DPF Editor

A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse

Øyvind Bech

Master's Thesis in Informatics – Program Development

Department of Informatics
University of Bergen

HØGSKOLEN I BERGEN
Department of Computer Engineering
Bergen University College

May 2011

Contents

List of Figures

Preface

Foreword

This is submitted as my master's thesis in the Master's Degree programme in Informatics – Program Development. Joint degree between the University of Bergen and Bergen University College.

A note on personal pronouns

This work has (fortunately) not been carried out in a solitary vacuum. A not insignificant part of the research and development work has been done in co-operation with others. Not wanting to or being able to point out "who did what" at particular stages in the development process, I will be using the pronoun "we" when describing the active subject. When a decision or action was unambiguously my own, I have pointed this out by describing myself as "the author".

Acknowledgements

I would like to thank my supervisor, Yngve Lamo, for great patience and support. Also, Adrian Rutle deserves extra thanks for invaluable support and input on development. My co-student Dag Viggo Lokøen has been a great discussion partner and co-developer, contributing solidly to the design and implementation of DPF Editor. I would also like to thank all the other members of the DPF group at HiB and UiB: Uwe Wolter, Florian Mantz, Alessandro Rossini, Anders Sandven, and Suneetha Sekhar.

For proofreading, general input on structural matters, and unparalleled philosophical discussions, special thanks go to my good friend Nils Torvald Østerbø. I also would never have started on this project, nor finished it, without the absolute support of my marvellous wife, Ellen. Thank you!

Bergen, 31st May 2011

Chapter 1

Introduction

This thesis is part of an ongoing project at Bergen University College and the University of Bergen started in early 2006, called *Diagram Predicate Framework* (DPF). Covered here is the design, construction and evaluation of a model (specification) editor for the Eclipse platform, based on DPF.

1.1 Motivation

In modern life, models are ubiquitous. Architects and designers visualize buildings and building projects by use of models, engineers model construction projects as a matter of routine. The goal of all this activity is often to get to grips with complex subject matter through *abstraction*: simplification and concretization of some desired aspects of a system under study in a specific domain. This process of abstraction – removing details from a system and considering only what is perceived as the essential features – has also been a driving force behind the development of software engineering as a profession. Since the invention of the programmable computer, progressively more high-level computer languages have come to dominate software engineering. Parallel advancements have been made regarding *modelling*; diagrammatic modelling languages, techniques and tools have been continuously developed to answer to the needs of the software industry.

Perhaps inevitably, the synthesis of text-based programming languages and diagrammatic modelling methodologies has become an objective in several contexts, none more so than in *Model-driven engineering* (MDE). Within this engineering paradigm, the models themselves have become the primary software artefacts, and the ultimate goal is to automatically generate working software from pure diagrammatic (model) representations of domain knowledge.

Hopefully, having modelling tools and notations that possess proper semantics can facilitate this goal. Being based on a strong mathematical foundation, DPF promises to be able to define diagrammatic modelling languages that possess stringent, non-ambiguous semantics.

In this thesis, we will argue that the general industry practice of mixing diagrammatic modelling languages with textual constraint languages such as OCL, leads to added (meta)model complexity. This complexity in turn puts demands on developers and domain experts. It also leads to synchronization issues when models are evolved, as well as making it technically difficult to preserve constraints in model transformations. Our proposed solution to these issues is a multi-layer model editor based on the DPF. DPF provides us with stringent semantics as well as diagrammatic constraint definitions, bypassing the need for separate metamodelling hierarchies for models and constraints respectively.

In view of this, and based on previous development efforts as well as analysis of existing tools, we will try to answer the following research question: *Is it possible to construct a DPF specification editor supporting metamodelling at an arbitrary number of meta-levels?*

We will try to answer this question by designing and implementing a model editor for the DPF formalism, based on concepts adapted from the previous efforts and containing functionality based on the DPF. Hopefully, this will be the first of many steps towards a fully-fledged software suite based on DPF that can be applied for research purposes as well as in educational and industrial scenarios.

1.2 Structure of thesis

In addition to this introductory chapter, the thesis has been laid out as follows:

In **Chapter 2: Research method**, we give a short presentation of the research method on which this thesis is based and briefly describe how this method has been applied to the thesis.

In **Chapter 3: Background**, we introduce the Model-driven engineering methodology (MDE). We also introduce the Diagram Predicate Framework (DPF) and relate this framework to MDE concepts. Lastly, we discuss metamodelling, both as a concept within MDE and in relation to DPF.

In **Chapter 4: Previous efforts and current situation**, we discuss the previous efforts towards a working editor for DPF, pointing out their strengths and weaknesses, particularly in relation to our own project. We also discuss the current situation regarding a DPF tool in particular and metamodelling tools in general. Lastly, we formulate the requirements for a DPF specification editor and outline the general problem description for our thesis.

In **Chapter 5: Technology platform and methodology**, we describe our choice of technology and how this creates a technological context for our research. We also briefly discuss our choice of development methodology.

In **Chapter 6: Design and development**, we outline the design and development of our tool. This includes the process of naming both the tool and its individual components, the overall architecture of the solution, and the individual packages that make up this architecture. Further, we describe the editor and its (sub)package structure in more detail. Then, we discuss the implementation of metamodelling capabilities, typing of a diagram's graphs,

and semantics validation. Lastly, we discuss some of the process-related issues we encountered during the development phase.

In **Chapter 7: Tool demonstration**, we show the tool in use on a simple example application. This gives a brief demonstration of our tool's capabilities, employing both its metamodelling functionality as well as its ability to validate semantics based on a metamodel's constraints. The demonstration serves as a validation of the tool's capability to perform a modelling task based on the DPF.

In **Chapter 8: Concluding matters**, we conclude on our tool's capabilities, in relation to the DPF as well as previous efforts. We also describe some possible avenues for further work, both directly and indirectly related to our tool.

1.2.1 A note on definitions

All definitions and diagrams given in chapter 3 of this thesis, directly relating to DPF, are taken from Rutle [62]. DPF is still a work in progress, and definitions given in other articles and presentations may differ slightly in form and/or content.

Chapter 2

Research method

In this chapter, we give a short presentation of the research method on which this thesis is based and briefly describe how this method has been applied to the thesis.

Computer science is a young discipline, where no single unified research method has been established [10, 31]. Building on resources from – among other fields – logic, mathematics, physics, engineering, social sciences, and linguistics, researchers in computer science have had to adapt research methods from other disciplines. It is not even entirely obvious whether computer science qualifies as 'science' in a traditional definition of the term [13, 12].

Denning et al. [10] point out that there are three major paradigms, or cultural styles, by which research activity is typically carried out within computer science. They label these styles *theory* (as rooted in mathematics), *abstraction* (as rooted in the experimental scientific method), and *design* (as rooted in engineering). Making a similar taxonomy, Glass [31] extracts the (general) scientific method as a separate approach, and labels the three remaining methods *analytical*, *empirical* and *engineering*. Tedre [73], calling the paradigms traditions, follows this pattern and makes the case that "The variety of research approaches within and among those traditions [in computer science] might bring about ontological, epistemological and methodological confusion" [73, page 107]. In light of this, it becomes important to conduct computer science research solely within one of the mentioned paradigms. According to Tedre, "It is notoriously difficult to conduct research in the intersection of research traditions without making a mess of it" [73, page 107].

Solheim and Stølen [71] claim that a large part of computer science research can be labeled *technological research*. This type of research is contrasted to what is labeled "classical research", i.e. the approach usually called the scientific method (see table 2.1). In technological research, the focus is on creating new or improved *artefacts*. Such artefacts can for instance be new programming languages, security protocols, methods, or – as in our case – programs. Technological research is carried out in an iterative process, containing the three steps *problem analysis*, *innovation*, and *evaluation*.

We recognize the research carried out in relation to this thesis as technological research, and have carried out the research and development in accordance with the method's steps in the following manner:

Problem analysis: Initially, we investigated existing solutions and products in order to evaluate to what extent these could provide a basis for our artefact. We concluded that no single product or application fulfilled our need for a multi-layer diagrammatic model editor, functioning as a plug-in to the Eclipse platform. New research and development was needed, building on the theoretical framework provided by the Diagram Predicate Framework. This is discussed in chapter 4.

Innovation: The innovation part of this thesis work has been to create DPF Editor, a multi-layer diagrammatic model editor for Eclipse. The choice of technology is described in chapter 5, while the artefact itself and the process of constructing it is described in chapter 6. Our claim is that the finished artefact, DPF Editor, satisfies the requirements set forth in section 4.3.

Evaluation: In this thesis work, we have evaluated our artefact against the requirements set forth in section 4.3. Most aspects of how the artefact satisfied these were evaluated informally during the development phase, and fed back to requirements during iterations. Finally, two more elaborate case studies were made in order to evaluate whether DPF Editor was capable of fulfilling the given specification. One of these is described in chapter 7, while another can be found in [5].

In chapter 7, we also briefly discuss further approaches for evaluating the finished artefact.

	Classical research	**Technological research**
Problem	Need for new theory	Need for new artefact
Solution	New explanations (new theory)	New artefact
Solution should be compared to...	Relevant part of the real world	Relevant need
Overall hypothesis	The new explanations agree with reality	The new artefact satisfies the need

Table 2.1: Comparison of classical and technological research. From [71].

Chapter 3

Background

In this chapter, we introduce the Model-driven engineering (MDE) methodology. We also introduce the Diagram Predicate Framework (DPF) and relate this framework to MDE concepts. Lastly, we discuss metamodelling, both as a concept within MDE and in relation to DPF.

3.1 Model-driven engineering

Model-driven engineering (MDE)[1] is a software development methodology which emphasize the use of models as the *primary artefacts* in the development process [29]. This implies that software developers working within this paradigm should be able to automatically generate information systems directly from models, without first going through the step of writing (text-based) computer code. The goal is thus to move away from a code-centric approach towards a model-centric approach, thereby separating business logic from implementation details and getting domain experts more directly involved in the development process.

Unfortunately, no single definition of 'model' in the context of MDE has gained industry-wide acceptance. Although similarly themed, many different definitions exists. The challenge seems to be to achieve a definition that is wide enough, but still retains enough substance for practical use [48].

Gonzalez-Perez and Henderson-Sellers [33, page 1779] state: "From a simplistic point of view, we could say that a model is *a statement about a given subject under study* (SUS), *expressed in a given language*."[2] This definition is similar to one given by Seidewitz [67, page 1] ("a set of statements about some system under study"), but it includes an explicit reference to the *language* of the model.

Gonzalez-Perez and Henderson-Sellers [33, page 1779] go on to say: "we can say that the major reason that we need models for is to reason about the

[1] The term MDE is not used universally, 'model-driven development' (MDD) and 'model-driven software development' (MDSD) are also frequently used.

[2] The SUS (the modelled artefact) is given many designations in the literature: 'original', 'application domain', 'real system', and 'subject under study' are frequently used.

complexity of the SUS without having to deal with it directly [...] As a result, a suitable model would have to exhibit the appropriate structure for it to be useful. For this reason, we prefer to say that, for a statement about an SUS (expressed in a given language) to be a model, it needs to be homomorphic with the SUS that it represents."

This ties in with STACHOWIAK'S requirements(cited in [48]) that a model needs to possess three features (table 3.1).

Mapping feature	A model is based on an original
Reduction feature	A model only reflects a (relevant) selection of the original's properties.
Pragmatic feature	A model needs to be usable in place of the original with respect to some purpose.

Table 3.1: Model features according to Stachowiak [48]

Furthermore, models can be categorized as *prescriptive* or *descriptive* [33]. This reflects the role a model plays in relation to the SUS. A prescriptive model, created before the SUS, will act as a specification towards the SUS. A descriptive model will function as a documentation of the SUS. In MDE, models can also have both features, being used for both prescriptive and descriptive purposes. Changes in the SUS are propagated to the model and *vice versa*. The process where changes in a model are propagated to the SUS as the model evolves is often described as either *(meta)model evolution* or *application evolution* in the literature [72].

Model-driven architecture

The term Model-driven architecture (MDA) is often referenced in the literature. MDA is a reference implementation of MDE, specified by the OBJECT MANAGEMENT GROUP (OMG) [56]. Central to this standard is the notion of a platform-independent model (PIM), a model independent of any implementation technology [45]. The PIM models the system from a business-centric view.

By use of *model transformations*, the PIM is transformed into a platform specific model (PSM). This model corresponds to some existing technology layer, for instance EJB or Microsoft .NET. MDA requires one such PSM for each technology platform. The final step is to transform the PSM into runnable code [45].

In general, a model transformation is a set of rules, which specifies the way (part of) one model can be used to define (part of) another model. (See also section 3.3.4.)

3.2 Diagram Predicate Framework

Diagram Predicate Framework (DPF) is a research project initiated in 2006 by the Bergen University College (HiB) and the University of Bergen (UiB), Norway. The project involves several researchers from Norway and Canada [14].

Historical note: DPF has previously been branded as *Generalized Sketches* (GS) and *Diagrammatic Predicate Logic* (DPL). Earlier articles may use this nomenclature.

DPF is a graph-based specification format that takes its main ideas from both categorical and first-order logic (FOL), and adapts these concepts to software engineering needs. While in FOL the arity of a predicate is given by a collection of nodes only, the arity of a predicate in DPF is given by a graph, i.e. a collection of nodes together with a collection of arrows between nodes. Besides, the main difference between FOL and DPF is that FOL is an "element-wise" logic, i.e. variables varies over elements of sets. In contrast, DPF uses a "sort-wise" logic where variables vary over sets and mappings [63].

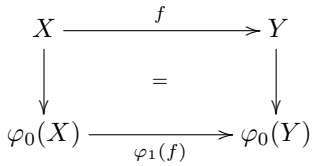
DPF aims to be a completely diagrammatic specification framework for MDE. The claim behind DPF is that any diagrammatic specification technique in software engineering can be viewed as a specific instance of the DPF specification pattern. DPF is a pattern, i.e. generic, in the sense that we can instantiate this pattern by a signature (see definition 3, below) that corresponds to a specific specification technique, like UML class diagrams, ER diagrams or XML [66].

The concepts of *graphs* and *graph homomorphisms* are essential to diagrammatic modelling in general and to DPF in particular. Following Rutle [62], we define these terms as the following:

Definition 1 (Graph) *A graph $G = (G_0, G_1, src^G, trg^G)$ is given by a collection G_0 of nodes, a collection G_1 of arrows and two maps $src^G, trg^G : G_1 \to G_0$ assigning the source and target to each arrow, respectively. We write $f : X \to Y$ to indicate that $src(f) = X$ and $trg(f) = Y$.*

(Note that this definition allows for multigraphs. In this type of graph, two vertices may be connected by more than one edge.)

Definition 2 (Graph Homomorphism) *A graph homomorphism $\varphi : G \to H$ is a pair of maps $\varphi_0 : G_0 \to H_0$, $\varphi_1 : G_1 \to H_1$ which preserve the sources and targets; i.e. for each arrow $f : X \to Y$ in G we have $\varphi_1(f) : \varphi_0(X) \to \varphi_0(Y)$ in H, such that the following diagram commutes:*



This gives a *structure-preserving* mapping between two graphs.

Arrows can also be *composed*. Following FIADEIRO'S [27, page 20] definition of Categories, we will denote the composition of arrows by using semicolons (;). For instance, the composition of arrows f and g will be denoted $f; g$.

3.2.1 DPF Specifications

DPF models are represented by (diagrammatic) specifications. The syntax of these specifications is given by the following definitions as stated by Rutle [62]:

Definition 3 (Signature) *A (diagrammatic predicate) signature* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ *consists of a collection of predicate symbols* Π^Σ *with a map* α^Σ *that assigns a graph to each predicate symbol* $\pi \in \Pi^\Sigma$. $\alpha^\Sigma(\pi)$ *is called the arity of the predicate symbol* π.

Table 3.2 shows a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ used for object-oriented modelling.

Definition 4 (Atomic Constraint) *Given a signature* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, *an atomic constraint* (π, δ) *added to a graph* S *is given by a predicate symbol* π *and a graph homomorphism* $\delta : \alpha^\Sigma(\pi) \to S$.

Definition 5 (Specification) *Given a signature* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$, *a (diagrammatic) specification* $\mathfrak{S} = (S, C^\mathfrak{S} : \Sigma)$ *is given by a graph* S *and a set* $C^\mathfrak{S}$ *of atomic constraints* (π, δ) *on* S *with* $\pi \in \Pi^\Sigma$.

Definition 6 (Type Graph and Typing Morphism) *A type graph is a distinguished graph* $TG = (TG_0, TG_1, src^{TG}, trg^{TG})$. *A typed graph* (G, ι) *which is typed by* TG *is a graph* G *together with a graph homomorphism* $\iota : G \to TG$. *The homomorphism* ι *is called a typing morphism.*

3.2.2 Instances and semantics

In DPF, the semantics of a predicate π is given by the set of its instances, $\iota : O \to \alpha(\pi)$ where each ι is a graph homomorphism into the arity of the predicate [62]. These semantics can be defined in different ways, according to what is suitable. In table 3.2, we have used set theoretical definitions to denote the semantic interpretation for each predicate. In a model editor, one could implement a validator for each predicate. Following Rutle, we define the semantics of predicates and the instance of a specification [62]:

Definition 7 (Semantics of Predicates) *A semantic interpretation* $[\![..]\!]^\Sigma$ *of a signature* $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ *is given by a mapping that assigns to each* $\pi \in \Pi^\Sigma$ *a set* $[\![\pi]\!]^\Sigma$ *of graph homomorphisms* $\iota : O \to \alpha^\Sigma(\pi)$ *called valid instances of* π, *where* O *may vary over all graphs.*

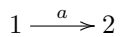
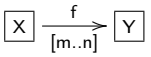
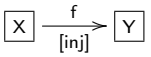
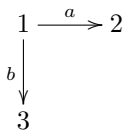
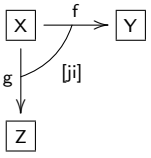
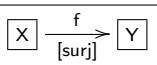
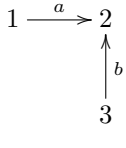
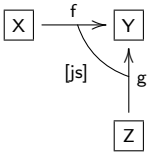
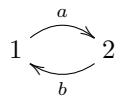
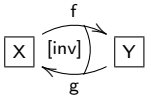
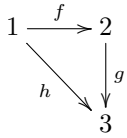
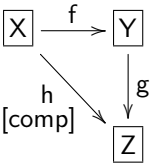
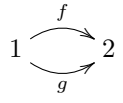
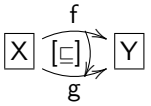
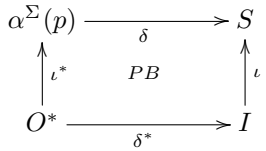
π	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
[mult(*m*,*n*)]			$\forall x \in X : m \leq \lvert f(x)\rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[injective]			$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
[jointly-injective]			$\forall x, x' \in X : f(x) = f(x')$ and $g(x) = g(x')$ implies $x = x'$
[surjective]			$\forall x \in X : \bigcup\{f(x)\} = Y$
[jointly-surjective]			$\forall x \in X, \forall z \in Z : \bigcup\{f(x) \cup g(z)\} = Y$
[inverse]			$\forall x \in X$, $\forall y \in Y : y \in f(x)$ iff $x \in g(y)$
[composition]			$\forall x \in X : h(x) = \bigcup\{g(y) \mid y \in f(x)\}$
[image-inclusion]			$\forall x \in X : f(x) \subseteq g(x)$

Table 3.2: A sample signature $\Sigma = (\Pi^{\Sigma}, \alpha^{\Sigma})$

Definition 8 (Instance of Specification) *Given a specification* $\mathfrak{S} = (S, C^{\mathfrak{S}}: \Sigma)$*, an instance* (I, ι) *of* $\mathfrak{S}$ *is a graph* I *together with a graph homomorphism* $\iota : I \to S$ *such that for each constraint* $(\pi, \delta) \in C^{\mathfrak{S}}$ *we have* $\iota^* \in [\![\pi]\!]$*, where* $\iota^* : O^* \to \alpha^{\Sigma}(p)$ *is given by the following pullback diagram*



Quote [62, page 37]: "To check that a constraint is satisfied in a given instance of $\mathfrak{S}$, it is enough to inspect only the part of $\mathfrak{S}$ which is affected by the constraint."

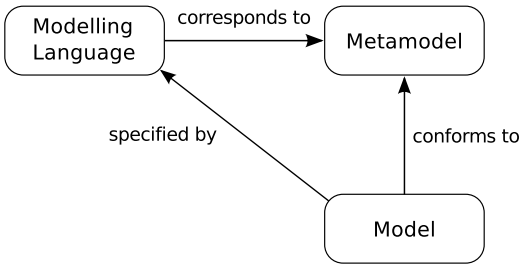


Figure 3.1: The figure illustrates the relationships between a model, its metamodel, and the modelling language that is defined by the metamodel. Adapted from [62].

3.3 Metamodelling

Metamodelling is a special kind of modelling: the resulting metamodels are models themselves. However, as with the term 'model', there does not seem to be any industry-wide consensus on these definitions [48]. Following Gonzalez-Perez and Henderson-Sellers, we will define the term metamodelling as "the act and science of engineering metamodels" [34, page 18]. Since the prefix 'meta' suggests that modelling has been performed twice, a metamodel can therefore be called a "model of models".

What makes this a special subject matter in the context of MDE is the fact that the models and metamodels are "made from the same stuff": they are of the same conceptual type. (This introduces a recursive flavour to the process and makes distinction between different model layers somewhat tricky. Consider for example the phrase "the Class class in UML". Does this refer to the Class class in MOF[3] or the Class class in the UML metamodel? Are those classes the same? And what does 'the same' mean? Answering questions such as these can often be non-trivial.) [34]

Metamodels can be viewed as (domain specific) *language definitions*. Metamodelling then becomes a mechanism for defining graphical modelling languages which is used in the way grammars, for instance given in Backus–Naur Form (BNF), are commonly used to define text-based languages such as programming languages [45].

In MDE, a metamodel will define a new modelling language and this language can in turn be used for repeated modelling, creating a model that *conforms to* or is an *instance of* its metamodel. We say that a metamodel defines the abstract syntax for its instances, and that any model that conforms to its corresponding metamodel is syntactically correct as defined by the modelling language [62] (figure 3.1).

This is a recursive process that gives rise to what is commonly called a *metamodelling hierarchy*. In a metamodelling hierarchy, the model at each layer, or meta-level, is at at a higher level of abstraction than the one below. The common way to terminate this hierarchical recursion is to define the "final" or "uppermost" metamodel layer in terms of itself, making it a *reflexive layer* (figure 3.2).

[3]See below

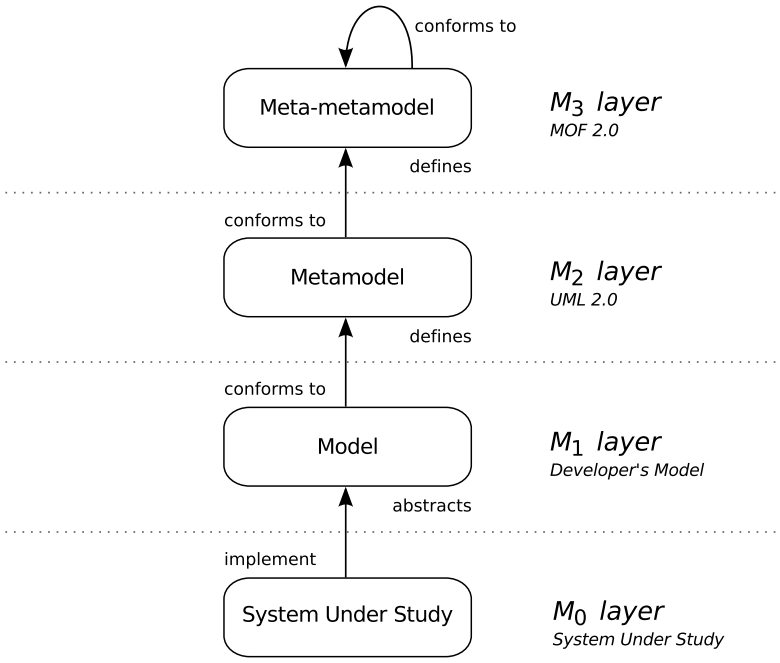


Figure 3.2: OMG's 4-layered hierarchy illustrating (meta)modelling languages and their corresponding metamodels. Adapted from [62] and [72]. Note how the MOF 2.0 layer is a reflexive layer, defined in terms of itself. Also note that the annotation is not standardized: some publications number the layers downwards, starting with the M_0 layer at the top.

The most widely used modelling language in MDE – and information system modelling in general – is the Unified Modeling Language (UML), developed by the OMG and currently in version 2.3 [57]. This language is designed as a four-layered architecture. The top layer, M_3, is defined in terms of the Meta Object Facility (MOF), which provides a meta-metamodel. This M_3-model defines the language used to build the UML metamodel (M_2). The various diagram standards of UML, such as Class, Object, and Sequence diagrams, are defined in this M_2 language. This is the layer most developers interact with, doing modelling at the M_1 layer, using CASE tools or just drawing models by hand [72].

3.3.1 Why do metamodelling?

In relation to modelling tools, there are two main reasons for doing metamodelling. First, without giving modelling languages a well-defined syntax, tool construction will be difficult. Likewise, automated reasoning over models will be hard to achieve. Second, the process of model transformation, a central technique of MDA (and MDE), depends on unambiguously defined models: Model transformation tools and the set of rules that constitute a model transformation are both generally created in terms of metamodels [45].

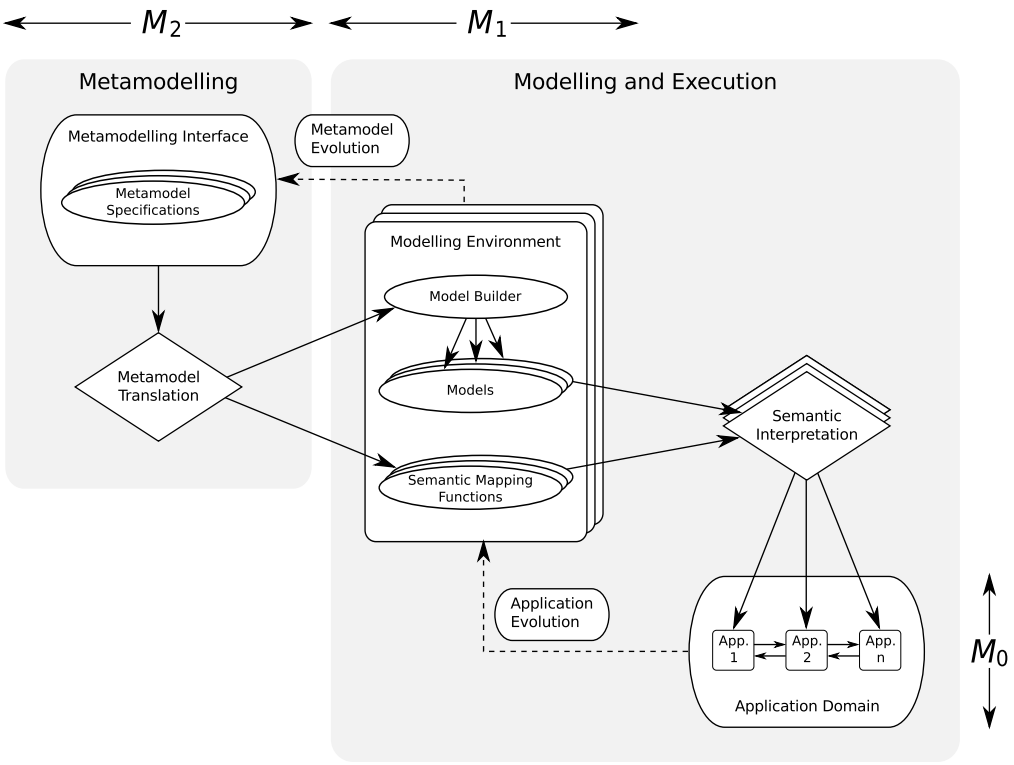


Figure 3.3: The metamodelling design process. Adapted from [72]. The figure describes the main elements that go into the process. At the M_2 layer, the toolsmith, or language designer, using a metamodelling interface, describes the model. The modelling environment is the "traditional" modelling environment here shown in interaction with a SUS, or application domain. Note how model evolution can occur at both meta-levels M_2 and M_1.

3.3.2 Domain-Specific Languages

The concept of a *Domain-Specific Language* (DSL) is also often introduced as a feature of metamodels. The metamodelling process opens up for designing specialized modelling languages for specific modelling tasks (figure 3.3). These languages are contrasted to more general-purpose languages, such as UML. Sometimes this concept is the main reason for doing metamodelling, other metamodelling tasks may be more general in nature. However, the specificity of a DSL or the generality of a general-purpose language is a relative factor, determined by the language designer [38].

In contrast to general-purpose modelling languages such as UML, a DSL gives the developer the ability to customize not only the (end-user) application to a particular domain, but also the *tools* used to make the application. According to Gronback [38, page 6], "...in the process of creating, maturing and extending your DSL or family of DSLs, you might end up with something akin to UML. The difference is that you're using your organization's family of models, transformation definitions, and generation facilities, which are tailored to your exact needs."

3.3.3 Constraints

Constraints are an intrinsic part of most modelling languages, augmenting the basic modelling tool palette for specifying business rules and providing more expressive power for technical requirements. When analyzing traditional modelling languages such as UML, we seek to make the distinction between *structural* or *built-in* constraints and *attached* constraints. Constraints considered structural are defined in the metamodel. These built-in constraints usually control such traits as multiplicity and uniqueness. Attached constraints are defined *outside* the metamodel. This type of constraint is usually defined with some textual constraint language such as the Object Constraint Language (OCL). Attached constraints usually control complex business rules and requirement features of a model [64, 62].

In MDE, attached constraints are generally applied through the use of textual constraint languages [61]. For instance, in MDA, the OMG recommends the combination of UML and OCL for the definition of platform independent models (PIMs) [56]. However, this general practice of mixing diagrammatic modelling languages with textual constraint languages may introduce several problems when applied in a metamodelling scenario.

First, the introduction of an additional language is an inconvenience, perhaps only slightly for the developer, but all the more for the domain expert, who now has to master *two* specification languages in order to communicate successfully with the developer.

Second, the tool designer has to take this into account, having to automate reasoning over both the graph-based structures of a MOF-based (or any diagrammatic) modelling language as well as over the expressions from an separate, attached constraint language such as OCL. Some very troublesome synchronization issues, related to changes made in constrained models, seem to arise from this situation [62].

3.3.4 Constraints and model transformations

As mentioned, model transformations are integral to MDE. They are used "everywhere", for instance facilitating code generation, refinement of models, refactoring, adaptation, and evolution of models and model hierarchies. A model transformation generates a target model from a source model using (at least) one model and a *transformation definition*. Generally, a transformation definition consists of one or more rules. As metamodels provide the modelling languages for both the source and target models, each rule describes how one or more constructs in the source language can be transformed into one or more constructs in the target language [45, 62].

There also exists an inherent problem regarding attached constraints and (rule-based) model transformations: As attached constraints are often defined in a textual language, the transformation rules – being defined in the abstract syntax of (diagrammatic) metamodelling languages – often just ignore the attached constraints. In doing so, information can become lost in the transformation process. The developer has to correct for this, possibly through the process

of manually entering additional code. Again we encounter a conflict with the MDE goal of pure model-based development. According to [64, page 14]: "This problem is closely related to the fact that the *conformance* relation between models and metamodels is not formally defined for MOF-based modelling languages, especially when OCL constraints are involved."

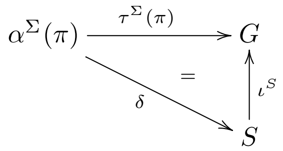
DPF seeks to remedy these issues by bypassing the notion of attached constraints altogether. Structural and attached constraints have thus been integrated in diagrammatic specifications. These (integrated) constraints are now defined by means of predicates that belong to a predefined signature. In this way, constraints are defined diagrammatically, and treated uniformly [62].

3.3.5 Metamodelling in DPF

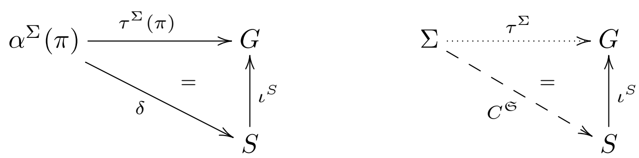
In DPF, the concept of metamodelling is tightly connected to the concept of a *modelling formalism*, which can be regarded as a modelling language. In order to reach the definition of a modelling formalism, we first need to define *typed signatures*, *typed constraints*, and *typed specifications* [62]. Furthermore, to define *conformant specifications*, we will need the definition of the Category of Instances, **Inst**. Informally, **Inst**($\mathfrak{S}$) is a category containing all instances (I, ι) of $\mathfrak{S}$. A formal definition is given by Rutle [62, page 40].

Definition 9 (Typed Signature) *A signature typed by a graph G is a signature $\Sigma = (\Pi^\Sigma, \alpha^\Sigma)$ together with a map τ^Σ assigning to each predicate $\pi \in \Pi^\Sigma$ a graph homomorphism $\tau^\Sigma(\pi) : \alpha^\Sigma(\pi) \to G$. $\tau^\Sigma(\pi)$ is called the typing of π. We use $\Sigma \triangleright G$ or $((\Pi^\Sigma, \alpha^\Sigma) \triangleright_{\tau^\Sigma} G)$ to denote a signature Σ typed by G.*

Definition 10 (Typed Atomic Constraint) *Given a typed signature $((\Pi^\Sigma, \alpha^\Sigma) \triangleright_{\tau^\Sigma} G)$, a typed atomic constraint (π, δ) added to a typed graph (S, ι^S) with $\iota^S : S \to G$ is given by a predicate symbol π and a graph homomorphism $\delta : \alpha^\Sigma(\pi) \to S$ such that $\delta; \iota^S = \tau^\Sigma(\pi)$.*



Definition 11 (Typed Specification) *Given a graph G and a typed signature $\Sigma \triangleright G$, a specification typed by the graph G is a specification $\mathfrak{S} = (S, C^{\mathfrak{S}}{:}\Sigma)$ together with a typing graph homomorphism $\iota^S : S \to G$ assigning to each element of S a type in G such that $\forall(\pi, \delta) \in C^{\mathfrak{S}} : \delta; \iota^S = \tau^\Sigma(\pi)$. We write $\mathfrak{S} \triangleright G$ or $((S, C^{\mathfrak{S}}{:}\Sigma) \triangleright_{\iota^S} G)$ to denote a specification $\mathfrak{S}$ typed by G.*



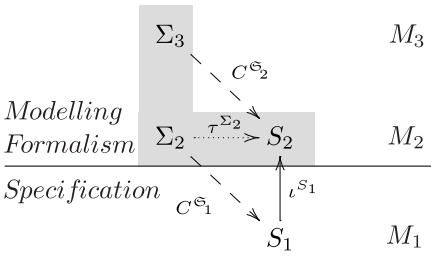
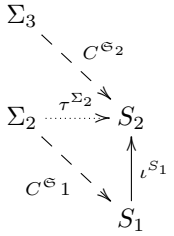


Figure 3.4: Modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ together with a specification $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$. The figure also displays the alignment with the metamodelling hierarchy from OMG. From [62]

Definition 12 (Conformant Specification) *Let* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2} : \Sigma_3)$ *be a specification with* $\Sigma_3 = (\Pi^{\Sigma_3}, \alpha^{\Sigma_3})$*, and* $\Sigma_2 \rhd S_2 = ((\Pi^{\Sigma_2}, \alpha^{\Sigma_2}) \rhd_{\tau^{\Sigma_2}} S_2)$ *a signature. A typed specification* $\mathfrak{S}_1 = ((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \rhd_{\iota^{S_1}} S_2)$ *conforms to* $\mathfrak{S}_2$ *iff* $(S_1, \iota^{S_1}) \in \mathbf{Inst}(\mathfrak{S}_2)$*. We write* $\mathfrak{S}_1 \blacktriangleright \mathfrak{S}_2$ *or* $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$ *to denote a specification* $\mathfrak{S}_1$ *conformant to* $\mathfrak{S}_2$*.*



We now have two kinds of specifications, *typed* and *conformant*. The difference lies in the fact that a conformant specification S1, in addition to be typed by a graph also satisfy the constraints of another specification, S2. This leads us to the notion of a modelling formalism [62]:

Definition 13 (Modelling Formalism) *A modelling formalism* $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$ *is given by two signatures* $\Sigma_2 = ((\Pi^{\Sigma_2}, \alpha^{\Sigma_2}) \rhd_{\tau^{\Sigma_2}} S_2)$ *and* $\Sigma_3 = (\Pi^{\Sigma_3}, \alpha^{\Sigma_3})$ *and a specification* $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2} : \Sigma_3)$ *which is called the corresponding metamodel of the modelling formalism.*

The modelling formalism (figure 3.4) is the key to achieving a metamodelling mechanism in DPF. "In a modelling formalism $(\Sigma_2 \rhd S_2, \mathfrak{S}_2, \Sigma_3)$, predicates from the signature Σ_3 are used to add atomic constraints to the metamodel $\mathfrak{S}_2$. This corresponds to metamodel definition. These constraints should be satisfied by all specifications $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$. [...] Moreover, predicates from the signature $\Sigma_2 \rhd S_2$ are used to add constraints to typed and conformant specifications; i.e. $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \rhd_{\iota^{S_1}} S_2)$ and $((S_1, C^{\mathfrak{S}_1} : \Sigma_2) \blacktriangleright_{\iota^{S_1}} \mathfrak{S}_2)$, respectively. This corresponds to model definition. These constraints should be satisfied by instances of these specifications." [62, page 53] In this fashion, the constraints cover the functionality of both structural and attached constraints as described in section 3.3.3 above.

To enable the termination of the metamodelling hierarchy, DPF also introduces a reflective modelling formalism. This formalism specifies its own metamodel, thereby terminating the metamodelling hierarchy [62, page 54].

In this manner, DPF provides us with the foundation to construct a diagrammatic (metamodelling) system with an arbitrary number of meta-levels and with all constraints contained in one closed formalism.

Chapter 4

Previous efforts and current situation

In this chapter, we will discuss the previous efforts towards a working editor for DPF, pointing out their strengths and weaknesses, particularly in relation to our own project. We also discuss the current situation regarding a DPF tool in particular and DSL/metamodelling tools in general. Lastly, we formulate the requirements for a DPF specification editor and outline the general problem description for our thesis.

4.1 Previous efforts

So far, three separate efforts have been made to produce drawing tools or model editors directly related to the DPF formalism. We will consider these efforts in turn.

4.1.1 Sketcher95

Sketcher95[1] is a tool for drawing Generalized Sketches. It is a drawing tool only, with no support for code generation or model transformations. The first version of Sketcher95 was created in 1995 by a group from Latvia [39]. Originally, the tool was developed for Windows 3.x as a 16-bit application. Later, it was ported to the 32-bit Win32 framework. At present, the source code and any comprehensive documentation for this project is missing [70], making further development difficult.

An executable version of Sketcher95 is, however, available. It remains partially unfinished. Notably, some common functionality is missing from the tool. There is little or no support for cut/paste, undo/redo or zooming. Also, there

[1] We are not aware of any official branding of this tool, and will refer to it as *Sketcher95*. This may be a misnomer. The application's title bar displays the name "Sketcher", while the "About" dialog states "Sketch Application version 2".

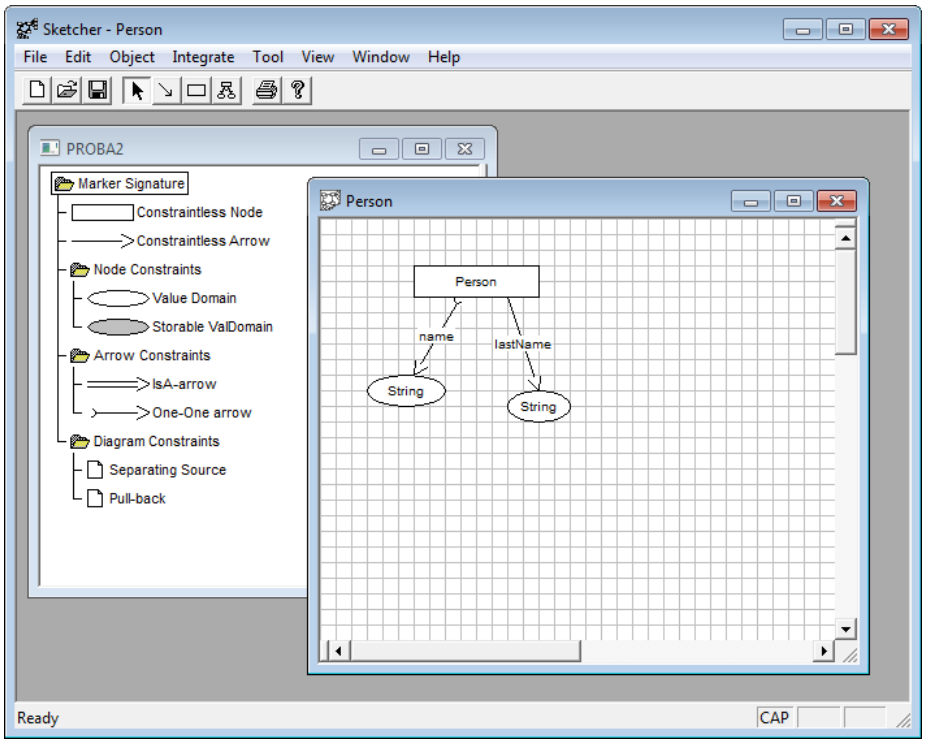


Figure 4.1: Sketcher95's main window. A *sketch document* (named "Person") is shown in the right edit window, its corresponding *marker signature* is shown in the edit window on the left.

seem to be some "selection tool shortcomings" [39, page 31] and lack of keyboard support in the edit windows. In addition, the tool suffers from irregular application failures, making continued use a challenging exercise. Although unfinished and as such unfit for general distribution, the application is runnable on modern versions of Microsoft Windows and has given us valuable insight into the subject matter.

Sketcher95's main interface is implemented as a Multiple Document Interface (MDI), typical for a Windows application of the mid to late 1990s. The main application window consists of the traditional elements menu, toolbar, and status bar, while various edit windows are contained inside the main window (figure 4.1).

In Generalized Sketches, one differentiates between *signatures* (similar to DPF signatures) and *sketch documents* (hereafter called sketches, similar to constrained graphs in DPF specifications). This is reflected in Sketcher95 where different types of edit windows exist for signatures and sketches. A signature is tied to a sketch by not allowing the user to edit a sketch without first opening an edit window for the corresponding signature. During the editing process, both signature and sketch windows are kept active by the tool.

When the user creates a new signature, the signature edit window is shown (figure 4.2). This windows allows for the editing of the three different kinds of constraints in a signature: *node constraints*, *arrow constraints*, and *signature*

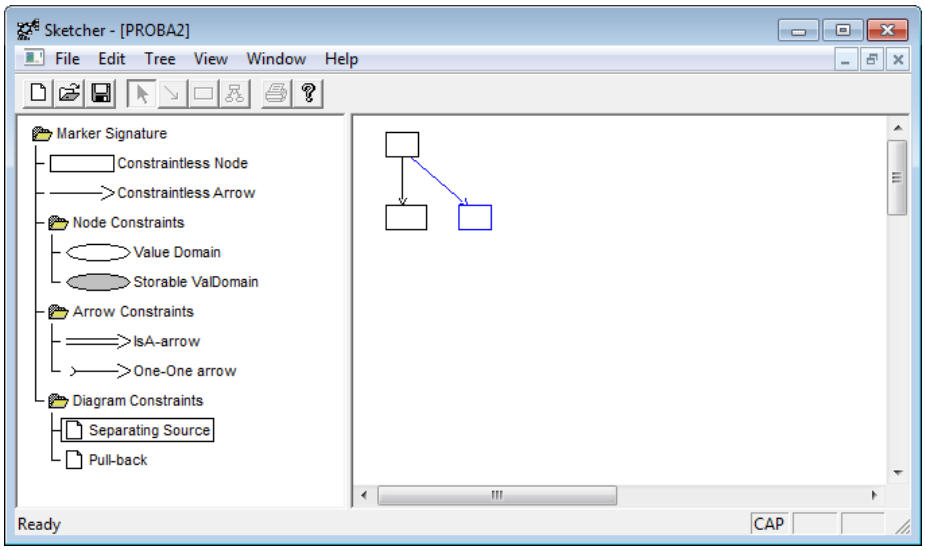


Figure 4.2: A sample signature in Sketcher95. The content of a diagram constraint is activated.

constraints. There is also an option to edit the visualization of unconstrained nodes and arrows.

As mentioned, a signature in Generalized Sketches corresponds to a signature (Σ) in DPF (definition 3, page 9). A *marker* [65] in Generalized Sketches corresponds to a predicate symbol in DPF. Applying a marker to an element of a sketch will make the marker a *constraint* on that element. Sketcher95 distinguishes between predicates on nodes, predicates on arrows, and predicates on diagrams, i.e. collections of nodes and arrows.

To visualize constrained nodes and arrows in sketches, the user can choose from a comprehensive palette of graphical primitives to make up both node shapes and arrow shapes and decorations. A node constraint has configurable elements for name, descriptions and marker. An arrow constraint is similar, but lets the user assign graphical primitives to both the tail, body and/or head of the arrow (figure 4.3).

A diagram constraint has a *shape* that is given as a collection of nodes and arrows. (This corresponds to the concept of a predicate symbol's arity in DPF.) When applying a diagram constraint to a sketch, the user must assign each arrow from the constraint to a corresponding arrow in the sketch. The application checks that all the arrows have the same direction and that the source and target nodes have the same constraints as they do in the corresponding diagram constraint.

To illustrate this, we will apply the predicate `[jointly-surjective]` from table 3.2, page 10. The arity of the predicate denotes the general shape of the predicate. To apply the diagram constraint to a sketch (figure 4.4), the user must

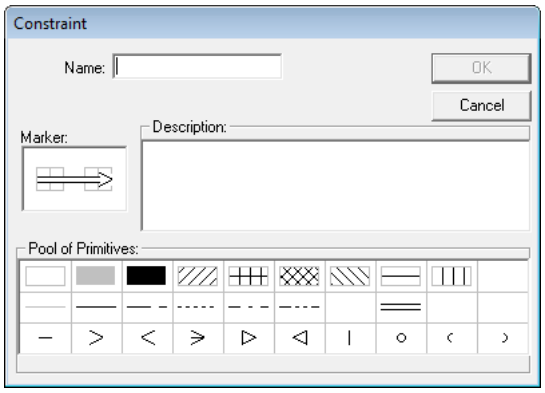


Figure 4.3: Arrow Constraints in Sketcher95

1. select the appropriate constraint to apply
2. select the arrows in the sketch that corresponds to the arrows in the diagram constraint
3. repeat the last operation until all arrows from the diagram constraint has been assigned an arrow in the sketch

A diagram constraint can also produce missing diagram elements. This can be used to have constraints produce new connections between two or more unconnected nodes when applied.

4.1.2 Sketcher .NET

Sketcher .NET was developed by Ørjan Hatland in 2006 as part of his master thesis [39] at the University of Bergen. The Microsoft .NET Framework was chosen as application platform, taking advantage of the modern programming language C# (version 1.1). Sketcher .NET was developed around the same concepts as Sketcher95, separating signatures and sketches in the same manner. The user interface was brought up to date and implemented as a *Tabbed Document Interface* (TDI). The signatures were also relocated to a docking toolbar (figure 4.5).

Unfortunately, this application did not reach a finished state, and several pieces of functionality are left unimplemented. As the source code is available and written in a modern programming language, Sketcher .NET could have been used by us as a base for further development. There are, however, some weighty reasons[2] for not doing so:

- The solution would be restricted to the Windows platform[3]
- HiB focuses on the teaching of Java based technologies
- Other development platforms such as Eclipse include more advanced modelling features

[2]See also section 5.1, page 29

[3]We are aware of the mono framework [53], but do not believe that the mono implementation of Windows Forms (WinForms) would be suitable for this project.

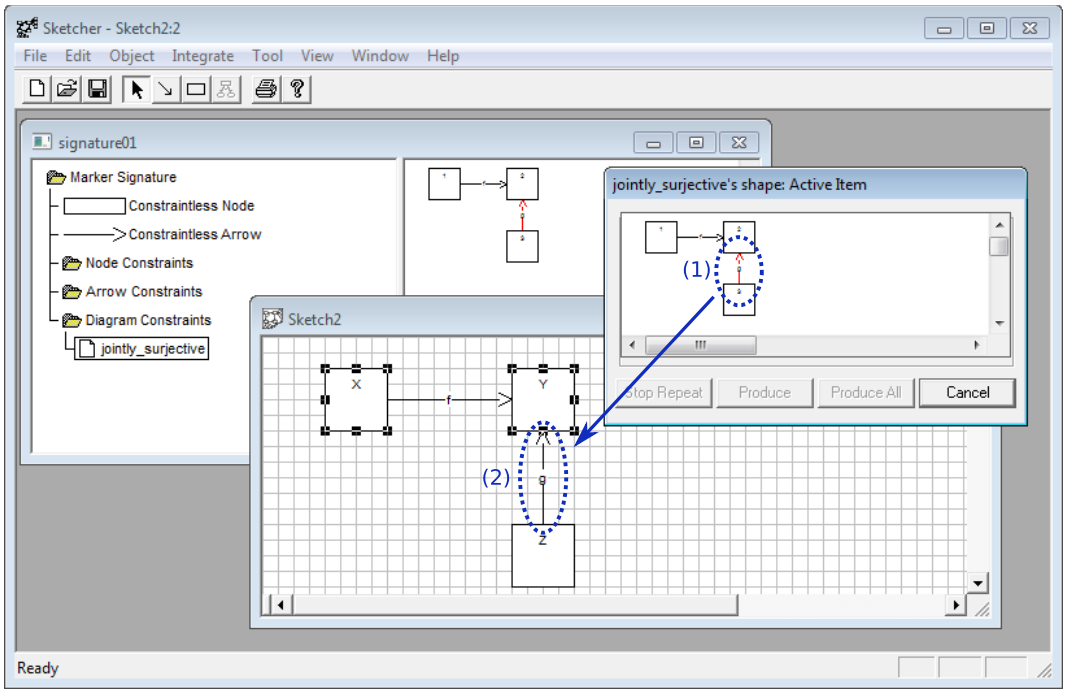


Figure 4.4: Applying the shape of the diagram constraint `[jointly-surjective]` in Sketcher95. The user's selection to mark an arrow in the constraint (1) and then an arrow in the sketch (2), is marked with dotted ovals in the figure.

4.1.3 A GMF solution for Eclipse

In 2008, as part of his master thesis [70] at the University of Bergen, Stian Skjerveggen did development work on an editor for the Diagram Predicate Logic framework (figure 4.6). Retaining the main ideas from the previous attempts, this project changed the developmental focus from a Windows-only solution to a OS-independent plug-in running in the Eclipse [21] platform (section 5.1.1, page 30, for a detailed description). The plug-in was developed using the Eclipse Graphical Modeling Framework (GMF), "a set of generative components and runtime infrastructures for developing graphical editors based on [Eclipse Modeling Framework] EMF and [Graphical Editing Framework] GEF." [75](see also [18] and [37])

Although considerable progress was made, time did not permit for this project to reach a finished state. Among the features left unimplemented are support for metamodelling, dynamic generation of the tool palette, semantic validation of constraints added to a graph, and automatic routing of constraints that span connections. Additionally, Skjerveggen concluded that GMF was not well suited to the task of making a DPF editor.

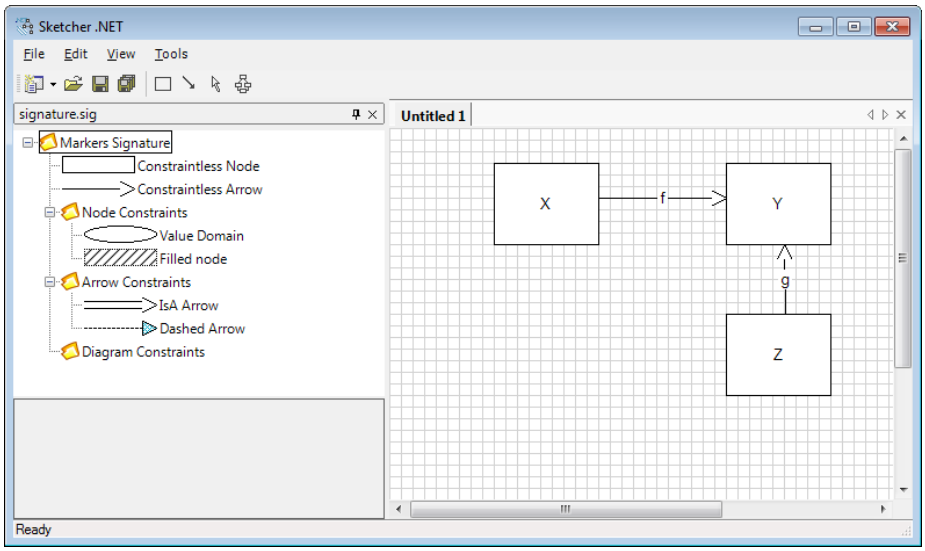


Figure 4.5: Sketcher .NET main window, showing the tabbed interface and a dockable signature window.

This project would probably be the best candidate for a base for further development. As we shall see, we chose not to continue development along the same trail, our main reasons being:

- Architectural changes made in the DPF
- Some reluctance on the author's part to continue using GMF
- An added emphasis on research problems regarding metamodelling

It is important to point out that without Skjerveggen's work, we could very well have been pursuing a similar, GMF-based solution.

4.2 Current situation

There still exists no working tool to create diagrammatic specifications based on the Diagrammatic Predicate Framework. As stated, all the previous efforts have been discontinued, and must be considered prototypes for our own development activity. The previous effort can be considered the "best" so far, as it prototyped several central DPF concepts on Eclipse and made some headway towards being a real integrated development tool running inside an IDE, not just a sketching application.

There is also a clearly stated need for a practical implementation of a DPF (meta)modelling tool. The aim is to try out theoretical constructs in practice and perhaps use the tool as a "sounding board" for new DPF features. There also seem to be a need for a tool for educational purposes, as there exists plans to start up a course in MDE-based methods – including DPF – at Bergen University College.

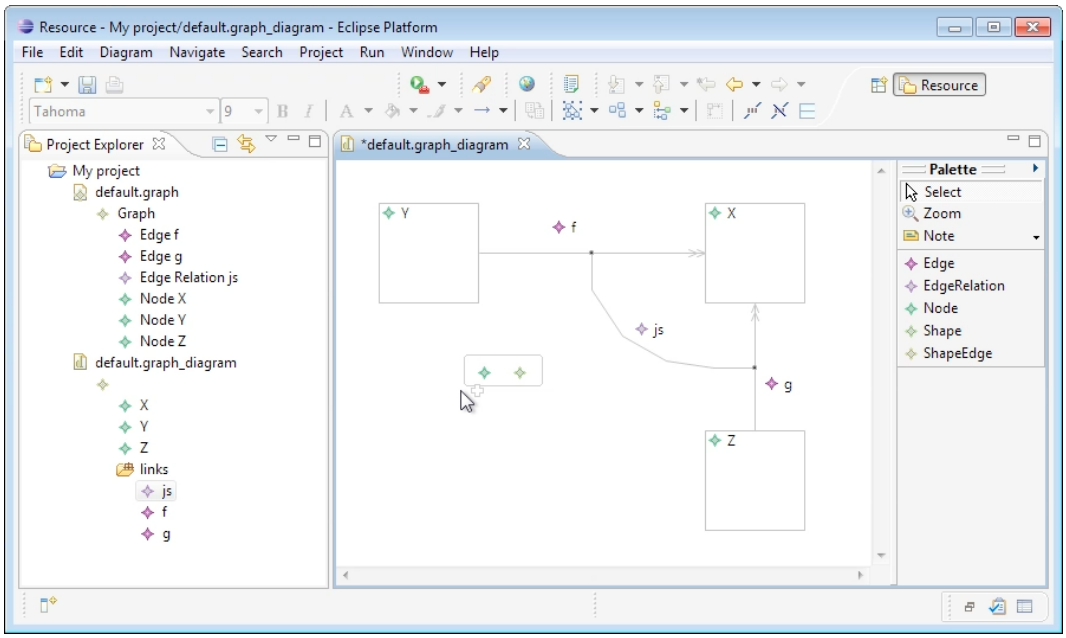


Figure 4.6: Stian Skjerveggen's GMF-based solution running as an Eclipse plugin. In this screenshot, a tree list (displaying the contents of both the graph and the corresponding diagram), a toolbar, the diagram itself being edited, and a dockable palette can all bee seen.

4.2.1 Related tools

There is an abundance of visual modelling tools available, both as open source software and as closed-source commercial products. Some of these tools also possesses metamodelling features, letting the user specify a metamodel and then use this model in some new editor. A comprehensive list of such tools is beyond the scope of this thesis; here we give a brief presentation of five metamodelling tools currently available, The Visual Modeling and Transformation System, AToM^3, The Generic Modeling Environment, MetaEdit+, and MetaDepth.

VMTS

The Visual Modeling and Transformation System (VMTS) is an n-layer metamodelling environment which supports editing models according to their metamodels [50]. Tool support for VMTS is available [78]. The tool allows for an arbitrary number of (meta)modelling layers, but has no support for a completely graph-based constraint language, as it uses a text-based language, Object Constraint Language (OCL), for the specification of constraints. It runs on the Microsoft .NET framework.

AToM^3

AToM^3 (A Tool for Multi-formalism and Meta-Modeling) is a tool for multi-paradigm modelling [2, 43]. The two main tasks of AToM^3 are metamodelling

and model-transformation. Formalisms and models are described as graphs. From the metamodel of a formalism, AToM^3 can generate a tool that lets the user create and edit models described in the specified formalism. Some of the metamodels currently available are: Entity-Relationship, GPSS, Deterministic and Non-Deterministic Finite state Automata, and Data Flow Diagrams.

AToM^3 is freely available. The tool does not allow for an arbitrary number of (meta)modelling layers, nor is there support for a completely graph-based constraint language (constraints can be specified as OCL or Python expressions). The tool is implemented in Python and Tcl/Tk, and runs on most platforms.

The Generic Modeling Environment

The Generic Modeling Environment (GME) [49] is a configurable toolkit for creating domain-specific modelling and program synthesis environments. The configuration is accomplished through metamodels specifying the modelling paradigm (modelling language) of the application domain [32].

The GME metamodelling language is based on the UML class diagram notation and OCL constraints. Metamodels specifying the modelling paradigm are edited in the tool's editor and saved to file. New editors can then be instantiated, based on the newly generated metamodels. In order to simplify the editing process, both models and metamodels are edited in the same environment. Model visualization is customizable through built-in decorator interfaces. All GME modelling languages provide type inheritance, and GME supports various concepts for modelling, including hierarchy, multiple aspects, sets, references, and explicit constraints.

The tool does not allow for an arbitrary number of (meta)modelling layers, nor is there support for a completely graph-based constraint language. GME's architecture is based on Microsoft Component Object Model (COM), making it extensible by any language that supports COM. The drawback of this approach is that GME only runs on the Microsoft Windows platform.

MetaEdit+

MetaEdit+ is a commercially available integrated environment for Domain-Specific Modeling (DSM) [77].

The tool lets the user produce Domain Specific Modelling (DSM) languages through a graph-based modelling formalism. It also includes a graphical editor for specifying concrete syntax for DSM languages for use in model editors. Support for metamodel evolution is also provided. A series of pre-defined modelling languages are included, providing support for feature modelling, financial service models, logic etc.

The tool does not allow for an arbitrary number of (meta)modelling layers, but appears to support a graph-based constraint language. The tool is distributed as commercial software. MetaEdit+ is made with Smalltalk, and it runs on both Microsoft Windows and X11-based operating systems.

Tool	N-layer	Single formalism	Cross platform	Source available
Sketcher95	No	Yes	No (Windows 32)	No
Sketcher .NET	No	Yes	No (.NET)	Yes (C# 1.1)
GMF Solution for Eclipse	No	Yes	Yes (Eclipse)	Yes (Java/GMF)
VMTS	Yes	No	No	Yes (C#)
AToM^3	No	No	Yes	Yes (Python)
GME	No	No	No	Yes (COM)
MetaEdit+	No	Yes	Yes	No
MetaDepth	Yes	No	Yes	Yes (Java)

Table 4.1: Comparison of features of model editors

MetaDepth

The MetaDepth [11] framework is a framework for multi-level metamodelling. The system permits building systems with an arbitrary number of meta-levels through deep metamodelling. The framework allows the specification and evaluation of derived attributes and constraints across multiple meta-levels, linguistic extensions of ontological instance models, transactions, and hosting different constraint and action languages. Constraints and actions can be defined using Java or EOL [46]. At present, the framework is text-only; it does not yet contain support for a graphical concrete syntax.

Tool overview

Table 4.1 shows a summary of the features we consider as most important for our development effort. The column **N-layer** indicates whether a tool lets the user edit models on an arbitrary number of meta-levels. The column **Single formalism** indicates whether there is support for a completely graph-based constraint language (all constraints contained in one closed formalism). **Cross platform** indicates whether the tool easily can be run on more than one platform, and the column **Source available** indicates whether the source code is available, either through an open source license or otherwise.

(Note that MetaDepth has the most complete set of features by our evaluation criteria. However, the tool does not yet contain a graphical concrete syntax, and, as the tool was in its early development stages when work commenced on our project, we did not consider MetaDepth as a basis for our work.)

As we have tried to show, we are presently unaware of any open-source diagrammatic metamodelling tool that gives software developers the ability to construct domain specific languages at an arbitrary number of meta-levels while maintaining all constraints contained under one closed formalism.

4.3 Requirements and problem description

In the following chapters, we will describe our effort to make a modelling tool with an emphasis on metamodelling features. In this section, we formulate the concrete requirements for this tool and also outline the general problem description for our thesis.

Requirements based on previous efforts

The tool to be constructed should include the central features present in the previous DPF tools. This includes functionality for:

- Drawing nodes and arrows in a specification diagram
- Creating and maintaining a model graph for the specification diagram
- Keeping the integrity of the graph of the diagram (no dangling arrows)
- Editing existing diagrams:
  - Moving and resizing nodes
  - Deleting nodes and arrows
  - Support for undo and redo
- Persisting a specification
- Creating a specification based on a given metamodel

This list is not exhaustive. As we have employed an agile development methodology (section 5.2, page 36), several *user stories* have been made, each relating to one or more central tool features.

Additionally, the tool should also be *multi platform*, i.e. runnable on all popular operating systems.

New requirements

The central new feature to explore in relation to the previous efforts, is metamodelling. It is our view that DPF will provide a sound theoretical basis for constructing a metamodelling tool that allows the user to edit models on an arbitrary number of meta-levels. Also, we believe that the DPF strategy of keeping all constraints within one closed formalism will be beneficial for further work in this area, especially if features such as model transformations, code generation, versioning control, and metamodel evolution are to be added to the system.

This adds the further requirements that the tool must have functionality for:

- Editing specifications at an arbitrary number of meta-levels
- Adding constraints to a metamodel and validate these constraints at the instance layer

Problem description

In relation to the previous, we formulate the following research question:

Is it possible to construct a DPF specification editor supporting metamodelling at an arbitrary number of meta-levels?

We will try to answer this question by designing and implementing a model editor for the DPF formalism, based on concepts adapted from previous efforts and containing the functionality as outlined above. We also believe that, if successful, this editor will be an important step towards a general modelling platform based on DPF.

Chapter 5

Technology platform and methodology

In this chapter, we describe our choice of technology and how this creates a technological context for our research and development. We also briefly discuss our choice of development methodology.

5.1 Technological context

Our choice of technology platform for this project was somewhat straightforward, as the DPF research group already had voiced a strong interest in the Eclipse platform and thereby expressed a preference for a DPF tool running in Eclipse. Skjerveggen [70] discusses the choice of Eclipse over other technologies such as Microsoft .NET in some detail. In our view, the following points were important when settling on Eclipse as our development platform:

Eclipse is licensed under an Open Source license: Eclipse is distributed as Free and Open Source Software(F/OSS) [26], making it an ideal candidate for development work in an academic environment.

Eclipse has a large feature set: Eclipse comes bundled with EMF (see section 5.1.2), a collection of industrial-grade modelling tools based on Ecore/EMOF.

Eclipse is based on Java, an industry leading programming language[1]: Programming courses at Bergen University College are mostly Java-centric. As such, more students should be familiar with the Eclipse framework than with systems based on other programming languages.

(As regards our choice of development tools peripheral to the Eclipse platform (in particular source control and automated builds), see section 6.9.2, page 55.)

[1] According to the TIOBE Programming Community Index for April 2011 (`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`), Java is the most widely used programming language industry-wide, ahead of C and C++.

5.1.1 Eclipse

Eclipse [21] is an open-source software development environment managed by the Eclipse Foundation. The platform has become an industry standard Java IDE. Grown to a fully-fledged multi-language software development environment, Eclipse originated from IBM VisualAge for Smalltalk™ and IBM VisualAge for Java™ [24], both products originally written in the Smalltalk language.

Eclipse is completely built around the concept of *plug-ins*. Everything in Eclipse, with the exception of a small run-time kernel, is implemented as a plug-in. Support for plug-ins is facilitated by the Eclipse component Equinox, an implementation of the OSGi R4 core framework specification. [25] A plug-in can facilitate further extension of Eclipse by declaring an *extension point*. By declaring an extension point, a plug-in exposes a minimal set of interfaces and related classes for others to use; other plug-ins declare extensions to that extension point, implementing the appropriate interfaces and referencing or building on the classes provided [9].

The Standard Widget Toolkit (SWT)

Although almost completely built in Java, Eclipse does not base its user interface on either the Abstract Window Toolkit (AWT) [59] or Swing [81]. Instead, Eclipse has applied its own widget toolkit, called The Standard Widget toolkit (SWT) [55]. The entire Eclipse user interface (UI) is based on SWT [9].

Where AWT delegates all its widget functionality to native widgets and Swing emulates all widgets in the toolkit by implementing a 100% pure Java solution, SWT employs a hybrid approach by using native widgets where possible and emulated widgets in the cases where native functionality is unavailable. This has the disadvantage that porting the framework from one platform to another becomes a more laborious process than for Swing [79].

On the other hand, the advantages are also significant. The "lowest common denominator" (LCD) problem, as seen in AWT, disappears. The LCD problem occurs when a widget toolkit only implements the features common to all elements in a large set of native UIs, eschewing innovative and productive features that is present in one or more, but not all, UIs that the toolkit covers [81].

Swing tries to tackle this problem by emulating all widgets, implementing them in pure Java. Compared to Swing, SWT becomes more lightweight, having done away with the emulation of a large numbers of UI features that Swing emulates, although the underlying UI already features an implementation. As the emulation of UI features seldom is perfect, Swing often displays a look and feel that experienced users of a native UI will recognize as "not the real thing" as opposed to SWT's use of native widgets. There is also a speed advantage, as native drawing routines often outperform calls to the graphic context from Java Swing. [55]

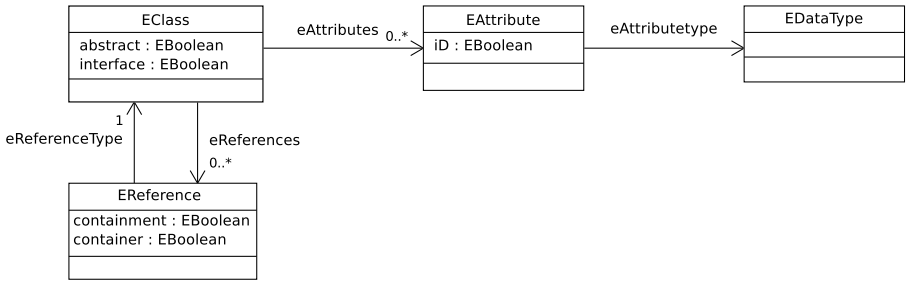


Figure 5.1: A simplified subset of the Ecore metamodel. Adapted from [8].

5.1.2 The Eclipse Modeling Framework

To quote the EMF project's web site [18]: "The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model." Lately, what was formerly known as EMF has itself become part of the larger Eclipse Modeling Framework Project (EMF), where the "old" EMF is denoted EMF (Core). When nothing else is noted, we will mean EMF (Core) when discussing EMF.

EMF's metamodel is called Ecore. Ecore is a reflective model, which means that it is its own metamodel. Ecore is almost identical to a subset of MOF (section 3.3, page 11) called Essential MOF or EMOF. This makes EMF a MOF-based modelling language (EMF can actually read and write EMOF serializations transparently) [8]. A simplified subset of the Ecore metamodel is shown in figure 5.1.

In EMF, several tools are available to let the developer create and modify models. The most basic form of model editing is done through a simple tree-based editor (figure 5.2). Models can also be created through XML documents, annotated Java interfaces or by importing UML models created in Rational Rose [8].

EMF supports serialization of Ecore models to the XMI [58] format. This facilitates easy serialization and de-serialization of models [8].

EMF also includes facilities for generation of Java code based on the structure of an Ecore model. This is done through the intermediate stage of using an *generator model* as basis for the code generation process. This model contains additional information that's not present in the data model itself, "but instead resides in a generator model made up of decorators for Ecore model objects." [8]

5.1.3 The Graphical Editing Framework

The Graphical Editing Framework [37] (GEF) bundles three components; *Draw2d*, the *GEF (MVC) framework*, and *Zest*[2]. These components are distributed as separate Eclipse plug-ins, but they are often described as a coherent whole.

[2]Zest [80] is a visualization toolkit and constitutes a set of (graph) visualization components built for Eclipse. The entire Zest library has been developed in SWT/Draw2d. Although a part of GEF, we have not employed Zest in our project.

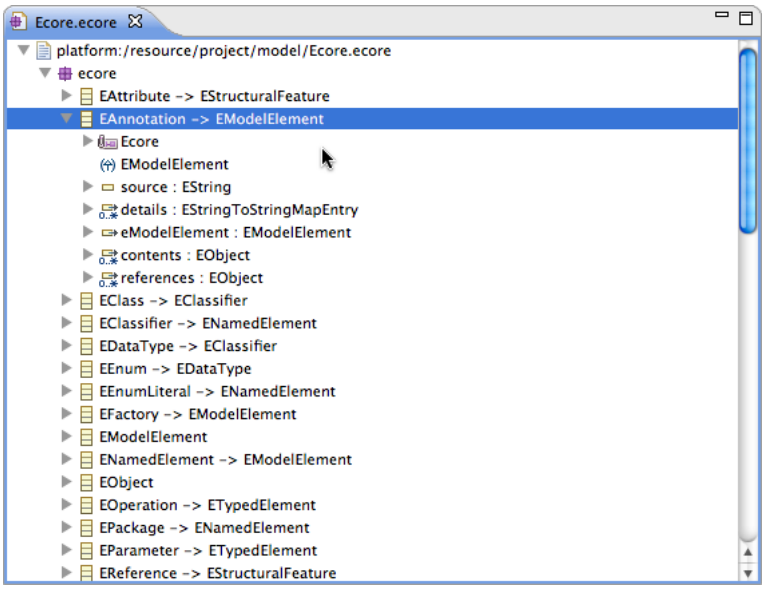


Figure 5.2: EMF's tree-based Ecore editor. Shown here is a part of the Ecore metamodel itself begin edited.

Draw2d

Draw2d [15] is a lightweight framework for two-dimensional layout and rendering built on top of SWT. As such, it can be distributed and used independently from Eclipse. Draw2d introduces the concept of *figures*. A figure is a lightweight (having no counterpart in the underlying native UI) graphical component in Draw2d. Draw2d uses an SWT Canvas object as the medium for drawing figures and for handling user input events. Draw2d also has support for text, by means of labels and several "rich text" features. In addition, Draw2d has a built-in concept of *connections*, i.e. connection lines between figures. These lines can receive text decorations as labels and graphical endpoint decorations for displaying arrows etc.

Input events are handled by an `org.eclipse.draw2d.SWTEventDispatcher` object that adds SWT event listeners on a Canvas. Paint events are forwarded from figures to an `org.eclipse.draw2d.UpdateManager`, which handles the job of laying out and repainting figures. This happens in two phases. First, invalid figures are laid out, creating damage regions. Second, all damaged areas of the graph are repainted [15].

Draw2d figures can be composed in parent-child relationships. This means that a parent figure is responsible for painting (and optionally lay out) its children, and children are not painted on areas outside its parents'.

Prime Draw2d features are [15]:

- Layout and rendering support
- Various figure and layout implementations
- Borders

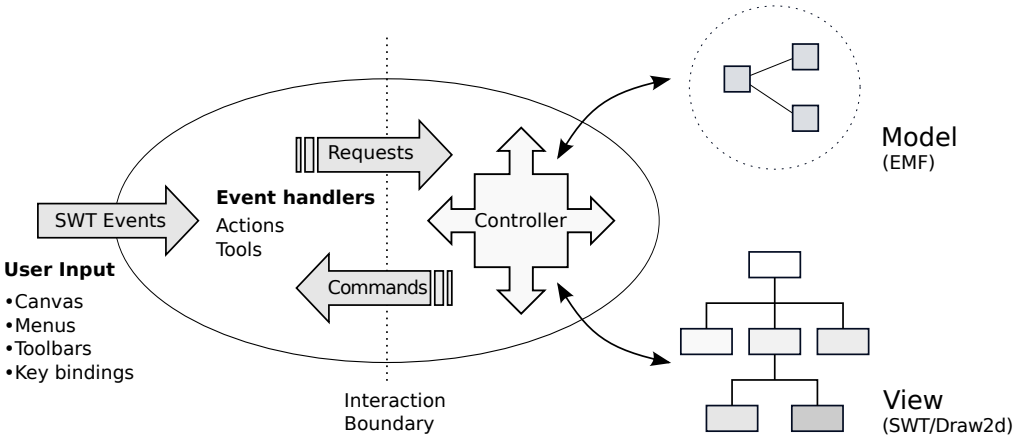


Figure 5.3: GEF (MVC) overview. GEF (MVC) can be loosely defined as the region inside the oval. The framework provides input handlers (Actions and Tools) that turn SWT events into requests. Requests and Commands are used to encapsulate interactions and their effects on the model. The Controller provides the link between an application's model and view. Adapted from [38] and [36].

- Cursors and Tooltip support
- Connection anchoring, routing, and decorating
- Multiple, transparent layers
- Flexible coordinate systems
- Overview window (thumbnail with scrolling)
- Printing

This provides a flexible system for laying out and drawing diagrams to screen, making the handling of user input and drawing routines fairly transparent for any GEF application.

GEF (MVC) framework

The GEF (MVC) framework, hereafter called GEF, is layered on top of Draw2d. GEF expands on SWT and Draw2d by adding editing capabilities to widgets and drawings. This is achieved by implementing a Model-View-Controller (MVC) pattern [47] where the views are implemented either as an SWT-based tree control or a Draw2d-based figure. The model is typically implemented as an EMF model, although plain Java objects might be used in simpler cases. This leaves the implementation of both the controller and the overall plug-in functionality to GEF [7].

In the GEF MVC pattern, the model provides all user-modifiable and user-viewable data. This also means that all data persistence is relegated to the model. The controller or view should not persist or create any new data. Also, the model should not maintain any reference to the controller or view, making the coupling to the model as loose as possible [9].

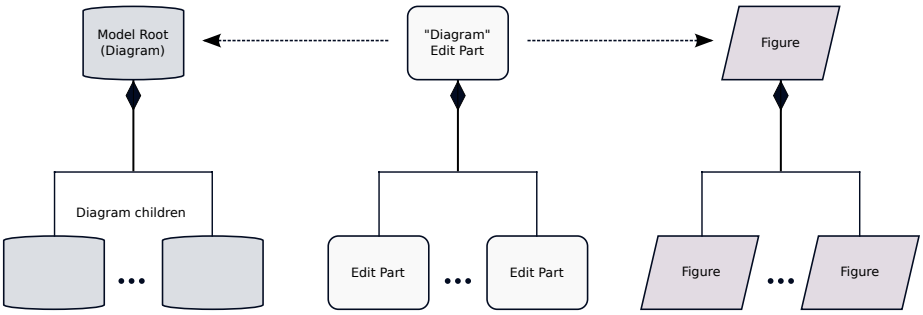


Figure 5.4: MVC hierarchies in GEF. The typical arrangement is that `EditParts` maintains children. Usually, this corresponds to a similar containment found in the model. For example, the model may consist of a diagram containing nodes. There would then be a corresponding diagram EditPart which contains multiple node EditPart children. This parent-child relationship of EditParts carries over into their corresponding figures. The parent's figure will contain the children's figures [36]. Adapted from [38] and [36].

Controller

The GEF controller (figure 5.3) initially registers with the model as a listener to receive updates when changes are made to the model. Subsequent input from the user is then propagated first to the model and then to the view.

The GEF controller is made up from a collection of instances of `org.eclipse.gef.EditPart`. As with figures, `EditParts` can maintain children. The model's structure is typically represented in diagrammatic form as Draw2d figures. When this is the case, the collection of `EditParts` mirrors the structure of both the model and view hierarchies [36] (figure 5.4).

There are two main types of `EditParts` in GEF: graphical and tree. `TreeEditParts` use SWT widgets to create a tree view for user interaction. `GraphicalEditParts` use Draw2d figures as their view. Different types of GraphicalEditParts constitute different types of model objects; for instance, GEF distinguishes between *child* EditParts and *connection* EditParts. An EditPart's main responsibilities are to create and maintain an associated view, child EditParts, connection EditParts, and support editing of the model. [38]

An implementation of GraphicalEditPart is also responsible for creating the figures that represent its corresponding model object(s). This is achieved by overriding the `getFigure()` method.

EditParts can also be extended by the use of *edit policies*. These are instances of `org.eclipse.gef.EditPolicy`, which are pluggable contributions each implementing some portion of an EditPart's behaviour. Using edit policies, EditParts can be extended without changing their class hierarchy [38].

Factories and figures

The creation of EditParts, and subsequently of figures for the view, is done through the use of an `EditPartFactory` (figure 5.5). Every GEF canvas maintains exactly one edit part factory.

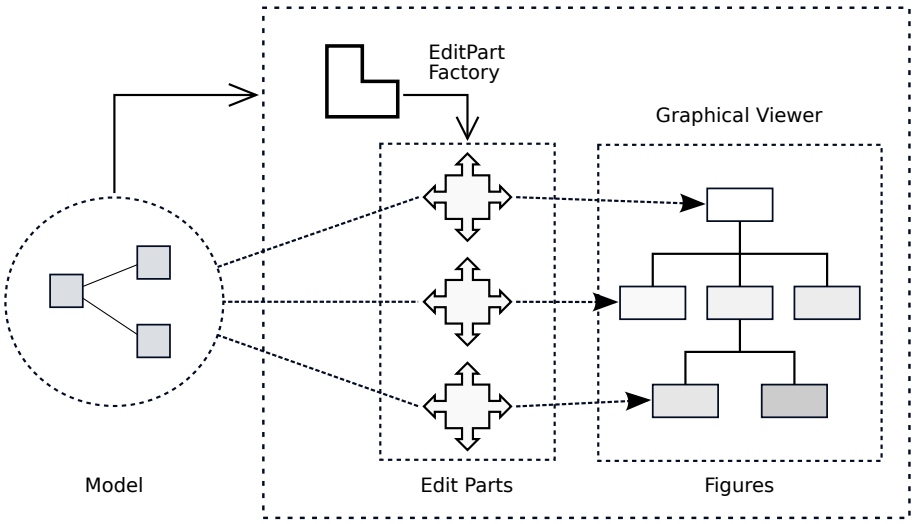


Figure 5.5: Edit part factories in GEF. Subsequent figures are created by the EditParts when their `getFigure()` methods are called by the framework. Adapted from [38].

The GEF canvas thus contains and maintains a collection of figures that visually represents the underlying model. Each EditPart must create and manage figures representing the model object associated with that edit part. Each figure in the GEF canvas must implement the `org.eclipse.draw2d.IFigure` interface. Similar to the `java.awt.Graphics` class in AWT, the `org.eclipse.draw2d.Graphics` class provides drawing primitives [36].

Complex figures

As mentioned, Draw2d figures can be composed in parent-child hierarchies. In GEF, a figure belonging to a specific EditPart automatically becomes a child of that edit part's parent figure. Custom figures can be made by extending the Figure class and overriding the Paint method. All figures with children must declare a `LayoutManager` responsible for positioning and sizing the child figures. (Available layout classes are `BorderLayout`, `FreeformLayout`, `FlowLayout`, `GridLayout`, and `StackLayout`) [36].

5.1.4 Implementing GEF-based Eclipse plug-ins

The user interface of Eclipse is centred on the concepts of *editors*. Basic text editors and source code editors for popular programming languages are provided with Eclipse. Plug-ins that need to present their own editing functionality can do so by implementing the `org.eclipse.ui.IEditorPart` interface. Editors are typically built to modify some file type, and thus follow the classic open-modify-save paradigm [9]. The files themselves are typically located somewhere in the Eclipse *workspace*, a container directory for one or more Eclipse projects.

A special case of the Eclipse editor is the graphical editor, which behaves as an ordinary editor (modifies some file type), but does so by letting the user manipulate graphical entities, such as a diagram, rather than text. GEF provides the `org.eclipse.gef.ui.parts.GraphicalEditor` class for this purpose. This class can be extended in order to create a custom GEF-based Eclipse editor [36].

When a user manipulates figures in an editor, input events are created and GEF translates these events into discrete GEF *requests*. These requests are forwarded to any EditPart belonging to the figure that was manipulated. The EditPart consumes the requests and produce GEF *commands*. GEF commands encapsulate changes to the model that can be applied and undone [7]. This is used in a Command pattern [30], making undo and redo in an editor easy to implement.

Palette

A common supplement to a GEF editor is a *palette*, creating a common "visual repository" for tools associated with the visual editor. This gives the user a familiar visual pattern for creating and manipulating content in the visual editor. In addition, context menus and other pop-up menus/tooltips etc. can easily be created in GEF [36].

5.2 Development methodology

As our method of development, we have selected Extreme Programming (XP). XP is a software development methodology developed by Kent Beck [6]. It is an early and perhaps the best known example of what is called *agile methods*. [74]. This is a common description for "small-scale" development methods, adhering to the *agile manifesto* [52]. During the last ten years, XP and other agile methods have gained mainstream acceptance in the IT industry. As the method is taught as part of a graduate course on modern software development methods at HiB, it was a natural choice for use in this project.

XP can be considered a conglomerate of different and seemingly unrelated development *practices*. Beck [6] argues that, although apparently unrelated, these practices are interconnected and interdependent. Key practices are Pair Programming, Test-Driven Development, Refactoring, Continuous Integration, and Simple Design [52]. We will briefly discuss how we applied these and other practices in the next chapter.

Chapter 6

Design and development

In this chapter, we outline the design and development of our tool. This includes the process of naming the tool and its individual components, the overall architecture of the solution, and the individual packages that make up this architecture. Further, we describe the editor and its (sub)package structure in more detail. Then, we discuss the implementation of metamodelling capabilities, typing of a diagram's graphs, and semantics validation. Lastly, we discuss some of the process-related issues we encountered during the development phase.

6.1 Finding names

We wanted to find a name for our tool that was both descriptive and concise. Although our project is not an official Eclipse project, we opted to conform to Eclipse development guidelines where possible. The reason for this was partly to provide familiarity for other developers and partly to ease the process of packaging and deploying the plug-in at a later stage.

6.1.1 Tool branding

The Eclipse development guidelines include the Eclipse User Interface Guidelines [23] and the Eclipse Project Naming Policy [22]. According to the latter, the policy for project names is:

Descriptive Name: Projects are encouraged to use a descriptive name – a name that is useful when placed into a box on an Eclipse architecture diagram.

Nicknames: If a team strongly prefers to use a nickname style project name, instead of a purely descriptive project name, there should also be a longer "more official, but less often used" (MOLU) combination nickname-descriptive project name.

Acronym: Most descriptive names are sufficiently long that it can be convenient to abbreviate them in some way.

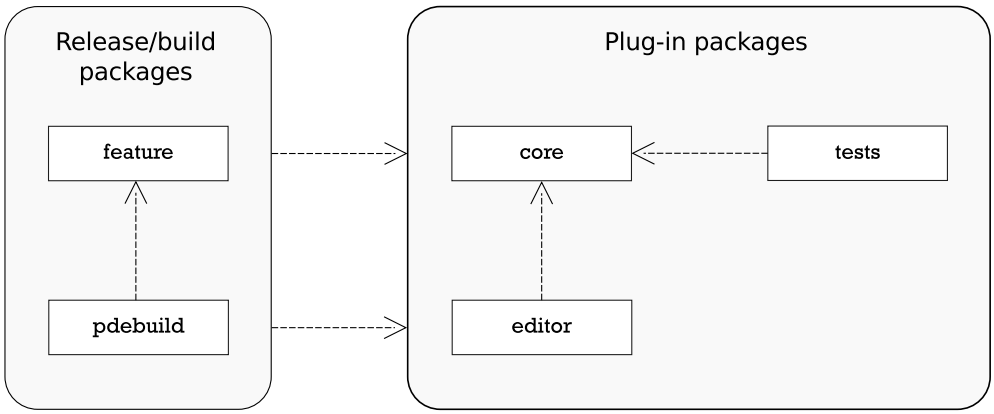


Figure 6.1: The DPF Editor package structure as it exists at present. The boxes in this diagram indicate Java packages, and the arrows indicate dependencies between these. The dependencies go in the direction of the arrows: for instance, the `tests` package depends on the `core` package. All packages have the prefix `no.hib.dpf` (omitted for clarity). Also, each package is located within a separate Eclipse project.

Not a Product Name: To avoid confusion between Eclipse projects and commercial products, Eclipse projects may not be named after commercial products and vice versa.

Regarding this, we also took into account the fact that this is just the first of – hopefully – many DPF-related tools. A fitting name for such a collection of tools would be "DPF Tool Suite", "DPF Tools" or similar. To emphasize that this project primarily concerns itself with producing a (diagrammatic) drawing and modelling tool for editing DPF specifications, we settled on "DPF Editor" as the branded name for our plug-in. The abbreviated form will be "DPFE".

6.1.2 Namespace and naming conventions

Our solution was implemented using the Java programming language. Related to the branding of the plug-in was the choice of a suitable Java *namespace*. The Eclipse Naming Conventions [19] recommends naming packages in conformance with Sun's (later Oracle's) guidelines. These guidelines can be found in the official naming conventions [42]. We followed this naming convention and settled on `no.hib.dpf` as our root namespace. Individual Java packages were named accordingly. We also followed the guidelines when naming classes, interfaces, methods, and so forth.

6.2 Tool architecture

We placed our implementation of the DPF metamodel in the package `no.hib.dpf.core` and the GEF editor code in the package `no.hib.dpf.editor`. In addition, three more packages make up the current implementation of DPF Editor

(figure 6.1): `no.hib.dpf.core.tests`, `no.hib.dpf.pdebuild` and `no.hib.dpf.feature`. These packages are implemented as separate Eclipse *projects*, named accordingly.

The packages can be categorized as either "release/build packages" or "plug-in packages". The contents of the release/build packages are used in the automated build process only. The remaining packages contain the components of the plug-in itself.

6.3 Release/build packages

These packages are used in the automated build process. See section 6.9.2 for a description of this process.

6.3.1 no.hib.dpf.feature

The Eclipse *feature* concept lets developers combine several Eclipse plug-ins into a single *feature package*. A feature package is seen as an atomic piece of software by the end user, and can be installed in a single step. The feature *manifest* also lets the developer specify which other features and plug-ins a feature needs in order to run, if any.

At present, DPF Editor consists of two different plug-in projects, namely `core` and `editor`. We have utilized the feature concept, and packaged our plug-ins in one single feature. Upon build, DPF Editor is compiled into a single jar file for further distribution. Any later additions to the plug-in architecture should also be added to the main feature.

The *feature* definition is located in a separate Eclipse project, containing the `no.hib.dpf.feature` namespace.

6.3.2 no.hib.dpf.pdebuild

PDE/Build [20] is an Eclipse framework for building plug-ins and features for Eclipse. This is a flexible framework, providing support for a large number of build scenarios. We have defined a simple build script within PDE/Build to build our plug-in code.

There are two configuration files in this project. The `build.properties` file was copied from the PDE source as per instructions on how to use PDE/Build. The other file, `build-dpf.xml` is an ant [1] build file which actually runs the build. Its skeleton was copied from the PDE/Build source, and modified for our use.

PDE/Build also contains functionality for deploying finished artefacts to an update site. This functionality is left inactive at the moment, but the plan is to start using it when the DPF Editor becomes ready for wider distribution.

The PDE/Build code is located in a separate Eclipse project, containing the `no.hib.dpf.pdebuild` namespace.

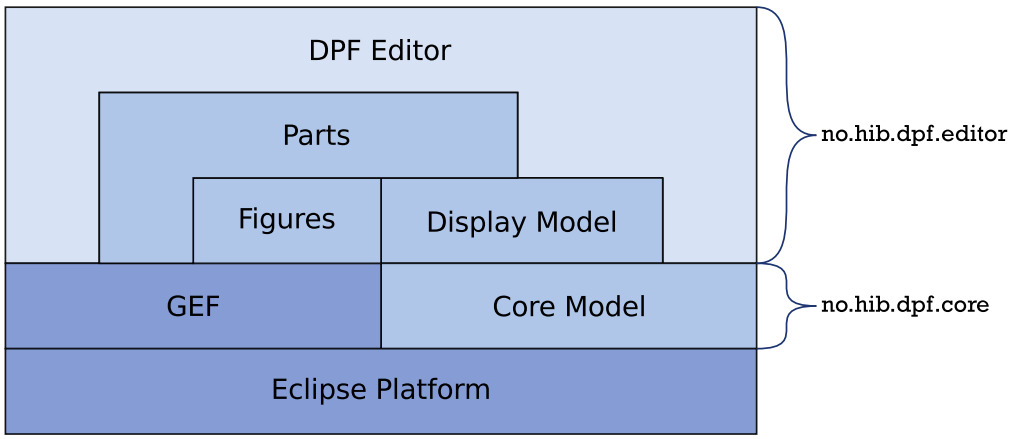


Figure 6.2: The main component architecture of the DPF Editor plug-in packages. The Core Model component is contained within the `no.hib.dpf.core` package, the Display Model, Figures, Parts, and DPF Editor components all are parts of the `no.hib.dpf.editor` package.

6.4 Core and Display Models

In this and subsequent sections we will make a short presentation of the content of DPF Editor's plug-in packages. An illustration of the component architecture of these packages can be seen in figure 6.2.

The purpose of our plug-in is to make the user able to edit DPF specifications (definition 5, page 9). The editor we want to implement is a *diagrammatic* editor, i.e. the user should be able to edit a specification graph by means of manipulating a diagram. In our case, this diagram consists of three types of objects, *nodes*, *arrows*, and *constraints*. The class definitions for these types of objects are located in the `no.hib.dpf.core` package.

6.4.1 The core package

As the name hints at, the `no.hib.dpf.core` package contains our implementation of the core DPF (meta)model. The contents of this package were developed using the EMF modelling tools. Using EMF for this task was a crucial element of our development effort. The main reasons for settling on Eclipse and EMF as an implementation platform have been given previously (section 5.1, page 29). In addition, we also intended to experience the model-driven development process for ourselves. We believe that a hands-on experience with modelling tools yields insights into the MDE process that is otherwise difficult to obtain. In section 6.9, we discuss some of our experiences with using Eclipse/EMF as a modelling tool.

The modelling was primarily done using EMF's tree-based Ecore editor (section 5.1.2, page 31). Making use of a generator model, we generated Java code from our Ecore model. In this way, we got class skeletons and simple getters and setters functionality "for free". More complex methods we had to implement

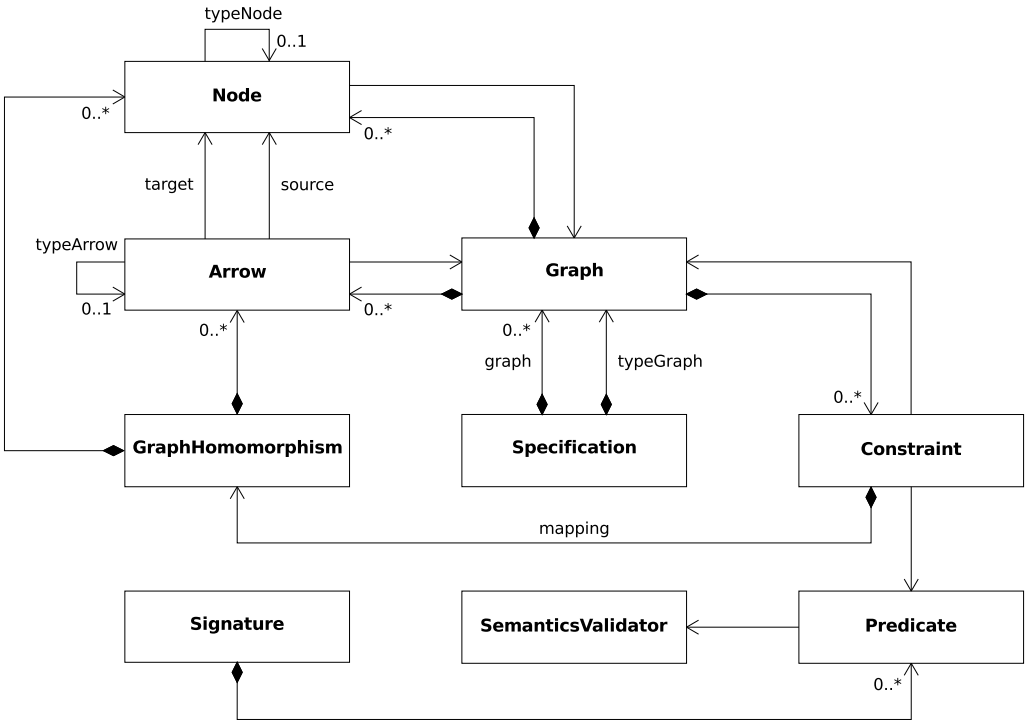


Figure 6.3: The main classes that make up the `no.hib.dpf.core` package as modelled in Ecore. This figure omits some of the less important auxiliary classes. Fields and methods, as well as some association's names, have also been left out for clarity. Associations with multiplicity equal to 1 have been drawn without a multiplicity indicator. In the figure, we recognize the `Node`, `Arrow`, and `Constraint` classes: Note that nodes and arrows can reference other instances of the same type, in order to easily maintain typing homomorphisms between graphs. Much important functionality is located in the `Graph` class, and instances of this class will interact with instances of the classes mentioned, as well as being managed by instances of the `Specification` class. Another DPF key concept is modelled in the `Signature` class, and instances of this class maintain a set of predicates, also being referenced by constraints. Lastly, instances of `SemanticsValidator` are referenced by individual instances of `Predicate`.

ourselves, applying Test-Driven Development (TDD) as the main development practice.

The content of the `no.hib.dpf.core` package models the concepts of the DPF. As mentioned, classes containing definitions for *nodes*, *arrows*, and *constraints*, designated `Node`, `Arrow`, and `Constraint`, respectively, are present in this package (figure 6.3). Furthermore, classes representing the core DPF concepts *signature*, *predicate* and *specification* are present, designated `Signature`, `Predicate` and `Specification`. Finally, classes representing *graphs* and *graph homomorphisms* are present, as well as a class that represents a generic semantic validator. These last classes are named `Graph`, `GraphHomomorphism`, and `SemanticsValidator`. Instances of `Graph` will contain (and be referenced by) instances of classes `Arrow`, `Node`, and `Constraint`. `GraphHomomorphism` instances will be used to apply constraints to graph elements. `SemanticsValidator` instances will serve to validate semantics (see section 6.8, page 53).

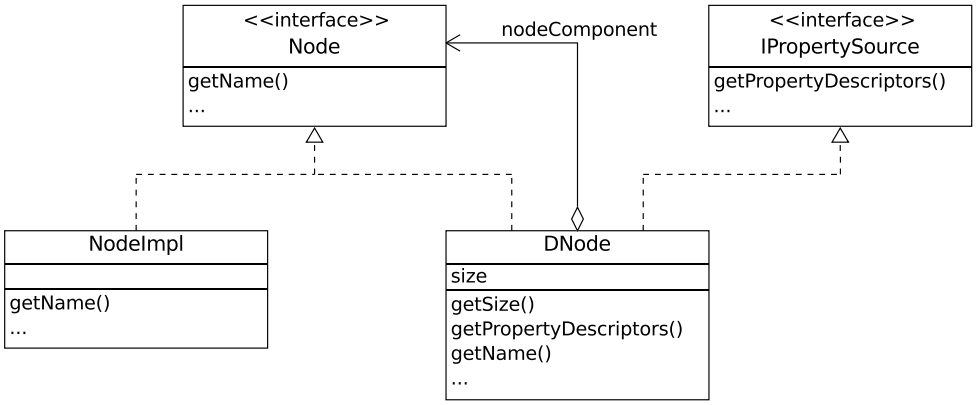


Figure 6.4: The DECORATOR pattern, as implemented in the Display Model. This figure displays the solution for the `DNode` class, but similar solutions were implemented for `DArrow` and `DConstraint`. For illustrational purposes, only a limited number of methods are shown. The interface `Node` and its implementation `NodeImpl` are located in the `no.hib.dpf.core` package. The `DNode` class belongs to the `no.hib.dpf.editor.displaymodel` package. `IPropertySource` is an Eclipse interface.

It should also be pointed out that the Java implementation of this hierarchy – as generated by EMF – consists of both *interfaces* confirming to the model's class hierarchy and Java *classes* implementing these interfaces. For instance, the `Node` class as modelled in Ecore yields a Java generated `Node` interface as well as an implementation of this interface located in the `NodeImpl` class. This way of generating code facilitates the use of the ABSTRACT FACTORY pattern [30], which decouples the creation of objects from their use.

6.4.2 The Display Model

In addition to the Core Model component, we had to implement user-manipulable counterparts to some classes in the `no.hib.dpf.core` package, namely `Node`, `Arrow`, and `Constraint`. As the user should be able to manipulate instances of these classes on-screen, two structural additions were needed:

- Position and size information for movable and sizeable objects
- Implement the `IPropertySource` interface for supply of display properties to the model's corresponding EditParts

This structural addition was achieved by implementing a simplified variant of the DECORATOR pattern [30]: Display Model counterparts to the Core Model classes `Node`, `Arrow`, and `Constraint`, named `DNode`, `DArrow` and `DConstraint` respectively, are put in the sub-package named `no.hib.dpf.editor.displaymodel`. Each of these classes implements the interface from their Core Model counterpart and at the same time maintains a reference to an instance of an implementation of this interface. Figure 6.4 illustrates how this is implemented for descendants of the `Node` interface.

Using this implementation method, we can instantiate classes from the `no.hib.dpf.core` package inside our editor. These instances have added function-

ality and data, but this additional information is decoupled from classes in the Core Model. The consequence of this is that we can regard the Display Model and Core Model components as one and the same in the context of GEF and the DPF Editor.

In addition to these couplings, the graph of the diagram itself must be managed. This management is done by the class `no.hib.dpf.editor.displaymodel.DPFDiagram`. This class is responsible for maintaining a reference to a `Graph` object from the Core Model. This `Graph` object serves as the specification graph for the diagram being edited.

A note on concrete syntax

As the above arrangement hints at, the concrete syntax ("how [...] modeling concepts are rendered by visual and/or textual elements" [3, page 1]) of our editor is hard-coded to consist of arrows, two-dimensional nodes, and constraints connected to these. Future solutions may focus on decoupling the editor's concrete syntax from the editor environment. For instance, the implementation of UML-like class definitions with compartmentalized graphical representations, containing attributes and operations, would require such a decoupling. The division of the model into Core Model and Display Model should hopefully be of assistance in this task. See also section 8.2.3, page 67.

6.4.3 Persistence/XMI

Although the Display Model and the Core Model components have been "fused" together by implementing the Decorator pattern, one issue remains when it comes to persist the contents of these models.

As mentioned in section 5.1.2, EMF models are easily serialized to the XMI format. This means that we are handed an easy means to persist the Ecore part of DPF specifications edited in DPF Editor.

The Display Model, being constructed as Plain Old Java Objects (POJOs), does not offer this convenience. Thus, we adopted the functionality that came with Eclipse's "Shapes" plug-in example (see section 6.9.3, page 55), namely using the `writeObject()` method of `java.io.ObjectOutputStream` to serialize the Display Model objects to binary "snapshot" files. Leaving the references to the Core Model instances `transient`, we could thus serialize only the "display model" part of the model objects, leaving the "core" part to be serialized by means of XMI.

Deserialization was handled by applying the `readObject()` of `java.io.ObjectInputStream` to serialized display models. When deserializing, the process is reversed, reading both Display Model and Core Model objects separately before joining them together in a single operation. In order to achieve this, we needed some method of identification for Core Model objects, so they could be referenced from restored Display Model counterparts. This was achieved by letting the `Node`, `Arrow`, and `Constraint` classes inherit from the new class `no.hib.dpf.core.IDObject`. This class generates a unique ID (a string representation of a random `java.util.UUID`) and implements a single method, `getId()`

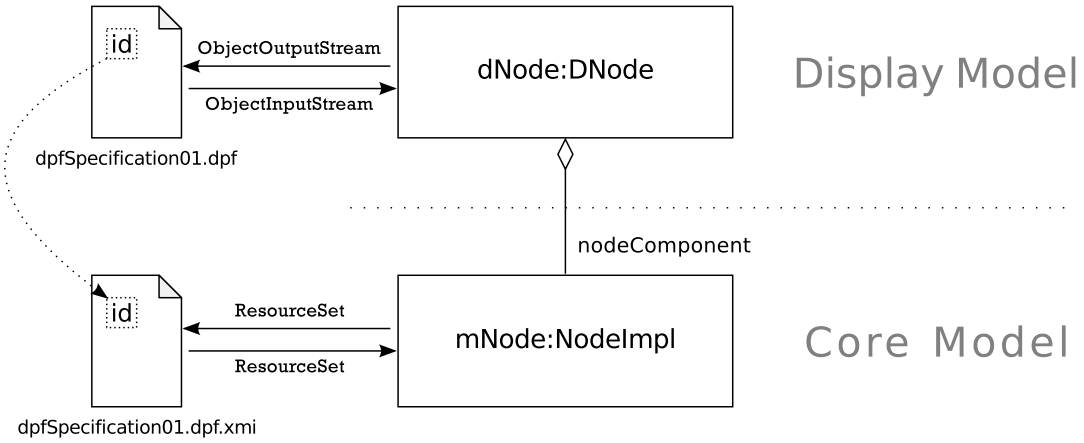


Figure 6.5: Serializing a single node in the model. This depicts only part of the process of serializing/deserializing a complete specification, but the pattern is the same for all objects. A DNode instance in the Display Model is serialized using `java.io.ObjectOutputStream` and deserialized using `java.io.ObjectInputStream`. Its counterpart from the Core Model is serialized to and deserialized from an XMI file using `org.eclipse.emf.ecore.resource.-ResourceSet`.
The ID of the NodeImpl instance is kept in the serialized version of the DNode, facilitating re-coupling of the DNode and NodeImpl instances after deserialization. The two files, together containing the serialized specification, have the same name, but different suffixes.

that returns this unique ID. This functionality lets us maintain a Core Model reference in a Display Model instance throughout the serialization-deserialization operation by just serializing the ID in the Display Model instance. Figure 6.5 shows how this works for a single node in a specification.

This has the downside of creating two separate files for each DPF specification edited in the plug-in. At present, the only method of synchronizing such a pair of files is by giving them identical names, albeit with different suffixes. This is planned resolved in a future version of DPF Editor. The file that is associated with the DPF Editor is the file containing the display model, having a `.dpf` file name suffix. The file containing the Core Model instance is given a `.dpf.xmi` file name suffix.

6.5 The editor

In addition to the display model, the `no.hib.dpf.editor` namespace contains three major components: the EditParts, the figures, and the main editor implementation (see figure 6.2, page 40). In this section, we will examine these components in turn.

6.5.1 Controllers/EditParts

As mentioned in section 5.1.3, the GEF controller is made up from a collection of instances of `org.eclipse.gef.EditPart`[1]. In DPF Editor, this collection is placed in the `no.hib.dpf.editor.parts` sub-package. For each user-editable class in the model, there exists a `EditPart` class that implements a GEF controller. In our case, these EditParts are designated `NodeEditPart`, `ArrowEditPart`, and `ConstraintEditPart`. In addition, an EditPart corresponding to the `DPFDiagram` object in the model has been built. This class (`DiagramEditPart`) serves as the main container for the whole edit area belonging to the editor. The diagram edit part is responsible for the container's layout and its edit policies.

To keep track of changes in the model, the EditParts in DPF Editor implement a simple variant of the OBSERVER pattern [30]. This is done by having all EditParts implement the `java.beans.PropertyChangeListener` interface. Additionally, all Display Model elements maintain references to instances of `java.beans.PropertyChangeSupport`, which trigger change events in a connected instance of `PropertyChangeListener`. To set up this pattern, EditParts instances add themselves as listeners to their corresponding model elements (`NodeEditPart` to `DNode` etc). When changes are committed to the properties of a model element, the element calls the `firePropertyChange()` method on its `PropertyChangeSupport` instance to notify any listeners that a change has been made. This pattern keeps the model elements and EditParts loosely coupled without the model elements having to directly reference any EditPart.

6.5.2 Figures

GEF relies on instances of `org.eclipse.draw2d.IFigure` to render graphical components to the display. For DPF Editor, we implemented separate figures for nodes, arrows, and constraints. Realizing these components, we kept close to previous efforts' visualizations, as well as those given by the DPF. See table 3.2, page 10 for proposed visualizations for a selection of constraints.

Our implementation is placed in the `no.hib.dpf.editor.figures` subpackage. For nodes and arrows, we built our code on existing classes from the "Shapes" plug-in example (see section 6.9.3, page 55). Node figures were realized by extending `org.eclipse.draw2d.Figure` and arrow connections by extending `org.eclipse.draw2d.PolylineConnection`. We encountered a more challenging task when faced with implementing the visualizations for multi-line constraints. Initially, we implemented an "arrow-spanning" constraint, the type where a constraint visualization spans two arrows as in the predicates `[jointly-injective]` and `[jointly-surjective]` from table 3.2. Implementing this kind of visualization as a Draw2d connection required extending `PolylineConnection` and overriding the `outlineShape()` method of that class. Later, we also implemented constraint visualizations belonging to predicates with an arity containing just one arrow (for instance `[mult(m,n)]`), as well as visualizations spanning two parallel arrows (`[inverse]`, `[image-inclusion]`).

[1] All Eclipse namespaces are documented at [16].

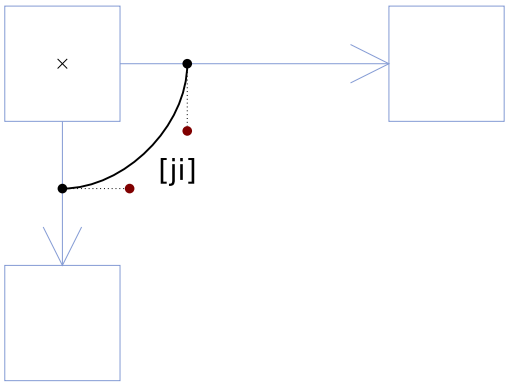


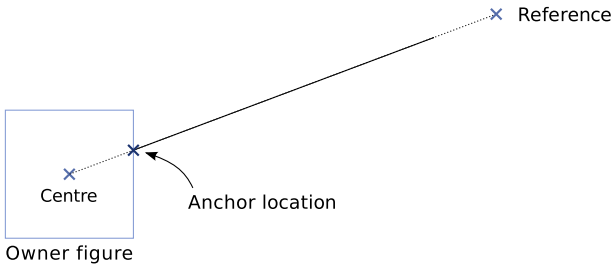
Figure 6.6: An arrow-spanning constraint visualization, in this case representing the `[jointly-injective]` predicate from table 3.2. The two anchor points for the constraint line are emphasized with black dots. The two Bézier control points are marked with red dots in the figure (these are not drawn to the display as the control points are fixed and not user-editable). The centre of the adjoining node, marked "x", is also used to calculate on which side of the constraint the control points should be placed and to which side of the node the constraint should be drawn. The label "[ji]" is also drawn to the graphic display by the figure.

In contrast to arrows, which are anchored to figures (see below), constraint visualizations that span arrows (as shown in figure 6.6) must be anchored to some point on these arrows. These points are calculated by traversing the arrows a fixed distance from a node to which the arrows are connected. Having calculated these points, we implemented the arrow-spanning constraint visualizations by drawing a cubic Bézier curve [28] between the two points, calculating the control points for the curve using the two control points and the adjoining node for reference. See figure 6.6 for a stylized view of this kind of constraint visualization. Constraints with single arrow visualisations usually just print a label text to the GEF canvas.

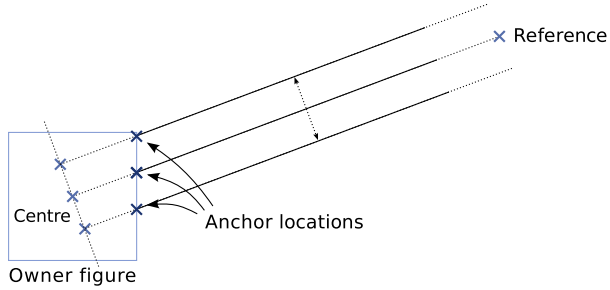
Routing and Draw2d customization

Draw2d features a *layout* mechanism that is usually triggered when an on-screen item is added, removed, moved, or resized. Layout features *routing* of connections. In the simplest case, routing involves drawing a (connection) line between two nodes' anchors. An 'anchor' in this setting is simply a point on the canvas. When only one arrow connection between nodes is required, the functionality available in the class `org.eclipse.draw2d.ChopboxAnchor` is sufficient (figure 6.7(a)). Instances of this class calculate the point at which the bounds of a node's figure are intersected by the line travelling from some reference point to the centre of the figure [38].

When the user has drawn more than one arrow between nodes, however, using `ChopBoxAnchor` results in two or more arrows being drawn on top of each other. This is clearly not desirable from a usability standpoint. As Draw2d does not offer a suitable alternative, we had to implement one ourselves. The class `no.hib.dpf.editor.figures.MultipleArrowsChopboxAnchor` provides a solution for this situation. The class expands on `ChopBoxAnchor` by allowing for



(a)



(b)

Figure 6.7: Figure 6.7(a) shows the `org.eclipse.draw2d.ChopboxAnchor` class in principle. The ChopboxAnchor's location is found by calculating the point at which the bounds of a figure are intersected by the line travelling from some reference point to the centre of the figure [38]. Adapted from [38] and [35].
Figure 6.7(b) shows the `no.hib.dpf.editor.figures.MultipleArrowsChopboxAnchor` class in principle. The figure shows a situation where three arrows' connections are connected to a node's figure. The line travelling from some reference point to the centre of the figure is used to generate one or more vectors (two in this example, shown with dotted arrows), orthogonal to the line. The vectors' length depends on the size of the owner figure. Based on each such vector, a new (offset) internal reference point and a new line are defined from which new anchor points on the figure's bounds are found. Symmetry across two figures is achieved by having the leftmost and/or lowest figure always calculate the anchor points from left, and *vice versa*.

more multiple arrows entering or leaving a node getting separate anchor points. For each arrow connection, a vector (in the geometric sense of the word), orthogonal to the arrow's direction, is created. The vector's length depends on the node's dimensions, being set at a fixed rate to the node's internal size. This vector is then used to calculate a new internal reference point in the node. Then, the arrow connection is translated using the vector, and a new anchor point on the bounds of the node's figure can be calculated (figure 6.7(b)).

By using `PolylineConnection` instances, we let Draw2d do more advanced routing than just drawing straight lines. Polyline connections are automatically routed around any figures that lie on their path. This routing is performed by an instance of `org.eclipse.draw2d.ConnectionRouter` [15]. In our case, this has the implication that constraint connections, being dependent on the positions of the arrows they span, must be laid out after the arrow connections have been laid out. Unfortunately, Draw2d does not contain a mechanism that lets us delay the layout of constraints until after all the connections have been laid out.

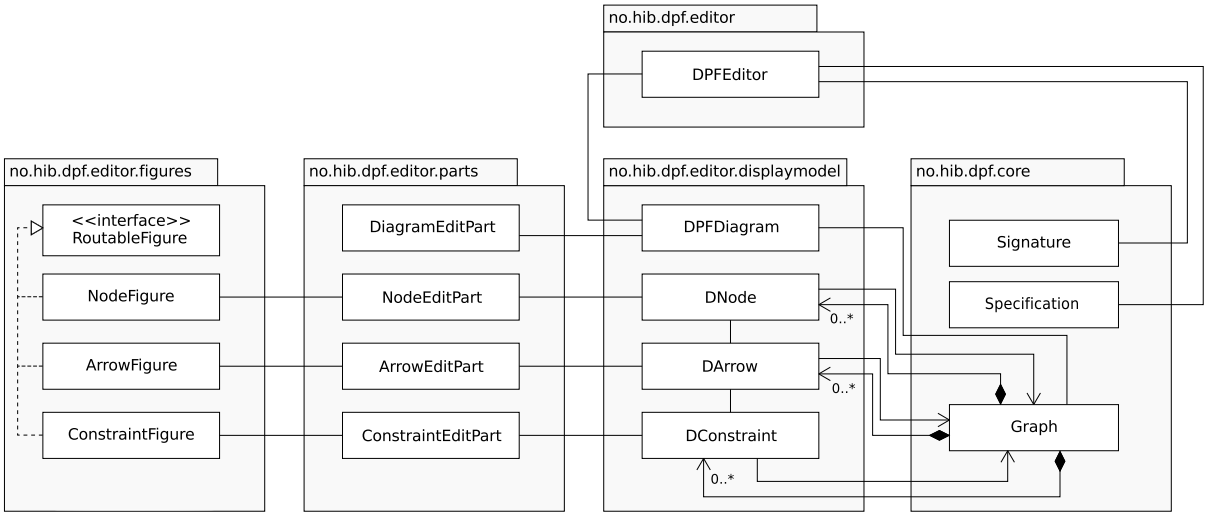


Figure 6.8: The `DPFEditor` class and its interaction with the non-GEF classes and packages. This figure shows only some selected classes from each package and their references across package borders. Fields, methods, and association names have been left out for clarity. Associations with multiplicity equal to 1 have been drawn without a multiplicity indicator. Note that the `Graph` object contains and interacts with `Node`, `Arrow`, and `Constraint` objects from the `displaymodel` package. The `Specification` class' containment of `Graph` (figure 6.3) is omitted for clarity. Classes in the packages `no.hib.dpf.editor.parts` and `no.hib.dpf.editor.figures` implement the GEF "controller" and "view" functionality, respectively.

We got around this problem by introducing a new interface, `RoutableFigure`, that was implemented by our figures and connections. This interface published a method, `getRoutingPriority()`, that lets the implementing class signal its priority in the routing queue. Lastly, we custom-designed a `ConnectionRouter` that overrides `org.eclipse.draw2d.AbstractRouter` and lays out elements which implements `RoutableFigure`.

6.5.3 The DPF editor

The functionality that ties all the plug-in classes together is placed in the class `no.hib.dpf.editor.DPFEditor`. This class extends the abstract class `org.eclipse.gef.ui.parts.GraphicalEditor`, a general implementation of a GEF editor. Its main responsibilities are:

- Creating and maintaining a reference to a `Signature` object
- Creating and maintaining a reference to a `Specification` object
- Creating and maintaining a reference to a `DPFDiagram` object
- Creating the palette
- Creating actions for constraint applications

At present, the `Signature` object contains a set of predicates that closely match the signature definition from table 3.2. This definition is not hard-coded, but loaded from a file in the user's Eclipse workspace. If this file cannot be found

when the plug-in is loaded, it is generated by the editor. Note that this signature definition (from the Core Model) in its present state lacks a visualization component. The corresponding constraint visualizations are, as indicated above, implemented as fixed classes residing in the `no.hib.dpf.editor.figures` subpackage.

The `Specification` object belongs to the CORE MODEL. This object contains a main graph object that contains references to all nodes, arrows, and constraints that are under editing. It also contains a type graph object (see below).

As mentioned, the `DPFDiagram` instance ties together the graph from the specification and the DISPLAY MODEL data structure (figure 6.8). The editor handles the `DPFDiagram` instance as the root of the display model, and it is this object (and its referenced objects) that is serialized to a `.dpf` file when the contents of the editor are saved to disk. In the plug-in's manifest file [9] (`plugin.xml`), the `.dpf` file name suffix is associated with the plug-in, making Eclipse load the plug-in when the user activates this type of file.

Plug-in

The class `no.hib.dpf.editor.DPFPlugin` extends `org.eclipse.ui.plugin.-AbstractUIPlugin`. This gives us support for plug-in preferences, i.e. the ability to save and restore user preferences through a simple interface. By implementing a SINGLETON pattern [30], an instance of this plug-in class is available project-wide.

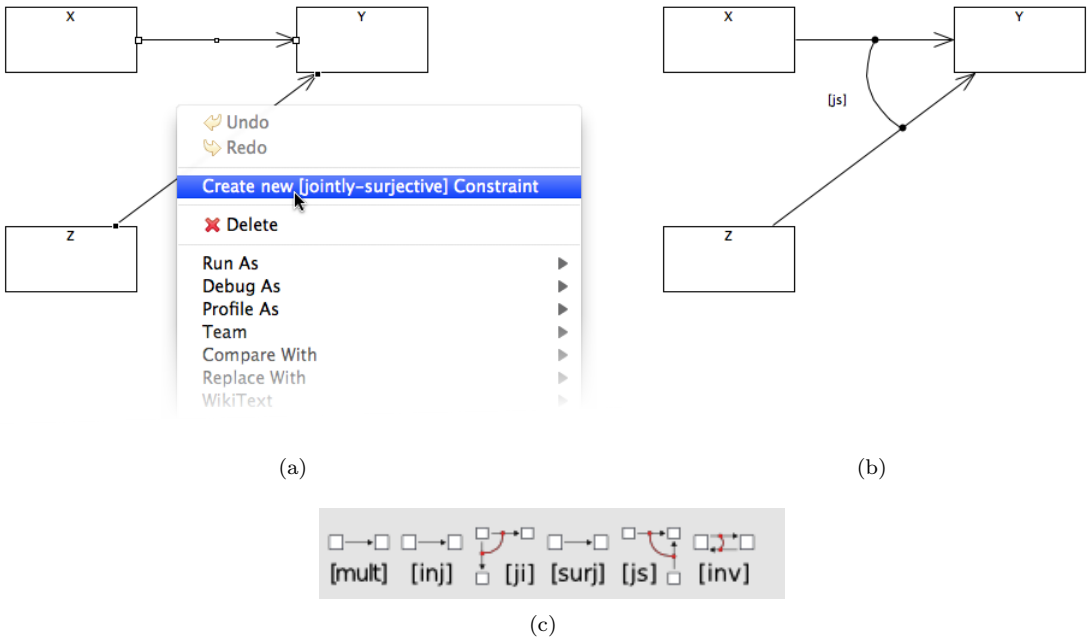
6.6 The user interface

In addition to editing items on the GEF canvas, the user interacts with DPF Editor through the *palette*, *constraint actions*, the *creation wizard*, and the *preference page*. We will briefly describe these features here.

6.6.1 The palette

The GEF editing paradigm places editing tools in a palette. The palette in DPF Editor functions in the same way as a common toolbar, icons representing selection tools as well as nodes and arrows that can be added to the specification being edited. The palette is dockable; it can be docked on either side of the editing canvas. It can also be shown as a standard Eclipse *view*, i.e. it can be put in any dockable window in the Eclipse editor. Figure 6.11, page 53 shows the DPF Editor palette in use.

When the user selects a *type graph* that differs from the default (see below), the palette will be configured to display a selection of nodes and arrows from that particular type graph.



(a) (b)

(c)

Figure 6.9: Applying a constraint to a diagram. The partial screenshot in figure 6.9(a) shows a situation where the user has selected two arrows. No dangling arrows are allowed in graphs, so to make a valid graph, DPF Editor adds the three nodes adjacent to the arrows to the selection graph. Then, all available predicates are compared to this new "selection graph", trying to find a graph homomorphism from the arity of the predicate into the selected subgraph. In this particular configuration, the `[jointly-surjective]` predicate's arity can be applied. When invoking the context menu, the user is presented with the choice of applying a constraint from this predicate onto the diagram. The partial screenshot in figure 6.9(b) shows the resulting visualization, the constraint being drawn between the two arrows. The partial screenshot in figure 6.9(c) shows part of DPF Editor's toolbar, displaying available constraints from a signature.

6.6.2 Constraint actions

As stated earlier, DPF constraints can be added to specifications. The user can *constraint* a graph in the editor by using the mouse to select the nodes and arrows that shall be constrained, thereby creating a *subgraph*, called the 'selection graph'. Then, the system decides if any constraint is available for the selection, trying to find a graph homomorphism from the arities of all available predicates into the selection graph. If any such constraint is available, the user can select it, either from the context menu (figure 6.9(a)), or by clicking a constraint button on the toolbar (figure 6.9(c)).

Recall that a constraint is given by a predicate symbol and a graph homomorphism from the arity of a predicate into a graph (definition 4, page 9). In our case, this graph is the main `Graph` object contained by the `Specification` being edited. Also, the editor maintains a reference to a `Signature` object. This signature contains the predicates we can choose from when creating a new constraint. In order to automate this process, we have to provide functionality for finding a graph homomorphism from one graph (the arity of each predicate) into another (the main graph of the specification).

Finding graph homomorphisms

We have implemented a simple algorithm for finding graph homomorphisms from one graph to another. This algorithm works by taking two graphs, source graph G and target graph H. We then want to find at least one graph homomorphism $G \to H$. The algorithm achieves this by producing all permutations of nodes in the target graph H and then checking for possible graph homomorphisms $G \to H$. This kind of algorithm possesses an exponential running time, and is as such not suited for large graphs. Indeed, according to Hell and Nešetřil [40], for each fixed simple graph H, the process of deciding whether a simple graph G contains a homomorphism to H is NP-complete if H is not a bipartite graph.

For our practical purposes, this will most likely remain unproblematic. First, the predicate arities in question (graph G) are not very large, and matching two small graphs (typically ≤ 5 nodes each) does not have a prohibitive cost in computing power. Second, the set of nodes and arrows selected by the user (graph H) must not have a node count that exceeds the node count from the arity graph (graph G). If so, the search will default to false.

There is a possibility that this solution will not suffice for all types of predicates and graphs to be created with DPF Editor in the future. In such cases, a solution where the user manually constructs the graph homomorphism may be implemented. This solution would then "kick in" when node count in one or both graphs exceed a pre-set limit.

6.6.3 The creation wizard

In Eclipse, an editor plug-in is normally associated with a certain file type. As mentioned, DPF Editor is associated with files having the `.dpf` file name suffix. The standard way for a user to create such a file, is to invoke a creation wizard which steps the user through the process of creating a new file used for a plug-in's workspace.

At present, our plug-in possesses one such creation wizard, which lets the user specify

- The name and placement of the file containing the specification[2]
- An optional specification containing a type graph (see below)

The wizard was implemented by extending `org.eclipse.jface.wizard.Wizard` and implementing `org.eclipse.ui.INewWizard`. Figures 7.2(a) and 7.2(b), page 61 displays the wizard being used.

6.6.4 The preference page

In order to let the user specify certain values that define how the plug-in should look and feel, we have created a simple preference page. This was achieved by

[2] As mentioned, the file gets a "twin" with the `.dpf.xmi` extension when Core Model objects are persisted.

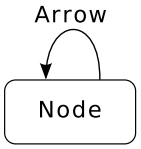


Figure 6.10: The graph of the default metamodel, consisting of a single node and a single arrow self-referencing the node. This graph will type any other graph.

extending `org.eclipse.jface.preference.PreferencePage`. At present, the preference page for DPF Editor lets the user select the colouring for nodes and whether type names and/or arrow names should be displayed on-screen.

In addition to this, we have added zoom capabilities to our editor, letting the user change the zoom level of a diagram being edited. Also, the user can superimpose a grid onto the editing area. Nodes can be made to snap to this grid, easing the process of aligning nodes in a diagram.

6.7 Metamodelling and type graphs

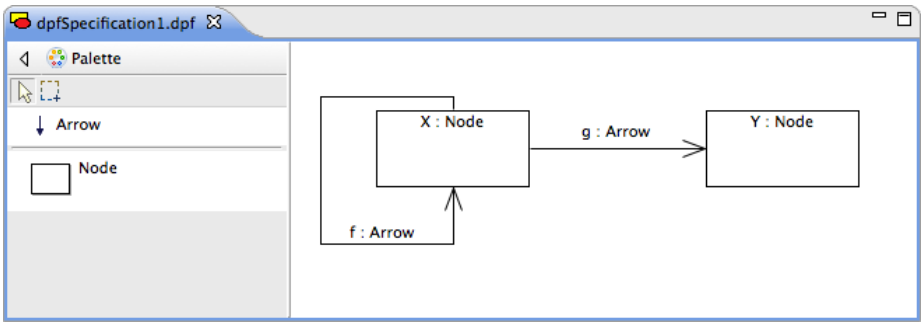
In order to positively answer our research question (section 4.3, page 27), we had to augment our basic DPF specification editor to support metamodelling at an arbitrary number of meta-levels. This was achieved by letting the `Specification` class maintain a reference not only to a main graph (the graph under editing), but also to a type graph (definition 6, page 9) that types the nodes and arrows in the main graph.

The default type graph

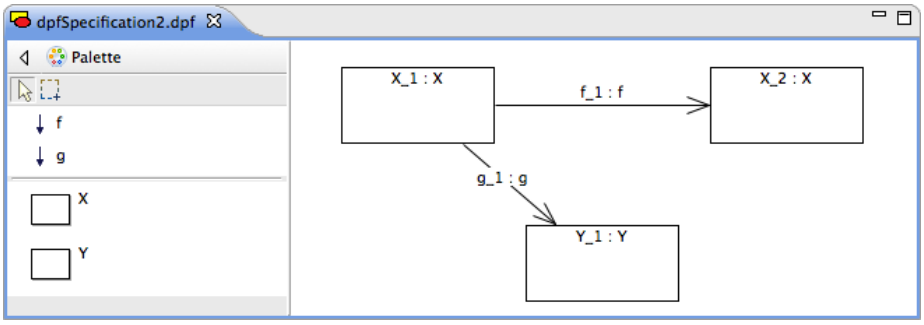
In the default case, the user does not specify a type graph. In this case, the editor creates a default type graph, containing two elements, a node and an arrow (figure 6.10). Applying this type graph, nodes in the typed graph will be of type Node and arrows in the typed graph will be of type Arrow. Any Arrow may reference any Node, and therefore the default type graph can type any graph created by a diagram in the editor. Figure 6.11(a) shows how the palette looks when the default type graph is used.

Creating custom type graphs

Instead of using the default type graph, the user can specify that the graph belonging to another specification should be used as a typing graph. This is done in the creation wizard by specifying a previously saved specification file (`.dpf.xmi`-extension). Upon wizard completion, the palette displays the nodes and arrows available from the new typeset. DPF Editor will also respect the resulting typing homomorphism by restricting the user's ability to draw connections between nodes in the diagram when any connections are not found between the nodes' corresponding types in the typing graph. Figure 6.11 shows an example of a specification being used to define the type of another specification.



(a)



(b)

Figure 6.11: Two specifications being edited in DPF Editor. The first is defined using the default type graph. The second specification has been created using the graph from the first specification ("dpfSpecification1.dpf.xml") as the type graph. Note how the node and arrow names from figure 6.11(a) are repeated in the palette in figure 6.11(b). Also, in order to respect the type homomorphism, the editor will not let the user draw an arrow from a "Y" node to an "X" node in the lower diagram.

6.8 Semantics validation

As noted in chapter 3, in an editor one could implement a validator for each predicate. The task of constructing a *general-purpose* validator for DPF semantics is beyond the scope of this thesis. For prototyping purposes, however, we have implemented some (hard-coded) validators. These descendants of the `no.hib.dpf.core.SemanticsValidator` class have been made for the `[mult(m,n)]`, `[inverse]`, and `[jointly-surjective]` predicates as given in table 3.2, page 10. Other predicates have – for the time being – been equipped with a default validator that always returns true.

When deciding whether a specification graph is *valid* in DPF, we have to regard the constraints that have been applied to the specification's type graph.

Listing 6.1 shows a simplified implementation of the specification validator. In general terms, it implements the pullback diagram from definition 8, page 9. This method is triggered whenever the diagram is changed, i.e. when a node and/or an arrow is added or removed.

```
public boolean isSpecificationValid(Specification spec) {
  boolean isValid = true;
  for (Constraint c : spec.getTypeGraph().getConstraints()) {
      Graph oStar = spec.createOStar(c);
      isValid &= c.getPredicate().validateSemantics(oStar);
  }
  return isValid;
}
```

Listing 6.1: A simplified listing of the semantics validator.

6.9 The development process

From the onset, we had to take into account two key factors regarding development. First, in accordance with Skjerveggen's [70] previous efforts, we had committed ourselves to using EMF modelling tools for the data model. Second, we had also rejected GMF as a workable platform, thereby defaulting to the use of GEF as our toolkit of choice.

6.9.1 Development method and XP practices

As mentioned, we selected the XP methodology [6] as the basis for our development effort. As no real customer representative were available to us on a continuous basis, we had to adapt and downscale the XP practices related to planning and customer tests slightly. Several members of the DPF group were available for planning meetings. Informal tests and presentations were carried out throughout the autumn 2010 and winter 2011. Our focus was on employing *simple design* [52], getting as much functionality as needed while constantly seeking to minimize code size and simplify the overall design.

For most of the development part of this project, we had the advantage of being two developers working on the code. This allowed us to employ *pair programming* [52] as the primary coding (and modelling) practice.

A key XP practice is *test-driven development* (TDD), the practice of driving the development process by working in short cycles, adding a failing test, then making it work. [52] The set of EMF tools that comes shipping with Eclipse lets the user set up tests related to models created using the Ecore modelling tools. These tests are based on the application of the JUnit testing framework [44]. JUnit is now the de facto standard for unit testing Java source code [51, page 84]. It might be noted that, as tests are generated up front on the basis of modelled classes, this way of doing TDD deviates somewhat from the "pure" practice. Using the generated tests, we could nevertheless drive the development of method functionality and refactor all non-public functionality, as we would otherwise have done in TDD.

6.9.2 Source control and automated builds

We used available source control resources at HiB as our source code repository throughout the development process.

To achieve continuous integration and automated building, we installed and set up Hudson [41], an open source *continuous integration* [52] (CI) server. Using this tool, we achieved having the latest build available as reports on any failing tests that may have been committed to source control.

Agile methodologies such as XP explicitly state that the amount of documentation work products external to the code should be minimized. This practice is sometimes paraphrazed as "the source code is the documentation". Shore and Warden [68] point out that (written) documentation should be created until practices are in place to replace it. Also keeping in mind that participants will enter and leave our project on a steady basis, *handoff* documentation will also be required at times. We have practiced *collective code ownership* as well as pair programming to distribute code knowledge better among project participants. In addition, we have set up a wiki[3] to concentrate work products such as to-do lists and documentation snippets. This wiki is readable and editable by all project members.

6.9.3 Special practices regarding GEF

As GEF requires the developer to code every element by hand, the work needed to construct even basic GEF-based applications can be substantial. Also, commands and property listeners are typically used for object instantiation, inter-module messaging, and communication. This has the advantage of making GEF very loosely coupled and thereby easy to extend. The downside is that even simple debugging becomes non-trivial, as most messages are encapsulated inside command queues that are not easily monitored in a debugger. This, and other factors such as the sheer code size[4], makes the learning curve of GEF quite steep.

To the author's knowledge, there does not exist any professional book that solely focuses on GEF. Some good sources do exist, most notably [9, 38, 36, 69]. Moore et al. [7] is probably the most complete source on the subject, but has become increasingly outdated, as GEF has been developed further. This lack of documentation has probably affected our productivity. Neither of the project participants had any notable prior expertise in developing GEF applications. Consequently, we initially resolved to what can be described as "exploratory programming" – we developed small subsets of our plug-in piecemeal in order to learn or discover how certain aspects of GEF manifests themselves. This practice loosely resembles what in Scrum [76] is called a "sprint".

To get started using the GEF platform, we ran the example GEF applications that come shipped with Eclipse. We managed to use the "Shapes" plug-in example as a basis for our work, in the process re-working the code to our

[3] See http://en.wikipedia.org/wiki/Wiki

[4] Version 3.6 of the GEF(MVC) class hierarchy contains over 350 Java classes and interfaces. Draw2d adds a further 250.

needs. This had the advantage of getting a workable editor quickly up and running. The disadvantage was a large piece of code that we had to "decipher" over time as the development progressed.

6.9.4 Tool shortcomings

Modelling tools

We initially did some development on the Core component using the *Ecore Diagram Editing tool* [8], a graphical modelling editor that is included in EMF's set of tools. Unfortunately, this tool sometimes seemed to generate invalid model definitions, thus not giving the impression of a tool ready for production use. We were also not satisfied with the layout algorithm, having to struggle to control the graphical layout of references within the model. As a consequence, we fell back to using the tree-based Ecore editor as the main model definition tool. This turned out to be a satisfactory solution, apart from the issues we encountered with refactoring (see below).

Lack of refactoring support

Successful application of TDD relies heavily on being able to apply the XP method *refactoring* for improving code quality and generally guide the design process [68]. Doing refactoring using Ecore with EMF tools, however, did prove to be a troublesome experience. The Ecore modelling tools currently lack support for automated refactorings such as changing method and class names.

The modify-save-generate cycle thus needed to do simple refactorings was both time and labour intensive, making us put off simple refactoring tasks and focus more on planning than we would have done in a traditional Java programming scenario. Of course, this does not comply very well with an agile approach, where the developer ideally lets TDD and refactorings guide the design process rather than doing large parts of the design work up front.

Chapter 7

Tool demonstration

In this chapter, we intend to show the tool in use on a small, but hopefully sufficiently advanced example. This will be a brief demonstration of DPF Editor's capabilities, employing both its metamodelling functionality as well as the tool's ability to validate semantics based on a metamodel's constraints. This demonstration serves as a validation of the tool's capability to perform a modelling task based on the DPF.

7.1 Demonstration setup

We will set up the demonstration by specifying an example information system that models students' course enrolments at universities. The system will consist of:

A set of classes:

- Person
- Organization
- Activity

Further concretized as:

- Student
- University
- Course

References between entities:

- Universities can reference students and courses
- Students can reference courses and universities
- A course can reference students

A set of constraints:

- A student must reference at least one and at most four universities

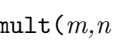
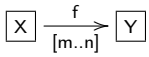
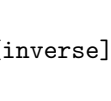
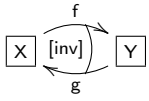
π	$\alpha^{\Sigma_2}(\pi)$	Proposed vis.	Semantic interpretation
`[mult(m,n)]`			$\forall x \in X : m \leq \lvert f(x)\rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
`[surjective]`			$\forall x \in X : \bigcup\{f(x)\} = Y$
`[inverse]`			$\forall x \in X$, $\forall y \in Y : y \in f(x)$ iff $x \in g(y)$

Table 7.1: The signature Σ_2 used in the metamodelling example

- A student must be referenced by at least one university
- References between students and universities must be inverse
- References between students and courses must be inverse

We implement this system by constructing a metamodelling hierarchy in DPF. This hierarchy will consist of four metamodelling layers and one instance layer. At the uppermost M_4 layer (figure 7.1), we place a graph corresponding to DPF Editor's default type graph (section 6.7, page 52).

We then continue modelling, defining specifications at layers M_3, M_2, and M_1. Each specification's graph is typed by the graph at the layer M_{n+1}.

At the M_3 layer, we define the specification $\mathfrak{S}_3 = (S_3, C^{\mathfrak{S}_3} : \Sigma_4)$, which models the concepts *Class* and *Reference*.

At the M_2 layer, we define the specification $\mathfrak{S}_2 = (S_2, C^{\mathfrak{S}_2} : \Sigma_3)$, which models the concepts *Person*, *Organization*, and *Activity*. No constraints have been added to the hierarchy at layers M_3 and M_2, i.e. the signatures Σ_4 and Σ_3 are empty.

At the M_1 layer, in addition to type the specification $\mathfrak{S}_1 = (S_1, C^{\mathfrak{S}_1} : \Sigma_2)$ using the graph from layer M_2, we also let the specification be defined by a signature Σ_2 (table 7.1), which contains the predicates that we will need to constraint the model at the instance layer, M_0. Specification $\mathfrak{S}_1$ contains definitions for *Students*, *Universities*, and *Courses*.

At the instance layer, M_0, we will instantiate a model typed by $\mathfrak{S}_1$ that represents an extract of a system describing students' course enrolments.

7.2 Tool preparation

In order to implement the information system as described above, we had to implement the individual predicates belonging to the signature Σ_2. Two predicates – `[inverse]` and `[mult(n,m)]` – were already present in the system. The remaining predicate – `[surjective]` – was added before we began modelling. As there is no stand-alone predicate editor available for DPF Editor yet, the predicate had to be hard-coded into the system, including functionality for semantic validation (see section 6.8, page 53).

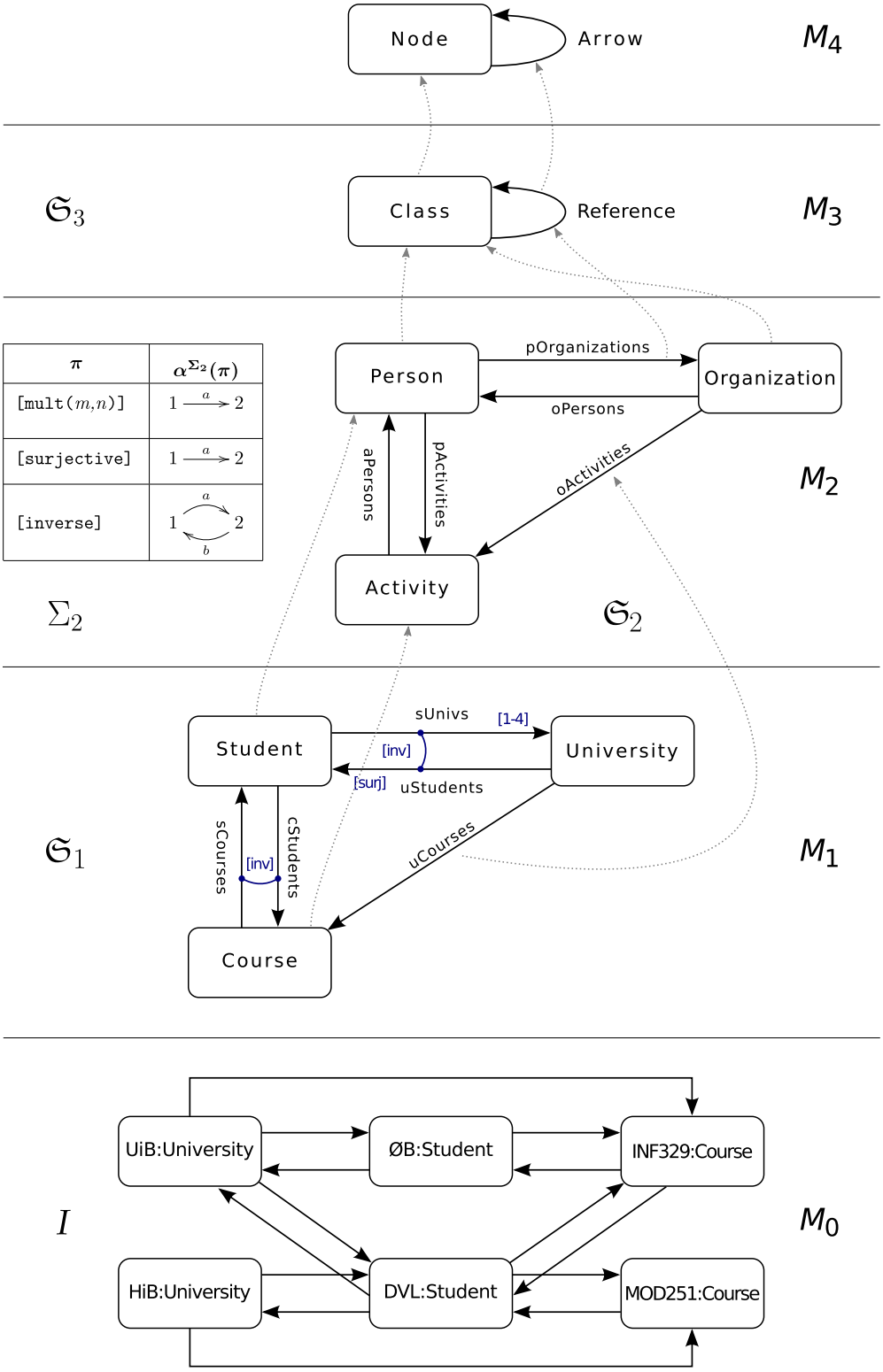


Figure 7.1: A metamodelling example. The figure shows the metamodelling hierarchy that we wish to model using DPF Editor. Layers are labeled M_4 though M_0, echoing the naming scheme from OMG's 4-level hierarchy. The graph at the M_4 layer corresponds to the default type graph in DPF Editor. The graphs at layers M_3, M_2, and M_1 belong to specifications $\mathfrak{S}_3$, $\mathfrak{S}_2$, and $\mathfrak{S}_1$ respectively. At layer M_0, an instance of $\mathfrak{S}_1$, named I, is shown. Constraints applied to $\mathfrak{S}_1$ are shown in blue, and some (but not all) typing morphisms are indicated with grey, dotted arrows.

7.3 Tool demonstration

In this section, we demonstrate the essential steps for implementing the information system as described above. DPF Editor runs inside Eclipse, and the user activates the tool's editor by selecting a project folder and invoking the wizard for creating a new DPF Specification Diagram (figure 7.2(a)). The new file, containing the specification, is given the name `m3.dpf`.

Then, the user can edit the specification $\mathfrak{S}_3$ at the M_3 layer, creating definitions for *Class* and *Reference*. Figure 7.3(a) shows a screenshot of this editing process.

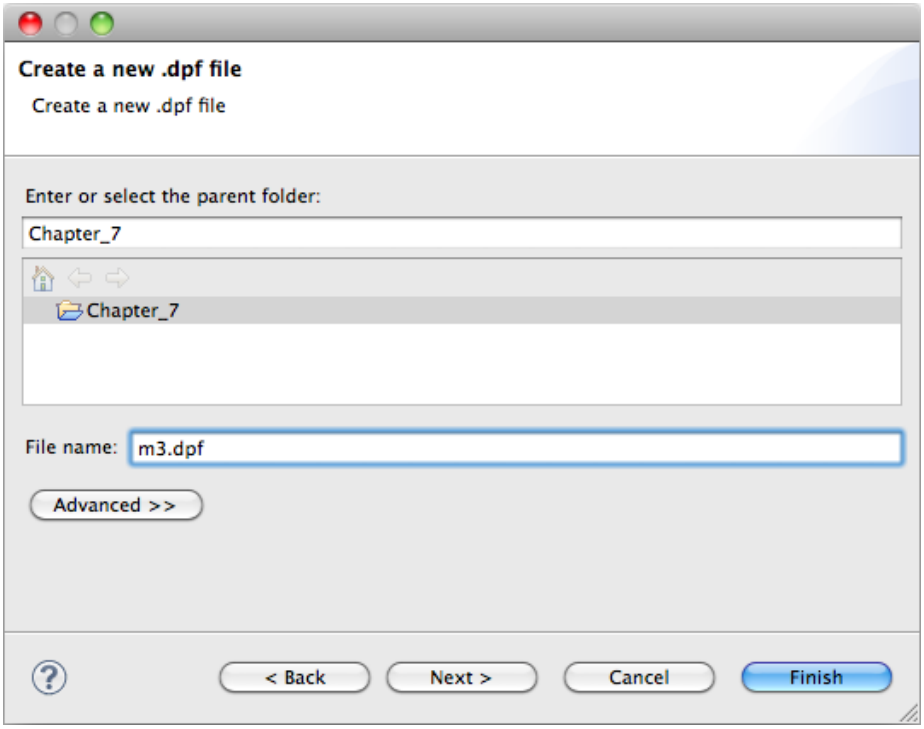
Now, we want to use the graph from the specification $\mathfrak{S}_3$ at layer M_3 as a type graph for a new specification $\mathfrak{S}_2$ at layer M_2. This is achieved by invoking the wizard for creating a new DPF Specification Diagram once more. This time, in addition to specify that our file shall be called `m2.dpf`, we also specify that the file `m3.dpf.xmi` will contain the type graph for our new specification (figure 7.2(b)). Figure 7.3(b) shows a screenshot of the editing of the specification $\mathfrak{S}_2$ at layer M_2. Classes *Person*, *Organization*, and *Activity* are added to the diagram, including numerous reference types between them.

We repeat the pattern of invoking the wizard for creating the Specification Diagram at layer M_1, using the graph from the specification $\mathfrak{S}_2$ at layer M_2 as type graph. Figure 7.3(c) shows a screenshot of the editing of the specification $\mathfrak{S}_1$ at layer M_1. The user can add constraints as required, as the tool was preloaded with a set of predicates corresponding to the signature as described in table 7.1.

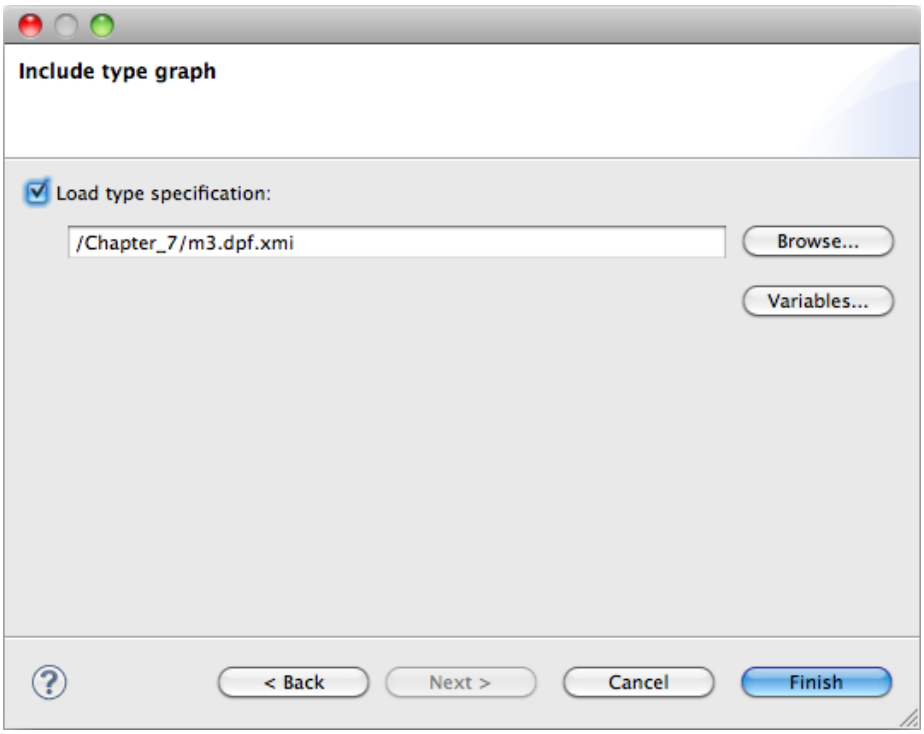
Having finished this last layer of specifications, we can use DPF Editor to make instances of specification $\mathfrak{S}_1$. One such instance is shown in figure 7.4(a). If the user tries to make an instance that breaks the validation of any constraint's semantics, the tool displays a status message to that effect, as shown in figure 7.4(b).

7.4 Further evaluation

As a graduate course in Model-driven engineering (MDE) had been planned for the spring semester 2011 at Bergen University College (HiB), the author had hoped to be able to arrange for students to participate in a field experiment designed for the testing of DPF Editor. Unfortunately, the course was postponed, and the opportunity to get user feedback from participants external to the development project went away. Hopefully, the planned course will be held at a later stage, opening up for extensive user-testing of the tool.

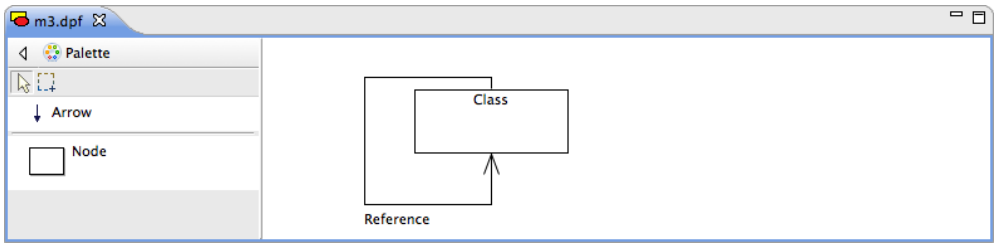


(a)

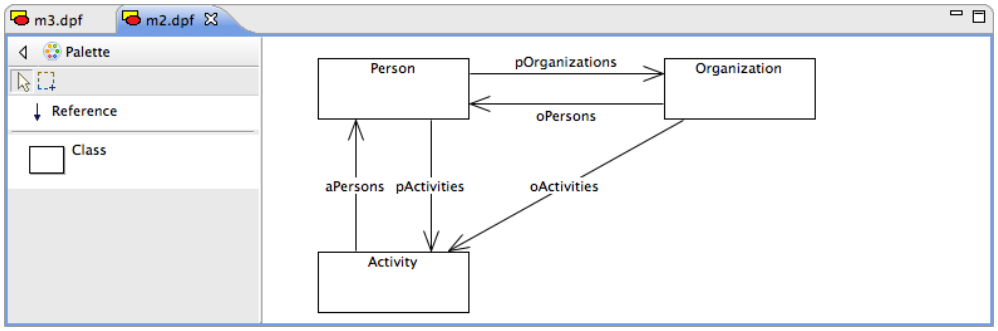


(b)

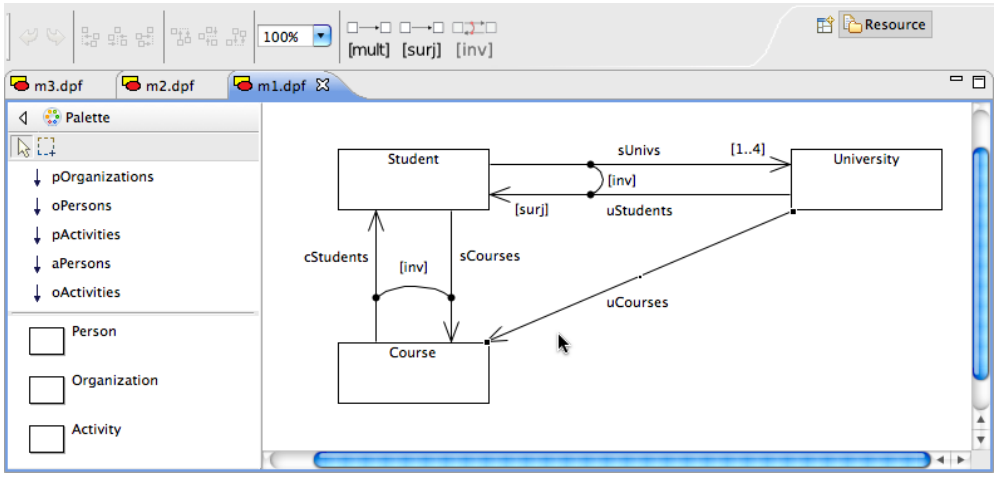
Figure 7.2: The new DPF Specification Diagram Wizard. Figure 7.2(a) shows how the user enters the name of the file that will contain a new specification. (Technically, the file will contain only the Display Model belonging to the specification. See section 6.4.3, page 43 for a description of DPF Editor's persistence strategy.) Figure 7.2(b) shows how the user selects an existing DPF graph file (`.dpf.xmi`) and specifies that it shall be used as a type graph for a new specification.



(a)

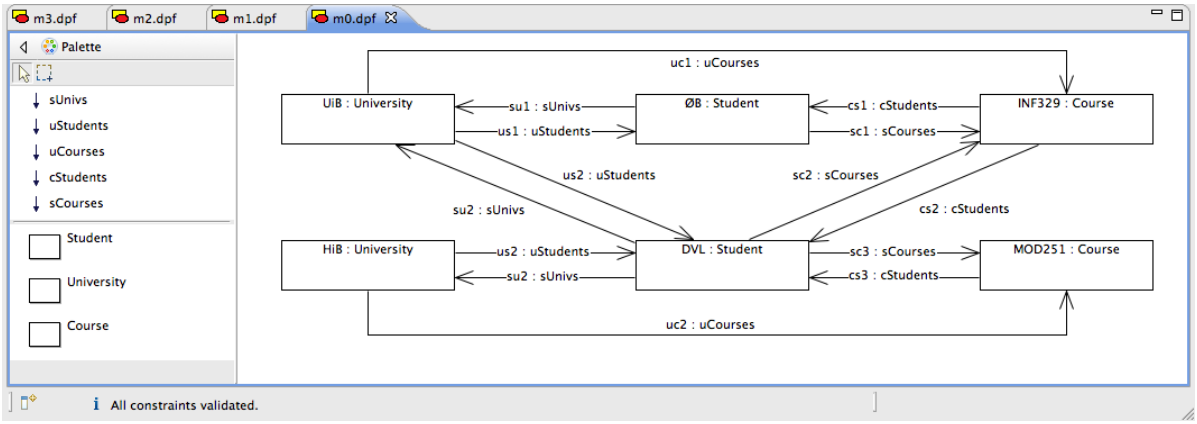


(b)

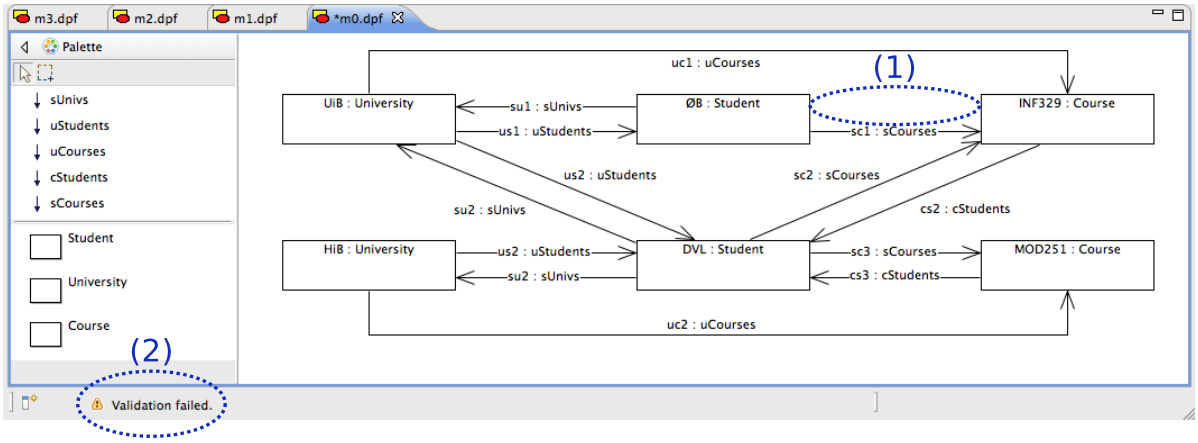


(c)

Figure 7.3: Editing the specification $\mathfrak{S}_3$ at layer M_3 through specification $\mathfrak{S}_1$ at layer M_1. In figure 7.3(a), DPF Editor is configured by the default type graph, enabling the user to create a diagram consisting of the types *Node* and *Arrow*. In this diagram, a single node type, *Class* is created, as well as a single arrow type called *Reference*. In figure 7.3(b), DPF Editor is configured by the type graph belonging to the specification $\mathfrak{S}_3$ at layer M_3, enabling the user to create a diagram consisting of the types *Class* and *Reference*. In this diagram, classes called *Person*, *Organization*, and *Activity* are created, as well as numerous reference types between these classes. No constraints are added to the two first specifications. Figure 7.3(c) shows the editing of the specification $\mathfrak{S}_1$ at layer M_1. In this screen, DPF Editor is configured by the type graph belonging to the specification $\mathfrak{S}_2$ at layer M_2, enabling the user to create a diagram consisting of the types *Person*, *Organization*, and *Activity*. The editor is also configured to let the user apply constraints from the signature Σ_2. In this diagram, entities called *Student*, *University*, and *Course* are created, as well as numerous reference types between these classes. Also, constraints from the signature Σ_2 have been added to the specification. Note how, as the reference *uCourses* have been selected by the user, the toolbar icons representing the constraints `[mult]` and `[surj]` have both become enabled. The remaining constraint, `[inv]`, is not enabled, as the current selection does not allow for the creation of a graph homomorphism from the corresponding predicate into the selected subgraph.



(a)



(b)

Figure 7.4: Figure 7.4(a) shows the editing of an instance I at layer M_0. The diagram being edited in this screenshot corresponds to the instance graph at layer M_0 in figure 7.1. Note that the tool shows the legend "All constraints validated" as the content of the instance conforms to the specification $\mathfrak{S}_1$ defined at layer M_1. Figure 7.4(b) shows an instance at layer M_0 failing the semantic validation. In this screenshot, we have removed a student reference (`cs1`, dotted blue oval marked **(1)**), thereby invalidating two instances of the constraints `[inv]` ("references between students and universities must be inverse") and `[surj]` ("a student must be referenced by at least one university"), from the specification $\mathfrak{S}_1$. In the lower left corner (dotted blue oval, marked **(2)**), the tool informs us that "Validation failed".

Chapter 8

Concluding matters

In this chapter, we conclude on our tool's capabilities, in relation to the DPF as well as previous efforts. We also describe some possible avenues for further work, both directly and indirectly related to our tool.

8.1 Conclusion

In this thesis, we have described how we constructed a diagram editor as a plug-in for Eclipse based on DPF. The finished artefact can be run in Eclipse, provided that Eclipse Modelling Tools [18] have already been installed.

We have improved on previous efforts (section 4.1, page 18) by implementing a true multi-platform solution, runnable in Eclipse on all major operating systems. To our knowledge, there does not yet exist any metamodelling tool available for the Eclipse platform that contains the described functionality. This also sets our editor apart from any related tool presently available.

In our view, the functionality now available does fulfil the requirements we set forth in section 4.3, page 27, and can be summarized as follows:

Diagram editor: The user can edit specification diagrams in a graphical editor by creating, moving, deleting, and connecting nodes and arrows. If a node is removed, any connected arrows are also removed, prohibiting dangling arrows in the diagram. The resulting graphs can be serialized to (and de-serialized from) XMI files. The editor also supports undo and redo operations.

Metamodelling and specification typing: This is probably the main new piece of functionality available in our plug-in. The plug-in now has the ability to create a metamodelling hierarchy of an arbitrary number of meta-levels, thereby letting the user create specifications as new domain specific languages. When applied as typing specifications, these DSLs will establish new editor configurations. This is achieved upon creation of a new specification, by specifying in the editor that the specification being created shall be typed by the graph from a previously edited specification.

Typing has been implemented by creating type homomorphisms between meta-levels in the metamodelling hierarchy.

Applying and validating constraints: The user can select a constraint, either from the toolbar or from a context menu, and apply this constraint to a specification diagram being edited. The tool itself guides this application process, making sure that any new constraint is created with a valid graph homomorphism from the predicate's arity onto the diagram's graph. Typed specifications can be validated against any constraint that has been applied to the type graph, provided these constraints have been associated with a corresponding constraint validator.

Auxiliary functionality: Specification diagrams can be printed to a system's printer using Eclipse's standard printing commands. A preference pane lets the user specify the properties for user settings. For instance, if the user does not wish to display arrow names in the diagram this can be turned off in the preference pane. Also, the user can change the zoom level while editing, as well as superimpose a grid on the editing area for easier placement of nodes in the diagram.

Integrated development environment: We have also set up an integrated environment to facilitate further development work. This environment consists of a wiki, a continuous integration system, and source version control.

8.2 Suggestions for further work

This section contains a selection of proposals for further work. The author does not wish to make any prioritized list, as the focus and direction of development may change on a later stage.

8.2.1 Signature editor

Today's plug-in offer a single pre-defined signature for modelling use. This signature is hard-coded, and future users of DPF Editor will probably wish to define their own signature(s). A two-stage solution for this can be suggested:

1. Create a simple signature editor that lets the user compile signatures by choosing between pre-defined predicates
2. Expand on this solution by enabling the user to edit individual predicates

A simple editor

The editor functionality for this first stage ought to be straightforward to implement as an Eclipse plug-in, perhaps by employing a wizard or similar "helper" functionality. In conjunction with this, a set of pre-defined predicates would need to be implemented by members of the DPF group. Candidate predicates for this kind of collection could be found by evaluating predicates previously

π	$\alpha^{\Sigma}(\pi)$	Proposed vis.	Semantic interpretation
[injective]	$1 \xrightarrow{a} 2$	X $\xrightarrow[\text{[inj]}]{\text{f}}$ Y	$\forall x, x' \in X : f(x) = f(x')$ implies $x = x'$
[surjective]	$1 \xrightarrow{a} 2$	X $\xrightarrow[\text{[surj]}]{\text{f}}$ Y	$\forall x \in X : \bigcup\{f(x)\} = Y$

Table 8.1: Two similar predicates in a signature. Both predicates possess the same arity and only slightly dissimilar visualizations.

defined in the DPF literature. (See [14] for an updated list of papers.) This could also lead to a theming of predicates for easy selection by the end user.

As the system exists today, the difficulty of defining a new predicate in DPF Editor varies according to several parameters. If the predicate to be created possesses the same arity and a similar visualization – perhaps only with a different label – as some previously defined predicate, the only new significant component to develop would be the semantic validation checker for that particular predicate. For an example of such similar predicates, compare the two predicates [injective] and [surjective] listed in table 8.1.

If a predicate requiring a new type of visualization and corresponding semantics validation is needed, the required effort will be of a more substantial character, depending on the complexity of the visualization and semantics in question.

An advanced solution

The second stage – enabling the user to edit individual predicates – can also be broken into two distinct sub-tasks. If the goal is to let the user define a new predicate with new semantics but using pre-defined arities and visualizations, the predicate editor will depend on the system to possess some general *semantics validation* system (see below). If the goal is to go beyond this and implement a "free-form" editor, giving the user the power to edit both predicates' arities and their corresponding visualizations, a *generalized method of creating visualizations* would also be needed.

A note on DPF conformance

So far, we have avoided discussing how the editor conforms with DPF modelling formalisms (definition 13, page 16). This stems from the fact that our Ecore-modelled implementation of a DPF Signature is of the un-typed kind, not typed as given in definition 9, page 15. This is a conscious decision, as we saw no need to include an implementation of typed signatures before a working signature editor was ready. The implication is that although being able to construct a metamodelling hierarchy, DPF Editor does only conform partially to modelling formalisms as they are defined within DPF.

8.2.2 Semantics validation system

The current code includes a prototypal solution for validating a specification based on the constraints applied to the specification's type graph. The solution includes hard-coded validators for each predicate we have chosen to include. A general solution is an obvious extension to the project, and would have to include features for describing the semantics of predicates in some high-level language. Even without a signature editor, this component will probably be of benefit for future developers, as it would lessen the workload needed to implement a validator for a single semantic.

8.2.3 A more flexible graphical notation

As the system exists today, all visualizations (nodes, arrows, and constraints) are hard-coded in the editor as subclasses of Draw2d figures and connections. To obtain a more flexible tool, it would be desirable to decouple the visualizations from the Display Model. As with the concept of a signature editor, this goal may be achieved in more than one stage:

A widget solution: A set of simple graphical elements akin to a GUI widget toolkit [54] can be constructed. An editor could be built, letting the user build visualizations piecemeal from these "parts", included as graphical primitives in the editor. The resulting visualizations could then be made to replace the current, default implementation in parts or whole.

Concrete syntax in a metamodelling hierarchy: Baar [3] proposes to extend the metamodelling approach to also include concrete syntax definition. By introducing so-called *display manager* classes that would bridge a model's abstract syntax and the visual language, the concrete syntax could be completely decoupled from the diagram editor. This would enable a separate metamodelling hierarchy for concrete syntax in addition to the existing hierarchy for abstract syntax. A major undertaking, this solution would dictate radical refactoring and the need for more advanced widget-based editors.

Also worth mentioning in this context is the design of the nodes and arrows themselves. Currently, little effort has gone into the aesthetic aspects of the diagram editor, and a less distracting user experience can probably be achieved by doing work in this area. Related to this is also the matter of having user-movable arrow and constraint anchors, where the automatic layout as it exists today can be "tweaked" interactively by the user.

8.2.4 Layout and routing

Automated layout seems to become an issue when dealing with medium-sized to large diagrams. Due to layout problems, we encountered problems using EMF's modelling tools while modelling a decidedly average-sized model (section 6.9.4,

page 56). There seems to be a big usability gain to be capitalized on in this matter.

Today's editor contains a simple routing algorithm, based on Draw2d's `ShortestPathConnectionRouter` class. The problem of finding routing algorithms that produce easy-readable output is a focus of ongoing research [60], and this problem applied to DPF Editor can probably be turned into a separate research task.

8.2.5 Code generation

Perhaps the real utility for an end user of DPF Editor will only become manifest when some sort of (preferably stand-alone) running system can be generated from specifications. We have already done some introductory work on code generation [4], and further work in this area is already commenced. Several solutions can be envisaged, for example Java code generation for class modelling or SQL code generation for data modelling.

Related to this topic is the concept of model transformation. DPF has been built with support for transformations [62], and this is probably one of the main goals for a DPF tool to achieve. Along with support for (meta)model evolution, support for this feature probably lies further into the future than most of the other features mentioned.

8.2.6 Make editor code testable

At present, very little of the editor code itself is covered by unit tests. The reasons for this are twofold. First, we used an existing example editor as a base for the development of the DPF editor. This code was not equipped with unit tests. Second, most editor components are tightly coupled to parts of the Draw2d and/or GEF frameworks and thus remain hard to test.

To remedy this, a project that focuses on the decoupling of the editor classes from Draw2d and GEF will probably make it easier to create unit tests for the editor. This project can probably also be done separately from the other development efforts, making it a good candidate for a stand-alone effort.

8.2.7 Moving beyond GEF

If DPF Editor is to remain an Eclipse-based plugin, it is hard to imagine diagrammatic editing being implemented in another platform than GEF. However, this does not mean that the code needs to remain directly layered upon GEF. *Graphiti* [17] shows promise, especially from a usability standpoint. Graphiti may very well turn out to be a more suitable graphical platform on which to build a diagrammatic editor for DPF.

Bibliography

[1] Apache Ant. *Project Web Site*. `http://ant.apache.org/`.

[2] AToM^3: A Tool for Multi-formalism and Meta-Modelling. *Project Web Site*. `http://atom3.cs.mcgill.ca/`.

[3] Thomas Baar. Correctly Defined Concrete Syntax for Visual Modeling Languages. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS 2006: 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 111–125. Springer, 2006.

[4] Øyvind Bech and Dag Viggo Lokøen. *DPF to SHIP Validator Proof-of-Concept Transformation Engine*. `http://dpf.hib.no/code/transformation/dpf_to_shipvalidator.py`.

[5] Øyvind Bech, Adrian Rutle, Yngve Lamo, and Alessandro Rossini. DPF Editor: A Multi-Layer Diagrammatic (Meta)Modelling Environment. Submitted to MODELS 2011: International Conference on Model Driven Engineering Languages and Systems.

[6] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[7] Bill Moore and David Dean and Anna Gerber and Gunnar Wagenknecht and Philippe Vanderheyden. *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM Corp., Riverton, NJ, USA, 2004.

[8] Frank Budinsky, Ed Merks, and David Steinberg. *EMF: Eclipse Modeling Framework 2.0 (2nd Edition)*. Addison-Wesley Professional, 2006.

[9] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison-Wesley Professional, 3 edition, 2008.

[10] Douglas. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32:9–23, January 1989.

[11] Juan de Lara and Esther Guerra. Deep Meta-modelling with MetaDepth. In Jan Vitek, editor, *TOOLS 2010: 48th International Conference on Objects, Components, Models and Patterns*, volume 6141 of *LNCS*, pages 1–20. Springer, 2010.

[12] Peter J. Denning. Is computer science science? *Commun. ACM*, 48:27–31, April 2005.

[13] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden*, April 2002.

[14] DPF: Diagram Predicate Framework. *Project Web Site*. `http://dpf.hib.no/`.

[15] Draw2d. *Project Web Site*. `http://www.eclipse.org/gef/draw2d/`.

[16] Eclipse contributors and others. *Eclipse Platform API Specification* . `http://help.eclipse.org/helios/topic/org.eclipse.platform.doc.isv/reference/api/overview-summary.html`.

[17] Eclipse Graphiti. *Project Web Site*. `http://www.eclipse.org/graphiti/`.

[18] Eclipse Modeling Framework. *Project Web Site*. `http://www.eclipse.org/emf/`.

[19] Eclipse Naming Conventions. *Project Web Site*. `http://wiki.eclipse.org/Naming_Conventions`.

[20] Eclipse PDE/Build. *Project Web Site*. `http://www.eclipse.org/pde/pde-build/`.

[21] Eclipse Platform. *Project Web Site*. `http://www.eclipse.org`.

[22] Eclipse Project Naming Policy. *Project Web Site*. `http://wiki.eclipse.org/Development_Resources/HOWTO/Project_Naming_Policy`.

[23] Eclipse User Interface Guidelines. *Project Web Site*. `http://wiki.eclipse.org/User_Interface_Guidelines`.

[24] Eclipse Wiki. *FAQ Where did Eclipse come from?* `http://wiki.eclipse.org/FAQ_Where_did_Eclipse_come_from%3F`.

[25] Equinox OSGi. *Project Web Site*. `http://www.eclipse.org/equinox/`.

[26] Joseph Feller, Brian Fitzgerald, Scott A. Hissam, and Karim R. Lakhani. *Perspectives on Free and Open Source Software*. The MIT Press, 2007.

[27] José L. Fiadeiro. *Categories for Software Engineering*. Springer, May 2004.

[28] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer graphics: principles and practice (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[29] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[30] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, November 1994.

[31] Robert L. Glass. A structure-based critique of contemporary computing research. *Journal of Systems and Software*, 28(1):3–7, 1995.

[32] GME: Generic Modeling Environment. *Project Web Site*. `http://www.isis.vanderbilt.edu/Projects/gme/`.

[33] Cesar Gonzalez-Perez and Brian Henderson-Sellers. Modelling software development methodologies: A conceptual foundation. *J. Syst. Softw.*, 80:1778–1796, November 2007.

[34] Cesar Gonzalez-Perez and Brian Henderson-Sellers. *Metamodelling for Software Engineering*. Wiley Publishing, 2008.

[35] Graphical Editing Framework. *Draw2d Programmer's Guide*. `http://help.eclipse.org/galileo/topic/org.eclipse.draw2d.doc.isv/guide/guide.html`.

[36] Graphical Editing Framework. *GEF Programmer's Guide*. `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.gef.doc.isv/guide/guide.html`.

[37] Graphical Editing Framework. *Project Web Site*. `http://www.eclipse.org/gef/`.

[38] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[39] Ørjan Hatland. Sketcher .NET – A drawing tool for generalized sketches. Master's thesis, Department of Informatics, University of Bergen, Norway, June 2006.

[40] Pavol Hell and Jaroslav Nešetřil. On the complexity of h-coloring. *J. Comb. Theory Ser. B*, 48:92–110, February 1990.

[41] Hudson. *Project Web Site*. `http://hudson-ci.org/`.

[42] Java Naming Conventions. *Project Web Site*. `http://www.oracle.com/technetwork/java/codeconventions-135099.html`.

[43] Juan De Lara, Hans Vangheluwe. Using AToM^3 as a Meta-CASE Tool. In *Proc. 4th International Conference on Enterprise Information Systems*, pages 642–649, Ciudad Real - Spain, April 2002.

[44] JUnit testing framework. *Project Web Site*. `http://www.junit.org/`.

[45] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[46] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The epsilon object language (eol. In *In: Proceedings European Conference in Model Driven Architecture (EC-MDA) 2006*, pages 128–142. Springer, 2006.

[47] G. Krasner and S. Pope. A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[48] Thomas Kühne. What is a model? In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2004.

[49] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. *The Generic Modeling Environment*, volume 17, pages 82–83. ACM Press, 2001.

[50] László Lengyel, Tihamér Levendovszky, and Hassan Charaf. Constraint Validation Support in Visual Model Transformation Systems. *Acta Cybernetica*, 17(2):339–357, 2005.

[51] Steve Loughran and Erik Hatcher. *Ant in action: java development with ant, second edition*. Manning Publications Co., Greenwich, CT, USA, 2007.

[52] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[53] Mono. *Project Web Site*. `http://mono-project.com/`.

[54] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, March 2000.

[55] Steve Northover and Mike Wilson. *Swt: the standard widget toolkit, volume 1*. Addison-Wesley Professional, first edition, 2004.

[56] Object Management Group. *Web site*. `http://www.omg.org`.

[57] Object Management Group. *Unified Modeling Language Specification*, May 2010. `http://www.omg.org/spec/UML/2.3/`.

[58] Object Management Group, XML Metadata Interchange (XMI®). *XML Metadata Interchange Specification*. `http://www.omg.org/spec/XMI/2.1.1/`.

[59] Oracle. *Abstract Window Toolkit (AWT) (Java SE Documentation)*, April 2010. `http://download.oracle.com/javase/6/docs/technotes/guides/awt/`.

[60] Tobias Reinhard, Christian Seybold, Silvio Meier, Martin Glinz, and Nancy Merlo-Schett. Human-friendly line routing for hierarchical diagrams. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 273–276, Washington, DC, USA, 2006. IEEE Computer Society.

[61] Adrian Rutle. Towards a Formal Diagrammatic Framework for MDA. In *ECOOP-DS 2008: 18^{th} ECOOP Doctoral Symposium and PhD Students Workshop*, pages 37–40, 2008.

[62] Adrian Rutle. *Diagram Predicate Framework: A Formal Approach to MDE*. PhD dissertation, Department of Informatics, University of Bergen, Norway, 2010.

[63] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Diagrammatic Formalisation of MOF-Based Modelling Languages. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS Europe 2009: 47^{th} International Conference on Objects, Components, Models and Patterns*, volume 33 of *LNBIP*, pages 37–56. Springer, 2009.

[64] Adrian Rutle, Alessandro Rossini, Yngve Lamo, and Uwe Wolter. A Formalisation of Constraint-Aware Model Transformations. volume 6013, pages 13–28. Springer, 2010.

[65] Adrian Rutle, Uwe Wolter, and Yngve Lamo. Diagrammatic Software Specifications. In *NWPT 2006: 18^{th} Nordic Workshop on Programming Theory*, October 2006.

[66] Adrian Rutle, Uwe Wolter, and Yngve Lamo. Generalized Sketches and Model-Driven Architecture. Technical Report 367, Department of Informatics, University of Bergen, Norway, February 2008.

[67] Ed Seidewitz. What models mean. *IEEE Softw.*, 20:26–32, September 2003.

[68] James Shore and Shane Warden. *The art of agile development*. O'Reilly, first edition, 2007.

[69] Vladimir Silva. 2D Graphics with GEF and Zest. In *Practical Eclipse Rich Client Platform Projects*, pages 173–208. Apress, 2009.

[70] Stian Skjerveggen. (Towards an) Implementation of a Graphical Editor for Diagrammatic Predicate Logic in the Eclipse Platform. Master's thesis, Department of Informatics, University of Bergen, Norway, June 2008.

[71] Ida Solheim and Ketil Stølen. Technology research explained. Technical Report A313, SINTEF ICT, Oslo, Norway, March 2007.

[72] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Metamodelling. In *Model-Based Engineering of Embedded Real-Time Systems*, volume 6100 of *Lecture Notes in Computer Science*, chapter 4, pages 59–78. Springer, 2010.

[73] Matti Tedre. Know your discipline: Teaching the philosophy of computer science. *JITE*, 6:105–122, 2007.

[74] The Agile Alliance. *Project Web Site*. `http://www.agilealliance.org/`.

[75] The Eclipse Graphical Modeling Project. *Project Web Site*. `http://www.eclipse.org/modeling/gmp/`.

[76] The Scrum Alliance. *Project Web Site*. `http://www.scrumalliance.org/`.

[77] Juha-Pekka Tolvanen and Steven Kelly. Metaedit+: defining and using integrated domain-specific modeling languages. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 819–820, New York, NY, USA, 2009. ACM.

[78] VMTS. *Project Web Site*. `http://www.aut.bme.hu/Portal/Vmts.aspx`.

[79] Yannick Saillet. *Migrate your Swing application to SWT*. IBM developerWorks, 2004. `http://www.ibm.com/developerworks/java/tutorials/j-swing2swt/`.

[80] Zest. *Project Web Site*. `http://www.eclipse.org/gef/zest/`.

[81] John Zukowski. *The Definitive Guide to Java Swing, Third Edition (Definitive Guide)*. Apress, Berkely, CA, USA, 2005.