lockStatus == UNSPECIFIED
3     return parent.getLockStatus()
4   else
5     return lockStatus
```

This method for determining the lock status of an element means that to determine the lock status of an element, its own status is given precedence. If this status is unspecified, the nearest predecessor in the element tree with an unspecified lock status is used. This means that the user can easily lock a large group of elements, by simply locking a folder or other group high in the element tree structure. It also means that the user can unlock a specific element in such a locked subtree, by simply changing that elements lock status. This new lock status is also propagated down the tree, meaning a whole subtree can be unlocked in the samme manner as locked.

4.6.1.1 Lock password

As a further protection against editing, the author is able to password-protect the lock status of elements, which means that the lock status can not be changed without knowing the password.

When specifying the lock password, there should be an easy way for setting the same password for each locked element. An option is to have a common workspace-level password, which is transparently used as the lock password for elements. The user is able to set a workspace-wide password, through a menu option. This password is persisted across sessions, but is only stored locally on the users machine, meaning that it is not exported along with other artifacts from the tool.

When changing the lock level for an element, the following is done:

- If the element is already locked, the password for the element is checked. If it is the same as the workspace password, the operation is allowed. Otherwise, the user is prompted for the element password. If the correct password is entered, the operation proceeds to the following step, otherwise it aborts.
- The lock level is changed to the one specified by the user, and the elements password is set to be the workspace password.

The element password is, as opposed to the workspace password, persisted along

with the element itself, meaning that the password-protection is kept when the element is exported.

This solution provides a sense of security, although it is quite easy to break. Given the current way of persisting elements in XMI, the password is easily extracted from an exported element. The password could be encrypted, but then it may still be removed by simply altering the XML. To counter this, more advanced security features should be put in place, but this is counter-productive to the intended use of the tool and the locking feature. The purpose of locking is simply to discourage editing of elements by users who should only read or comment – if such a user really wants to make changes, then there is no way to stop him. Hypothetically, he could simply redo the project, copy-pasting the content. In addition, it should be easy to recover the password, if forgotten by the author.

4.6.1.2 Specifying lock level

The status of the lock level belongs on the existing *Management & Tracing* page which exists in all editors, as this page contains all administration-related information regarding an element. A "slider" widget is added to this page, which allows the user to change between different lock levels easily.

In addition to the previously mentioned locking levels – unlocked, partially locked and completely locked, the user should be able to specify a fourth option: *use parent's lock status.* This option clears the elements own lock status, and directs it to use the lock status of it's parent (which might also be set to use it's own parent's lock status, recursively).

4.6.2 Reporting

The RED tool contains a plugin for generating and outputting reports based on a set of elements created by the user in the tool. The current functionality enables the user to select a set of elements, choose a layout from a pre-defined set of report templates, and get a report in HTML which can function as an initial requirements specification document.

For the model fragments to serve their purpose as a communications tool between analysts and developers, they must be included in the generated reports. To include the model fragment diagrams in the generated reports, GMF contains utility classes which can convert a diagram to an image, which can be

passed to the report generation templates for inclusion. This requires that the template for requirements be modified to accommodate these new images.

The details of how the reporting component is designed and implemented can be found in Friis (2012).

Chapter 5

Implementation

This chapter describes various low-level technical issues, solutions and decisions. First, the structure of the codebase is described, by giving an overview of the various plugins which make up the system, as well as describing the package structure within these plugins, in order to make it clear how the code is organised. Then, the implementation details of the various components are described, along with any major difficulties and low-level technical decisions. Finally, the deployment procedure for the tool is described.

The purpose of this chapter is to give an overview of how the solution is implemented, in addition to decribing how various technical issues have been solved. This is particularly useful for developers wishing to extend the existing functionality, by providing information on where and how the various components are implemented.

5.1 Structure

This section describes the structure of the codebase, and how it is divided into plugins, in order to provide an overview of how the code is organised.

5.1.1 Plugins

The structure of the plugins containing the modelling and weaving functionality is based on the structure preferred by GMF. As described in Section 4.1, EMF is used to generate the domain model, in addition to providing the EMF.Edit framework, which provides advanced editing capabilities for the generated model.

On top of these, GMF creates a diagram plugin, which contains all the basic functionality required for a single visual diagram editor. Multiple diagram editors require multiple plugins, which means that different functionality are divided into separate plugins.

The overall plugin structure used in the RED system is as shown in Figure 5.1. This is a modular architecture, in which major components are isolated from each other, promoting separation of concerns, modularity and encapsulation. Inside each of these components, further layers are used, such as a data layer, logic layer and presentation layer.

The UML editor, the weaving editor and the weaving process is contained in the *ModelElement* component. This structure is shown in more detail in Figure 5.2. This component consists of the following plugins, which are depicted in the figure:

- `dk.dtu.imm.red.modelelements` (ModelElements Model) – This plugin contains the domain model of the modelling component, which is described in Section 3.5. This plugin contains mainly code generated by EMF, as well as the code implementing the logic for the weaving process.
- `dk.dtu.imm.red.modelelements.edit` (ModelElements Edit) – This is completely generated by EMF, with no modifications at this point. It serves as the editing framework for the model used by the diagrams.
- `dk.dtu.imm.red.modelelements.diagram` (UML Editor) – Generated by GMF, this plugin contains the UML fragment diagram editor. The majority of the code in this plugin is generated, with minor modifications. Major changes to the generated code is achieved by modifying the templates from which the code is generated.
- `dk.dtu.imm.red.modelelements.diagram.custom` (UML Custom) – This code contains additional functionality required by the diagram plugin, such as custom shapes, policies etc. The benefit of placing custom code like this in a separate plugin is that it reduces the risk of overwriting when re-generating code from GMF.

- `dk.dtu.imm.red.modelelements.weave.diagram` (Weave Editor) – Like the plugin for the UML fragment diagram editor, this plugin contains the weave editor.

- `dk.dtu.imm.red.modelelements.weave.diagram.custom` (Weave Custom) – As above, this plugin contains custom functionality required by the weave diagram plugin.

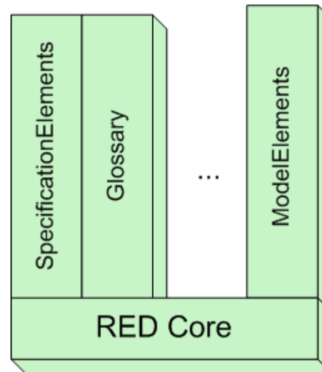


Figure 5.1: RED plugin structure

5.2 External libraries

5.2.1 JPL – Java Prolog Bridge

In order to perform the weaving process, the model fragments need to be converted to a Prolog representation, have some transformations applied to them,

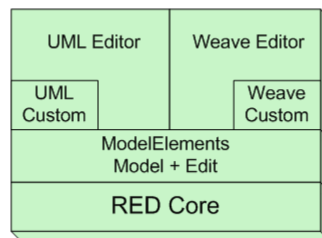


Figure 5.2: Modelelement plugin overview

and then converted back to Java. In order to contain and control the entire process within the tool, the 3rd party library JPL is used to start a prolog session, consult the converted model, call various clauses and finally convert the model back to a Java representation.

JPL consists of a set of Java classes providing an interface to Prolog, and some C functions hidden from the developer, which provide the actual bridge. JPL is distributed as part of the SWI-Prolog implementation, and is contained in two .dll-files, *jpl.dll* and the required *swipl.dll*, as well as the *jpl.jar*, which contains the Java interface. To include JPL in the project, these files have been added to the build path of the `dk.dtu.imm.red.modelelements` plugin. In order to load the .dll files, a call to `System.loadLibrary("swipl"); System.loadLibrary("jpl");` is required before JPL can be used.

5.3 GMF

5.3.1 Fragment Editor

Figures 5.3 to 5.5 show the final version of the UML model fragment editor integrated in the RED tool, with various types of UML models. The figures show the fragment editor as a page inside a requirements editor, along with the properties-view in the lower right corner.

5.3.1.1 Models

As described in Section 4.1, GMF diagram editors are implemented by creating a number of models, from which the editor code can be generated. The GMF models used to generate the fragment editor are:

- `activitydiagram.gmfgraph` – containing graphical definitions of the various figures used in activity diagrams
- `classdiagram.gmfgraph` – containing graphical definitions of the various figures used in class diagrams
- `statediagram.gmfgraph` – containing graphical definitions of the various figures used in state machine diagrams
- `usecasediagram.gmfgraph` – containing graphical definitions of the various figures used in use case diagrams

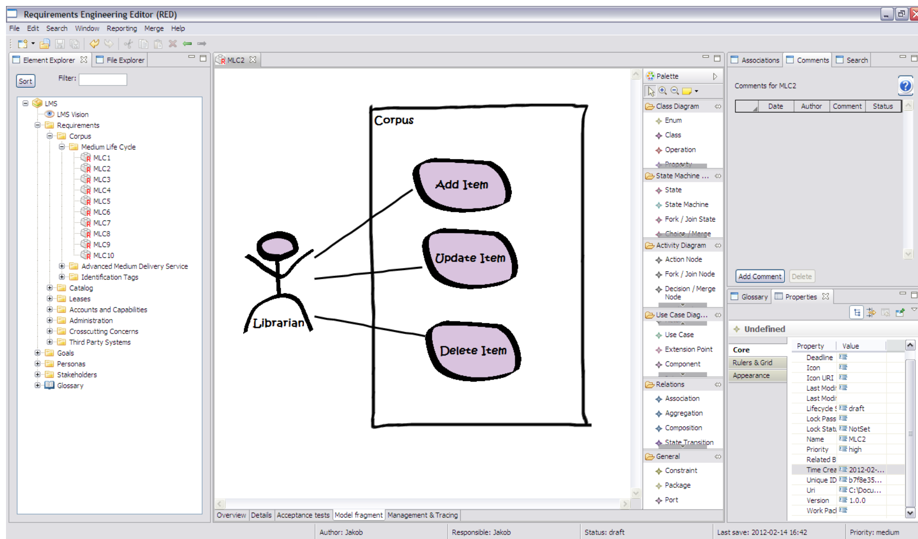


Figure 5.3: UML editor with use case diagram

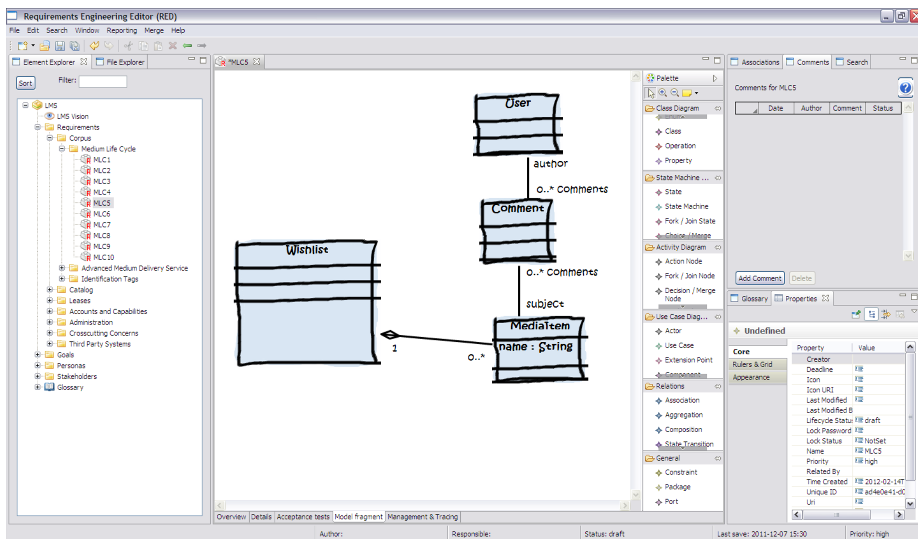


Figure 5.4: UML editor with class diagram

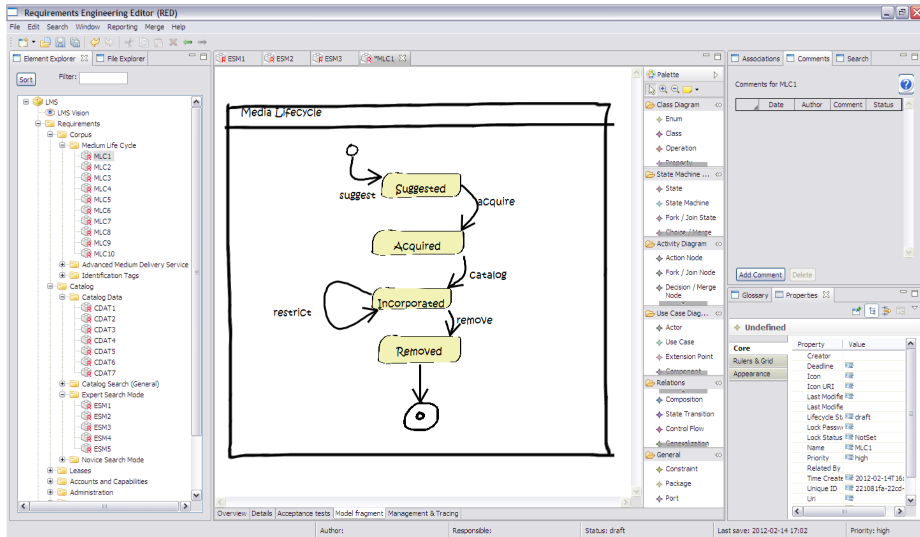


Figure 5.5: UML editor with statemachine diagram

- `modelelement.gmfgraph` – containing graphical definitions of various common figures which does not belong to one of the above, such as packages.
- `modelelement.gmftool` – describing the various tools and the structure of these in the toolbar
- `modelelement.gmfmap` – mapping the graphical elements to domain elements
- `modelelement.gmfgen` – defining the details of how the diagram should be generated

5.3.1.2 Dynamic Templates

During the development of the fragment editor, it became evident that several changes to generated code was needed in order to fit the requirements of the tool. For example, the following issues requires editing of generated code to be resolved:

- Setting the default smoothness of connections from `NONE` to `NORMAL`, as described in Section 5.3.3.2, requires that the method `refreshSmoothness()` is overriden in all generated connection classes

- A bug in GMF means that when dragging a node into or out of a compartment, a spurious duplicate node appears in the diagram. This is resolved by adding a call to `super.refreshSemantic()` last in all generated compartment edit parts' `refreshSemantic()` methods
- When a node consists of a compartment, a line is added above the compartment as part of its border. To remove this, `setBorder(null)` must be called on the compartment figure in the generated compartments edit part
- Elements like a Port, which is affixed to the side of another node, need a custom `BorderItemLocator` in order to be placed correctly, if they should be centered on the border instead of being placed next to it
- In order to allow the user to insert newlines into labels, and have them automatically wrap if they get to long compared to the node in which they are placed, labels need a `WrapTextCellEditor` as their text edit manager, instead of the default one, and they need the `textWrap` property to be set to true when created.

Manually editing the generated code is dangerous and should be avoided for various reasons.

- It is repetitive, as the specified changes must be made in many different classes
- There is a high risk that the changes are not applied to future elements, and the developer may forget or not know about the changes which need to be made
- A change in a generated method requires that the method be annotated with `@generated NOT`, in order for it not to be overwritten the next time the code is generated. This is easy to forget.
- Further, other pieces of code in such an annotated method may depend on information in the graphical models. If this information changes, the code may not be updated properly.
- There is a risk of losing compatibility when the framework is updated

In order to avoid manually editing generated code, it has been made a principle that all such code changes, which apply to a broad set of generated classes, must be made in the templates which are used for the actual code generation. These templates exists in the GMF plugin `org.eclipse.gmf.codegen`, and are written in the Xpand template language.

In order to modify these templates, they are copied to the `dk.dtu.imm.red.models` project. Care must be taken to ensure that they are placed correctly in a folder hierarchy which matches the hierarchy in the GMF plugin. See Figure 5.6 for a specific example of how the structure is mirrored in the actual Eclipse projects. Note that templates which are not modified need not be copied to the `dk.dtu.imm.red.models` project. The code generation runtime is capable of merging the two sets of templates.

To specify that these modified templates should be used, the option *Use Dynamic Templates* must be set to `true` in the relevant GMF generator model (.gmfgen file), and the *Dynamic Templates Directory* must be set correctly. In this case, it is set to `/dk.dtu.imm.red.models/model-gmf-templates`.

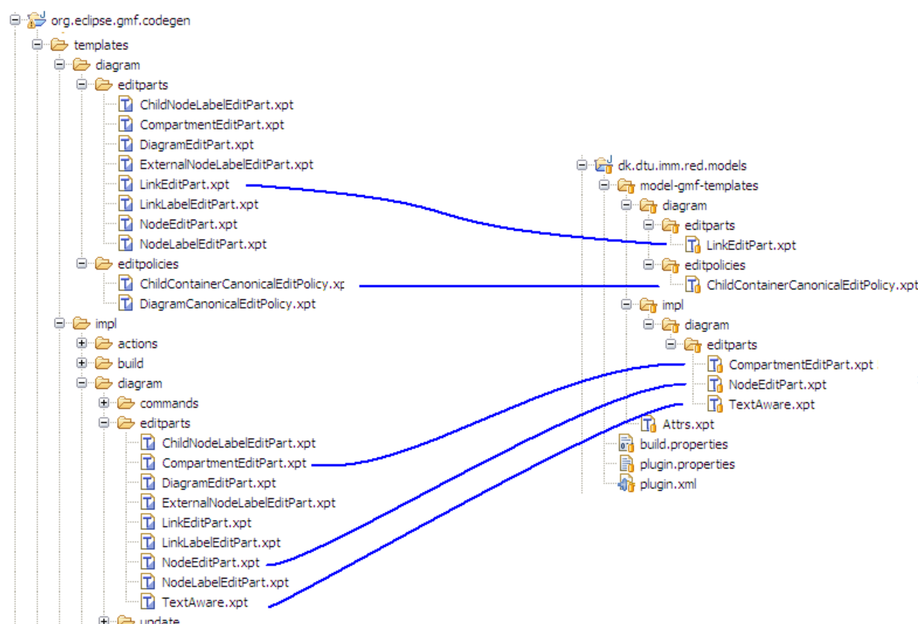


Figure 5.6: Required structure of GMF templates

5.3.2 Weave Editor

Figures 5.7 and 5.8 show the Weave Editor, containing a number of packages and weave connections, specifying how the weaving should be done.

Figure 5.7 shows the editor with two fragments, with a weave connection be-

tween them. This specifies that the two fragments should be woven together, resulting in a fragment where the *Media*-class from the fragments have been merged.

Figure 5.8 shows, that when multiple fragments are woven together, the fragment compartment can be collapsed, hiding the fragment contents. The result resembles a UML package diagram, focusing on the packages and how they should be woven together. This figure also shows the weave annotations, which specify additional weave details. In this case the annotation in the upper right corner specifies that the element *Medium* in *MLC1* should be woven with element *Media* in *MLC3*, even though their names do not match. The annotation on the connection between *MLC4* and *MLC5* specifies that *Librarian* in *MLC4* should be woven with *Libarian* in *MLC5*, due to a spelling error in *MLC5*, while the elements *User* in the two fragments should not be woven.

Figure 5.9 shows how the compartment of a model fragment, or any element capable of containing other elements for that matter, can be collapsed using the small "+"-button in the upper left corner.

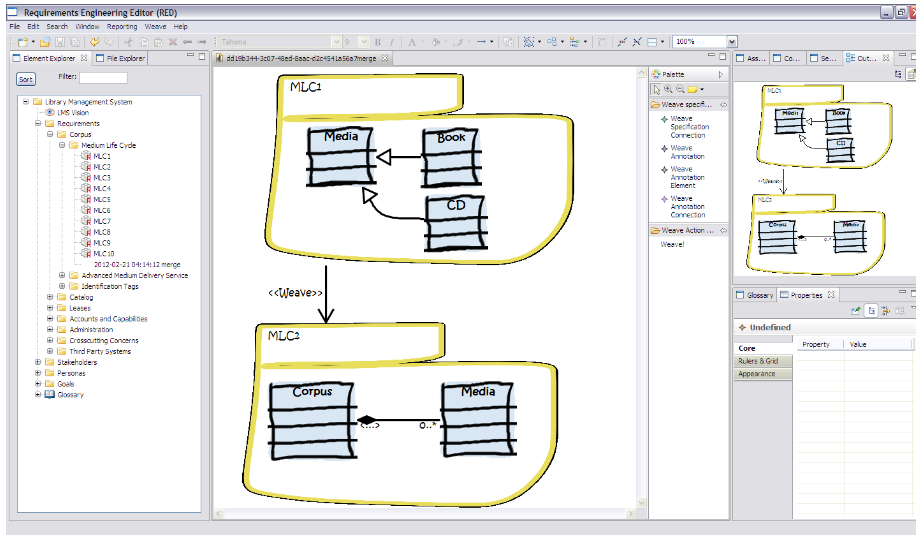


Figure 5.7: Weave Editor with two fragments

5.3.2.1 Models

Like the model fragment editor described below, the weave editor is implemented using a set of GMF models. Since the weave editor needs to be able to draw the

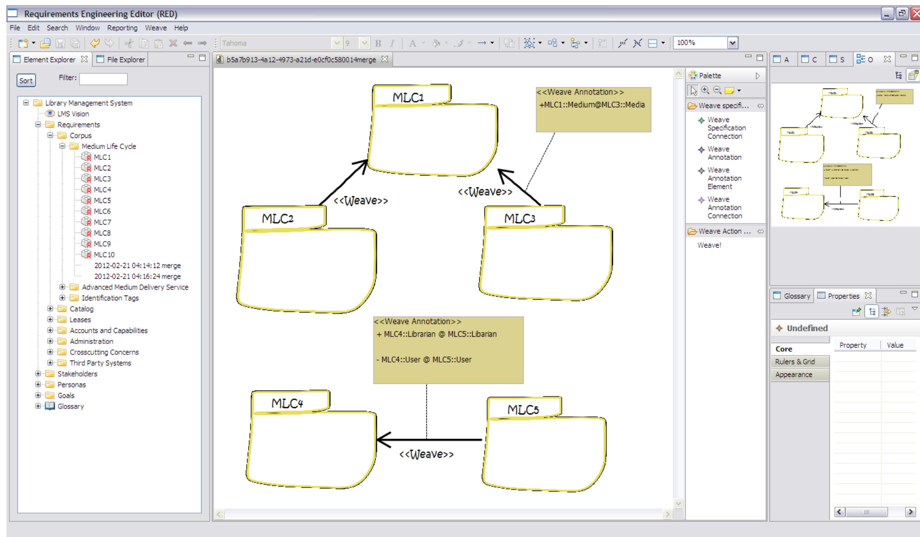


Figure 5.8: Weave Editor with 5 fragments, with collapsed compartments

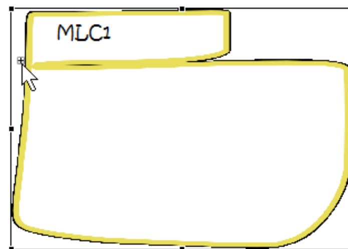


Figure 5.9: A compartment is collapsed using a small button in the upper left corner

same graphical figures as the fragment editor, it reuses all the graphical definition models. In addition to these, the weave editor requires the ability to draw a weave specification connection, which is defined in `weaveeditor.gmfgraph`.

Since the weave editor requires different tools, and a different mapping to the domain model than the fragment editor, it requires its own tooling and mapping model, which are provided in `weaveeditor.gmftool` and `weaveeditor.gmfmap`.

5.3.2.2 Dynamic Templates

As in the case with the fragment editor, there are some issues which is best fixed by changing the templates from which the figures' and diagram code is generated. The issues described in Section 5.3.1.2 are fixed the same way for this editor, in addition to these additional issues:

- It should not be possible to delete elements or modify labels in the weave editor. This is enforced by the edit policy

```
1        dk.dtu.imm.red.modelelements.merge.diagram.custom.edit....
             policies.NoDeleteComponentEditPolicy
```

  which should be installed as an edit policy on all connections and nodes, except for elements which are specific to the weave editor.

- It should not be possible to change the source or target of a connection, unless it is a weave editor specific connection. This is done by overwriting the `getInsteadCommand()` in the various edit parts *EditHelper*.

- It should be not possible to drag elements out of compartments. This is done by returning an instance of `UnexecutableCommand` from the compartments' `getMoveCommand()`-methods

5.3.3 Sketchiness

In order to achieve the sketchy, hand-drawn look which is required in the UML fragment editor, it is necessary to look at the options for modifying the graphical appearance of the shapes and connections in GMF, as well as the font used.

5.3.3.1 Shapes

The GMF graph metamodel contains a few standard shapes, such as rectangles, ellipses and a polygon shape which can be defined by a list of points. The generated code using these shapes refer to various basic shapes defined in Draw2d, which could be subclassed to provide new paint()-methods. These new paint()-methods could be used to draw the shapes required to give the diagram the desired look. However, the graphics context which is passed to these paint()-methods, and is responsible for the actual drawing of the shapes, only supports drawing images, and basic shapes with standard line types. The ability to change the line width and style, such as using a dotted or dashed line instead of a solid one, is not sufficient to get the hand-drawn look which we require, as described in previous chapters.

Instead, the use of images seems to be a better approach. By creating the required shapes and figures in an illustrating software and providing them as image files, two benefits are gained:

- The designer has more complete control over the look of the figures, and any 3rd party image manipulation software which can output standard image formats can be used
- To change the shapes and figures in the future, only a new image file has to be supplied. No code changes are necessary, only a rebuild of the software since the images exists as resources in a plugin bundle.

To support using images in GMF figures, the GMF graph metamodel provdes a custom figure, which consists of a name and a qualified class name, which is the class which implements the actual figure. The custom figure can contain child nodes, such as labels, layout constraints and layout strategies, just as the regular shapes, which is required in order to display the various textual data such as names of diagram elements.

The custom figure must implement the IFigure interface, which means that it can extend the Draw2d class ScalableImageFigure, which draws a given image filling the entire figure and handles scaling. It supports non-uniform scaling, which is important, as the user will surely want control over the actual shape and dimensions of the diagram elements. The SVGFigure, which is another figure type supported in newer versions of GMF, does not support non-uniform scaling, which is the major reason why this type was not selected.

There are, however, a couple of disadvantages to using the ScalableImageFigure:

1. The anti-aliasing functionality does not seem to work, which results in somewhat pixelated figures
2. Transparent backgrounds has not been made to work either, which is noticeable when a figure overlaps another figure, or is contained in a compartment with a background color
3. The non-uniform scaling means that outlines are not scaled correctly. A very stretched figure will thus have a bigger linewidth in the stretched dimension than in the non-streched one

In an attempt to alleviate problems 1 and 3, another approach is used in the case of Class Diagrams. This diagram type was chosen for experimentation because it contains elements with relatively simple shapes, namely classes and enums, which are both rectangular and divided into other rectangular subsections for operations and properties.

The figures for these elements are implemented as a custom figure with a custom border. The custom figure class is responsible for drawing a colored background with an irregular shape. The custom border draws the outline of the shape, using an image of the border divided into 8 parts: 4 sides and 4 corners. The corners are placed in fixed positions relative to the dimension of the figure being drawn, and then the side lines are stretched to match the figure's size. This ensures that the line width is constant, irrespective of stretching in the direction orthogonal to the line, and it reduces aliasing to a non-noticable problem, as shown in Figure 5.10. The Action node shows the anti-aliasing problem, which makes it pixelated. It also shows the stretching problem, where the vertical lines are much thinner than the horizontal lines, due to the vertical scaling. This is not apparent in the Class node. In addition, the Use Case shows the transparency problem.

This approach seems to work fairly well for classes. However, it can not easily be transfered to other shapes, such as diamonds, ellipses or more complex shapes. Some shapes, such as diamonds, can use the same method if rotation of the drawn line can be implemented correctly (keeping in mind that the angle of the sides varies with the non-uniform scaling), but rounded lines, such as those used in circles or ellipses, can not use this approach.

5.3.3.2 Connections

Connections are handled differently than node-like shapes in GMF. The visual representation of a connection is defined by a source- and target node, anchor-

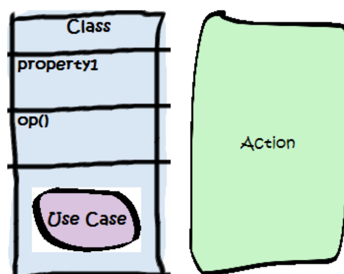


Figure 5.10: Example of UML nodes. The Class is drawn as a background, four corners and four side lines, while the Action and the Use Case are single images.

points on this node and various bendpoints, which allows the user to change the connection from a straight line to one that, for example, routes around obstacles.

The ability to insert bendpoints into the connection is very important, as it gives the user more freedom to layout the diagram. As standard, the GMF connections consist of straight lines between these bendpoints, but the framework also has an option for setting the *smoothness* of the line to one of the following values: NONE, LESS, NORMAL or MORE. A comparison of these settings and their effect on a simple connection with a bendpoint is shown in Figure 5.11.

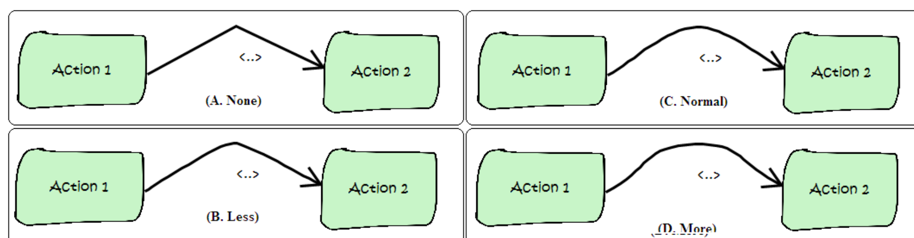


Figure 5.11: Comparison of smoothness values

Based on a short evaluation, the NORMAL smoothness setting, shown in figure 5.11 part (C), is chosen, as it was rated to resemble a hand-drawn stroke the most.

The connection decorations, such as arrowheads, or the filled and unfilled diamonds on compositions and aggregations, are implemented using the simple *Polygon Decoration* in the GMF graph model, by defining the simple shapes using a few points. These could also be implemented using the *Custom Decoration* element from the graph metamodel, perhaps using images for a better look, but it was assessed that the current solution is sufficient.

5.3.3.3 Font

The font is another important factor in making the diagrams look hand-drawn. Several fonts exists, which are designed to look like regular handwriting, so choosing one seems to be a matter of finding one which looks comparatively good.

One font, *Alpha Mack AOE* (Figure 5.12), was suggested, and looked particularly promising, until it was discovered that it lacked certain important symbols such as square and curly brackets.

Instead, the fonts which are bundled with GMF by default were compared, and a font named *Kristen ITC* is chosen (Figure 5.13). The font is configured in the GMF graphical definition model, where a property named `Basic Font` can be set to a value from a list of pre-defined fonts. The benefit of this font is that it comes with GMF, which means less work adding a new font to the framework.

Figure 5.12: Alpha Mack AOE

Figure 5.13: Kristen ITC

5.4 Installation procedure

This section first describes the installation procedure for end users who wish to install and use RED. This is a quite simple procedure, since no installation tools are required.

Next, a description is given on how to setup a development environment, for developers wishing to extend the codebase with new functionality. This requires a number of plugins and tools, the acquisition of which are described here.

5.4.1 Users

The installation procedure for users is exactly the same as described in Friis (2012). The tool is built using a standard build procedure using Eclipse and the plugin-development environment, which produces a folder containing the application, consisting of an executable file, all required plugins and various configuration elements. This folder can easily be distributed, which means that no installation procedure is required from the users' viewpoing – they simply unpack the folder and run the executable file.

5.4.2 Developers

In order to extend the functionality of the tool, a suitable development environment has be to set up for two reasons: various Eclipse-plugins are used for the actual development such as EMF and GMF for code generation and model editing, and a number of plugins are required by the tool itself, such as Eclipse core plugins, EMF, GMF etc.

The starting point for the setup procedure is the Eclipse IDE for RCP and RAP developers[1], which contains the plugins required for building RCP applications, most notably `org.eclipse.pde` (Plugin Development Environment) and `org.eclipse.rcp`.

In addition to this, the following plugins are required, either for development purposes, or because they are required by the tool. The version numbers given are the ones used at the time of writing. Updates may be available, but compatibility can not be guaranteed.

- Agile Grid v. 1.2.0 (`org.agilemore.agilegrid`) – this plugin provides the 3rd-party table-widgets, which are used in various places such as comments, associations and acceptance tests. It is available from `http://agilegrid.sourceforge.net/`, which also provides installation procedures.

- Eclipse Modeling Framework Runtime and Tools v. 2.7.1 (`org.eclipse.emf`) – the EMF tools are used for creating and maintaining metamodels, while the runtime provides the code containing the underlying EMF functionality. It is available from `http://www.eclipse.org/modeling/emf/`.

[1] http://www.eclipse.org/downloads/

- Graphical Editing Framework v. 3.7.1 (`org.eclipse.gef`) – GEF is required by GMF, and is avaiable from `http://www.eclipse.org/gef/`.
- Graphical Modeling Framework Runtime v. 1.5.0 (`org.eclipse.gmf.runtime`) – this plugin contains the core GMF functionality, and is required both for development and during runtime of the tool. It is available from `http://www.eclipse.org/modeling/gmp/`.
- Graphical Modeling Frameowrk Tooling v. 2.4.0 (`org.eclipse.gmf.tooling`) – the tooling plugin provides editors for creating and manipulating GMF models, which is required for GMF development. It is available from the same site as the runtime.

Chapter 6

Evaluation

6.1 Evaluation criteria

In order to evaluate the extent to which the work described in this thesis has met the goals declared in previous chapters, a number of evaluation criteria are listed:

- Usability – Is the tool usable, and how is the performance?
- Functionality – Has the required functionality been implemented, and is it working properly?
  - Is the functionality stable?
  - Does it correctly implement the specifications?

The following sections looks at these questions in turn.

6.2 Usability

As stated in previous chapters, the usability of the tool is of high importance, since students using the tool in a classroom setting should not spent time learning the tool or struggling with poor usability, but rather focus on the learning material.

In order to get a solid evaluation of the tool's usability, a thorough usability test should be performed, using real users, who have not been involved in the development of the tool. These users should, in a controlled and monitored setting, be assigned a number of tasks which can be solved using the tool. The observations of their behaviour, as well as their explicit feedback, should then be analysed to deduce where improvements to usability is needed.

In addition to this classic usability study, a prolonged study should be performed, for example using the students of the Requirements Engineering course for a semester. These students should be subjected to creating a requirements specification in the tool, which will give them a good amount of experience using the tool. At the end, as well as during the course, these experiences should be collected.

In order to improve the quality of the data collected, some measures can be taken. To avoid bias, the students could be split in two groups: a test group, which should create their work using the tool, and a control group which should create the same work, but without the tool. This would help filter out issues stemming from the assignment itself, although it might be unfair to the control group (or the test group, if the tool proves to reduce productivity or quality). To alleviate this, the students could instead be made to work without the tool for the first half of the course, and with the tool for the second half. This way, they would have a better background for evaluating the usefulness of the tool.

Although desirable, such a usability study has not been made during the course of this thesis. A major reason for this is timing, since the course only runs once a year in the fall semester at DTU. Being developed during the fall and winter, a version fit for usability testing was not available in time.

An issue with the usability testing is that a certain knowledge of requirements specifications, and the context in which the tool is to be used, is required in order to provide meaningful data for a usability test. As such, students of the course, or possibly other sofware analysts/architects/engineers, are required, making it difficult to perform the study outside of the course. As such, the usability study has been deferred, and stands as an open task, ready to be performed in the future.

As a substitute for a real usability study, the course lecturer, Harald Störrle, has been given a prototype of the tool a month before the submission of this thesis. This has resulted in numerous improvement requests, which have served to fix broken functionality, add new features and improve usability.

6.2.1 Performance

An aspect of the tool's usability, which is testable without a group of test persons, is measurable metrics, such as the performance of various operations.

The evaluation of the performance of the tool is split into two categories: how responsiveness the tool is under various situations (startup, normal operation, heavy load), and how big the latency of various operations are.

6.2.1.1 Responsiveness

The responsiveness of the tool, with regards to the components added during this thesis, is especially relevant when working with diagram editors, as these seem to put the highest load on the system.

Eclipse uses a strategy of lazily loading required plugins, i.e not loading plugins before they are used. This gives a shorter application startup time, and a lower average memory use, but may introduce heavy load when running the tool, if a large plugin or a number of plugins suddenly need to be loaded, as a response to a user action.

In the tool, this is especially noticable when opening the *requirements editor* for the first time since starting the application. This prompts the RCP framework to load GMF and all associated plugins, which may give rise to a noticeable delay on low-end machines.

As soon as the plugins and resources are loaded however, the graphical editor is very responsive. A test was carried out, where over 60 model elements (States) and over 30 connections were created. The tool showed no sign of slow responsiveness at any point, even when several model elements were dragged around in the editor, prompting elements and connections to be redrawn.

6.2.1.2 Latency

In addition to the responsiveness described above, the latency of certain operations, which may potentially run for a longer time, also affects the user experience.

Most notably is the weaving process, which may take some time, considering the amount of steps in the process:

- The model is converted to Prolog
- Weave transformations are applied
- The model is converted back to Java
- A diagram is created for the model

Appendix A shows some results for the first step, which shows that the latency is in the area of 0-100ms, depending on the size of the model (100ms is achieved with models with 4000 elements, which would be far more than the common use case).

Various, informal experiments show that the latency for the remaining operations is negligible, so measurements or analysis of the runtime of these will not be provided.

6.3 Functionality

The tools functionality is evaluated in terms of completeness and correctness. Completeness is a metric which determines to which degree the goals stated in the beginning of this report has been covered, while correctness covers whether the implemented functionality behaves as specified.

6.3.1 Completeness

The goals, as stated in Section 1.2, are as follows:

- Add UML modelling capabilities to the requirements engineering tool created by Friis

- Provide functionality for extracting and weaving UML model fragments
- Improve the existing tool with various requests for functionality.

The evaluation of whether these goals have been accomplished is done as an objective evaluation of whether the given functionality is present in the tool. From this, the following results are apparent:

- The tool now contains UML modelling capabilities, in the form of a graphical UML editor added as a page to the existing requirements-editor. The user is able to model Class, Use Case, State and Activity diagrams. The diagrams can be saved, and are also included in the report generation process.
- The user is able to extract model fragments for display in a graphical diagram editor, the weave editor. In this editor, it is possible to specify weaving details, by drawing so-called weave connections between elements. These connections can be annotated, allowing the user to specify the weave at a high level of detail.
- A basic weave-operation is available, which performs a simple model weave. This part is not complete, though a prototype working on simple cases is implemented.
- The following additional functionality has been added:
  - The user can lock elements at two different lock levels: one which disables editing, and one which also disables commenting.
  - A visual folder editor is added, which gives the ability to re-arrange folders in a graphical editor.
  - Comments can be exported in a comma-separated format, allowing them to be imported in another tool.
  - Various usability issues have been resolved.

6.3.2 Correctness

Correctness is ensured by:

- Unit testing the parts of the code which is suitable for unit testing. Since a lot of the tool's codebase is GUI and generated diagram code, not everything is suitable for unit testing, and it can not be used to ensure complete coverage.

- Exploratory black-box system testing, which is done by trying out various use cases a number of times, noting down any bugs and stability issues. The testing is done in an exploratory manner, meaning that detailed test scripts are not written beforehand. Instead, the test is designed as it is carried out. This provides a more agile approach, which is suitable when the tester, developer and designer is the same person. Since the turnaround time for the tests is very low, many details may be explored.

6.3.2.1 Unit testing

The code related to the UML modelling and the weaving part, which is suitable for unit testing, is the code responsible for implementing the weaving process, notably the code which converts the model to Prolog, and the code which converts the Prolog back to Java. This, as opposed to the rest of the modelling component, is not auto-generated, and not UI related.

Most of the other functionality implemented, mainly improvements to the core functionality such as locking of elements and exporting of comments, benefits from unit testing, where a high code coverage can be obtained.

A high code coverage has been found to be especially good at uncovering stability issues resulting from boundary cases caused by the input data, such as null pointers. In order to measure the code coverage of the unit tests, the third party tool *EclEmma 2.0.1* [1] has been used, which analyses the code coverage when unit tests are run. The results of this analysis has been used to write additional test cases to achieve a code coverage close to 100% for some parts.

6.3.2.2 System testing

In addition to the unit testing described above, exploratory black-box testing has been performed on the tool. This testing process has been carried out in an ad-hoc manner, concurrent with the development of new features, by trying out the newly implemented features in the tool in as many ways as possible, noting down whenever something did not behave as expected, and whenever errors occured or exceptions were thrown.

The criteria, by which these tests has been carried out, are:

- Regression – has the new functionality broken anything?

[1] http://www.eclemma.org/

- Core functionality – does the system correctly implement the use case at hand?
- Boundary testing – does the system function correctly in “corner cases”?

To some extent, whenever an issue is found and fixed, a unit test is written to ensure that the issue does not reappear. This has been used most to ensure the correctness of the Java–Prolog–Java conversions in the weaving process.

Chapter 7

Conclusion

This thesis set out to solve two primary goals: bridging the gap between between clients and developers of software development projects, and improving the Requirements Engineering Editor (RED) tool for the students in the 02264 Requirements Engineering course at the Technical University of Denmark.

The RED tool has been created by Friis (2012), who has built the foundation for a requirements engineering tool dedicated to assisting the students in the course creating requirements specifications. The initial version of the tool provides a prototype, intented for evaluation by students in the fall of 2012, and subsequent improvement in the future by both graduate and undergraduate student projects.

This thesis has improved the tool, and achieved the main goals, by (a) creating a graphical modelling editor, allowing users to create small UML models attached to requirements; (b) creating a method for extracting these UML models, weaving them together into a draft analysis model, and exporting this as either an image, or in a Prolog representation resembling the XMI structure commonly used to serialise UML models; and (c) providing various improvements to the set of basic features, such as locking of elements for reviewing, exporting of comments for integration with inspection tools, a graphical folder structure editor and minor improvements to navigation.

The graphical modelling editor enables users to draw small UML fragments, which can be used to clarify and exemplify the intention and meaning of individual requirements. The benefit of this is twofold: it improves the communication between, on one hand, the analyst and client, who together specify the various requirements, and on the other hand the developer who is responsible for reading the specification and implementing the requirements. In addition, coupled with the ability to weave together sets of these UML fragments, it provides a trace between requirements and the draft analysis model, which is the result of the weaving process.

The weaving component consists of an editor for specifying weave details, the integration of a simple model weaver written in Prolog by H. Störrle, and an editor for evaluating and improving the weave result. The weaving resembles the UML Package Merge, with some key differences: there is support for weaving elements beyond the few elements, for which explicit merge transformations are defined in OMG (2011) (Packages, Classes, DataTypes, Assocations, Properties, Operations and Enumerations), and the user is able to annotate the weaving model with instructions for the model weaver. This allows the user to specify that two elements should be woven, even though they do not match (for example if they have different names), or that two elements should not be woven, even though they match.

The various additional functionality and usability improvements has been implemented at the request of H. Störrle, the primary stakeholder for the tool. The focus has been on providing functionality which integrates the RED tool with other tools created in the course context, such as the Formal Inspection Tool, FIT by Petrolyte.

In conclusion, this thesis has built upon the vision of providing advanced tool support for students of requirements engineering. The primary success criteria for the work done in this thesis is whether future students will benefit from and enjoy using the functionality provided, and whether the functionality will be improved, expanded, and used as inspiration for new ideas in the future.

7.1 Limitations and Future Work

Above, the work produced in this thesis has been summarised. This section will conclude this thesis by listing the known limitations and deficiencies of the current implementation, the areas uncovered during this project which need further study, as well as a vision for future improvements and features.

One of the main goals has been to provide forwards traceability from requirements, by establishing a link between requirements and design models through the UML model fragments. The output of the weaving process is a UML model consisting of a set of UML fragments woven together, and can be used as the starting point for the work on analysis- and design models. This model contains implicit links back to requirements, since the identifiers of the various model elements can be traced back to the original model fragments. In this manner, one can trace the requirements which underlies the various elements in a design model, and document that the design provides adequate coverage of the requirements.

Due to time constraints, no additional work has been put into this tracing aspect of the tool besides laying the foundation and describing the possibilities for future work. Requirements tracing is an important research topic, and automatic tracing support will likely prove to be very beneficial for change management and systems evaluation.

The model fragments created in the tool are designed to have a hand-drawn look, with the purpose of conveying the fact that the fragments are created at an early stage, and are still open to critique and editing. An evaluation of whether this message is correctly interpreted by real users is needed, and can, for example, be achieved by doing an empirical study among the students of the Software Engineering course. In addition, some work is needed to improve the visual appearance of the model fragments. Despite some efforts, some functionality such as transparent backgrounds and anti-aliasing has not been achieved – functionality, which would drastically improve the overall visual quality of the diagrams.

The future students of the course should be used as subjects for evaluating areas such as usability, the provided functionality and identifying features which lack in the current version.

Finally, more work can be done on integrating the tool with 3rd party systems. Due to lack of time, the user is not able to import previously exported comments. This functionality would improve the integration with FIT. The weave result model can only be exported as Prolog and not XMI, which would enable direct integration with full-blown UML modelling tools such as MagicDraw and the coming implementation of VMQL in this setting.

Appendix A

Performance comparison of reflective versus non-reflective visitor implementation

In order to evaluate the performance impact of using reflection to implement the visitor pattern, the two methods were compared by measuring the execution time of a number of trial runs.

The two different implementations are shown in Listings A.1 and A.2.

```
public void visit(Object object) {
  for (Method method : this.getClass().getMethods()) {
    if ("visit".equals(method.getName())) {
      if (method.getParameterTypes()[0].isAssignableFrom(
            object.getClass()) &&
            method.getParameterTypes()[0] != Object.class) {
        method.invoke(this, new Object[] {object});
        break;
      }
    }
  }
}
```

Listing A.1: Visit()-method with reflection

```
public void visit(Object object) {
  if (object instanceof Class) {
    visit((Class) object);
```

```
4   } else if (object instanceof State) {
5     visit((State) object);
6   } else if (object instanceof UseCase) {
7     visit((UseCase) object);
8   } else if (object instanceof Enum) {
9     visit((Enum) object);
10  } else if (object instanceof ActivityNode) {
11    visit((ActivityNode) object);
12  } else if (object instanceof StateMachine) {
13    visit((StateMachine) object);
14  } else if (object instanceof Component) {
15    visit((Component) object);
16  } else if (object instanceof Package) {
17    visit((Package) object);
18  }
19  ...
20 }
```

Listing A.2: Visit()-method without reflection

The performance is measured by creating a UML model with a large number of elements, and converting it to prolog while measuring the time it took. The UML model is created by first creating a root package, and a pointer `pointer` which points to this package, and then entering a `for`-loop iterating n times. For each iteration, three random model elements and a package are added to the packaged pointed to by `pointer`, and then `pointer` is set to point to the new package. This creates a tree of elements n levels deep, each containing three random elements and a package.

This test case is run 20 times each for three different models of different sizes: 400, 2000 and 4000 elements. This is done to provide a data set which is large enough to provide an average which can be used for comparison, and also to see how the performance of the methods scale. The test is run on a PC running Windows XP, with a Intel Core 2 Duo P8600 2.40 Ghz CPU and 2 GB RAM. The test results are shown in Table A.1.

Even though there are some outliers in the data, the trend seems to be that the method using reflection is between 2 and 5 times slower than the method without reflection, and that the execution time seems to rise linearly with the number of elements in the model, though this can not be stated definitively from this set of data.

	With Reflection			Without Reflection		
	n = *400*	n = *2000*	n = *4000*	n = *400*	n = *2000*	n = *4000*
1	29 ms	89 ms	112 ms	5 ms	20 ms	26 ms
2	13 ms	41 ms	114 ms	8 ms	13 ms	54 ms
3	16 ms	42 ms	84 ms	7 ms	16 ms	28 ms
4	8 ms	44 ms	109 ms	2 ms	14 ms	53 ms
5	7 ms	68 ms	82 ms	3 ms	14 ms	27 ms
6	7 ms	43 ms	80 ms	1 ms	13 ms	49 ms
7	7 ms	41 ms	108 ms	2 ms	41 ms	22 ms
8	7 ms	46 ms	81 ms	1 ms	19 ms	50 ms
9	7 ms	71 ms	109 ms	2 ms	18 ms	22 ms
10	7 ms	37 ms	95 ms	1 ms	11 ms	48 ms
11	8 ms	39 ms	106 ms	2 ms	11 ms	23 ms
12	7 ms	37 ms	83 ms	1 ms	9 ms	49 ms
13	7 ms	62 ms	104 ms	2 ms	11 ms	22 ms
14	7 ms	37 ms	81 ms	1 ms	10 ms	51 ms
15	7 ms	40 ms	107 ms	2 ms	11 ms	22 ms
16	7 ms	38 ms	108 ms	1 ms	11 ms	50 ms
17	7 ms	65 ms	81 ms	2 ms	11 ms	23 ms
18	7 ms	38 ms	106 ms	1 ms	11 ms	48 ms
19	7 ms	39 ms	80 ms	2 ms	35 ms	22 ms
20	7 ms	37 ms	80 ms	1 ms	11 ms	40 ms
Average						
	9 ms	48 ms	96 ms	2 ms	15 ms	37 ms

Table A.1: Performance test data. n is model size

Appendix B

Weaving user guide

This appendix describes how to:

- create model fragments
- collect a set of model fragments in a weave model
- specify weave details
- evaluate and manually change the weave result

B.1 Creating model fragments

Model fragments are created as sub-parts of *Requirements*. In the Requirements-editor, a tab labelled *Model fragment* contains the fragment editor, as shown in Figures B.1 and B.2.

Model elements are created using the toolbar, which is highlighted in Figure B.2. This contains the various elements which are currently supported, categorised by diagram type. Note that the categories do not imply any restrictions on how elements can be mixed in a diagram.

Various properties regarding elements, such as name, can be viewed and edited in the *Properties* view, also highlighted in the lower right corner of the figure. The *Outline* view, highlighted in the upper right corner, gives an overview of the entire diagram, which is useful if the diagram can not be fitted on a single screen.

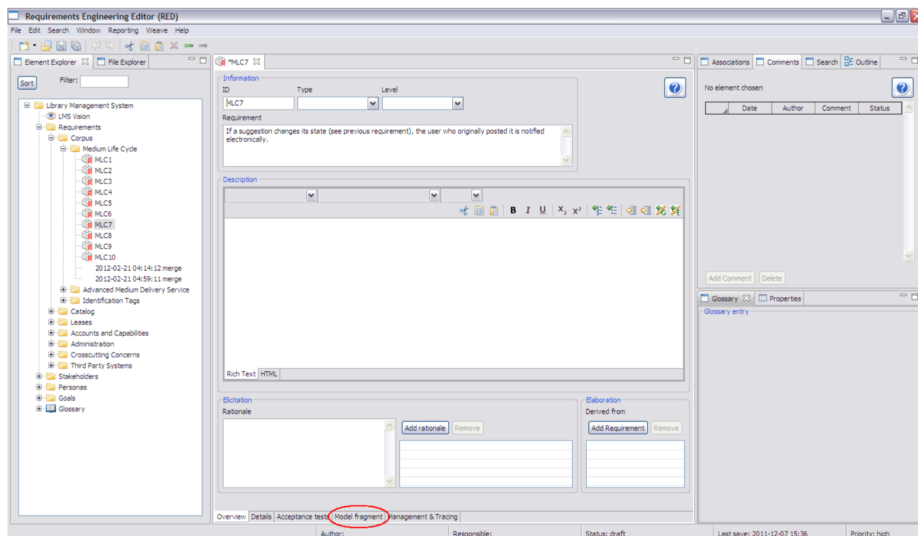


Figure B.1: Model fragment tab in Requirements editor

B.2 Creating a weave model

A weave model is created by specifying a set of model fragments (i.e requirements) in a dialog.

This dialog is accessed from the main menu, under the *Weave*-menu, which contains a *Create Weave Model* item. Selecting this item opens the dialog shown in Figure B.3. From here, requirements can be selected, and a weave model created by pressing *Finish*. The created weave model will automatically open in a new editor, and will also be accessible from the Element explorer view, under the name of *[date] merge*, where [date] represents date of creation.

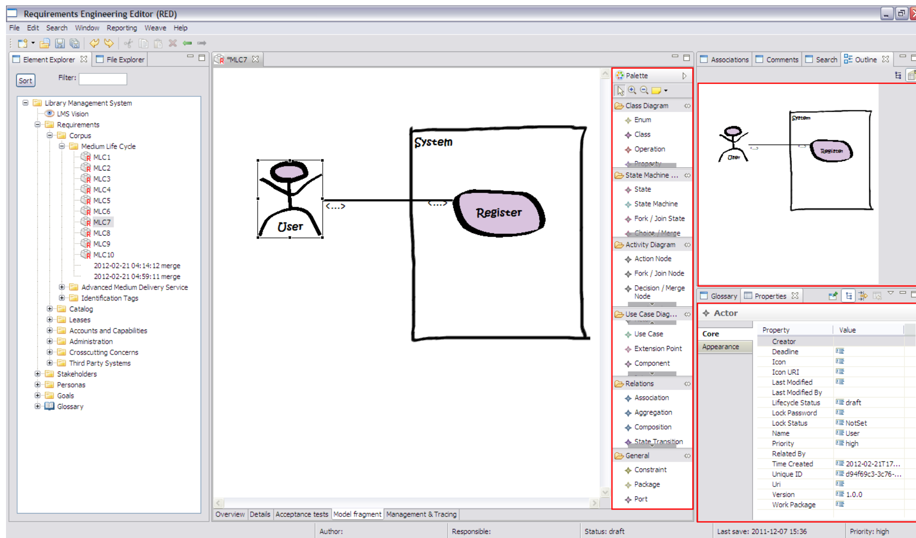


Figure B.2: Model fragment editor. Toolbar, outline and properties are highlighted

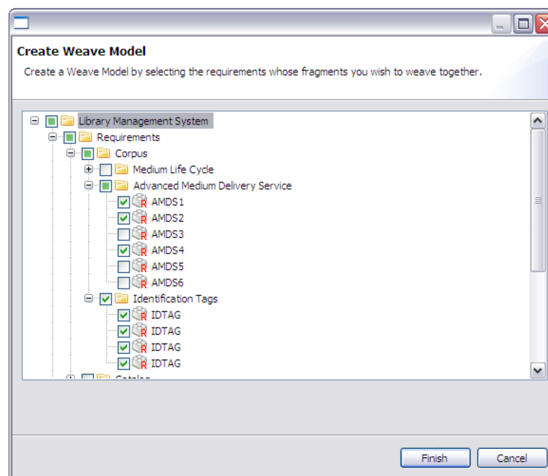


Figure B.3: Dialog for creating a weave model

B.3 Specifying weave details

The weave editor can be used to specify weave details, and to initiate the weave process.

Figure B.4 shows the weave editor, as it looks containing two model fragments, originating from requirements *MLC7* and *MLC8*. The two fragments are connected by a *weave connection*, labelled <<Weave>>, which is connected to a *weave annotation*. This weave annotation can be used to specify further weave details, such as specifying that two elements which would not normally match should be woven anyway.

The toolbar contains the following tools:

- *Weave Specification Connection*, which is the <<Weave>> connection between model fragments, used to specify that two fragments should be woven. This connection can be made between all model elements.
- *Weave Annotation*, which is the <<Weave annotation>> box connected to the weave specification connection. This contains details regarding the attached weave connection.
- *Weave Annotation Element* are added to the Weave Annotation, and contains the actual annotation details.
- *Weave Annotation Connection*, which is used to connect weave connections and annotations. This is the dotted line in Figure B.4.
- *Weave!* starts the weave process based on the contents of the weave editor.

B.4 Evaluate the weave result

After activating the *Weave!* tool in the weave editor, the weave process is executed, and the weave result editor opens. The weave result is also accessible from the Element Explorer view, under the name `[date] result`, where [date] is the date of creation.

The weave result editor contains 2 pages, in addition to the standard Management & Tracing page: *Prolog representation* (Figure B.5) and *Draft model* (Figure B.6).

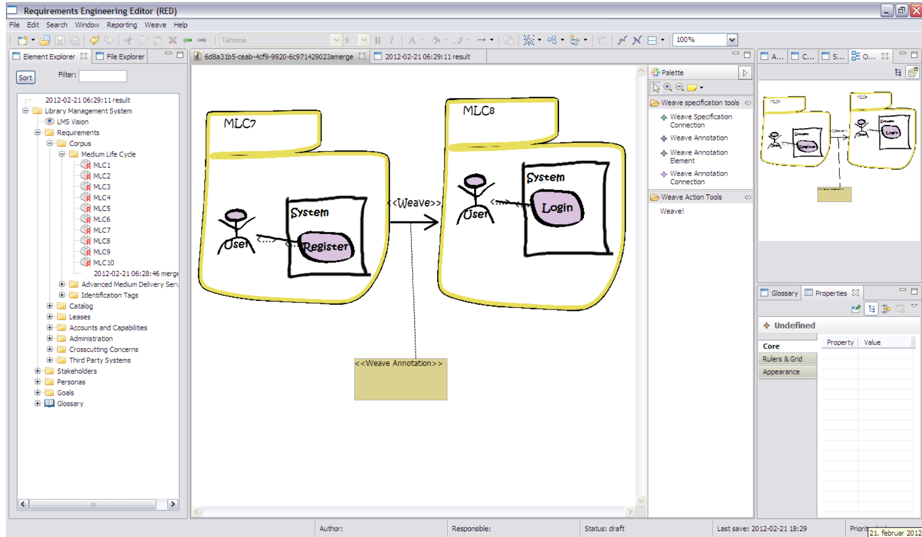


Figure B.4: Weave Editor

The Prolog representation page contains two text editors containing the Prolog code as it looks before and after weaving. The code can be edited by right-clicking the editor and selecting *Unlock Prolog Editing*.

Another way of altering the Prolog is to save it to a file using the button in the lower left, editing the file, and then pressing *Reload Prolog*, which reads the file into the editor. Options are available for replacing the unique ID numbers of model elements with simpler ID numbers, and for removing diagram layout information.

Note, that editing the Prolog representation is only recommended for expert users who know the way the models are represented in Prolog.

After editing the Prolog, in the top editor, the *Weave* button can be pressed to re-do the weave transformations. This will cause the contents of the lower editor to update with the new result, and the diagram on the *Draft model* page to be updated. If the Prolog in the bottom editor is edited, the *Refresh Model Fragment diagram* button recreates the diagram in the Model Fragment page.

The diagram in the *Draft model* page can be edited using the UML tools familiar from the model fragments editor, and can be exported as an image.

2012-02-24 08:19:33 result

Information

Below are two editors containing the weave model, and weave result model, represented in Prolog.
The weave model is the model from the weave editor, and contains the original model fragments, as well as any weave annotations.
The weave result model contains the model after the weave process, which is also shown in the "Model Fragment" page in this editor.

The models can be edited by right-clicking and selecting "Unlock Prolog Editing". Alternatively, the Prolog can be exported to an external file, from which it can be edited and then reloaded into the editor.

NB: Editing the Prolog code is only for expert users!

Weave model (pre-merge)

```
me(weaveAnnotation-'d7776c3b-7047-45c6-bc4c-660aecb66d73-weave', [from-id('0f039587-2513-42c3-a67d-95d759a6ed37-weave'),
to-id('f6554add-c0a6-40e5-bf59-2c860435248b-weave'), body-'default']).
me(weaveAnnotation-'b11b3acc-3bac-4cc6-b879-6d80883912ff-weave', [from-id('32622d2c-8c48-439d-8a86-696ee9e7c824-weave'),
to-id('0602b245-39a2-434b-b568-7c584fa62bae-weave'), body-'default']).
me(property-'1-weave', [name-'', type-id('32622d2c-8c48-439d-8a86-696ee9e7c824-weave'), association-id('02f76888-0cab-4a6a-b98c
-e1be622c0b3b-weave') ]).
me(property-'2-weave', [name-'', type-id('0638638f-ad3b-4ccc-9335-a177436ba389-weave'), association-id('02f76888-0cab-4a6a-b98c
-e1be622c0b3b-weave') ]).
me(association-'02f76888-0cab-4a6a-b98c-e1be622c0b3b-weave', [memberEnd-ids(['1-weave', '2-weave'])]).
me(class-'0638638f-ad3b-4ccc-9335-a177436ba389-weave', [name-'A', ownedMember-ids(['1-weave'])]).
me(class-'32622d2c-8c48-439d-8a86-696ee9e7c824-weave', [name-'B', ownedMember-ids(['2-weave'])]).
me(package-'0f039587-2513-42c3-a67d-95d759a6ed37-weave', [name-'RequirementA', ownedMember-ids(['0638638f-ad3b-4ccc-9335-
a177436ba389-weave', '32622d2c-8c48-439d-8a86-696ee9e7c824-weave', '02f76888-0cab-4a6a-b98c-e1be622c0b3b-weave'])]).
me(generalization-'9bf5fe61-bb74-4209-b66a-a2d6ad5e41cb-weave', [from-id('a832338e-cd5f-4e6b-83bb-e825ac5eafaf-weave'), to-id
('03384ab3-1898-40d4-9faf-7fdcddc9d7b2-weave')]).
me(class-'03384ab3-1898-40d4-9faf-7fdcddc9d7b2-weave', [name-'A', ownedMember-ids(['9bf5fe61-bb74-4209-b66a-a2d6ad5e41cb-
weave'])]).
me(class-'a832338e-cd5f-4e6b-83bb-e825ac5eafaf-weave', [name-'C', ownedMember-ids(['9bf5fe61-bb74-4209-b66a-a2d6ad5e41cb-
weave'])]).
me(class-'0602b245-39a2-434b-b568-7c584fa62bae-weave', [name-'H']).
```

Weave model

External file

Output 'simple' id-numbers | Suppress diagram-specific information

Save prolog to file | Change file location | Reload Prolog

Prolog saved to file: None

Weave Result Model (post-merge)

```
me(package-'f6554add-c0a6-40e5-bf59-2c860435248b-weave',

  [ ownedMember-ids(['02f76888-0cab-4a6a-b98c-e1be622c0b3b-weave', '03384ab3-1898-40d4-9faf-7fdcddc9d7b2-weave',
'0602b245-39a2-434b-b568-7c584fa62bae-weave', 'a832338e-cd5f-4e6b-83bb-e825ac5eafaf-weave']),
    name-'RequirementA'
  ]).
me(class-'a832338e-cd5f-4e6b-83bb-e825ac5eafaf-weave',

  [ name-'C',
    ownedMember-ids(['9bf5fe61-bb74-4209-b66a-a2d6ad5e41cb-weave'])
  ]).
me(association-'02f76888-0cab-4a6a-b98c-e1be622c0b3b-weave',
  [memberEnd-ids(['1-weave', '2-weave'])]).
me(property-'2-weave',

  [ name-'',
    type-id('03384ab3-1898-40d4-9faf-7fdcddc9d7b2-weave'),
    association-id('02f76888-0cab-4a6a-b98c-e1be622c0b3b-weave')
  ]).
me(property-'1-weave',
```

Refresh Model Fragment diagram

External file

Output 'simple' id-numbers | Suppress diagram-specific information

Save prolog to file | Change file location | Reload Prolog

Prolog saved to file: None

Prolog representation | Draft model | Management & Tracing

Figure B.5: Prolog representation page in weave result editor

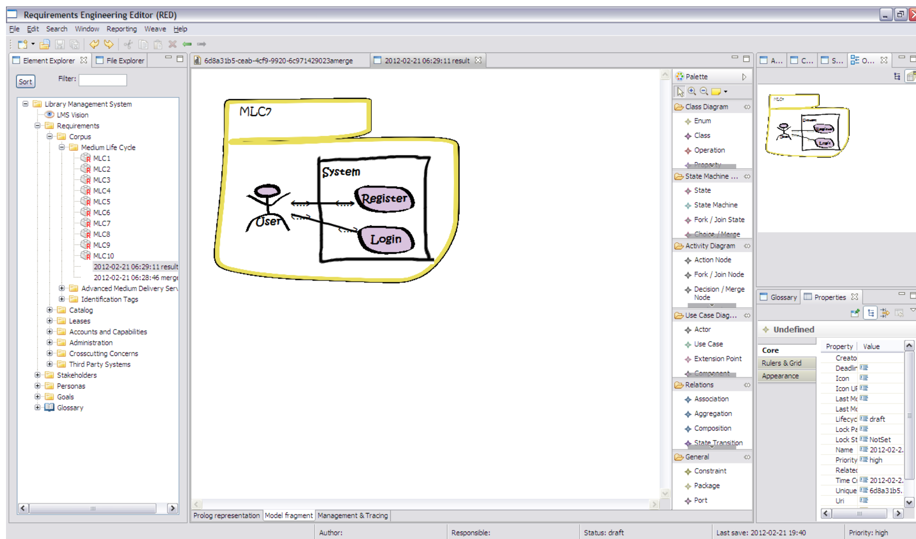


Figure B.6: Model fragment page in weave result editor

Bibliography

Amer Al-Rawas and Steve Easterbrook. Communication Problems in Requirements Engineering: A Field Study. *Proceedings of the First Westminster Conference on Professional Awareness in Software Engineering*, pages 47–60, 1996. URL http://www.cs.toronto.edu/~sme/papers/1996/NASA-IVV-96-002.pdf.

Grady Booch, Ivar Jacobson, and James Rumbaugh. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004. ISBN 0321245628.

Bill Buxton. *Sketching User Experiences: Getting the Design Right and the Right Design*. Morgan Kaufmann, 2007.

Brian Dobing and Jeffrey Parsons. Understanding the Role of Use Cases in UML: A Review and Research Agenda. *J. Database Manag.*, 11(4):28–36, 2000.

Brian Dobing and Jeffrey Parsons. Current Practices in the Use of UML. In *Perspectives in Conceptual Modeling*, volume 3770 of *Lecture Notes in Computer Science*, pages 2–11. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-29395-8. URL http://dx.doi.org/10.1007/11568346_2.

Marcos Didonet Del Fabro, Jean Bezivin, Frederic Jouault, Erwan Breton, and Guillaume Gueltas. AMW: a generic model weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005. URL http://www.sciences.univ-nantes.fr/lina/atl/www/papers/IDM_2005_weaver.pdf.

Kenneth D. Forbus, Ronald W. Ferguson, and Jeffery M. Usher. Towards a computational model of sketching. In *Proceedings of the 6th international*

conference on Intelligent user interfaces, IUI '01, pages 77–83, New York, NY, USA, 2001. ACM. ISBN 1-58113-325-1.

Anders Friis. Basic Tool Support for Requirements Engineering. Master's thesis, Technical University of Denmark, DTU Informatics, 2012.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94 –101, April 1994.

Deborah Hix and H. Rex Hartson. *Developing user interfaces: Ensuring usability through product and process*. Wiley, 1993.

Matthias Jarke. Requirements tracing. *Communications of ACM*, 41:32–36, December 1998. URL http://doi.acm.org/10.1145/290133.290145.

M. Esperanza Manso, José A. Cruz-Lemus, Marcela Genero, and Mario Piattini. Models in software engineering. chapter Empirical Validation of Measures for UML Class Diagrams: A Meta-Analysis Study, pages 303–313. Springer-Verlag, 2009. ISBN 978-3-642-01647-9. URL http://dx.doi.org/10.1007/978-3-642-01648-6_32.

OMG. OMG Unified Modeling Language (OMG UML) Infrastructure, August 2011.

Rita Petrolyte. FIT - An Online Inspection Support Tool. Master's thesis, Technical University of Denmark, DTU Informatics, 2011.

Keng Siau and Lihyunn Lee. Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML. *Requirements Engineering*, 9:229–237, 2004. ISSN 0947-3602. URL http://dx.doi.org/10.1007/s00766-004-0203-7.

Ian Sommerville and Pete Sawyer. *Requirements Engineering - A Good Practice Guide*. John Wiley & Sons, 1997.

Harald Störrle. A prolog-based approach to representing and querying software engineering models. In *VLL*, pages 71–83, 2007.

Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9:529–565, 2010. ISSN 1619-1366. URL http://dx.doi.org/10.1007/s10270-009-0145-0.