(Towards an) Implementation of a Graphical Editor for Diagrammatic Predicate Logic in the Eclipse Platform

Stian Skjerveggen

Department of Informatics
University of Bergen

June 2008

Contents

List of Figures

Preface

Foreword

This is my master thesis in the Program Development Master's Programme at the Department of Informatics, University of Bergen and the Bergen University College.

Software modeling for me has always been about the UML and class diagrams. But I've often found that the theory and practice are not the same, and my models usually gets thrown away or replaced through endless refactorings. When I first took on this master thesis, my expectations were that I might get to learn more about modeling software systems. What I have learned is that class diagrams are just the tip of the iceberg. People use diagrams in many different ways; as a quick sketch to show some aspect of a system or as a blueprint to derive code from.

I am very grateful of getting the opportunity to work with the Eclipse Platform. Having used it exclusively as a Java editor in the past, this project has shown me that editing Java code is not at all the only thing Eclipse is capable of. I hope to bring some of the things that I have learned while working with Eclipse with me and put them to use in the future.

Acknowledgements

First of all I would like to thank Stine for her patience and understanding while I have been working on this thesis and couldn't give her the attention she deserves. I wish to thank all those who have contributed to this thesis, I would especially like to thank Uwe Wolter, Adrian Rutle and Yngve Lamo for all their help and valuable comments. Without them, I could not have completed this thesis.

Stian Skjerveggen,
Bergen, June 2008.

Chapter 1

Introduction

This master thesis is mainly about a new approach to software modeling. It is a part of an ongoing project at the Bergen University College and the University of Bergen started in early 2006, called *Generic Diagrammatic Software Sketches*[1]. The project aims at investigating the theoretical and practical aspects of using *Diagrammatic Predicate Logic* (DPL)[2] [30] in specification of software.

1.1 Motivation

The diversity and heterogeneity of modeling languages make the needs for model integration and model transformation mechanisms more relevant than ever. Especially in *Model-Driven Engineering* (MDE) where the primary software artifacts are graphical models of the system. These models can be considered blueprints and most of the software is derived, either by hand-coding or code-generation, from these blueprints.

Most of the graphical notations that are in use today does not have proper semantics. The problem with this is that it can lead to ambiguous constructions and semantic relativism, so the need for formal specification methods have become a vital issue.

Diagrammatic Predicate Logic (DPL) is a specification formalism that is able to define diagrammatic modeling languages with a strong mathematical foundation. It is a graph-based specification format which takes ideas from both *Category Theory* and *First-Order Logic* and adapts them to software engineering needs.

1.2 A short problem description

The primary goal of this thesis is to explore how the *Eclipse Platform* can be beneficial for the Generic Diagrammatic Software Specification project, and

[1] `http://gs.hib.no`
[2] The DPL framework is called *Generalized Sketches* in earlier publications

how a graphical editor in the Eclipse Platform can be put together. The Eclipse Platform is designed for building integrated development environments, with all its functionality based on various *plug-ins*. The objective is to explore the plug-ins that are made especially for modeling software and find out how to utilize them in the DPL framework.

1.3 Structure of thesis

The structure of this thesis is based on a template provided by Carsten Helgesen of the Bergen University College. While some things didn't fit in the template, the structure has been altered slightly. **Chapter 2** provides a background for the thesis and explains the context the project is to be used in. **Chapter 3** discusses the existing solutions and provides a general description of the foundation of this thesis, while **Chapter 4** shows a general overview of the Eclipse Platform and explains why it was chosen as a base platform for the thesis. **Chapter 5** discusses some approaches on how to solve the problems from the project description **Chapter 6** evaluates how the Eclipse Platform performs according to the requirements and provides some thoughts on further steps.

Chapter 2

Background

2.1 Software Engineering

The term "software engineering" was first coined at the first NATO Software Engineering Conference in 1968. The conference was held to discuss what was then called the "software crisis" [28]. The term was used to describe the impact of the increase in computing power and the increasing complexity of the problems that could be solved. What was previously unrealisable computer applications could now be implemented. But the immaturity of the software engineering profession resulted in projects that were running over time, low quality software and unmanageable projects with code that was difficult to maintain. The increase in computer power had led to a decrease in software quality. In Edsger Dijkstra's 1972 ACM Turing Award lecture, "The Humble Programmer" [6], Dijkstra stated that:

> [The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.
>
> – Edsger Dijkstra, *The Humble Programmer*

In the 1940s all machine code was written by hand. Some early tools such as macro assemblers and interpreters were developed in the 1950s, and in the 1960s optimizing code compilers were developed, helping productivity and quality. It was first then that the really big programmer projects could be realized. The resulting software were orders of magnitude larger and more complex than previous software systems. The early experience in building these systems showed that informal software development was not good enough. Projects was delayed by large amounts, budgets were exceeded and the produced software was difficult to maintain and performed poorly.

To solve this crisis, several software engineering processes and methodologies were created with various degrees of success. There is no 'ideal' approach to software engineering because of the wide diversity of software systems, this means that there is a need for a diversity of approaches to software development.

In the 1970s the focus shifted to collaboration. Several collaborative software tools were developed, such as Unix, code repositories and so on. By the 1980s, personal computers and personal workstations had become common, and the rise of consumer software began. The first commercial object oriented programming language, Smalltalk, was released in 1980 and introduced the term *object oriented programming*. Smalltalk was influenced by *Simula*, a language developed in Norway which was the first introducing the techniques we now know as object oriented programming.

2.2 Object Oriented Programming

With the introduction of *objects* which define their own data and behavior, one could begin to use a higher level of abstraction in the discussion of software system design. The concept of Object Oriented Programming (OOP) led to a variety of new object-oriented programming languages, database system as well as software modeling approaches. OOP remained the dominant programming methodology, largely because of *C++* and the popularity of graphical user interfaces, which is well suited for OOP. Instead of thinking in terms of processes and how a computer would execute the processes, the focus was shifted to objects and their behavior. Object orientation could be described as a black box technology were the implementation is abstracted away from the user.

Mainly due to the introduction of OOP, the level of abstraction in software specifications has become higher. Raising the level of abstraction has been a continuous goal of computer scientists since the beginning of computer science. A widely used technique is to document system specifications and design using a set of models. These models are graphical representations that describe business models. Because they are a graphical representation it is easier to understand them for software engineers than by using a detailed description in a natural language. In software engineering, the software specifications has to be both descriptive and comprehensible to be able to communicate its needs to the different participants in a software project.

2.3 Graphical Modeling Languages

Graphical modeling languages have been around for a long time. In the 1970s, Peter Chen's Entity-Relationship Diagrams (ERD) [4] became very popular in data modeling. The "standard" for modeling software today is the Unified Modeling Language (UML). It is a family of graphical notations that help in describing and designing software systems, and in particular, software systems that are built using object oriented programming. In the late 1980s and early 1990s there were many different types of graphical notations around. The three "founders" of the UML; Grady Booch, James Rumbaugh and Ivar Jacobson all had their own object oriented modeling approaches and worked together to unify them. The UML is a fairly open standard which is controlled by the Object Management Group (OMG), which is an open consortium of companies.

The fundamental reason as to why one uses a graphical modeling language in describing the design of a software system is because a programming language is not at an appropriate level of abstraction. Despite the usefulness of a graphical notation describing the system design, there are many disputes in the software engineering world about their role. One perspective is that the only important thing is working code, not beautiful models. As Jack Reeves put it, "*The code is the design*" [15]; pointing out that the only thing that truly is in sync with the code is the code itself. Other software engineering approaches, such as *Agile Software Development* [19] and in particular *Extreme Programming* [2], use graphical modeling as a *sketch*. With this usage, developers use a graphical model to help communicate some part or aspect of the system, not the whole system up-front. This is more because of the agile software's nature of being able to respond to changing requirements and of responding to the changes rather than following a plan.

Martin Fowler mentions in his book, *UML Distilled* [15], different ways of using UML. The most common one is using UML as a quick sketch to communicate some aspect of the software, as it is done in Agile Development. Another usage is UML as a blueprint. This is used more in the Big Design Up Front (BDUF) methodologies; often associated with the *Waterfall Model* [28] of software development. Here all of the system's design should be completed and perfected before the implementation begins and because of this more sophisticated tools are needed to handle all the details that are required.

2.4 Model-Driven Engineering

With increasingly sophisticated software engineering tools it is possible to start with Model-Driven Engineering (MDE). This is a recent concept, where the model is the primary engineering artefact. Various degrees of model integrations are used, either by hand-writing the code using the model as a blueprint, or by generating the code from the models using code generators. As MDE continues its evolution it adds greater focus on the architecture of software systems and automation in software development, and with it comes a higher level of abstraction in software engineering. This abstraction allows for simpler models with a bigger focus on the problem space, while earlier methods were more process-oriented.

2.4.1 Model-Driven Architecture

The best known approach to MDE is the Model-Driven Architecture (MDA) from Object Management Group, launched in 2001. MDA provides a set of guidelines for structuring software specifications expressed as models. The MDA is focused on *forward engineering*; which is to produce code from abstract human elaborated specifications. One of the aims of MDA is to separate design from architecture. This is done by dividing the development work into two areas.

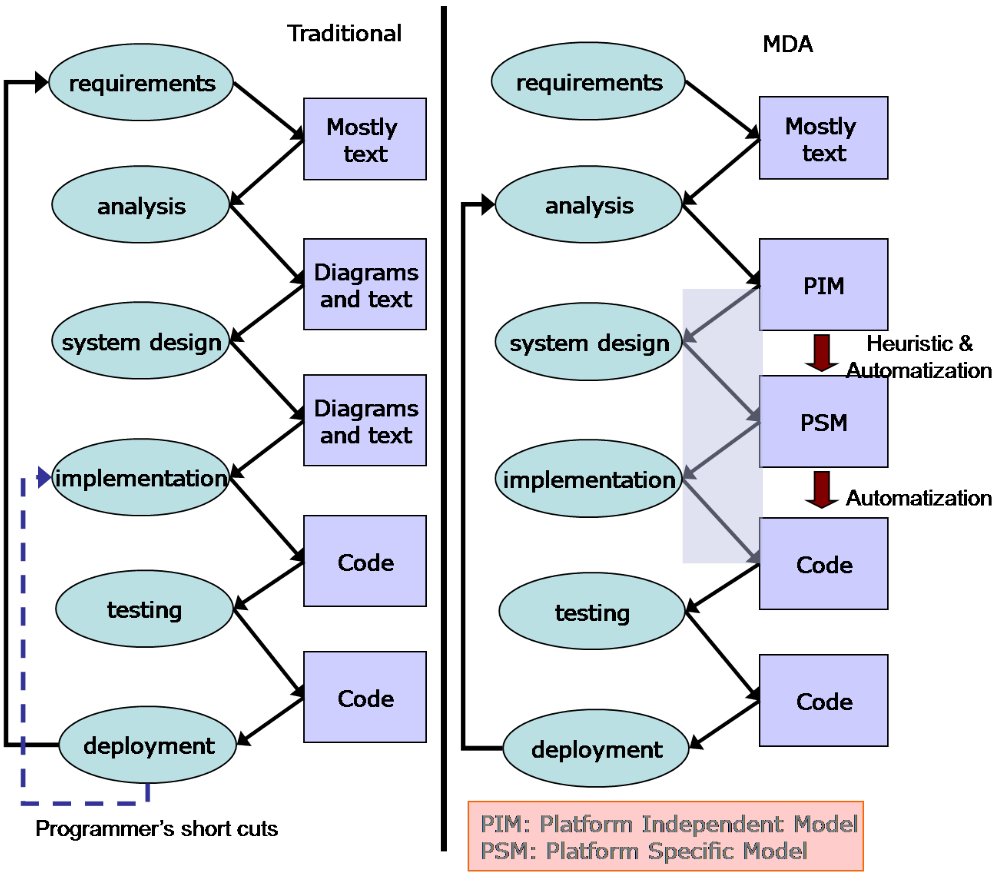


Figure 2.1: Comparison between standard software engineering steps and MDA

An application is represented by creating a *Platform Independent Model* (PIM) which is a UML model that is independent of any particular technology or platform. Tools will then translate the PIM into a *Platform Specific Model* (PSM). A comparison between a traditional approach and the MDA approach is seen in figure 2.1.

In traditional software design one would go on to create the system specific or language specific diagrams after the system design phase, and from there on implement the system using the created diagrams. The result would be some code which then will undergo some sort of testing before an eventual deployment of the software. To shorten the process instead of going back to re-evaluate the requirements and analysis to create updated diagrams, programmers will often go directly back to the implementation phase and do changes there; thereby leaving the diagrams not in sync with the code. In the MDA approach the platform (system and/or programming language) specific diagrams will be automatically created, and the code will be generated from these diagrams. The good thing about this is that the model will always be in sync with the code, since the code is derived directly from the model. In Figure 2.1 an comparison of the system development process in traditional development and MDA is shown.

There are some concerns about MDA however. The MDA approach is set by a variety of technical standards, some of them yet to be defined or yet to be implemented in a standard manner. Take for example QVT (Queries/Views/-Transformations) which is the OMG standard for model transformation which there exist no full implementation of, although there are several projects which are partially QVT-compliant.

As mentioned earlier, in MDA the application platform and the implementing technology are chosen independently of the input models (the PIM). This provides flexibility and survives changes made in the realization technologies and software architectures. In addition, the domain model can be modified (and regenerated) as a response to changes in requirements independently of the application platform. This transformation between PIM and PSM are specified by a transformation definition language and executed by transformation tools. In order to get inter-operability and models that also work with other MDA-compliant environments, the models as well as the transformations between them are required to be defined formally. This means that there is a need for techniques that can be used to specify formal models and formal model transformations. Many tool vendors claim that they are fully MDA compliant, while in truth, the PIM to PSM transformations are not 100% automated.

2.5 The problem with modeling languages

The problem with UML is that it is a general-purpose language. It is not intended to be a complete development method [25]. It is intended to support all, or at least most, of the existing development processes. But because it is meant as a general-purpose modeling language the standard is so complex that the standard is often open for multiple interpretations [15] also known as *semantic relativism.* Criticism against UML claims that UML is bloated and overly large and complex because, as it is a general-purpose language, it contains many diagrams and constructs that are either redundant or rarely used. Because of its complexity it makes it harder to learn and adopt, which could be why the most frequent use of UML is as a sketch where precision and details is not as important.

But in the field of Model-Driven Engineering, having precise formal semantics for the diagrammatical notation is required [30]. This means that a good modeling should be *graph based*, *formalized* and *expressive* enough to capture the aspects of the universe it is modeling. The vast majority of the formal semantics for diagrammatic languages that were built according to MDE's needs employed a First-Order or similar logic systems based on string-based formulas [30]. This makes writing models complicated and prone to errors, and as mentioned earlier; it is easier to communicate a graphical model rather than a detailed natural language description. Another point made is that since software models are graph-based, modeling languages that use a string-based logic, such as UML with OCL expressions, rather than a graph-based logic may fail to express all the properties and aspects of the software system in an intuitive way [26]. A graphical modeling language, or a diagrammatical model is thus better than a string-based model. As it is graph-based it makes the relation between the syntax and the semantics of the model more compact and easier for the domain experts to understand.

2.6 Diagrammatic Predicate Logic

Diagrammatic Predicate Logic(DPL) [30, 31] is a generic formalism. It is a specification formalism to define diagrammatic modeling languages with a mathematical foundation. DPL is a graph-based specification format that borrows its ideas from both categorical and first-order logic and adapts them to the needs of software engineering.

2.6.1 Category Theory

In mathematics, category theory deals in an abstract way with mathematical structures and relationships between them. Instead of focusing on the individual objects (or groups) possessing a given structure, Category Theory emphasizes the *morphisms* (relations) between the objects. Category theory characterizes objects in terms of their “social life” [9]. A *morphism* is an abstraction derived from structure-preserving relations between two mathematical structures. The most typical application of category theory in the area of computer science is in algebraic development techniques, but it can also be put to use in concurrent and object-oriented systems, software architecture and service-oriented software development [9].

2.6.2 First Order Logic

First-order logic (FOL) is a formal deductive system used in mathematics, philosophy, linguistics, and computer science. It goes by many names, including: first-order predicate calculus (FOPC), the lower predicate calculus, the language of first-order logic or predicate logic. Unlike natural languages such as English, FOL uses a wholly unambiguous formal language interpreted by mathematical structures. FOL is a system of deduction extending propositional logic by allowing quantification over individuals of a given domain of discourse. FOL can be applied in computer science in the areas of language semantics, formal specifications, model checking and logic programming among others.

2.6.3 Modeling and the DPL Framework

Diagram specifications (DS) in DPL are categorical structures which consists of a graph in which some diagrams are marked with predicates from a predefined signature. A diagram is a visual representation of a part of the graph, while a signature, denoted by Σ, is a collection of diagrammatic predicate symbols. To make the DPL formalism suitable for software engineering it is a generalization and adaption of the categorical sketch formalism where signatures are restricted to a limited set of predicates: limit, colimit and commutative diagrams [27]. Signatures and diagram specifications are the concepts in the DPL framework which are used to represent modeling languages and models. The definitions of these concepts are as follows (The definitions are taken from [26]):

Definition 1 *A diagrammatic predicate signature* $\Sigma := (\Pi, ar)$ *is an abstract structure consisting of a collection of predicate symbols* Π *with a mapping that assigns an arity (graph)* $ar(p)$ *to each predicate symbol* $p \in \Pi$*, i.e.* $ar : \Pi \rightarrow Graph$*.*

Definition 2 *A diagram* (p, δ) *labeled with the predicate* p *in a graph* $G(S)$ *is a graph homomorphism* $\delta : ar(p) \rightarrow G(S)$*, where* $ar(p)$ *is the arity of* p*.*

Definition 3 *A* Σ*-specification* $S := (G(S), S(\Pi))$*, is a graph* $G(S)$ *with a set* $S(\Pi)$ *of diagrams in* $G(S)$ *labeled with predicates from the signature* Σ*.*

2.6.4 A DPL Example

The following example is adopted from [26]. The Σ_{UML}-specification in Figure 2.2 specifies the class diagram of a simplified software system of persons, companies and employment. Every person must work for zero or one company, but a company must hire one or more persons. The first constraint is set by a predicate *[singlevalued]* on the arrow *worksFor* while other constraint is set by the predicate *[total]* on the arrow *hires*. For more information about the example, see [26].

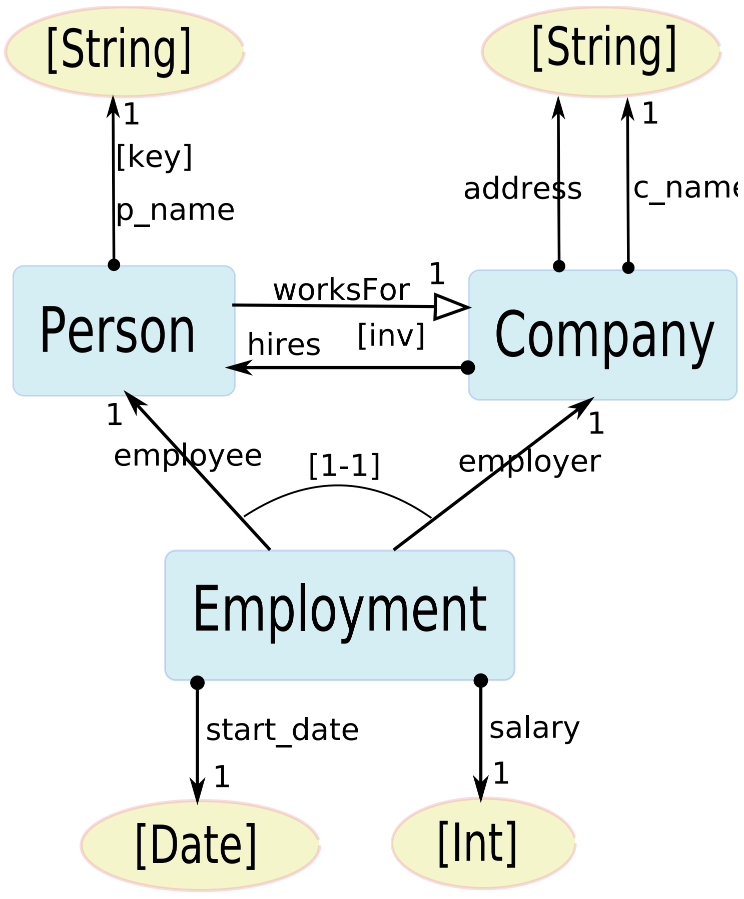


Figure 2.2: A Σ specification

Chapter 3

Description of problem

Note: In this chapter the term "generalized sketches" is used frequently. As mentioned earlier, the name was later changed to Diagrammatic Predicate Logic as the "sketch" concept can be misleading. This chapter is however about previous application, so the original term "generalized sketches" will be used.

3.1 Today's situation and existing solutions

There exists two solutions for creating models in DPL. But unfortunately, neither of them are complete. The first version, *Sketcher95*, was created in 1995 by a group in Latvia to draw generalized sketches. It was originally a 16-bit application, but was later rewritten to support 32-bit. Unfortunately, the company that the group worked for stopped the project. Which resulted in an application that has a few bugs, and not suitable for practical work. The source code for the program has also regrettably been lost, but an executable version still exists, which is very valuable for this project.

Another project attempted to recreate Sketcher95 in the .Net platform for Windows. It is described in Ørjan Hatlands master thesis [24]. Unfortunately not all of the functionality from Sketcher95 was implemented, and the features missing were left for further development.

3.1.1 Sketcher95

The first version of Sketcher95 was created in 1995 by a group from Latvia. It was originally created as a 16-bit application for Microsoft Windows. A year later, probably due to the popularity of Microsoft Windows 95 and the 32-bit platform, a pure technical job of rewriting the product to a 32-bit version was done. Due to some problems with the funding of the project, the company the programmers worked for stopped it. The lead programmer of the project relocated to USA shortly after. Unfortunately, the source code from the project was not conserved after this and the whereabouts of it is not known. Fortunately

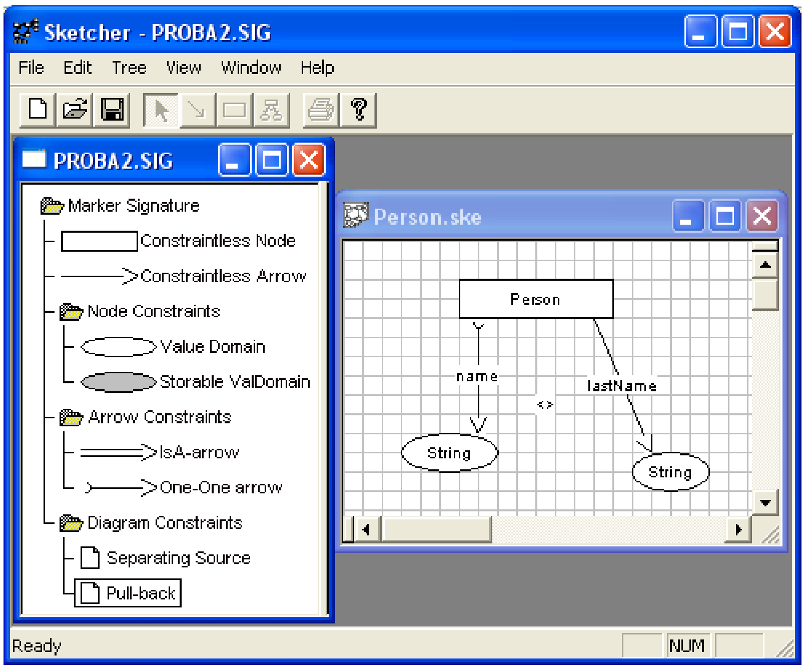


Figure 3.1: Sketcher95 in action. To the left is the *marker signature*, the right shows a *signature sketch* based on the marker signature.

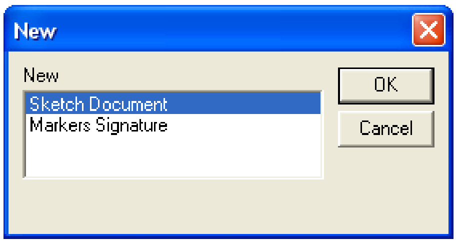
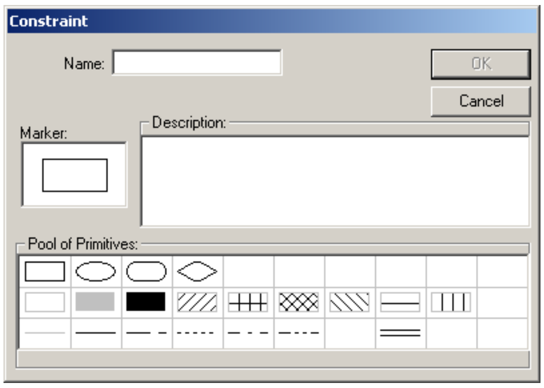


Figure 3.2: The "Create new document" dialog from Sketcher95.

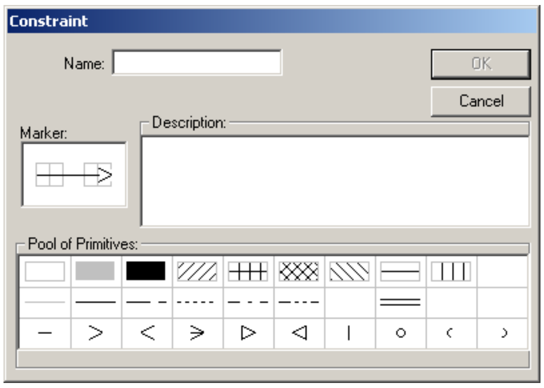
an executable version of the application is still available, but the source code and documentation is as mentioned missing.

Sketcher95 is a 32-bit Microsoft Foundation Classes application, implemented as a Multiple-Document Interface application. This means that it has a single parent window that serves as a container for several other child windows. A screenshot of Sketcher95 is shown in Figure 3.1. The main window in Sketcher95 has a menubar and a toolbar containing shortcuts to the most common used features offered by the menubar and a statusbar at the bottom. The child windows in the application provides the user interface to the two main parts of the generalized sketches formalism; signatures and sketches.

When a user wants to create a new document two choices are presented; creating a *Marker Signature* or creating a *Sketch Document* (figure 3.2). A sketch document depends on a marker signature, as the generalized sketches formalism is that every sketch is based upon a specific signature [30, 31]. When creating



(a) Node Constraint



(b) Arrow Constraint

Figure 3.3: Node and Arrow Constraints

a marker signature, the user is presented with a marker signature window (as seen to the left in figure 3.1).

The marker signature starts out empty except for a *Constraintless Node* and a *Constraintless Arrow*, which are always available. The user has the choice of creating three kinds of *constraints*. Constraints on a *node* (Figure 3.3(a)), constraints on an *arrow* (Figure 3.3(b)), or constraints on a *diagram* (Figure 3.4. Each constraint has a *marker* (hence the wording "marker signature"). A *node constraint* consists of three elements; a name, a description and a visual notation. The visual notation shows how the node will be presented; a square, square with rounded edges, a circle or other shapes. An *arrow constraint* has the same elements, a name, description and visualization, but the visualization is divided in three parts. This is because an arrow has three parts, a tail, a body and a head.

In the DPL framework a signature (Σ) is defined as a collection of diagrammatic predicate symbols (See section 2.6.3). These predicate symbols, or markers as they are called in Sketcher95, are to be interpreted as constraints imposed on the diagrams that are labeled by these markers. The predicates are divided in three parts, predicates on nodes, predicates on arrows and predicates

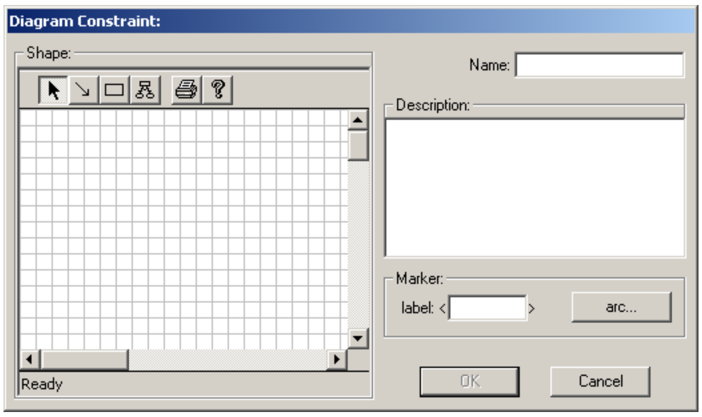


Figure 3.4: Diagram Constraint

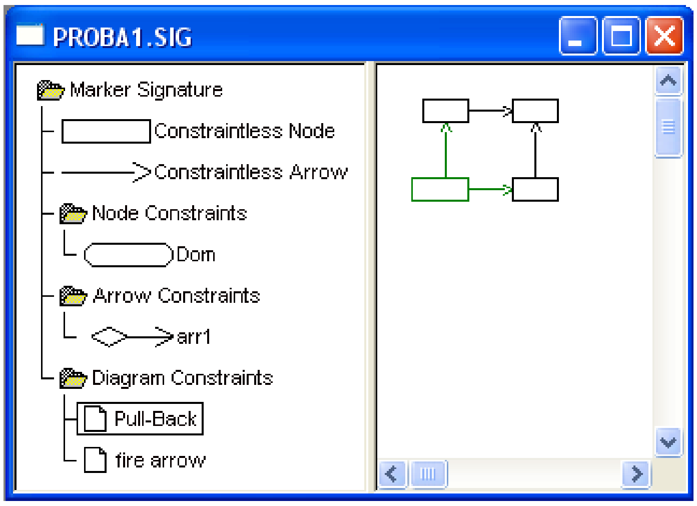


Figure 3.5: Sample signature in Sketcher95

on diagrams, i.e. collections of nodes and arrows.

The marker signature thus holds a collection of node constraints, arrow constraints and diagram constraints. In Sketcher95 all the marker collections are associated with folders, and it is possible to further organize them by adding sub-folders. Each marker is associated with an icon that depicts the visual notation of the constraint while the more complex diagram constraints have a simple icon, giving that the constraint may be too complex to be represented by an icon. A sample marker signature is shown in figure 3.5. In the figure a diagram constraint is selected.

A diagram constraint is, as explained, a collection of nodes and arrows. When a user is adding a diagram constraint to a sketch document a new dialog box comes up and the user must select which arrows in the sketch document corresponds with the arrows in the diagram constraint. To complete the process all the arrows must have the same direction and the source and target nodes must

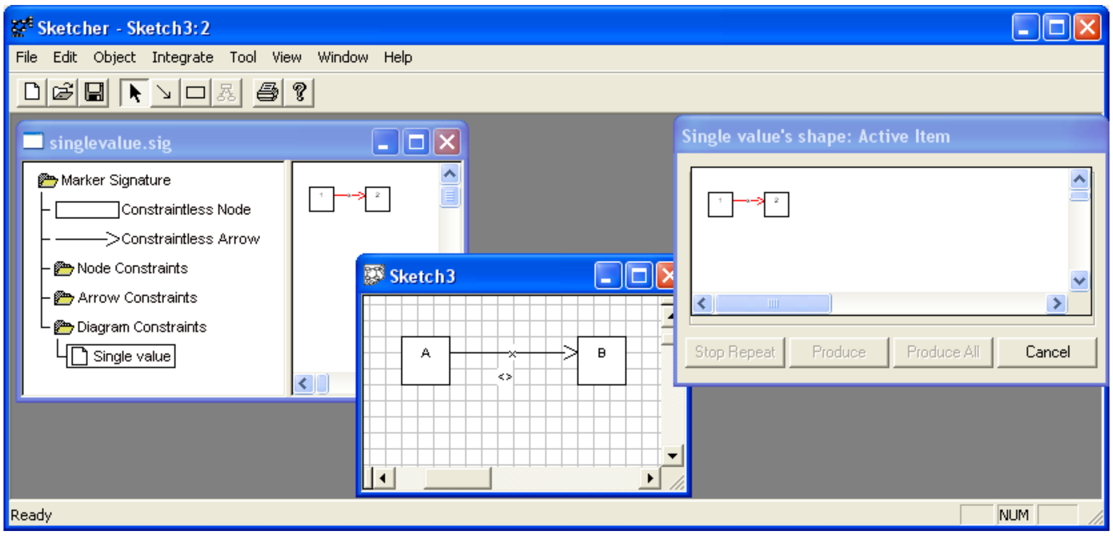


Figure 3.6: Adding a diagram constraint in Sketcher95

have the same constraints as they do in the corresponding diagram constraint. As an example, consider a simple predicate denoted by the name [single-valued]. The *arity* of the predicate denotes the general *shape* of the predicate. To add the diagram constraint to a sketch document, the user first have to choose which constraint to add. The next step is to choose the arrows in the sketch document that corresponds to the arrows in the diagram constraint as depicted in figure 3.6. The user then selects an arrow in the sketch document. If the diagram constraint contains several arrows, the process is repeated until all the arrows in the diagram constraint has a corresponding arrow in the sketch document. The diagram constraint can also include options to produce missing elements so it is possible to for example produce a connection between two nodes if there are no previous arrows between them in the sketch document.

The problem with Sketcher95, as previously mentioned, is that it remains unfinished and has some application errors. Most of the functionality is implemented, but because of the errors it is not a distributable program. There are also minor problems with the way things are visualized on the screen, some arrows are painted wrong or not in the right place and there are unpredictable application failures. There is also a lack of common GUI functionality such as copy/paste, undo/redo, zooming abilities and some tool shortcomings. A screen capture of Sketcher95 in Figure 3.7 shows the example from section 2.6.4.

3.1.2 Sketcher.Net

Sketcher.Net was developed by Ørjan Hatland in 2006 as part of his master thesis [24] at University of Bergen. The application was written in C# using the Microsoft.Net platform. Sketcher.Net uses the *Sketcher95* as a basis and takes the same approach as it when it comes to the separation of signatures and sketches. The user interface was updated to use a *Tabbed Document Interface*

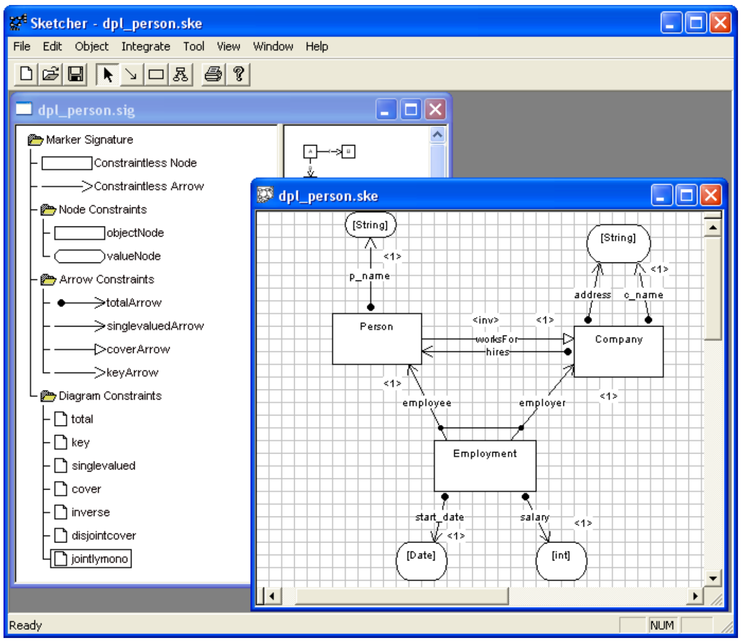


Figure 3.7: The example Σ-specification from Figure 2.2 in section 2.6.4 created in Sketcher95

to modernize the application from the traditional Multiple Document Interface. The result can be seen in Figure 3.8. The project remained as a basis for further work as all the GUI elements unfortunately did not get implemented. The key parts that are missing is mainly to complete the framework of basic operations for the diagrammatic predicate logic formalism and also a few graphical user interfaces for creating arrow and diagram constraints and other functionality.

As Sketcher.Net is written in C# and the code is available, it makes it a more viable project to continue on than Sketcher95. But eventually the Eclipse Platform was chosen as a base for this thesis. The reasons as to why will be discussed in greater detail later, but the short answer is that the features and functionality of the Eclipse Platform, coupled with many related modeling projects that exist in the Eclipse community was too great to ignore.

3.2 The Project: Background and General Idea

The current situation is that there exist no usable tool to create diagrammatic specifications based on the Diagrammatic Predicate Logic framework. The existing applications lack the functionality to be considered as more than prototypes and have been discontinued.

In the Java community the Eclipse Platform has become very popular in the later years. Also, the frameworks, subprojects and components of the Eclipse Platform supports most of the features that Model-Driven Engineering requires. In fact, many implementations of MDA specifications have been implemented as plug-ins to the Eclipse Platform.

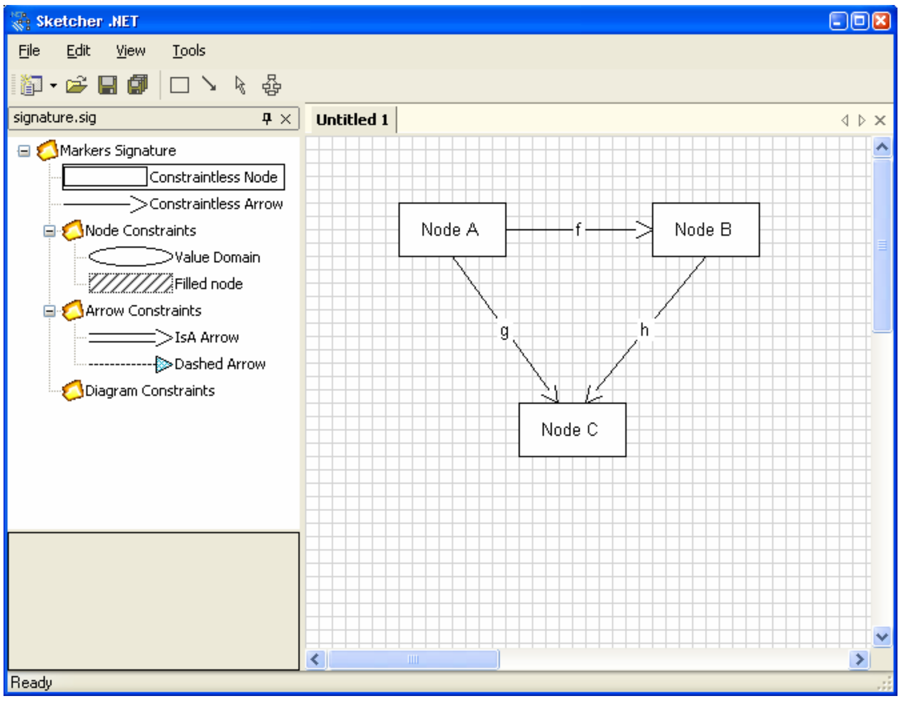


Figure 3.8: Sketcher.NET in action

There is also the fact that the Eclipse Platform provides many features for graphical editors as well and thus providing a standard look that can be used as a base for many different types of graphical editors. The objective of this thesis is to explore and discuss how the Eclipse Platform can be used or adapted for use in the Diagrammatic Predicate Logic Project and in Model-Driven Engineering in general. The foundation of the application is based on previous work such as Sketcher95 and Sketcher.NET, and development will be done by using their approaches to DPL. It should also, at some point in the future, support transformations between different graphical modeling specifications. Secondary tasks include the implementations of visual guides (Wizards) that guide the signature designer through the design phase.

The Diagrammatic Predicate Logic formalism is a big topic by itself and under permanent development, and an understanding of the formalism is needed to develop tools for it. In the area of graphical editing, there a are a few things that are common to any graphical editor; an underlying semantic model, a visualization of the model and serialization.

What makes this project so interesting is that it is a whole new approach to software specification and design. The new trend of Model-Driven Engineering makes this project very relevant. Finally, there is a lot of focus on the Eclipse Platform today, and learning more about it and how to manipulate and extend it is very appealing.

Chapter 4

Problem Analysis

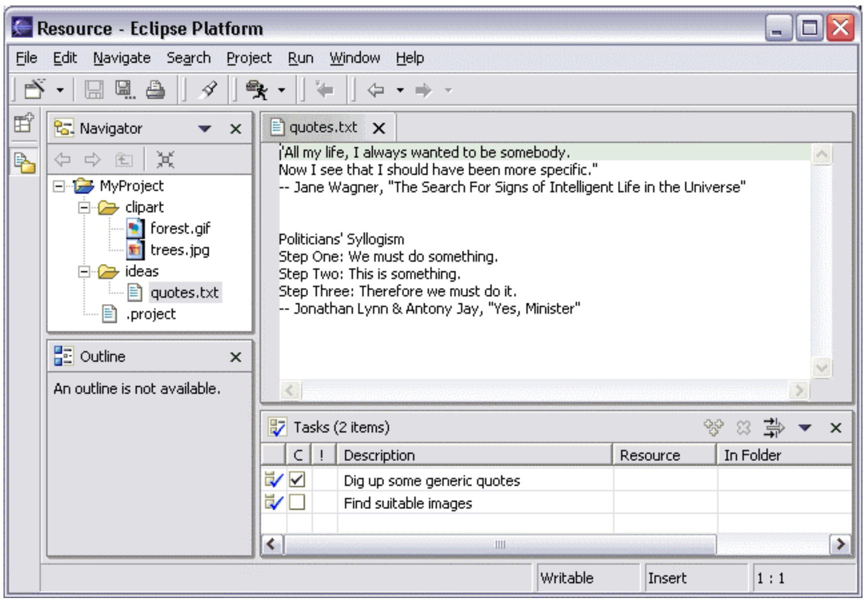
The basic problem at hand is to build a graphical editor for diagrammatic predicate logic in the eclipse platform. This can be divided in two different problems, with one being how to build a graphical editor in the eclipse platform and the other one is how to build a graphical editor for diagrammatic predicate logic (DPL). DPL is a generic formalism suited to define diagrammatic modeling languages with a strong mathematical foundation. But to produce any graphical editor for it, the internal structure of DPL must be obtained. To program something within the Eclipse Platform there is of course a requirement to understand how Eclipse works. Eclipse consists of many different *plug-ins* and it is important to determine the ones that can be beneficial for this project. A short list of the major parts in this process looks like the following:

- Obtaining the internal structure of the diagrammatic predicate logic formalism
- Implementing a generic graphical editor in the Eclipse Platform
- Customizing said editor to the needs of DPL

Getting the internal structure of the DPL formalism is mainly a theoretical task, consisting of browsing through various documentation and papers on DPL. As the project is about building a graphical editor that uses the DPL formalism, it is important to create a model of the DPL structure. This model will act as a metamodel for the graphical editor, and models built by the editor conforms to this metamodel.

The second task is to implement a graphical editor in the Eclipse Platform. This means that various aspects of a graphical editor must be covered. These include the following:

- A drawing surface the user interacts with
- A set of tools that defines the operations a user can perform, especially drawing tools



An outline is not available.

```
"All my life, I always wanted to be somebody.
Now I see that I should have been more specific."
-- Jane Wagner, "The Search For Signs of Intelligent Life in the Universe"


Politicians' Syllogism
Step One: We must do something.
Step Two: This is something.
Step Three: Therefore we must do it.
-- Jonathan Lynn & Antony Jay, "Yes, Minister"
```

	C	!	Description	Resource	In Folder
✓	✓		Dig up some generic quotes		
✓	☐		Find suitable images		

Figure 4.1: The Eclipse Platform User Interface

- Managing the connection between the visual model and the semantic model
- Storing the data generated by the graphical editor for future use and maybe collaboration with other applications

The third task is to extend the graphical editor with the features of the DPL formalism. This task is made easier by the fact that there exists a prototype of how an editor for DPL should work, Sketcher95 (See section 3.1.1).

4.1 Available Technology

As mentioned earlier, the Eclipse Platform was chosen as the basis of this project. This means that all the available technology comes from the Eclipse Project itself. Luckily the list of subprojects within the Eclipse Project is quite substantial and there are plenty of related projects to utilize.

4.2 The Eclipse Platform

To most people, Eclipse is a Java editor. While Eclipse is an excellent (and free) Java IDE, it is also a platform. The Eclipse Platform is designed for building integrated development environments (IDEs); it is an "IDE for everything and nothing in particular" [21]. This means that Eclipse can be used for practically anything, not just as a Java development environment. Figure 4.1 shows a screen capture of the Eclipse workbench. The workbench is made up of different *views* and an *editor*. Eclipse also has *perspectives* which allows for different arrangement and selection of different views and editors visible on the screen. The views provide information about some object the user is working on, like providing an outline of the document a user is editing via the *outline* view (shown in Figure 4.1 in the lower left).

To sum it up, Eclipse was chosen because of its features. The plug-in architecture of Eclipse enables extensions of other plug-ins and loosely coupled functionality. The Eclipse Modeling Framework contributes with a unified framework for creating models and provides both code-generation and serialization of the model. With the Graphical Modeling Framework you get a generated diagram editor for the model with all the common user interface functionalities like copy/paste, zoom and printing already in place. The next sections will provide a general overview and some short introductions to Eclipse and some of its frameworks.

4.2.1 History of Eclipse

The codebase in Eclipse originated from IBM VisualAge, developed by Object Technology International (OTI), a subsidiary of IBM. VisualAge was a family of computer integrated development environments (IDEs) which included support for a few popular programming languages like C++, COBOL, Fortran, Java and others. A version of VisualAge, VisualAge Micro Edition, was a re-implementation of the original IDE in Java, and this was the version that served as the foundation for the Eclipse Platform [1]. OTI started the development of Eclipse as a replacement of VisualAge and in November 2001 a consortium was formed to further the development of Eclipse as an open source project.

The initial Eclipse Board of Stewards included industry leaders such as Borland, IBM, Red Hat, SuSe, Rational Software and others. By the end of 2003 the initial consortium had grown to over 80 members. In 2004 Eclipse was reorganized into a non-profit organization and Eclipse became an independent body that drives the platform's evolution to benefit the providers of software development offerings and end-users. All technology and source-code provided to and developed by this fast-growing community is made available royalty-free via the Eclipse Public License [13].

4.2.2 Structure of the Eclipse Platform

The components that Eclipse is made up from are called *plug-ins*. A plug-in is the smallest unit of Eclipse Platform functionality that can be developed and distributed separately [21], in fact all of Eclipse's functionality comes from plug-ins except for a small kernel which is called the *Platform Runtime*. Small tools may be written as a single plug-in and more complex tools may have the functionality split across several plug-ins. Each plug-in has a *manifest* file which declares its interconnections with other plug-ins. The interconnection model is that a plug-in declares any number of named *extension points* and any number of *extensions* to other plug-ins extension points.

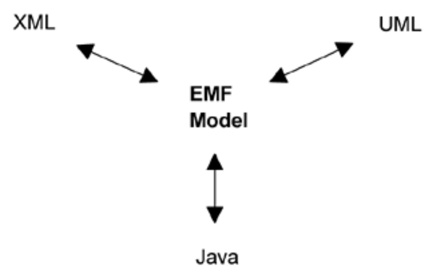


Figure 4.2: EMF unifies Java, XML and UML

The extension mechanism basically facilitates enhancement and communication between plug-ins. Each plug-in can define extension points that can be used either internally or by other plug-ins. This is the main idea of Eclipse; to extend, not replace. With the use of extension points one can enhance plug-ins or loosely couple chunks of functionality in a controlled manner. Eclipse also provides a mechanism for extending objects dynamically. A class that implements an "adaptable" interface declares that its instances are open for third-party extension of behavior, and any party can add behavior to existing types of adaptable objects.

One of the interesting projects for this thesis is the Eclipse Modeling Project. It is a project that focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling and standards implementations. One of the stronger arguments to use Eclipse is a subproject of the Eclipse Modeling Project, called the Eclipse Modeling Framework (EMF).

4.2.3 The Eclipse Modeling Framework Project

The EMF project[1] is a modeling framework and code generation facility for building tools and other applications based on a structured data model. EMF started out as an implementation to OMGs Meta-Object Facility (MOF) specification [17]. In the latest proposal, MOF 2.0, a subset of MOF is filtered out, called *Essential MOF* (EMOF). This subset is almost identical to the metamodel of EMF, Ecore. In fact, EMF can transparently read and write serializations of EMOF. The main purpose of EMF is to describe and build models. With some higher level code, these models can be used to generate code which can be used as a basis for any Java (or other programming languages) application. The model used to represent models in EMF is called *Ecore* [3]. Ecore is itself a EMF model, which means that it is its own metamodel.

For clarification, a simplified definition of a metamodel is that it is the set of rules, constructs and constraints that a model must conform to. An example of this could be that a UML class model must conform to the UML metamodel specified by the UML to be "legal".

EMF is intended to provide the benefits of formal modeling, but at a low cost of entry [14]. It is a open source framework targeting MDA development [23].

[1]`http://www.eclipse.org/modeling/emf/`

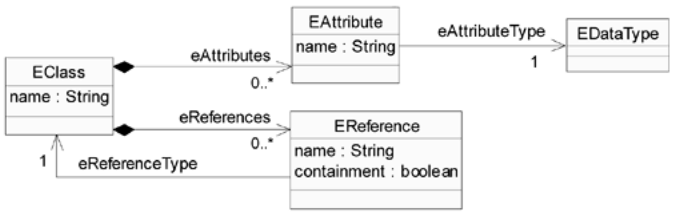


Figure 4.3: A simplified subset of the Ecore model, adopted from [3]

The input models can be created using annotated Java code, XML documents or modeling tools like Rational Rose, then imported into EMF. The code generator in EMF then turns the model into a set of Java implementation classes. Figure 4.2 shows how EMF unifies three important technologies: Java, XML and UML. Regardless of which technology is used to define it, an EMF model is the common high-level representation that “glues” them together [3]. With the code-generation, the focus is on the model itself, not its implementation.

The default serialized form of an Ecore model and instances of the model is XML Metadata Interchange (XMI). The XML Metadata Interchange is an OMG standard [18] for exchanging metadata information via XML. The most common use of XMI is as an interchange format for UML models although it can be used for serialization of other languages and metamodels, such as an EMF model. It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF).

A simplified subset of Ecore can be seen in figure 4.3. An `EClass` is used to represent a model class, which can have attributes (`EAttribute`) of a certain data type (`EDataType`), and it can reference other classes through `EReference`. This means that by using EMF we get a nice, extendable and unified framework for building meta-models.

The next sections provides more on the graphical aspects of the Eclipse Platform, beginning with a short introduction of the widget toolkit that Eclipse uses, and continues with an introduction of GEF and GMF.

4.2.4 Standard Widget Toolkit

For the GUI elements, Eclipse uses the Standard Widget Toolkit (SWT)[2]. SWT is a graphical widget toolkit for use with the Java platform, originally created by IBM as an alternative to the Abstract Window Toolkit (AWT) and Swing Java GUI toolkits provided by Sun Microsystems as part of the Java Platform, Standard Edition. SWT was designed to be a high performance toolkit; to be faster and less resource consuming than Swing. This is done by using the operating systems native widgets while Swing instead emulates them. SWT was chosen as the GUI toolkit for Eclipse because it was believed that the native look, feel and performance were critical to building desktop tools for demanding developers [1].

[2]`http://www.eclipse.org/swt/`

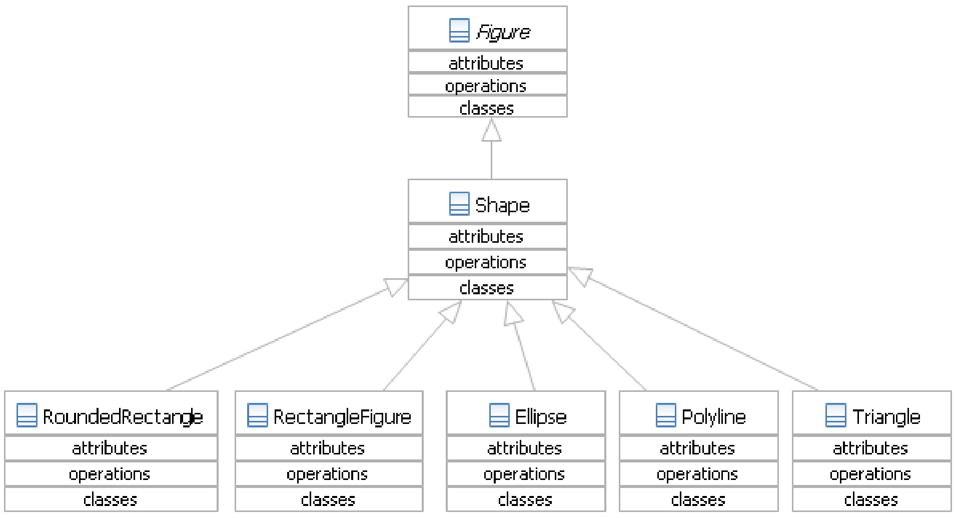


Figure 4.4: A simplified overview of the `Shape` hierarchy

4.2.5 Graphical Editing Framework

The Graphical Editing Framework (GEF)[3] provides a standard way to develop feature rich graphical editors in Eclipse. The editing possibilities of GEF allows the development of a graphical editor for nearly any kind of model, including an EMF model. Using GEF it is possible to build almost any kind of graphical editing application like class diagram editors, GUI builders and even WYSIWYG text editors.

For the actual graphical work, GEF depends on the Draw2d framework. Draw2D is a standard 2D drawing framework based on SWT. It provides the lightweight graphical system that GEF depends on for its display, and is provided with the GEF distribution in its own plug-in, *org.eclipse.draw2d*. Draw2D is not only for use by GEF. It is a self-contained graphical library and can be used by any Eclipse application.

Everything that is visible in a Draw2D window is drawn on a *figure*. The `Figure` class is the base implementation of every figure, and many subclasses of `Figure` provides useful additional functionality. These subclasses include `Shape` which provides an abstract support for a variety of shapes. Some examples of shapes are `Ellipse`, `Polyline`, `RectangleFigure` and so on. A simplified class diagram of the shapes is shown in figure 4.4.

The purpose of GEF is to facilitate the display of any model graphically using Draw2D figures, to support interactions from mouse, keyboard or the Eclipse Workbench and to provide common related components to these operations. GEF employs a strict Model-View-Controller architecture (see figure 4.5), an architectural pattern often used in software engineering to isolate the business logic from the user interface considerations. Separating the presentation from the data allows for a change in the visual presentation without affecting the

[3] `http://www.eclipse.org/gef/`

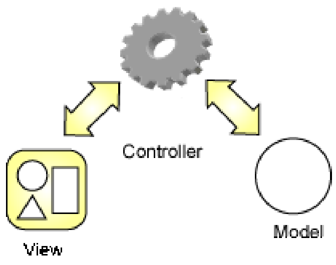


Figure 4.5: Model-View-Controller

underlaying computational system [28].

The **model** is any data that gets persisted. In GEF any kind of model can be used, but the model must employ some sort of a notification mechanism. It is very common to use EMF as the model in GEF, since EMF already includes a notification mechanism that is used throughout EMF. Whenever a model element in EMF is changed, a notification is sent via the EMF notification framework. The **view** is anything that is visible to the user. As explained before, GEF uses Draw2D for its graphical display, and thus the view usually are Draw2D figures. Each model object that is to be visualized needs a **controller** to coordinate the semantical object (from the model) with the notational object (from the view). The controller in GEF is called an *EditPart*, and they are the link between the model and the view. They are also responsible for editing, and contain helpers called *EditPolicies* which handle much of the editing task [7].

EditParts are the central element in GEF. Usually each model element class will have a corresponding EditPart so the class hierarchy of EditParts will likely be the same as for the model. The definition of an EditPart is done through the interface `org.eclipse.gef.EditPart`, but it is recommended that clients extend the base implementation, `org.eclipse.gef.AbstractEditPart` instead of implementing the interface [20].

To respond to user events generated from using the mouse or keyboard, GEF uses *tools*. A tool is a stateful object that translates low-level events, such as a mouse button being pressed, into high level *requests*. The type of request depends on the tool that was active. For example a creation tool will generate a creation request to the editpart whose figure was beneath the mouse pointer when the mouse button was pressed. The editpart does not handle the request itself, instead it delegates the request to the registered edit policy. The editpolicy will get the request and if it understands the request, it creates a *command* which will be executed to fulfill the request. An overview of the process is shown in Figure 4.6.

By using *commands* to execute requests, GEF employs the Command design pattern [16]. A command is the part that actually modifies the model. Commands also include support for execution limitations, undo and redo and also combining or chaining different commands.

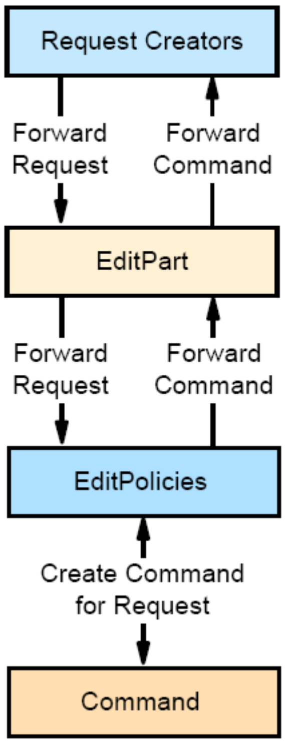


Figure 4.6: Communication chain Request – EditPart – Command. Adopted from [20]

4.2.6 Graphical Modeling Framework

> Let me be blunt: In the past, creating graphical editors within Eclipse using the Graphical Editor Framework (GEF) was slow and painful.
>
> – Chris Aniszczyk, Software Engineer, IBM

The drawbacks of using GEF is that every element has to be hand-coded. With larger model this evolves into very tedious work. GEF is a very complex framework and to create an application based on GEF requires the programmer to understand it.

The Graphical Modeling Framework (GMF)[4] provides a generative component and a runtime infrastructure for developing graphical editors based on EMF and GEF. It provides a more userfriendly way to utilize these two frameworks, by providing graphical interfaces to map elements from EMF and GEF together and using this information to generate a ready-to-use graphical editor.

An overview of the dependencies between GMF, GEF and EMF can be seen in Figure 4.7. Like GEF, GMF is also dependant of a domain model, but unlike GEF, GMF requires the model to be in Ecore format. Creating (or importing) the domain model is the first step in a GMF project. The next steps

[4]`http://www.eclipse.org/modeling/gmf/`

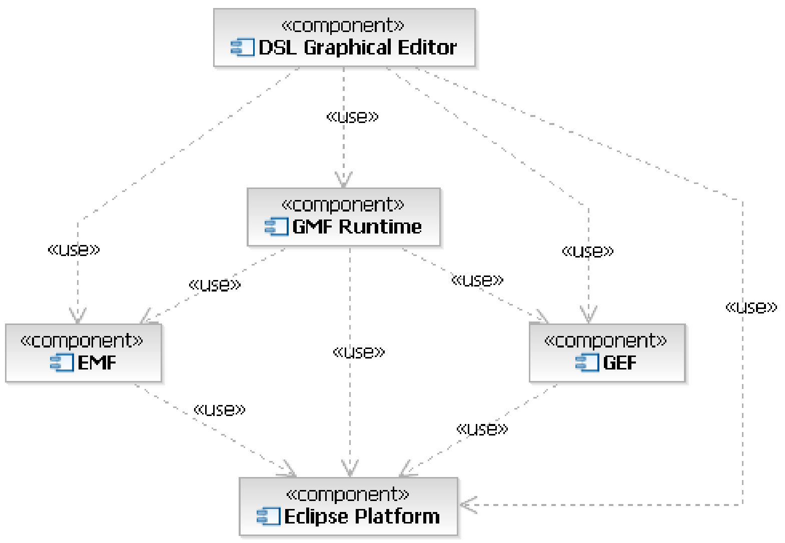


Figure 4.7: UML 2.0 component diagram shows the dependencies between the generated graphical editor, the GMF Runtime, EMF, GEF, and the Eclipse Platform. Adopted from [22]

are to create a graphical and a tooling definition model. The *graphical definition model* declares the various figures that will be shown in the editor. Although it has no direct relation to the domain model, the domain model can be used to generate a graphical definition model where the generative component of GMF tries to derive which elements are to be classified as nodes and which elements are connections. The *tooling definition model* defines which tools are to be available to the user, which basically are creation tools or actions for the graphical elements. These tools are eventually put in a *palette*.

The domain model, graphical definition and tooling definition gets mapped together in the *mapping definition model*. This model maps each tool with a notational (the visual element) and semantical (the domain element) element. This is a key model to GMF development and will be used as input to generate the *generation model*. The generation model is then used to generate the diagram editor plug-in, which will place itself in a new project in the Eclipse workspace. This plug-in can then be run, and instances of the domain model can then be created in the new generated diagram editor.

4.3 Summary of analysis and issues to resolve

Although it is very nice to get almost the entire graphical diagram editor for free from the Eclipse plug-ins, there is still much work to be done. First of all, the metamodel needs to be manually specified in an EMF model (Ecore). Next, GMF does not generate the diagram editor all by its own, many things needs to be manually specified. For instance, a connection generated by GMF is by default just a line. Any extra graphical visualization (like an arrowhead decoration on the target end) must be added manually. Every model object that is to be created in the diagram needs both a creation tool and a graphical figure, which is mapped together.

Any custom graphical figures has to be made manually, which requires a deep understanding of both the GEF and GMF frameworks, since GMF is using and extending GEF for the visuals and the controllers. As if that wasn't enough, the graphical figures are not done by GEF itself, instead this is handled by another plug-in, called *Draw2d*, which in turn is based on the *Standard Widget Toolkit* that Eclipse is using for all its graphical interfaces and drawing. With any complex and extensive framework comes a steep learning curve, and frameworks in Eclipse are no different.

When modifying or extending plug-ins there are also other factors to consider. The Eclipse Platform is made to be extended, and all of it's plug-ins as well. Which means that any plug-in must (or at least should) provide extension points for other plug-ins to use. And with this behavior there are also certain rules that one must follow. Like the Eclipse UI Guidelines [8] for example. Since this is based mostly on GMF which is supposed to create a standard for graphical editors, people would expect certain things from the product, like the palette that is used, the behavior of the tools, the different views that are created and so on.

EMF was not made with DPL in mind, which means that EMF somehow must be extended to support the extra constraints we impose on the objects. There are also aspects of the diagrammatical notation that is not supported at all by GMF. For example the arc between arrows in a DPL diagram is not something that GMF was made to do, and has to be worked around.

As the documentation and library-API's were piling up, it was clear that the project description and the scope of this thesis had to be shortened. For now, the requirements are:

1. Create a metamodel containing a simple graph using EMF and generate a graphical editor for it
2. Create additional visual shapes for the editor, including different types of connections
3. Mark nodes and arrows in the editor as being part of a diagram constraint

Since this is not the final piece in the parent project, all of this has to be documented clearly. In fact, other people involved with the project will continue working with Eclipse and its frameworks after this project is done, and hopefully some of the documentation in this master thesis will be of use to them.

Chapter 5

Solution and implementation

The design of the application tries to take the same approaches as the existing solutions, Sketcher95 and Sketcher.Net. But there are of course differences in the way the approaches are implemented, seeing as they have to follow the standards of EMF and GMF in Eclipse. The ultimate goal is to have the Eclipse version offer the same functionality as for example Sketcher95. Looking at how Sketcher95 does things will at least help the end-users communicate how they want the Eclipse implementation to perform.

By observing Sketcher95 it is possible to get the main structure of how the domain model of the DPL formalism should look like. The user interface of both Sketcher95 and Sketcher.Net is however dropped, as it would be too much work trying to bend the GMF generated editor into their shape. It is not impossible to do so however, but one of the points of using GMF is to get a standard look and feel for graphical editors in Eclipse. Users who have used other types of graphical editors in Eclipse expect certain things which the GMF generated editor conforms to.

5.1 The core of the DPL formalism

The initial problem is to make a domain model for the DPL formalism. As most papers on the formalism focuses mostly on the mathematical subject and the possibilities, there are not much detailed information on the actual structure except some definitions (see section 2.6). But observing Sketcher95 will bring some additional information.

Skertcher95 does its work in two distinct parts, creating a marker signature and creating a sketch document. The most interesting part here is the marker signature and what it contains. A marker signature starts out empty, except for two items, a **constraint-less node** and a **constraint-less arrow**. These two elements is the minimal requirements for making any graph. A marker signature can have three collections of **constraints**(which can be organized further into sub-collections). These are: **Node Constraints**, **Arrow Constraints** and **Diagram Constraints**. All of the mentioned collections are populated by **Markers**, one marker for each constraint.

A node constraint marker (see figure 3.3(a)) has three elements; a **Name**, a **Description** and a **Visual Marker**. The visual marker is selected by a pool of **primitives** and has three parts, a **Geometric Figure**, a body **Fill** and a **Border**.

An arrow constraint marker (Figure 3.3(b)) has the same **Name** and **Description** elements as a node constraint marker, but the **Visual Marker** is more extensive, containing two primitives for the **Tail**, two for the **Head** and one for the **Body**.

A diagram constraint marker (Figure 3.4 is more complex than a node or arrow constraint marker. It too contains a **Name** and **Description**. A diagram constraint marker also contains a **Shape** (or **Arity**) which can contain **Nodes** and **Arrows** with or without **Constraints**. It also has a **Diagram Marker** which basically is a label, and an **Arc**.

Observing Sketcher95 has provided most of the terms, but it is important to take in consideration that Sketcher95 currently is 13 years old, and terms and concepts has been given new names in the later years.

The problem however is how to model all these things in a metamodel that EMF can understand. The concept of splitting the functionality in two parts as Sketcher95 does will not work in the scheme of GMF, since it requires one metamodel. But as a beginning, the focus is on a simple graph. A (directed) **Graph** consists of a collection of **Nodes** that can be connected by **Arrows**. An **Arrow** has a **target** Node and a **source** Node. From the definitions in DPL, we can consider a specification as a directed graph in which some parts (diagrams) are marked with predicate labels taken from a predefined signature. In Sketcher95 these predicate labels are logically divided into three categories which are: predicates on a Node – interpreted as a node constraint, predicates on an arrow – interpreted as an arrow constraint and finally predicates on diagrams – interpreted as diagram constraints.

This translates (roughly) into the UML class diagram as shown in figure 5.1.

5.2 How to implement the model in EMF

As stated before, EMF requires a metamodel (see section 4.2.3). An EMF model can be defined using either annotated Java code, UML class models (from editors supporting XMI, like Rational Rose from IBM or the UML2 project[1]) or by using XML Schema. The easiest for a Java programmer is to use annotated Java code. For more information about how to define an EMF model, see [3].

To create the EMF model using Java code, one simply create the interfaces that makes up the metamodel. The various classes, their attributes and operations that is a part of the metamodel is marked with a `@model` tag in the

[1] `http://www.eclipse.org/modeling/mdt/?project=uml2`

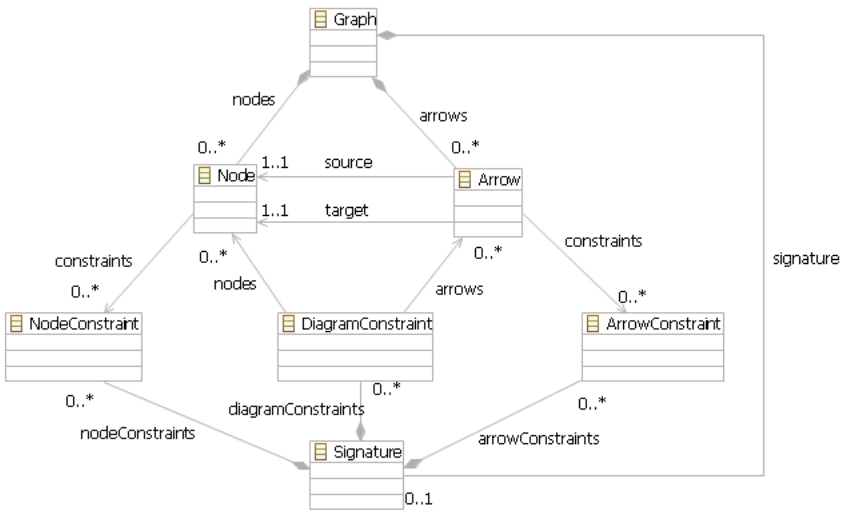


Figure 5.1: A rough UML class diagram of the structure

respective JavaDoc[2] comments. This also specifies any non-default values and references for the Ecore objects. Since this Java programmers already understand the syntax and the process only requires a text-editor, annotated Java offers the lowest cost of entry into EMF. EMF uses a subset of the JavaBean [29] simple property accessor naming patterns, which basically means that every attribute (or property) needs a `get` and a `set` method. An example of annotated Java is shown in Code Listing 5.1.

Listing 5.1: Example annotated Java interface

```
/**
 * @model
 */
public interface Edge {
  /**
   * @model
   */
  Node getSource();
  . . .
}
```

Regardless of the approach used, the result is two different models: the core model (Ecore) and a generator model (genmodel). These two models will drive the generation of a complete application. For each class in the model, the EMF code generator will create one interface and one implementing class, complete with accessors for every attribute (ie. getters and setters).

An important feature of EMF is the notification mechanism, and this is included in every setter in the implementing class. This means that when a user (or some feature) changes an attribute in a model element, a `Notification` is sent to any observers. the `Notification` includes the object that has been

[2]A documentation generator from Sun Microsystems for generating API documentation

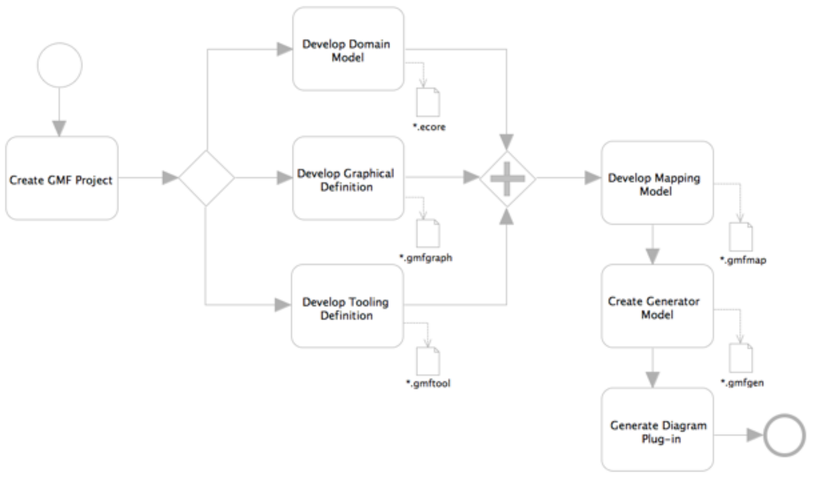


Figure 5.2: GMF Overview, adopted from [12]

changed, what kind of notification it is (like `Notification.SET` for a setter method), a *feature id* for the attribute along with the previous value and the new value of the attribute.

Every generated method has a `@generated` tag in their JavaDoc comments, so if is modified it is necessary to either remove or change the tag (appending 'NOT' to the tag is the convention) in order to prevent the modifications from being overwritten if the codes is generated again.

The EMF code generator also creates a *Factory* for creating instances of the objects in the model, and a *Package* interface that defines some properties of the package, a static constant reference to an implementation of the package and convenient access to all the metadata of the model. For more details on the Eclipse Modeling Framework see [3].

From the `genmodel` the user also has the option of generating *Edit code*, *Editor code* and *Testing code*. Generating editor code will produce an EMF editor project that can be used to create instances of the model. The testing code generates a testing project, complete with JUnit[3] tests.

With the now completed "semantical" model for the diagram editor, it is time to move on to GMF.

5.3 Letting GMF do its work

As explained in Section 4.2.6, GMF uses an Ecore file as input to derive the different "templates" that the diagram editor will be built from. An overview of the process in creating a diagram editor can be seen in Figure 5.2.

The graphical definition model is derived from the Ecore model, and put in a file with the extension `gmfgraph`. GMF can almost automatically decide

[3]A unit testing framework for Java

which of the elements to include in the diagram and give them some default visual figures; a `Node` gets a rectangle and a `Arrow` is represented as a line. But in order to do so, GMF needs to know the *diagram element* of the model. The diagram element is the "containing" element of the diagram, in this case it is the element `Graph`.

The `gmfgraph` consists of a *Canvas.* The canvas consists of all the nodes and connections from the Ecore model and a *figure gallery.* The figure gallery contains all of the *figure descriptors.* Every node and connection in the model has its own *figure descriptor.* The figure descriptor specifies the visual appearance of the nodes and connections. A figure for a node is by default a rectangle, while a connection is a line. There also exists several figure galleries that are intended for re-use and are included with GMF. An example of this is some common used figures in class diagrams, contained in `classDiagrams.gmfgraph`. These galleries can be loaded as resources into the `gmfgraph` so that their components can be used as part of the graphical definition.

The tooling definition model (`gmftool`) is also derived from the Ecore model. This defines the palette of the diagram editor, which means the creation tools for the diagram. Each model element that is to be created in the diagram has its own creation tool. You can also add *standard tools*, which are the standard tools in GMF like select, zoom etc. There is also support for *generic tools* which really isn't documented at all.

An important part to point out is that the actual palette is created programmatically in the generated diagram editor instead of using the Eclipse provided `paletteProvider` extension. Which is somewhat odd, considering that the runtime tutorial made by the GMF developers use the mentioned extension for the palette entries (see [11]). A consequence of filling it out programmatically is that palettes cannot be provided to other editors easily[4]. It was discussed why the mechanisms that were provided by the runtime was not used, but in the end the developers decided that it was not a common usecase to reuse palettes in different editors and the bug report[5] was eventually closed.

Aside from the palette, the *Tool Registry* in the `gmftool` can also contain other means of executing operations through *context menus*, *popup menus*, or even through the Eclipse toolbar. There is however little or no documentation around of how to actually do so, other than by trial and error.

The mapping model (`gmfmap`) combines the three existing models, the Ecore model, the graphical definition and the tooling definition into one model. Here each model element is mapped to a graphical visualization and a creation tool. The *wizard* for this is not really accurate, often the connections needs to be manually added. And since the names are often similar on connections there is a problem knowing which source and target attribute to use.

The last model in GMF is the *Diagram Editor Generation Model.* This contains all the previous information and some extra bits that are necessary to generate a graphical editor. There are usually no reasons to modify this manually.

[4]Original report of bug is found at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=126199`

[5]See discussion at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=168396`

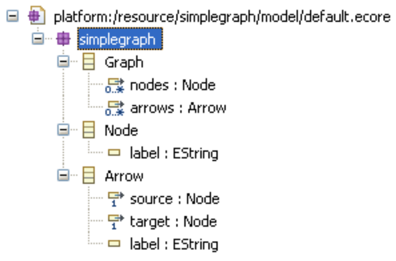

platform:/resource/simplegraph/model/default.ecore
simplegraph
Graph
nodes : Node
arrows : Arrow
Node
label : EString
Arrow
source : Node
target : Node
label : EString

Figure 5.3: The Ecore file of a simple graph

5.4 An Example Graph Editor

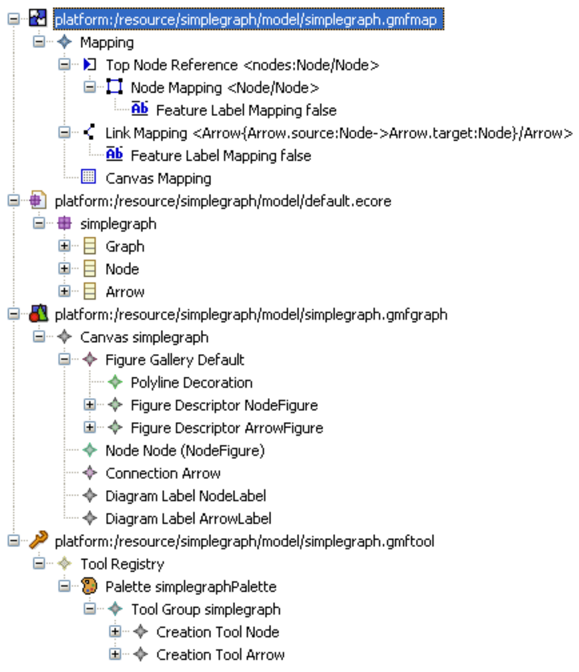
As an example, consider a simple directed graph only containing nodes and arrows. The Ecore of the domain model is shown in figure 5.3. When creating an Ecore it is important to make sure that every object is contained by a parent object, and ultimately, the root element which in this case is `Graph`. If there are some objects that does not have a proper parent, it will not show up in the diagram editor as GMF will not generate any tools or graphical definitions for it. It is also important to check on the multiplicities. Unless otherwise specified, an object contained in `Graph` will have the multiplicity [0..1] which means that a `Graph` will at most have one such object. To change this, set the "Upper Bound" property for the reference (in this example, `nodes` and `arrows` listed under `Graph` to `-1` (meaning a multiplicity of `*`). Every `Node` and `Arrow` does also have a `label` attribute of the type `EString`[6] to help separating the different objects of the same type.

From the Ecore the `genmodel` is derived and then the model code and edit code is generated. For now, neither needs any modifications.

Next the graphical definition model is created, marking `Graph` as the root element and `Node` as a node and `Arrow` as a connection. In the `gmfgraph` a *Polyline Decoration* is added and set as a *Target Decoration* in the the *ArrowFigure*. No extra specification is needed as the default visual of a polyline decoration is an open arrowhead. Both the `NodeFigure` and `ArrowFigure` has a child label and a corresponding accessor method. The label will show the contents of the `label` attribute in the model element.

The tooling definition model does not require any modifications, since a tool in this sense basically just a name, description and some icons. The three models, Ecore, `gmfgraph` and `gmftool` is mapped together in the Mapping Model. The wizard that is doing the work does have some issues deciding which element is a node and which is a connection however as it initially defines `Arrow` as a node. The `Arrow` element is changed to a *link* and its source and target feature are set accordingly, as well as defining the proper tool which will create the `Arrow` element. Another issue with `gmfmap` is that it does not know which *Diagram label* it is supposed to put the information from the `Node` and `Arrow` attribute to, so this needs to be specified manually in the *Feature Label Mapping* child of `Node` and `Arrow`. The resulting `gmfmap` file is shown in figure 5.4.

[6]`EString` is EMF's `String` equivalent



```
platform:/resource/simplegraph/model/simplegraph.gmfmap
  Mapping
    Top Node Reference <nodes:Node/Node>
      Node Mapping <Node/Node>
        Feature Label Mapping false
    Link Mapping <Arrow{Arrow.source:Node->Arrow.target:Node}/Arrow>
      Feature Label Mapping false
    Canvas Mapping
platform:/resource/simplegraph/model/default.ecore
  simplegraph
    Graph
    Node
    Arrow
platform:/resource/simplegraph/model/simplegraph.gmfgraph
  Canvas simplegraph
    Figure Gallery Default
      Polyline Decoration
      Figure Descriptor NodeFigure
      Figure Descriptor ArrowFigure
    Node Node (NodeFigure)
    Connection Arrow
    Diagram Label NodeLabel
    Diagram Label ArrowLabel
platform:/resource/simplegraph/model/simplegraph.gmftool
  Tool Registry
    Palette simplegraphPalette
      Tool Group simplegraph
        Creation Tool Node
        Creation Tool Arrow
```

Figure 5.4: The GMFMap of a simple graph

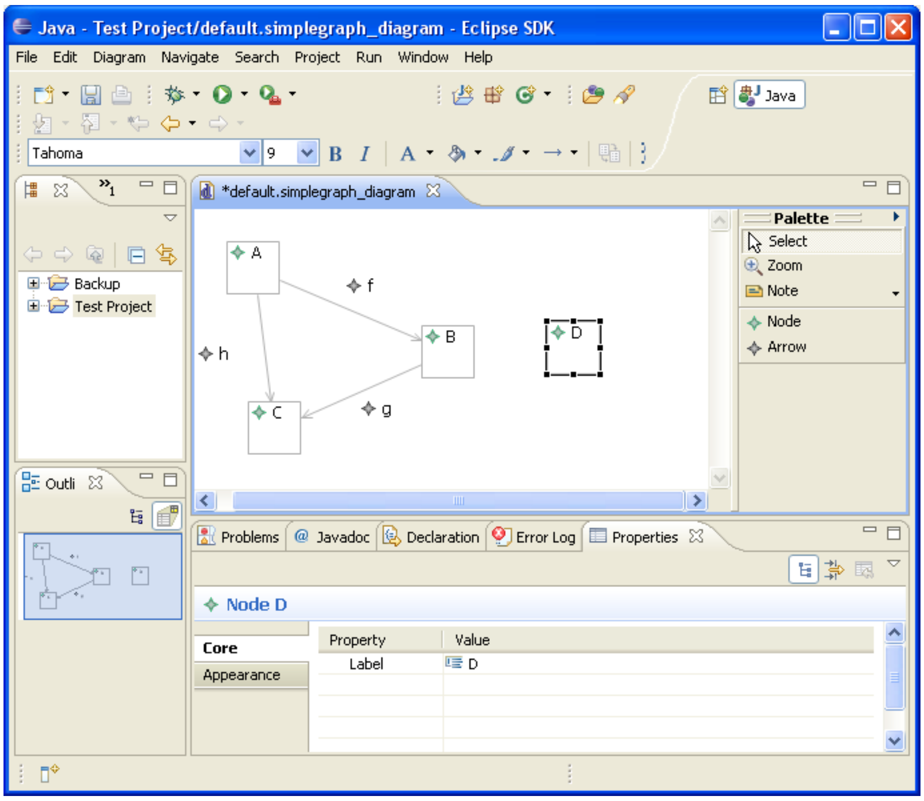
The mapping model is then transformed into a *Diagram Editor Generation Model*, and since it does not require any modifications, this is used as it is to generate a diagram editor. Assuming the event is problem-free, the generator will produce a new plug-in which can be loaded into the Eclipse workspace and be able to create diagrams like shown in figure 5.5, complete with a canvas to draw on, a palette with creation tools and default selecting and zooming capabilities, a property view and a outline view.

5.5 The GMF generated packages

For the simple graph diagram editor as described in the previous section, there is a total of 11 packages in the generated plug-in. These packages include a total of 76 java files, meaning that there is a lot of information to get a grasp of. The following is a brief explanation of each package:

`diagram.edit.commands` Commands are the part that actually modifies the model. This package contains commands that when executed creates or modifies the EMF resources.

`diagram.edit.edithelpers` An edit helper defines any behavior modification for a metamodel type. For example if one wants to add some initialization to the model elements, this is the place to do it. Every *editpart* has an edithelper, but the default generated edithelper class is empty. This is also the place to put edithelper advice, which basically are edithelpers for any specialization types (explained later).


Java - Test Project/default.simplegraph_diagram - Eclipse SDK
File Edit Diagram Navigate Search Project Run Window Help
Tahoma
9
Java
*default.simplegraph_diagram
Backup
Test Project
Outli
A
B
C
D
f
g
h
Palette
Select
Zoom
Note
Node
Arrow
Problems
Javadoc
Declaration
Error Log
Properties

Node D

Core	Property	Value
Appearance	Label	D

Figure 5.5: A diagram editor creating instances of simple graphs

`diagram.edit.parts` EditParts are the building blocks of GEF viewers, and are the controllers that ties the application's model to a visual representation. They are also responsible for requesting changes to the model.

`diagram.edit.policies` EditPolicies determines an EditPart's editing capabilities. Although it is possible to implement EditParts so that they handle all the editing responsibilities themselves, it is more flexible and object-oriented to use EditPolicies. As an example, when creating a connection between nodes, the `NodeItemSemanticEditPolicy` will decide if the specified connection type is legal, thus deciding whether or not the user may create a connection to or from the node.

`diagram.navigator` Deals with the navigator view in Eclipse and how to show the different model elements in the navigator.

`diagram.parsers` Parses different text strings. Not much documentation provided for this one.

`diagram.part` The "business" part of a diagram editor. Here the pieces that makes the editor application is put together.

`diagram.preferences` The preferences of the diagram editor. These include printing preferences, appearance preferences and others. The default implementation does not do much except to pass things to their superclasses.

`diagram.providers` Classes here provide instances of the different factories that are in use in the diagram editor application. For example the `EditPartProvider` provides the `EditPartFactory`.

`diagram.sheet` Makes the properties of the editparts displayable in the *properties view* (see Figure 5.5 at the bottom).

`diagram.view.factories` The View Factories for the notational elements.

5.6 Extending the Simple Graph

One of the requirements is to have **diagrams** in the graph. A diagram can be considered as a fragment of a graph closed in some technical sense. To simplify things for this thesis, a diagram is considered to be a collection of nodes and arrows.

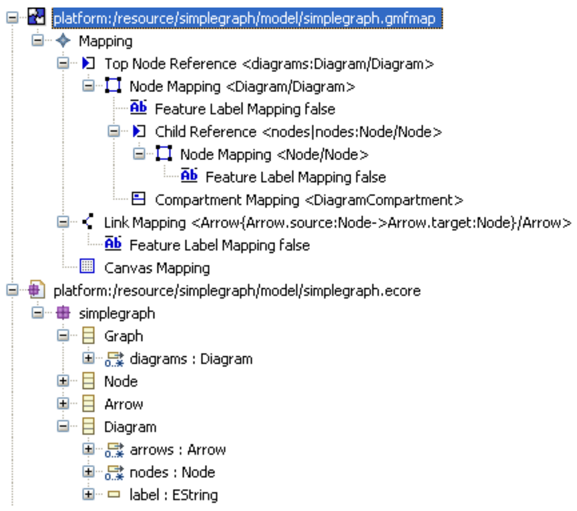
Then there is the issue of how to visualize this. In Sketcher95, a graph was drawn using only constraintless nodes and arrows. When adding a diagram constraint (see Figure 3.6), the user chooses which arrows in the graph that corresponds with arrows in the constraint. This leads to the conclusion that a diagram constraint contains, apart from a name and description, references to nodes and arrows.

The GMFMap model only allows for one containing class. Which means that the containment of nodes and arrows has to be moved to the new class, **Diagram**. This results in re-creating all the generated files, because this kind of change reflects badly in the pre-generated files and can lead to weird errors. While it is a pain to have to create new GMF models, it is better than trying to 'fix' the generated code. Unless of course there is a lot of custom code. The process of creating new `gmfgraph`, `gmftool` and `gmfmap` files is somewhat tedious if one does not have a clear metamodel.

5.6.1 Adding Compartments

The new GMFMap (and Ecore) is shown in Figure 5.6. The **Diagram** node is added and has a *Compartment* mapping, which means that nodes (and arrows) now are created inside a Diagram. The resulting diagram editor is shown in Figure 5.7. The placements of the nodes inside the diagram looks weird though. This is because of the default behavior of compartments. Originally meant to contain things like labels for attributes and such (in the case of a diagram editor), children in a compartment are placed in a 'stack', with every new element placed beneath the previous one. While this is great for an UML editor for example, it is not ideal for this project. To change this behavior, the generator model needs to be changed. Inside the `gmfgen` file, the property for the compartment can be changed. The property that needs to be changed is `List Layout` which needs to be set to `false`. The fixed result is shown in 5.8.

There are some issues though. Arrows can be created between nodes in different diagrams, while the intent is that diagrams should not have references to other diagrams. This behavior is determined by `Node`, and a node will allow



```
platform:/resource/simplegraph/model/simplegraph.gmfmap
  Mapping
    Top Node Reference <diagrams:Diagram/Diagram>
      Node Mapping <Diagram/Diagram>
        Feature Label Mapping false
        Child Reference <nodes|nodes:Node/Node>
          Node Mapping <Node/Node>
            Feature Label Mapping false
        Compartment Mapping <DiagramCompartment>
    Link Mapping <Arrow{Arrow.source:Node->Arrow.target:Node}/Arrow>
      Feature Label Mapping false
    Canvas Mapping
platform:/resource/simplegraph/model/simplegraph.ecore
  simplegraph
    Graph
      diagrams : Diagram
    Node
    Arrow
    Diagram
      arrows : Arrow
      nodes : Node
      label : EString
```

Figure 5.6: Adding compartment mappings for simplegraph in GMFMap and Ecore

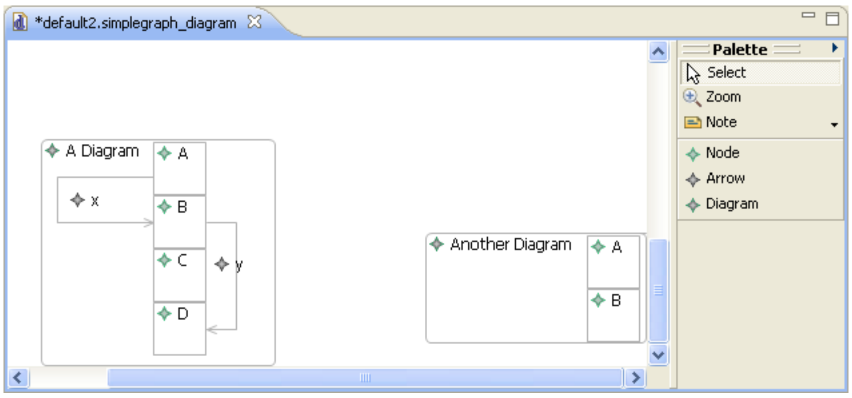


Figure 5.7: Weird behavior of node layout

creations of a connection start or end as long as the connection is of type `Arrow`. Since arrows should not go outside of the `Diagram` that is the parent of the `Node`, this behavior needs to be modified.

5.6.2 Modifying Connection creation

When the ‘Arrow Creation Tool’ is selected, the editpart that is directly under the mousepointer will determine whether or not it will allow a connection to be made. Neither the root `Graph` or `Diagram` allows this behavior and the mouse pointer will show that it is not possible to create a connection. The `Node` element however will allow this, and the behavior is determined by the class `NodeItemSemanticalEditPolicy` in the `diagram.edit.policies` package.

The method that returns the creation commands for connections between nodes are `getStartCreateRelationShipCommand` which will return a create

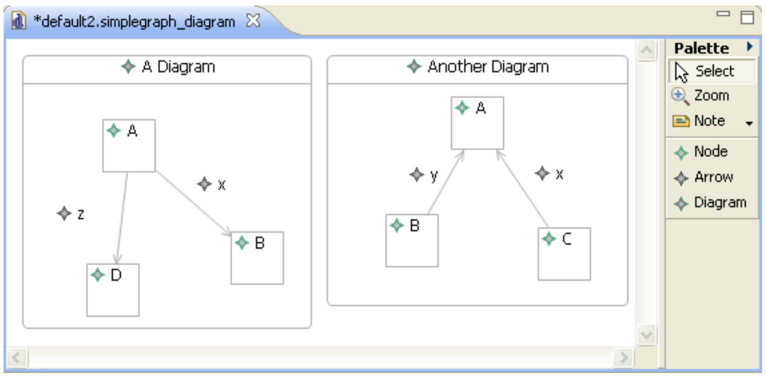


Figure 5.8: Node layout fixed

command if the node is a valid source end of the connection and `getComplete-CreateRelationshipCommand` which returns a create command if the node is a valid target of the connection. To have connections from a node inside a diagram limited to end on only other nodes in the diagram, checks are needed to decide whether or not the *container* of the target end is the same as the node on the source end.

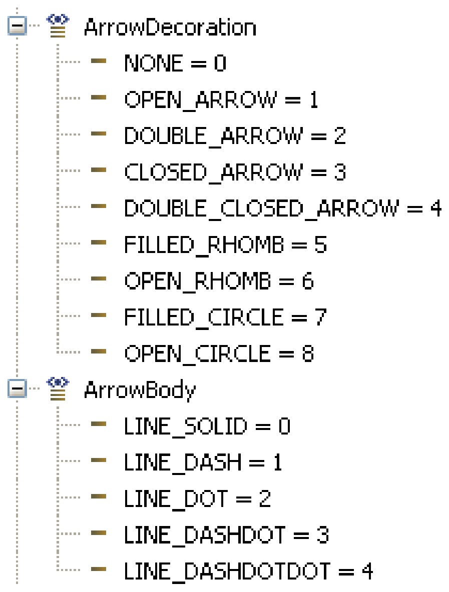
5.7 Creating instances of a simple graph

As shown in the previous section, diagrams now contain all the nodes and arrows in the graph. This is partly because of Sketcher95's separation of the concept of a signature and a sketch. But because of time-limitations this is not done here, and thus both the signature and the actual model-instance will be displayed in the same editor window.

This means that `Graph` needs some `Nodes` and `Arrows` of its own. It seems best to subclass Node into DiagramNode and GraphNode to avoid confusions. As usual, it is best to delete previous work and start over. Tedious, but at least it gets done faster.

5.8 Adding Constraints for Nodes and Arrows

Some new elements are introduced, `NodeConstraint` and `ArrowConstraint`. These will be used for adding constraints to the constraintless nodes and arrows. The constraint object consists of a name, a description and some visualization. For now, the visualization is a simple `Enumeration` type, and will be translated to visual decorations by the EditPart. Figure 5.9 shows the different enumeration values of the constraint.



- ArrowDecoration
  - NONE = 0
  - OPEN_ARROW = 1
  - DOUBLE_ARROW = 2
  - CLOSED_ARROW = 3
  - DOUBLE_CLOSED_ARROW = 4
  - FILLED_RHOMB = 5
  - OPEN_RHOMB = 6
  - FILLED_CIRCLE = 7
  - OPEN_CIRCLE = 8
- ArrowBody
  - LINE_SOLID = 0
  - LINE_DASH = 1
  - LINE_DOT = 2
  - LINE_DASHDOT = 3
  - LINE_DASHDOTDOT = 4

Figure 5.9: Enumeration of the different visual types

5.9 Listening for changes

In this section the subject is a `GraphArrow` and `ArrowConstraint`, which will be referred to as simply ‘arrow’ and ‘constraint’. The idea is to change the visualization of the arrow whenever the constraint changes. But since the constraint is not “owned” by the arrow, the arrow is by default oblivious to any changes made to the constraint. Which means that when the constraint’s values change, the arrow will not reflect these changes. An example would be that a constraint has the value of `arrowHead` changed from `NONE` to `OPEN_ARROW`, which should be reflected by the arrow visualization as an open arrowhead added to the arrow’s target connection.

The remedy this, the solution is to add *listeners* to the arrow EditPart. All the listeners to semantical children of an EditPart is automatically added, but since the constraint is not a semantical children of an arrow (it is contained by the `Graph` object so it can be accessible for other arrows) one must add it manually. This is done by overriding the `addSemanticalListeners` method.

To have a figure reflect changes on other (not directly related) figures it is necessary to make sure that the EditPart of the figure receives notification of the changes. This is done by adding a listener for the semantical element. The process of adding listeners to editparts is usually handled by the super classes of the EditPart in question in the method `addSemanticListeners` which automatically registers *listener filters* for any semantical children of the semantic element of the EditPart. Each listener filter is made up from a filter id string, the listener and an object to listen to. The object must of course be observable, but every EMF object is observable by default.

To manually add listeners, the `addSemanticListeners` method must be overridden. It is important to add a call to the super implementation of the method. Also, the `removeSemanticalListeners` method must be overridden to remove the manually added listener filters. In both methods, the operation of adding a filter is handled by the `addListenerFilter` method. For each listener filter that is to be added, simply add `addListenerFilter(filterId, this, myObject)` to the overriden `addSemanticalListeners()` method, and the corresponding `removeListenerFilter` calls in the `removeSemanticalListeners` method.

There is also the issue of adding the constraint to the list of constraints contained by the arrow. The original idea was to add constraints to an arrow by adding a connection between the arrow and the constraint. The constraint would then be added to the arrow by using the EditHelper of the connection. This initially worked out fine, until one tried to remove the arrow. It seems that unless a connection is connecting two elements defined as nodes, it will be considered as just a notation, so when trying to delete the connection only the view would be removed. The semantic `ArrowConstraintConnection` element would still remain in the model and be redrawn upon refreshing the diagram.

Usually when destroying an element, the semantical policy of the element would pick up the request, create a `DestroyElementCommand` which would dispose of both the View and the semantical object. When a connection between other connections is destroyed it is only removed from the view. The `getDestroyElementCommand` from the connection's semantical policy is never called. It seems the reason for this behavior lies in `ConnectionEditPolicy` and its method `shouldDeleteSemantic()`. In the end, `createDeleteViewCommand()` is called instead of the proper delete command. At this point there seems to be no solution as to get connections between connections in the way that is wanted.

After this setback the idea of adding constraints to arrows by connecting them was abandoned. Because of this, the user has to edit the properties of the arrow via the properties view and add constraints manually.

5.10 Changing visualization of an Arrow

When an `ArrowConstraint` has been added to an `Arrow`, the arrow now has a way of listening to the changes made in the constraint. But just listening to the changes does nothing. The EditPart of the arrow, `ArrowEditPart` is responsible for creating the visual figure for the arrow, and it does this by calling it's `createFigure` method. All of the figures in GMF are various extensions of Draw2D's `Figure`. All figures for connections in GMF are subclasses of `PolylineConnectionEx` which ultimately is a `Figure`. A simplified class diagram is shown in Figure 5.10.

Source and target decorators of an arrow are added to a `PolylineConnection` by the methods `setSourceDecoration` and `setTragetDecoration`. And since the arrow figure is extending this class it has access to these methods. Normally the figure generated by GMF's code generator would add the decorators specified in `gmfgraph` in the figure's `createContents` method. But since the requirements calls for a more dynamic approach, this behavior is modified.

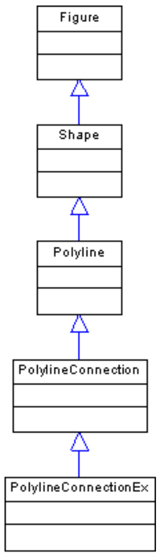


Figure 5.10: The connection figure hierarchy

When the figure is created, there are no decorators added to the connection. But when an `ArrowConstraint` is added to the connection, the decorators should be added as specified in the constraint. The connection is added as a listener to the constraints, as described in the previous section, and is notified of the constraint's properties. An added method to the figure implementation gets a list of the constraints that are added and calls a custom factory that creates the appropriate decoration. A class diagram is shown in figure 5.11.

5.10.1 Connection Decorators in GMF

By default, a polyline connection is just a line. The line has some options that can be set, such as line type (eg. dotted, dashed etc). There is also built-in support for decorating the connection. The ends of a `PolylineConnection` can be decorated by specifying a `RotatableDecoration` for either the source, end or both. The `RotatableDecoration` interface is, as the name suggests, designed to allow the decoration figure to be rotated based on the position of a specified reference point. Because of this feature, the decoration figure stays aligned with the line it is decorating when the lines angle is changed.

A `RotatableDecoration` is in short an `IFigure` that can be rotated. The interface only has two methods, `setLocation(Point p)` which sets the location of the figure, and `setReferencePoint(Point p)` which sets a reference point that is used to determine the rotation angle. For a target decorator, the *location* is where the connection ends, and the *reference point* is either where the connection begins or a bendpoint (see figure 5.12).

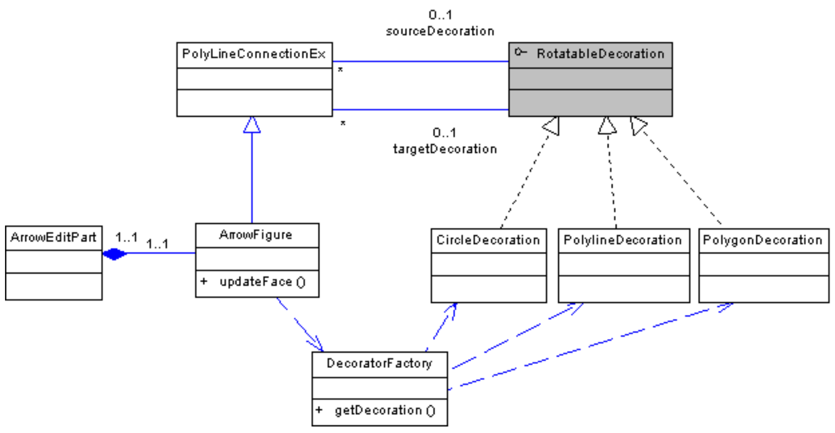


Figure 5.11: An ArrowFigure's updateFace() method is called, getting the appropriate decorator from DecoratorFactory

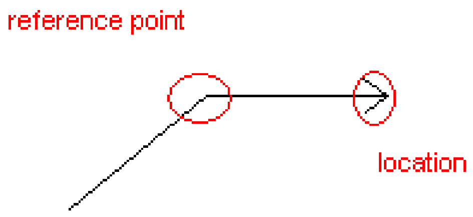


Figure 5.12: Location and Reference point in a connection

5.10.2 Simple geometric decorators

For simple geometric figures, making a decorator is relatively easy. There are already two classes in *draw2d* that does this, `PolylineDecoration` and `PolygonDecoration`. These takes a list of points as an argument and draws lines between the points. GMF also directly supports adding *template points* to a decoration, so all this can be done via GMF. For a simple open arrow-head, you would have to add the points {(-1,1), (0,0), (-1,-1)} to a `PolylineDecoration`. For a closed arrowhead, the same list of points applies, but the class used is `PolygonDecoration`. The last point is not needed, since the last point automatically connects to the first point. A *rhomb* (diamond) decorator is shown in figure 5.13. The decorations are not added to the end of the connections, it is actually "laid" over the line. So if the rhomb is not either filled or has a background color, the underlaying connection will show through the rhomb.

5.10.3 Circle Decorator

For more advanced decorations more work is needed. Although it is possible to make a circle out of points, it is not recommended. But first, the boundaries needs to be explained.

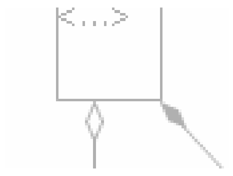


Figure 5.13: Rhomb decorator, filled and open

Each element in a diagram has a *boundary*, retrieved by the method `getBounds`, located in the `Figure` class from the *draw2d* package. This means that every element that inherits from `Figure` has this method. The method returns the smallest *rectangle* completely enclosing the figure. The reason for it is that if only one element has changed, there is no need to repaint the whole diagram. The boundaries helps GMF decide which part of the diagram to repaint.

So, for a circle, the bounds must encompass the whole of a circle, or else parts of it will not be painted correctly. The linewidth of the circle must also be taken into consideration. See code listing 5.2.

Listing 5.2: Calculating boundaries for a circle

```
public Rectangle getBounds() {
  if (bounds == null){
    int diameter = myRadius * 2;
    bounds = new Rectangle(
               myCenter.x - myRadius,
               myCenter.y - myRadius,
               diameter, diameter);
    bounds.expand(lineWidth / 2, lineWidth / 2);
  }
  return bounds;
}
```

Fortunately, draw2d already has an `Ellipse` class which can be used to make a circle. All the new class needs is some custom methods to set the radius of the circle and the width of the line used to draw the circle. The class also needs to implement `RotatableDecoration` to be accepted by GMF's `PolylineEx` class, which means that the `setLocation(Point)` and `setReferencePoint(Point)` methods must be implemented. The reference point can safely be ignored, as it makes no sense to rotate a circle. The `setLocation(Point)` is needed because it is used to determine the center of the circle. The `Ellipse` class handles the rest, such as the actual drawing of the circle (which uses `getBounds()` to determine the size of it.

5.10.4 Composed figures as decorators

Simple figures can be passed on as a list of points to `PolylineDecoration` or `PolygonDecoration` and generated automatically. But for a double arrowhead it isn't as easy, since there is no support for multiple lists of points.

The initial try on this was to have one list of points, draw the points and then shift the list of points by an amount, and then draw it again, leaving a double arrowhead. This approach means extending the original `PolylineDecoration` and overriding the method that does the drawing, `outlineShape(Graphics g)`. The code is listed in 5.3.

Listing 5.3: Outlining a shape

```
protected void outlineShape(Graphics g) {
  PointList points = getPoints();
  g.drawPolyline(points);
  points.performTranslate(-2, 0);
  g.drawPolyline(points);
}
```

The problem with this was that the rotation was not taken into account, and the second arrowhead was shifted two points to the right, not two points in the direction of the reference point. Also, the boundaries did not come out right. The second arrowhead was in some cases only partially visible. The solution for this was to add all the points to a list and then generate the bounds out of it with `PointList.getBounds()`, then manually adding the points to two separate lists of points, translate them using the reference point and then drawing the lines.

The requirements also called for a filled double arrowhead. The first solution was to do the same as with the double arrowhead as explained above, only using `PolygonDecoration` as a base.

These solutions are far from optimal as there is no way to change the points, all are static. A better solution for this particular problem is to build a more extensive path and go through GMF's own decorator setup. For a double arrowhead the points need to overlap, but this should not be a problem. The list of points is as follows: {(-1,1), (0,0), (-1,-1), (0,0), (-1,0), (-2,1), (-1,0), (-2,-1)}. The same approach also works for a filled double arrowhead, only the list must be extended more and closed.

5.10.5 Changing visualization on notification

Every EditPart has access to a `handleNotification` method. This method takes care of any incoming notifications from the elements that the EditPart is listening on. To have an arrow responding to a change in its constraints, this method has to be overridden and modified. Every notification is of the `Notification` type and has a reference to the object that has been changed, via the `getNotifier` method. By checking if the notifier is an instance of `ArrowConstraint` the appropriate method to change the visualization of the arrow figure can be called.

5.11 Adding Specialized Types

The mission now is to create a predefined Diagram. The diagram that is going to be created is a specialization of the standard `DiagramConstraint`, with Nodes and Arrows already included.

The *Element Type Registry* [10] provides a way for GMF to extend the metaclasses defined in EMF and provide specialized model elements for a GMF application. All the things that can be displayed, created, modified and deleted as logical elements in a GMF editor are described using *element types*. Each element type describes how its model element are to be displayed and how they are to be created, modified and destroyed, via edit helpers. There are two kinds of element types, *metamodel* types and *specialization* types. While a metamodel type corresponds directly to the `EClass` in the domains EMF model, the specialization class is used to extend the metamodel type. This is because it is not permitted [10] to define more than one metamodel type for a single `EClass` in a given client context. A specialization type can extend the editing behavior of the element type it specializes by its edit helper. While 'normal' element types has EditHelpers, specialization types have EditHelper *Advice*.

The default implementations of metamodel and specialization types are used to instantiate types that are registered through the `elementTypes` extension point. This registry helps GMF to find element types by their unique identifiers, their matching model object (or `EClass`) and the edit helper advice that applies to the object.

When editing an object, the object's editing behavior is defined by its EditHelper, which acts as a factory for edit commands. When a request to edit a model element is made, the response is to instantiate the commands that are defined by its EditHelper. The editing behavior for a specialized type is defined by its edit helper advice, and the specialized behavior decorates the default behavior for the metamodel type that it specializes. Which means that when an element is edited, advice from all of the specializations that match the object is contributed to the command.

GMF uses the extensible type registry in many areas. One area is the palette creation tools. A creation tool in the palette is associated with the element type that is to be created when the tool is selected. Other areas where the type registry is used are the Semantic Edit Policy, Icon Service (for providing metamodel types with icons) and the View Service.

To create a specialized type one has to add a `specializationType` to the `elementTypes` extension point. The `specializationType` contains a unique id, icon and name just as a normal `metamodelType`, but it also needs to know the id of the `metamodelType` it is specializing. The diagram editor generated by GMF also relies on a parameter called `semanticHint` (which the guide [10] does not mention) that has to be added as well. The `specializationType` also requires an EditHelperAdvice class, which is put in the `diagram.edit.helpers` package. The class needs to implement the `IEditHelperAdvice` interface, but since the `AbstractEditHelperAdvice` is already implementing it and provides a default implementation to it, it is best to extend it. To configure the element before its drawn, the best way is to override the `getBeforeConfigureCommand` method. The result of the method is of the type `ICommand` which means that it is required to build a custom command to return. The custom command is built by creating a new instance of `ConfigureElementCommand` and overriding the `doExecuteWithResult` method. In this method, the various elements that goes into the specialization type is added to the container and configured, and finally a `CommandResult.newOKCommandResult` is returned with a reference to the element that was configured.

When a creation tool is active, the edit commands for the editpart that the mousepointer is above will reflect whether or not the creation of the element is allowed. This behavior is defined by the editpart's editpolicy. The action of positioning a mousepointer above an editpart with a creation tool active will issue a request for the editpart's `getCreateCommand`. The request itself contains the ID of the element that is to be created, and the `getCreateCommand` will return the creation command (usually from the package `diagram.edit.commands`) if the creation of an element is legal; if the request is not legal it will return `null` and be reflected by changing the mousepointer to show that it is not allowed to put the element at that location.

In conclusion, when adding a specialization for an element there are several things that needs to be customized. The specialization type must be added to the extension point `elementTypes`, and it must be added to the `ElementTypes` provider in the `diagram.providers` package. In addition, an edithelper advice must be added, and the creation tool must be added to the palette via the editor's `paletteFactory`. To actually allow the creation of the specialized types, the editpolicy of the root diagram must be modified.

Chapter 6

Conclusion and further work

6.1 A summary of the results

The main function of this thesis is to evaluate the Eclipse Platform and point out the things that can be used and especially, *how* to use them. Also, how to customize the things that almost can be used. There are no doubts that GMF is well suited for making graphical editors. But the sheer complexity of the Eclipse Platform makes it extremely frustrating to start out with. There is a very steep learning curve for Eclipse plug-ins in general.

GMF is a relatively new framework, and as such, the documentation on it is not exceptional. The documentation that does exist deals mostly with beginner stuff, like how to create a simple diagram editor with the usual nodes and connections. When one are trying to expand the editor and create things that are not ‘standard’, there is not much documentation to speak of. There may be some clue as to how to almost do it hidden deep inside some of the provided examples, but that is if you are lucky. Another source of information is the GMF newsgroup, which has been very helpful. Apart from that, you are pretty much on your own. There has been a lot of trial and errors in this project, and countless hours adding debug breakpoints trying to figure out how GMF works.

I'm not saying that GMF is in any way bad, but it still has some issues.

The things that can be done:

- Creating a Graph consisting of Nodes and Arrows
- Assigning constraints to both Nodes and Arrows
- Changing visualization of Nodes and Arrows based on the constraints
- Creating Diagram constraints with a Shape
- Connecting a Node in the Graph with a Node in the Diagram

- Connecting an Arrow in the Graph with an Arrow in the Diagram
- Creating pre-defined Diagrams
- Creating customized tools
- Adding specialized element creation

Given more time, I think I would have spent it on trying to get a deeper understanding of the Eclipse Platform itself. Many of the features in GMF are standard features that are used throughout the entire Eclipse Platform, and many of the different approaches used are the standard ways of doing things in Eclipse. There is also the GEF framework to explore. When deciding what to start out out, I went for the GMF framework, only browsing through the EMF and GEF documentation as I went along trying to save time.

6.2 Further work

As this project has been a preliminary research on GMF and Eclipse Platform as a base for implementing the Diagrammatic Predicate Logic Framework there is understandable a lot of work left to do. The pure basics are shown and implemented as examples, but there exists no fully functional editor for the formalism as of yet. The following sections are some short commentaries on possible extensions.

Separate semantics from notation

At present time, the graph that is created in the diagram editor is created along with its own specification. The graph depends on having the constraints in the same diagram editor. The diagram editor should be able to load a predefined signature and make the graph based on it. To do this, a custom resource importer should be added, along with actions to load the diagram specification from outside the editor.

Generating palette items from a graph

When loading a specification, it would be nice to have some tools generated along with it. Instead of creating a default connection and then adding constraints to it, it would be better to have tools that did this automatically. But because of GMF's way of doing things, the creation of elements are bound to the `metaModelType` of the element, meaning that the element that is to be created must exist in the domain model of the diagram editor, and it's type must be specified in the plug-in manifest of the diagram editor.

One idea is to create new ECore elements by using the reflective EObject API [3], but that would require that GMF should be able to use dynamic EMF instead of generated code as the underlying diagram model. Such behavior is not yet supported by GMF, but it might get implemented in the future. There is an open request for such a feature at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=150177`, but the last note on it was in February 2008.

Another approach is to decouple to whole creation process in GMF, but that would take a substantial amount of time to do.

A third option could be to tinker with the code generation utility and the Jave Emitter Templates (JET) to add support for other inputs when generating the diagram editor. This will how ever not be very user-friendly.

Adding semantics to the diagram editor

At the moment there are no checks to decide if a diagram constraint that has been added is legal or not. This is of course not good, since the diagram constraint is important in DPL. Now there is only a visual connection between a node in a graph and a node in a diagram constraint. The graph portion should have some visuals that shows if the diagram constraint is not valid, or possible valid targets of elements, like the diagram constraint approach from Sketcher95. A possible example of how to do this is from the *review decorator* example that is provided with GMF. This plug-in adorns a note with an icon depending on the text inside the note. If the text is equal to 'passed' a green icon is put above the note, if the text is 'failed' a red icon is put up.

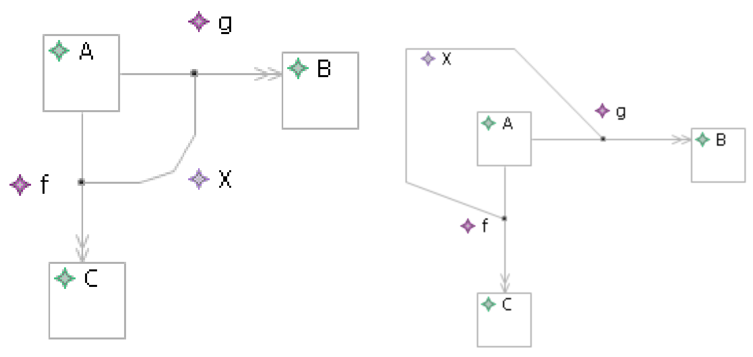
A more intuitive way of adding diagram constraints

Continuing on the previous section, at the moment a node in graph is linked with a node in a diagram constraint by connecting the two. A better solution would be to come up with something along the lines of Sketcher95's approach, by having some sort of wizard doing the job.

Arcs in diagram constraints

It is possible to create connections between connections in GMF. But the problem (as described in Section 5.9) is that connections are not considered semantical elements for some reason. Further study is needed on this part to discover why the edit commands are not gotten from the semantical edit policy and only the view is affected when removing the element normally.

There are also some issues with the visual of a connection between connections. GMF has a built-in obstruction-avoidance feature that makes connections try to take the "safest" route. This feature is shown in Figure 6.1. This is probably related to the `RoutingStyle` of the connection. According to the *GMF Newsgroup Q and A* available at `http://wiki.eclipse.org/GMF_Newsgroup_Q_and_A`, the routing style is applied in the View factory of the connection but I have not been able to get it to work properly.



(a) Desired result (b) The actual result

Figure 6.1: Connections between connections in GMF

Appendix A

Eclipse Setup

After numerous attempts with different “distributions” or whatever they call it, I have come to the conclusion that the simplest is to get Eclipse Classic. After that, use the built-in update manager to get GMF et al. By using the update manager at least you are ensured that the versions of GMF, EMF and GEF fit together. Most likely they don't anyway, but now you get someone else to blame but yourself. This is a good thing.

The next version of Eclipse (codenamed Ganymede) has a better update manager. Which is good, because from my experiences the one in the Europa version has a slight tendency to stall sometimes.

A.1 Where to find it all

Eclipse Classic `http://www.eclipse.org/downloads/`

EMF SDK `http://www.eclipse.org/modeling/emf/downloads/?project=emf` (XSD is not required at the moment; it is for XML Schema)

GEF and Draw2D SDK `http://www.eclipse.org/gef/downloads/`

GMF SDK `http://www.eclipse.org/modeling/gmf/downloads/`

SWT `http://www.eclipse.org/swt/` (Not really needed, but ok if you want to do small tests with graphical elements)

Trying to get some examples and SDK's is a whole different thing. The ones for EMF and GEF are fine I think, but the GMF SDK seems to have disappeared. My solution was to download the GMF SDK *after* installing the GMF plug-in from the Eclipse Updater. If you are not using the Eclipse Updater to install GMF first there will be some weird dependency problems and missing things, which means you have to install all of it again.

A.2 GMF Examples

The provided examples for the GMF project which can be installed in your workspace by selecting "New → Example..." were broken in my installation of GMF. It is possible to get new versions of all examples from the Eclipse CVS site. For more information about gaining access to the CVS, see `http://wiki.eclipse.org/index.php/CVS_Howto`

Although the examples on the CVS use the absolute latest versions of GMF (and dependencies) it is possible to get older versions that work with your GMF version by right-clicking the project → "Team.." → "Switch to another Branch/Tag".

Appendix B

Resources

Seeing as GMF has yet to release a book, here is a list of useful resources for dealing with GMF.

- `http://help.eclipse.org/help32/index.jsp` The section *GMF Developer Guide* has some very useful tutorials and various documentation on GMF. The latest version (3.3 at the time of writing) has some missing CSS files which makes some of the code examples very hard to read so its better to use the 3.2 version.
- `http://wiki.eclipse.org/GMF_Newsgroup_Q_and_A` Various questions that has been answered on the GMF newsgroup.
- `http://wiki.eclipse.org/GMF` The GMF wiki. The *GMF Documentation* link contains some examples (the mindmap example is pretty good) and the *Documentation Index* serves as an index for GMF documentation that is found in various locations.
- `http://www.eclipse.org/articles/` The Eclipse Corner Articles. Every article is written by members of the Eclipse development team or other members of the Eclipse community.
- *Eclipse Modeling Framework: a developer's guide* [3] — A book on EMF
- *Eclipse : building commercial-quality plug-ins* [5] — A book on creating plug-ins in Eclipse
- `http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html` A book on creating graphical editors using GEF and EMF. Provides a great overview on how GEF works.

Bibliography

[1] John Arthorne and Chris Laffra. *Official Eclipse 3.0 FAQs*. Addison-Wesley, 2004.

[2] Kent Beck. *Extreme programming explained : embrace change*. Addison-Weasley, 2000.

[3] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework: a developer's guide*. Addison-Weasley, 2003.

[4] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[5] Eric Clayberg and Dan Rubel. *Eclipse : building commercial-quality plug-ins*. Addison-Wesley, 2 edition, 2006.

[6] Edsger Dijkstra. Ewd 340: The humble programmer. *Communications of the ACM*, 10:859–866, 1972.

[7] The Eclipse Foundation. *GEF Programmer's Guide*. `http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html`[Online; accessed 23-May-2008].

[8] Nick Edgar, Kevin Haaland, Jin Li, and Kimberley Peter. Eclipse user interface guidelines, 2007. `http://wiki.eclipse.org/User_Interface_Guidelines`[Online; accessed 23-May-2008].

[9] José Luiz Fiadeiro. *Categories for Software Engineering*. Springer, 2005.

[10] Eclipse Foundation. Developer's guide to the extensible type registry. `http://help.eclipse.org/`[Online; accessed 22-May-2008].

[11] Eclipse Foundation. Tutorial: Configuring and extending the diagram palette, 2006. `http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.gmf.doc/tutorials/diagram/paletteConfigurationTutorial.html`[Online; accessed 22-May-2008].

[12] The Eclipse Foundation. Gmf tutorial. `http://wiki.eclipse.org/index.php/GMF_Tutorial`[Online; accessed 25-May-2008].

[13] The Eclipse Foundation. About the eclipse foundation, 2008. `http://www.eclipse.org/org/`[Online; accessed 23-May-2008].

[14] The Eclipse Foundation. Emf/faq, 2008. `http://wiki.eclipse.org/EMF-FAQ`[Online; accessed 23-May-2008].

[15] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language.* Addison-Weasley, 3 edition, 1995.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Weasley, 1995.

[17] Object Management Group. Meta object facility (mof) core specification, 2006. `http://www.omg.org/spec/MOF/2.0/`[Online; accessed 22-May-2008].

[18] Object Management Group. Meta object facility (mof) core specification, 2007. `http://www.omg.org/spec/XMI/2.1.1/`[Online; accessed 22-May-2008].

[19] Robert C. Martin. *Agile Software Development.* Prentice Hall, 2003.

[20] Bill Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* IBM RedBooks. IBM International Technical Support Organization, 2004. `http://publib-b.boulder.ibm.com/Redbooks.nsf/RedbookAbstracts/sg246302.html?OpenDocument`[Online; accessed 23-May-2008].

[21] Inc Object Technology International. Eclipse platform technical overview. Technical report, Eclipse Foundation, 2003. `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`[Online; accessed 1-May-2008].

[22] Frederic Plante. Introducing the gmf runtime. *Eclipse Corner Articles*, 2006. `http://www.eclipse.org/articles/Article-Introducing-GMF/article.html`[Online; accessed 23-May-2008].

[23] Adrian Powell. Model with the eclipse modeling framework, part 1: Create uml models and generate code. *IBM developerWorks*, 2004. `http://www.ibm.com/developerworks/opensource/library/os-ecemf1/`[Online; accessed 23-May-2008].

[24] Ørjan Hatland. Skethcer .net - a drawing tool for generalized sketches. Master's thesis, Department of Informatics, University of Bergen, June 2006.

[25] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Weasley, 1999.

[26] Adrian Rutle, Uwe Wolter, and Yngve Lamo. A formal approach to model transformations in software engineering. 2008. Proceedings, Wollic 2008, Submitted. Available at gs.hib.no.

[27] Adrian Rutle, Uwe Wolter, and Yngve Lamo. Generalized sketches and model driven architecture. Technical Report 367, Department of Informatics, University of Bergen, 2008. Presented at CALCO Young Researchers Workshop 2007.

[28] Ian Summerville. *Software Engineering*. Addison-Weasley, 8 edition, 2007.

[29] Sun MicroSystems. *JavaBeans(TM) API specification*. `http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html`[Online; accessed 25-May-2008].

[30] Uwe Wolter and Zinovy Diskin. The next hundred diagrammatic specification techniques: A gentle introduction to generalized sketches. Technical Report 358, Dept of Informatics, University of Bergen, July 2007.

[31] Uwe Wolter and Zinovy Diskin. Generalized sketches: Towards a universal logic for diagrammatic modeling in software engineering. 2008. Proceedings, ACCAT 2007, ENTCS, Accepted.