Visual OCL and Incremental Constraint Validation Framework for the DPF Workbench

Sidra Nadeem

Master's Thesis in Informatics – Program Development

Department of Informatics
University of Bergen

HØGSKOLEN I BERGEN
Department of Computer Engineering
Bergen University College

March 2013

Contents

Preface **viii**

1 Introduction **1**

1.1 Motivation 1

1.2 Organization of Thesis 3

2 Background **4**

2.1 Model Driven Engineering MDE 4

2.1.1 Domain Specific Modelling Languages 5

2.1.1.1 Feature of modelling language 8

2.1.2 Metamodelling 9

2.1.2.1 Metamodel Architecture 11

2.1.2.2 Metamodelling Process 12

2.1.3 Model Driven Architecture MDA 13

2.2 Unified Modeling Language (UML) 15

2.2.1 Classification of Diagram in UML 15

2.2.2 Example of UML 16

2.3 Constraints in MDE 17

2.4 Object Constraint Language (OCL) 19

2.4.1 OCL constraint 20

2.4.2 Basic structure of OCL Constraint 21

2.4.3 OCL Types 22

2.4.4 OCL Expressions and Queries 24
2.4.5 Kinds of OCL Constraints 25
2.4.6 Other concepts 28
2.4.7 Validation of Constraints 29
2.5 The Diagram Predicate Framework (DPF) 30
2.6 The Diagram Predicate Framework Workbench 33
2.6.1 The DPF Model Editor 35
2.6.2 Metamodel based Code Generation 35
2.6.3 The DPF Signature Editor 37

3 Constraints and their Validation in MDE 40

3.1 What are Constraints in MDE 40
3.2 Constraints and OCL in UML 41
3.3 Visualization of OCL 42
3.3.1 VisualOCL 42
3.3.2 Constraint Diagrams 45
3.4 DPF Semantic Validator 46
3.4.1 Problems with the DPF validation mechanism 48
3.4.1.1 Problems with the predefined DPF Signature . . . 49
3.4.1.2 Problems with DPF Workbench Signature Editor . 50
3.5 Summary 51

4 Design, Development and Solution 52

4.1 Development Process 52
4.1.1 Development Methodology 52
4.1.2 Coding Conventions 54
4.1.3 Tools 54
4.2 Problem Description 55
4.3 Eclipse OCL 56

4.3.1 Eclipse OCL CORE Component 56
4.3.1.1 APIs for Parsing Constraints and Queries 57
4.3.2 OCL Examples Component 59
4.3.3 OCL Examples and Editors Component 59
4.4 DPF OCL Checker 60
4.4.1 The OCL Checker 60
4.4.1.1 Tool Selection 60
4.4.1.2 Plugin Dependencies 60
4.4.1.3 Architecture 61
4.4.1.4 Implementation 61
4.4.2 Visualization of OCL 62
4.4.3 Relation to the Default DPF metamodel 62
4.4.4 Incremental Validation of constraints 67
4.4.5 Validation Feedback 72
4.5 Summary 75

5 Demonstration **77**
5.1 Demonstration Setup 77
5.1.1 Creating Signature 78
5.1.2 Defining Metamodel 81
5.1.3 Generating Model Editor from Metamodel 82
5.1.4 Generating Instance Editor from Model 84
5.1.4.1 Validating Constraints 84
5.1.4.2 Validation Feedback Reporting 85
5.2 summary 86

6 Evaluation and Conclusion **87**
6.1 Evaluation 87
6.1.1 Previous Approach 87

6.1.2 Our Approach 88
6.1.3 Evaluation of Our Approach 88
6.1.4 Related Work 90
6.1.4.1 Comparison with other visualizations of OCL approaches 90
6.1.4.2 Comparison with the other modelling Frameworks 91
6.2 Conclusion 92
6.2.1 Summary 92
6.3 Suggestion For Further Work 94
6.3.1 Validation 94
6.3.2 The DPF Workbench 95
6.3.2.1 Visualization of Models 95
6.3.2.2 Layout and routing 96

Appendix A Tutorial for Defining a New Signature in DPF Signature Editor 97

A.1 Installation 97
A.2 Define a new Signature File 98
A.2.1 Create a new dpf project 98
A.2.2 Create New Signature Wizard 98
A.2.2.1 Name of Predicate 101
A.2.2.2 Predicate Properties 101
A.2.2.3 Predicate Icon 101
A.2.2.4 Predicate Visualization 101
A.2.2.5 Predicate Semantic Validator 102
A.2.2.6 Predicate Syntax 102
A.3 Load newly defined Signature 102
A.4 Add Constraint on Model 105
A.5 Creating Instance of Model 105

Bibliography 107

List of Figures

2.1 A model may be a prescription or description of the system . . . 6

2.2 A Simple metamodel for UML class Model 7

2.3 A Simple metamodel for UML class diagram 10

2.4 The four metalevels of OMG with (meta)modelling languages and their corresponding metamodels 11

2.5 The four metalevels of OMG 13

2.6 Transforming a PIM into PSM using transformation tools 14

2.7 UML class diagram for flights 16

2.8 structural constraints in UML 17

2.9 Constraints in metamodelling 18

2.10 Structure of OCL constraint 21

2.11 Overview of types in OCL 23

2.12 arity of exclusive-or predicate 31

2.13 Graph homomorphism of exlusive-or predicate 31

2.14 The main component architecture of the DPF Workbench 34

2.15 Creating a model in the DPF Editor 36

2.16 Typing relation between different layers in DPF 37

2.17 Definition of new predicate using Signature Editor 38

3.1 Principle representation of a constraint 43

3.2 Example of Visual OCL Constraint 44

3.3 Example of Visual OCL Constraint for $Post$ condition 44

3.4 Each person has a different first name from last name 46

3.5 Model of different Classes (Student, University and Project) and association between them 47

3.6 The DPf editor All constraint validated 48

3.7 The DPF editor Validation Failure Error Message 49

3.8 Definition of the arity and the graphical icon of the injective predicate 50

3.9 Definition of the arity and the graphical icon of the injective predicate 51

4.1 Eclipse OCL Architecture 56

4.2 The OCL Class and the OCLHelper Class APIs 57

4.3 Interactive Xtext Console 59

4.4 An overview of the architecture of the DPF OCL Checker 61

4.5 Interactive Xtext Console 63

4.6 The DPF Workbench Signature Editor 64

4.7 Representation of OCL Constraint in the DPF Model Editor . . . 64

4.8 The graph of the default metamodel, consisting of a single node and a single arrow self-referencing the node. 65

4.9 Shape of the Predicate 65

4.10 A DPF model with two Constraints 68

4.11 Example on how our incremental algorithm works 70

4.12 Example on how our incremental algorithm works 73

4.13 The New Extension wizard showing the markers extension point selected. 74

4.14 Validation Errors in the Problem View 75

4.15 Validation Errors in the DPF Model Editor 76

4.16 The DPF Preferences Page 76

5.1 Example UML CLass Diagram 78

5.2 Definition of the arity and the graphical icon and OCL Semantics of the exclusive-or predicate 79

5.3 Definition of the arity and the graphical icon and OCL Semantics of the exclusive-or predicate 81

5.4 The graph of the default metamodel, consisting of a single Node and a single Arrow self-referencing the node. 81

5.5 Metamodel at M2 level 82

5.6 Model at M1 level 82

5.7 Instance at M0 level 84

5.8 a Valid instance at M0 level of a model at M1 level 85

5.9 Validation Feedback in Problem View 85

A.1 Create New DPF Project 98

A.2 Create New DPF Project 99

A.3 Create New Signature Wizard 99

A.4 Create New Signature Wizard 100

A.5 Create New Signature Wizard 100

A.6 Create New Signature Wizard 100

A.7 mult Predicate details 101

A.8 Define Predicate Shape 103

A.9 Example of predicate with ternary no of model elements 103

A.10 Create New Specification 104

A.11 Enter Specification Name 104

A.12 Load New Signature 105

A.13 Specification Editor 105

A.14 Validation Error 106

A.15 Validation Error in Problem View 106

List of Tables

2.1 Semantics of Predicates 32

5.1 The DPF predicates defined as OCL syntax 80

6.1 Comparison of DPF Workbench's Visualization of OCL with two other OCL Visualizations, VisualOCL and Constraint Diagram. . 91

6.2 Comparison of DPF Workbench with Eclipse Modelling Framework (EMF) w.r.t the OCL. 92

Preface

Foreword

This thesis is submitted to the Department of Informatics at the University of Bergen and Bergen University College as part of the Master degree Program, Joint Degree in Software Engineering.

The work done in this thesis was an extension to the Diagram Predicate Framework; an ongoing research project at the Bergen University College. Throughout the work in this thesis, I had the chance to learn many new concepts and different technologies. With the introduction of Model Driven Engineering, it makes it easier to understand the practical importance of modelling in the software industry these days. I also had the opportunity to work with lot of skilled people who really knows this field very well.

This project gave a chance to extend the features provided by the Diagram Predicate Framework Workbench; a reference implementation of the Diagram Predicate Framework concepts.

Acknowledgements

The work reported in this thesis has been carried out by me. I would like to thank my supervisor Yngve Lamo, for his invaluable guidance and support through the work with this master thesis. I would also like to thank Xiaoliang Wang, for providing feedback on technical aspects. I would also like to thank Florian Mantz in figuring out the direction of my project. I would like to thank the rest of the DPF project members especially Suneetha Sekhar. It would not be possible to finish my master thesis without the help of my brother Basit Rizwan, my husband Nadeem, and my daughter Javairia. Thank you. Lastly I want to thank my mother Shahnaz Akhter.

Bergen, 23 March 2013

Chapter 1

Introduction

1.1 Motivation

Over the past few decades, due to the advances in technologies and programming languages the developers are now capable of creating complex software programs. These computer programs are platform dependent and based on different technologies, each with its own domain concepts. As a result of this, developers often spend most of the time dealing with platform and technology-related issues [12], and fewer amounts of time are left to focus on domain concepts and design issues.

To overcome this complexity, different modelling languages have been introduced over the past few decades. The main purpose of these modelling languages is to provide abstraction mechanisms that help the developers to concentrate more on the design and the domain concepts rather than focusing on the underlying platform and technologies-related issues.

The first step in this area was taken in 1980s with the introduction of Computer Aided Software Engineering (CASE) tools that help developers to design artefacts of the software system as a graphical representation such as Class diagram, Entity relation diagram and data-flow diagram. The main goal of CASE was to automate the software development process by providing a set of tools for each phase of the software development life cycle. However, the models remain only for the documentation.

Due to the changes in technologies, software system needs to be upgraded and maintained rapidly. Since documentation of the system has been done as a separate task from implementation, a large amount of effort is needed to maintain the documentation and is also a time-consuming job. Because of this, there is a big chance that the documentation and implementation become out of sync, and documentation does not represent the system completely.

Model driven engineering (MDE) aims to overcome these issues [12] by using models as basic development artefacts, i.e. models are not only for the documentation but the actual implemented code is generated from the models. The basic idea of the MDE is to use models that describe some of the aspects of the system formally and then transform these models to other models or to implementation code by using transformation mechanisms. This facility automates the development process, to some extent, and allows developers to concentrate more on the problem domain rather than on the underlying technologies. A Model is based on a metamodel, and it should conform or validate to its metamodel.

Model validation is used for checking models with respect to their semantic and syntax [10]. Model validation is the process of checking models with their metamodel to verify that all the constraints defined in the metamodel are satisfied by the model. Object Constraint Language (OCL) is a formal language that is used to define constraints on metamodels and models. It was developed by International Business Machines (IBM) in 1995 as a business modelling language. Later, OCL has been adopted as a formal specification language within the Unified Modelling Language (UML) to define application-specific constraints on UML models. It became a part of the official Object Management Group (OMG) standard for UML from version 1.1 on. OCL is an expression and side effect-free language. It is not a programming language. OCL is a textual language that is used to specify constraints that cannot be expressed by Diagrammatic UML Constraints. When an OCL constraint is evaluated it will always return a value.

This master thesis will extend the features provided by the DPF Workbench with an OCL validator. The DPF Workbench which is a reference implementation of the Diagram Predicate Framework (DPF), is a language workbench for diagrammatic metamodelling and code generation. The DPF Model Editor at this point has only a hard coded java constraint validator. The DPF Workbench's Signature Editor can be used to define domain-specific constraints. However, it can only be used to define the semantics of predicates by java. To make the DPF validator more dynamic and efficient it will be extended with an OCL validator.

Through this thesis there will be introduced an OCL checker in the DPF Workbench based on `EclipseOCL` [15] plug-in. It will facilitate users to define new predicates by writing an OCL expression and also make visual syntax for the OCL constraints. Moreover, we will change the validation mechanism of the DPF Workbench towards validation of constraints and also the reporting of validation feedback to the user. In the end, this DPF Workbench OCL Validation extension will be demonstrated by the help of an example.

1.2 Organization of Thesis

This master thesis is organised in the following chapters:

Chapter 1 – Introduction
This chapter gives a brief introduction for the motivation, and also describes the organization of this master thesis.

Chapter 2 –Background
This chapter will give a brief introduction to Model Driven Engineering (MDE), Domain-Specific meta-modelling, constraints in MDE, the Object Constraint Language (OCL), the Diagram Predicate Framework (DPF), and the DPF Workbench.

Chapter 3 –Constraints and their Validation in MDE
This chapter will discuss how constraints are used in MDE, and why constraints are required for the modelling of software systems. This chapter also refers to some of the existing visualizations of OCL, including `VisalOCL` and `Constraint Diagrams`. At the end, the DPF Workbench's semantics validator will be introduced and we will also specify the problems with the existing DPF Workbench validation mechanism.

Chapter 4 –Problem Description
This chapter will provide the problem description which this thesis tries to solve, and it will also present the proposed solution to the problem.

Chapter 5 –Demonstration
This chapter will give a demonstration of the OCL validator extension in the DPF Workbench with the help of an example.

Chapter 6 – Conclusion and Summary
This chapter will give an evaluation of our tool, and a short conclusion to the work that has been done in this thesis. It also gives some suggestions for future work in the DPF Workbench.

Chapter 2

Background

In this Chapter, the theoretical background for this thesis is presented. We start with the introduction of Model Driven Engineering (MDE). We discuss some of the MDE's basic concepts, including domain specific modelling languages, metamodelling, Model Driven Architecture (MDA) and constraints in MDE. Then we proceed with the Object Constraint Language (OCL). At the end we present the Diagram Predicate Framework (DPF) and discuss the current DPF Workbench; a reference implementation of the DPF concepts.

2.1 Model Driven Engineering MDE

Over the past few decades, programming languages have evolved a bit from first generation to the second generation in terms of raising the level of abstraction, which allows developers to focus on design intents rather than the underlying technologies. With the recent advent of more expressive object oriented languages like C++, java and C# have raised the level of abstraction even further due to their encapsulation mechanism. Furthermore, the use of reusable class libraries allows developers to reuse program code instead of writing their own. The advancement in these areas allows the developer to focus more on the domain concepts and develop more complex and advanced applications instead of reinvent each thing from scratch.

The main drawback of this approach is the increase in the platform complexity because it is hard for anyone to have a complete overview of the whole system as the number of lines of code grows. Furthermore, today's middleware software's like J2EE and .NET, have thousands of classes and methods that make it even harder. Since changes in platform occur rapidly, so a considerable amount of effort is required to manually port code from one platform to another. Due to the increase in complexity, the developers have to spend more time to solve issues related to implementation rather

than concentrating on the domain requirements. This also makes it difficult to address the part of the software that will be affected by changes in the requirement or platform or language.

A problem which arises due to the rapid changes in the requirements during the last stages of the software-development process is that it would become difficult to maintain documentation. Since maintaining documentation requires a lot of effort and time. Models become only for documentation purposes and have no direct link with the implementation. So there is a chance that they might be out of sync, and developers might not put their effort to keep them in sync.

To overcome these issues a new branch of software engineering, Model Driven Engineering (MDE) has been introduced. MDE aims to address these problems and reduce complexity of software systems. It moves the development process from being code centric to model centric. In MDE, a model is considered as the basic artefact during the software-development process. This raises the level of abstraction by allowing the developers to focus more on the problem domain.

Models play a major role in MDE. Models may have different meanings. The most general definition of a model in [8] says, "a representation of something, either as a physical object which is usually smaller than the real object, or as a simple description of the object which might be used in calculations." In software engineering "A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made" [32].

According to [7] a model should posses these two abstraction characteristics:

- **reduction feature:** A model only represents some of the important properties of the original system.
- **mapping feature:** A model is based on an original system and can be use as a prototype for the original system.

Models can have different purposes; they can be used for describing the original system or they can be used for prescribing i.e. how a system will be implemented or for determining the aspects of the original system as shown in Figure 2.1.

2.1.1 Domain Specific Modelling Languages

Domain-specific modelling languages are used to formalize application behaviour, system requirements and application structure in a particular domain [12]. A domain-specific language is a language that only focuses on

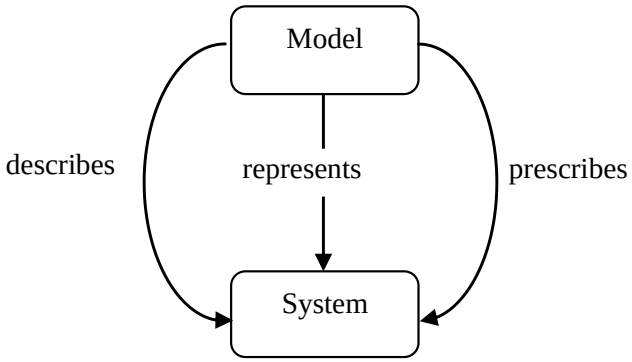


Figure 2.1: A model may be a prescription or description of the system

a particular domain, and it has limited expressive power targeting a specific domain [18]. The two main purposes of domain-specific modelling languages are, first it raises the level of abstraction by defining a solution directly by using domain concepts and rules instead of using programming. Second, from these higher-level specifications it generates a final product in a selected programming language or in some other form. A domain-specific modelling language is specified by persons who identify the modelling concepts in a domain and then formalize them by creating a metamodel [30]. The Domain-Specific Language(DSL) task will be easier if the language need only works for one problem domain, by doing this way one can easily focus on a restricted domain which is the area of interest.

Unified Modelling Language (UML) is one of the examples of general purpose modelling language [42]. "The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artefacts of software systems, as well as for business modeling and other non-software systems" [26]. UML is a general-purpose modelling language for software development, and it provides a general language to specify applications. A general example could be a metamodel describing the relationship between different concepts in a University domain that provides a language to define university domain concepts in a higher level of abstraction.

Figure 2.2 shows a simple metamodel of UML `Class diagram`. The purpose of the class diagram is to represent classes in a model. In object-oriented programming concepts, Classes have Operations (member function), Properties (attributes or member variables) and have associations with other classes. Thus, a UML class model provides a formal language in which these concepts can be described.

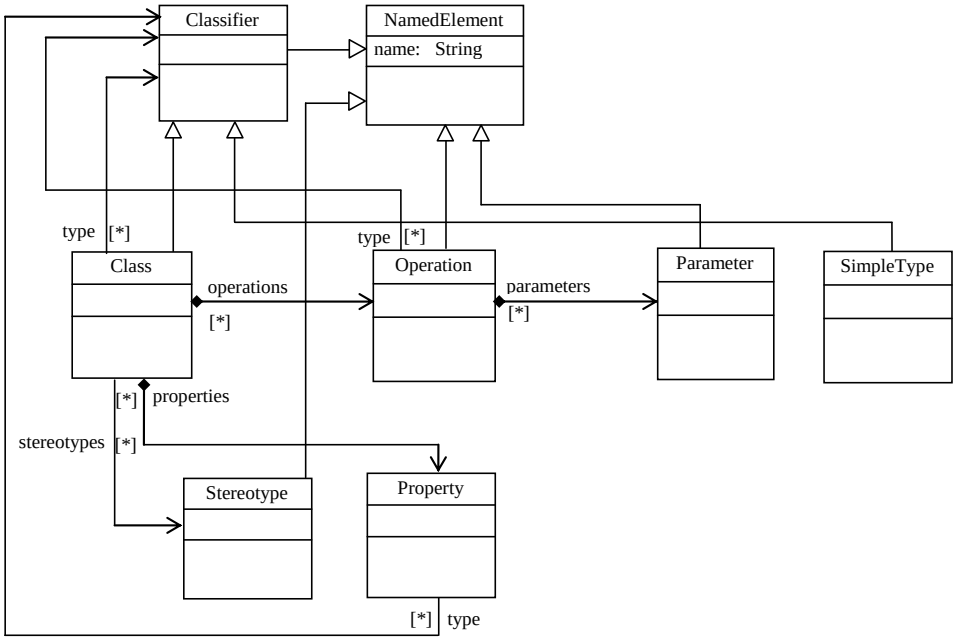


Figure 2.2: A Simple metamodel for UML class Model

2.1.1.1 Feature of modelling language

There are some of the features of a modelling language that a metamodel should be capable to describe [50]. These features are:

Concrete Syntax

It is the notation provided by the language that promotes the construction and presentation of models [50]. It defines the physical appearance of a language. It mainly has two forms that are typically used by languages: textual syntax and visual syntax. A textual syntax describes a model in a structured textual form. It can be of many forms but, mostly it consists of a collection of declarations and expressions. For example the following java code uses a textual syntax. It has a `Class` with a local attribute declaration and a method with return statement.

```
public abstract class Person
  {
     private String nameOfPerson;
     public String getName()
     {
       return nameOfPerson;
     }
  }
```

Listing 2.1: textual concrete syntax

A visual concrete syntax describes the graphical appearance of a language. It consists of a number of graphical objects, e.g. a class diagram shown in figure 2.3(b) is an example of a visual concrete syntax.

Abstract syntax

It defines the graphical appearance of the domain concepts and how they may be combined to create models. It consists of the domain concepts, relationship between them and how these concepts can be combined legally [50]. It only deals with the structure of concepts that can be used to define a language. For example, figure 2.3(a) shows an abstract syntax of a UML Class Diagram. It consists of `Class`, `association` and `property`.

Semantics

The semantics of a model describes what will be the effect when a model is executed. The abstract syntax of a language gives little information about the actual meaning of concepts in a language. For instance a language that uses type concepts of classes can permit the

creation of class objects according to the semantics of the language. It is important for the semantic of a language to be precise in order to have a clear understanding of what the language represents and what it means. A formal mathematical language is often used to define the semantics of a language.

Mappings

Languages will have relationships to other languages. This relationship may be expressed by translation (i.e. concepts in one language are translated into concepts in another language), semantics equivalence (i.e. the semantics of one language is equivalent to the semantics of another language) or through abstraction (i.e. two languages may be in different abstraction levels with each other). Capturing these relationships between languages is an important part of a language definition as it serves to place a language in the context of the surrounding world. This is called mapping. Mapping also exists between the components of one language, e.g. between the concrete and the abstract syntax of one language.

Extensibility

In order to support adaptability, languages have the ability to be extendable because it is not static, it evolves over the period of time. For example, new concepts may add to the language definition, and unused concepts will be deleted from the language. The extensibility allows the language to adapt to new domains and to progress to meet new requirements.

2.1.2 Metamodelling

Metamodels are another important concept in MDE. A metamodel is a model of a modelling language [50]. According to OMG's definition a metamodel "is a model of Models" [24]. A metamodel describes about what can be used to express a valid model in a certain modelling language [22]. The difference between a metamodel, and a model is that a metamodel is a model, but it must capture all the important concepts and features of a language that is being modelled.

A UML class diagram is a collection of classes and associations between them. Figure 2.3 shows a simple metamodel of UML class diagram and also shows the association between a class diagram and its metamodel as dashed lines.

The metamodel has three Classes `Class, Association, Property` and two bidirectional `Association`'s between them. The Class `Property` has two attributes, `LowerBound` and `UpperBound`. The Classes `Lecturer` and `Course` are instances of a `Class`. The multiplicity constraint `1..*` in the class dia-

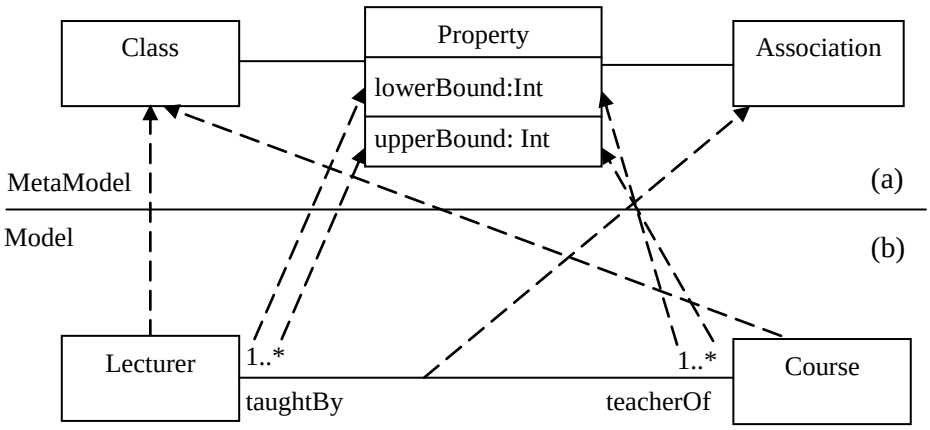


Figure 2.3: A Simple metamodel for UML class diagram

gram is specified by `LowerBound` and `UpperBound` of the `Property` Class in the metamodel. A metamodel has two main distinguishing characteristics:

- A metamodel specifies the abstract syntax, concrete syntax and semantics of the modelling language [50]. The abstract syntax specifies the set of modeling concepts, their attributes and relationships between them, as well as the rules for combining these concepts to create a model [26]. So a model should conform to the metamodel which specifies the modelling language.
- A metamodel should be part of a `modelling architecture` [50]. A `modelling architecture` enables a metamodel to be seen as a model, which itself is defined by another metamodel. Thus, all metamodels are described by another single metamodel. This single metamodel is often called meta-metamodel.

The system development is basically based on the use of languages to capture different aspects of the problem domain. The important benefit of metamodelling is that it has the ability to describe these languages in a unified manner. Metamodelling tackles the problem of language diversity by managing them uniformly, for instance; it can be possible to construct mapping between any number of modelling languages if they are described in the same metamodelling language. Another benefit of metamodelling is that different levels of abstraction can be defined and can be mapped to create a new language that will only focus on a particular domain [50].

A modelling language is used to define a metamodel, which is defined by a meta-metamodel. A metamodelling language is specifically designed to support design of languages so it should have the ability to capture all

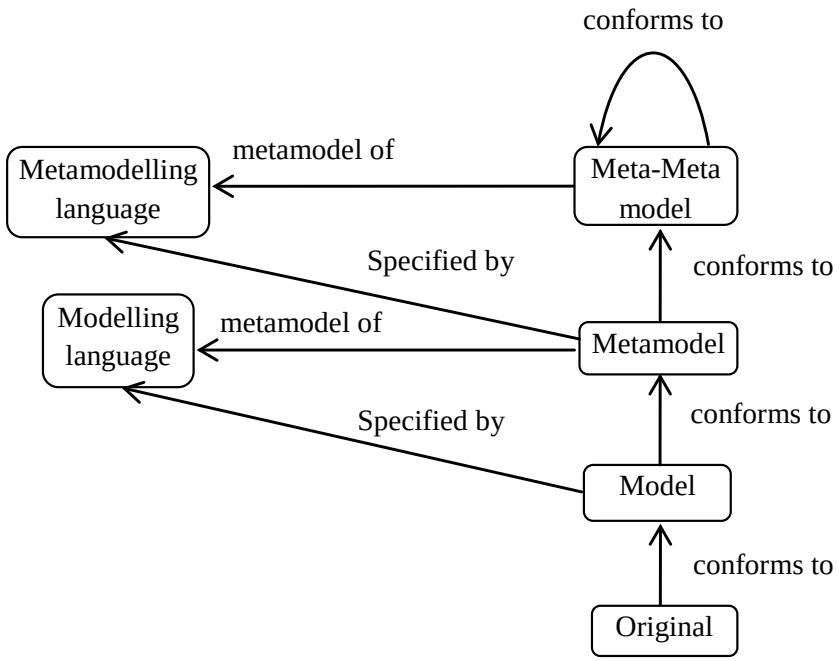


Figure 2.4: The four metalevels of OMG with (meta)modelling languages and their corresponding metamodels

the essential concepts of a modelling language, including its syntax and semantics.

A metamodelling language is a modelling language that is used to specify a metamodel, which itself serve as a metamodel for other modelling languages as shown in figure 2.4. So a metamodel, a model or an instance is specified by a modelling language at one abstraction level above, and they also conform to the metamodel of the modelling language. This metamodelling hierarchy stops when a modelling language is general enough to specify itself, like MOF in OMG's four layered metamodelling hierarchy.

2.1.2.1 Metamodel Architecture

The Object Management Group (OMG) [40] proposed traditional metamodel architecture. It is based on four metalevels in which a certain model resides, including:

M0 – level
It contains the original application data, i.e. the runtime instance of a model. For example, the actual data in rows in a relational database table or the representation of the model elements.

M1 – level
It contains a model, e.g. the classes of an object oriented system. At this level actual application, modelling takes place. This level may have instances at the M0 level.

M2 – level
It contains the metamodel that describes language concepts that are used to describe models at the M1 level. For example, figure 2.2 shows a metamodel of a UML class diagram.

M3 – level
It contains a meta-metamodel, i.e. the language concepts to create a metamodel which is at the M2 level. The Meta Object Facility (MOF) is an example of the M3 level; language that describes UML metamodel. These languages are general enough to describe all other languages, and they will even be used to describe themselves. That is why this is called reflexive metamodelling.

Figure 2.5 shows the OMG's four meta levels and relationship between them. The classes at M0 level are the instances of classes at M1 level, which themselves are the instances of classes at the M2 metamodel level, which are instances of meta-meta classes at M3 level, and the classes at M3 level are instances of classes at M3 level.

Meta Object Facility (MOF)
MOF [39] is at the M3 level of the OMG's four layers metamodelling hierarchy and hence at the meta-metamodel. It is an OMG standard language that is used to define modelling languages. It is used to specify, construct, manipulate and manage technological independent metamodels [22]. MOF is defined using MOF itself, i.e. it's a reflective language, and it provides a basis to define any other modelling language.

Unified Modeling Language (UML)
UML [42] is an industrial standard general purpose modelling language. It resides at M2 level of MDA four layer architecture. It is described by UML metamodel, which is an instance of MOF and described by MOF. It is used to describe and specify software systems.

2.1.2.2 Metamodelling Process

The complexity of a language being defined affects the task of creating a metamodel [50]. The metamodelling process has the following five steps.

- defines abstract syntax.

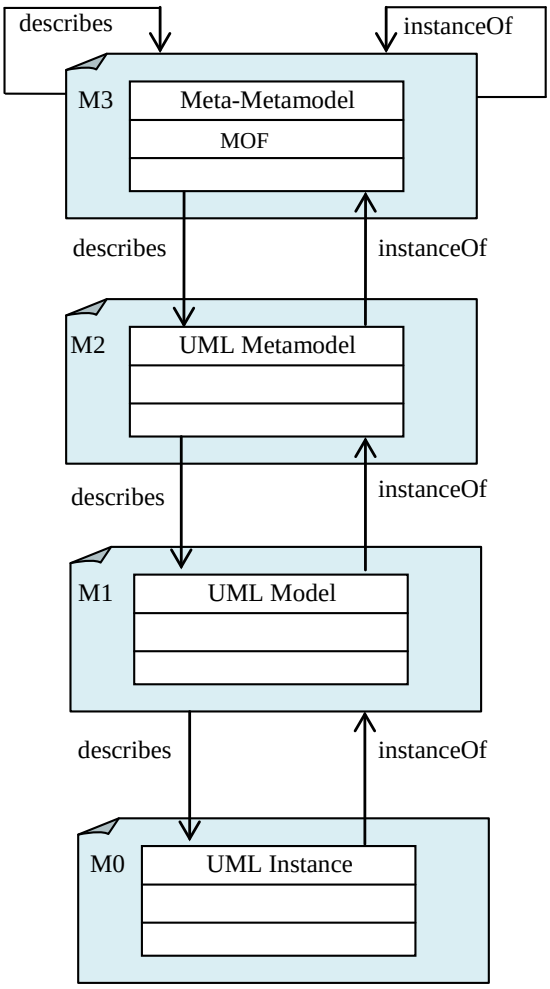


Figure 2.5: The four metalevels of OMG

- defines well-formedness rules.
- defines concrete syntax.
- defines semantics.
- define mapping to other languages.

2.1.3 Model Driven Architecture MDA

Model driven Architecture (MDA) [36] was initiated by Object Management Group (OMG) [40] in 2000 and it is the reference implementation of MDE concepts. It is based on four-layered metamodelling architecture and some set of technologies, including:

XML Metadata Interchange (XMI)

XMI [41] is an OMG. standard that is used to map MOF to eXtensible Markup Language (XML). It allows models, which are based on MOF to be formally described using XML. It allows different applications to exchange metadata information by using XML.

Object Constraint Language (OCL)

OCL [16] is an expression and a specification language. It is used to write expressions on models. It can be used both with UML and MOF models. It extends the expressive power of UML or MOF based models. Traditionally, it has been used with UML based models to specify constraints on models.

Informally, a platform "it is used to refer to technological and engineering details that are irrelevant to the fundamental functionality of a software component" [36]. Platform Independent Models (PIM) and Platform Specific Models (PSM) are two important terms used in Model Driven Engineering (MDE). A PIM is a technology independent model, and it does not have information about the technology which is used to implement the underlying system. On the other hand, a PSM is an implementation dependent model and describes an implementation of a system using a specific technology. Some examples of platforms are `J2EE`, `.NET` [36].

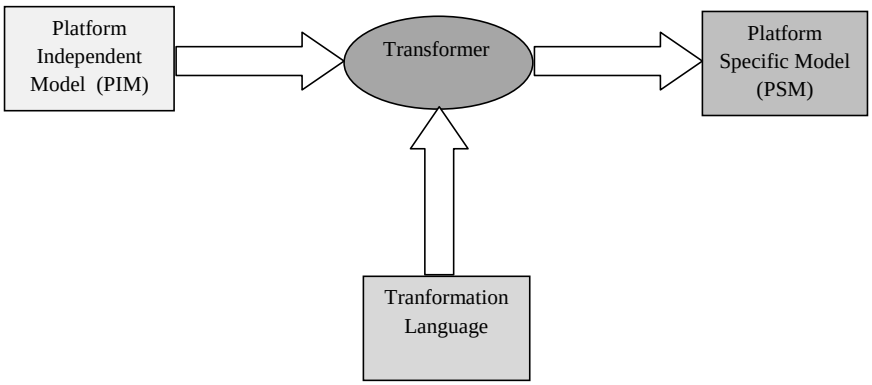


Figure 2.6: Transforming a PIM into PSM using transformation tools

The basic idea and purpose of PIM is to allow software developers to focus on the details of the system without going into the details of any specific technology or platform. By following this approach the developers can focus on domain concepts instead of thinking about which specific technology to use and what are the requirements for the technology. So it will raise the level of abstraction, which is the main idea behind MDE. PIM makes it easier for developers to have an overview of the whole or part of the system

to get a better understanding of the system as all the technology-related issues are hidden. It not only facilitates the developers, but it also helps people, e.g. domain experts to understand the system without having any knowledge of the platform.

A PSM is an extension of PIM with the details that how the system uses a specific platform. Depending on its purpose, a PSM provides more or less details. If it provides information that is needed to develop a system it will be an implementation that can be put into operation. It may be a PIM to be further refined and used as a PSM that can be directly implemented.

A PIM is transformed into a PSM by means of a model transformation as shown in figure 2.6. According to OMG " A model transformation is a process of converting one model to another model of the same system" [27]. A transformer takes PIM and mapping (Transformation Language) as input and produces PSM as output as shown in figure 2.6. It can be done manually or automatically.

2.2 Unified Modeling Language (UML)

Unified Modelling Language (UML) is a general purpose modelling language which is used for the specification, construction and documentation of the software artifacts [25]. It can be applied to all application domains, e.g. health, finance and implementation platforms, e.g., `J2EE, .Net`. It has been an OMG standard since November 1997. Since then it has been used as a dominant modelling language in software industry.

2.2.1 Classification of Diagram in UML

UML is a graphical notation language. The Models defined in UML are represented as a set of diagrams with each diagram focus on the different aspects of a software design.

UML has nine different types of diagrams to design different aspects of a system, and all these different diagrams are described by MOF. These diagrams are classified either as static or dynamic diagrams.

Static Diagrams
: These types of diagrams focus on the static structure of the system by using classes, attributes, operations and relationship between classes. These diagrams include class diagram, object diagram, Package diagram and component diagram.

Dynamic Diagrams
: These types of diagrams focus on the dynamic behaviour of the system

by using objects, show how different objects collaborate with each other and how the state of the object changes. These diagrams include sequence diagram, activity diagram, and state machine diagram.

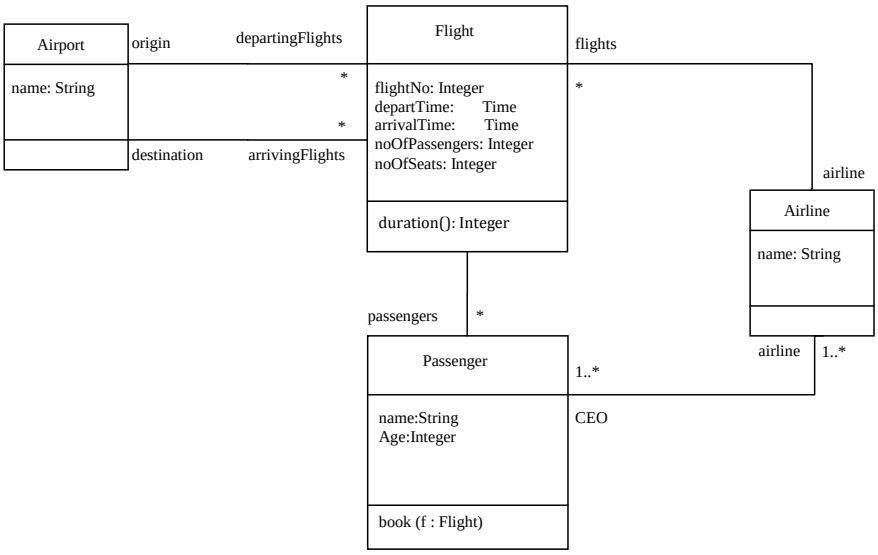


Figure 2.7: UML class diagram for flights

2.2.2 Example of UML

The example specifies a simplified model of an airport flight system. Figure 2.7 shows a UML class diagram of this model.

We briefly summarize the key aspects of the example.

- An `Airport` is the `origin/destination` of a `Flight`. An `Airport` has a `name` and has many `departingFlights` and `arrivingFlights`.
- An `Airline` has many `Flights`. Each `Airline` has a `name` and has at least One `CEO` who is also a `Passenger of the Flight`.
- A `Flight` has many `Passengers` and it has a unique `flightNo`.
- Each `Passenger` has `name`, `age` and `book` a `Flight`.

2.3 Constraints in MDE

A model defines basic building blocks in MDE, from which instances can be created. A constraint is a restriction, assertion, or condition related to some model elements. They specify properties that must be satisfied. Constraints are the well-formedness of the meta-classes [26]. They are defined in the form of `invariants` and must be satisfied by all the instances of the meta-class for a model to be meaningful [26]. UML which is a MOF based modelling language, allows simple diagrammatic constraints like multiplicity and ordering constraints to be directly added into the model. These constraints are part of the UML model structure and are called `structural` constraints.

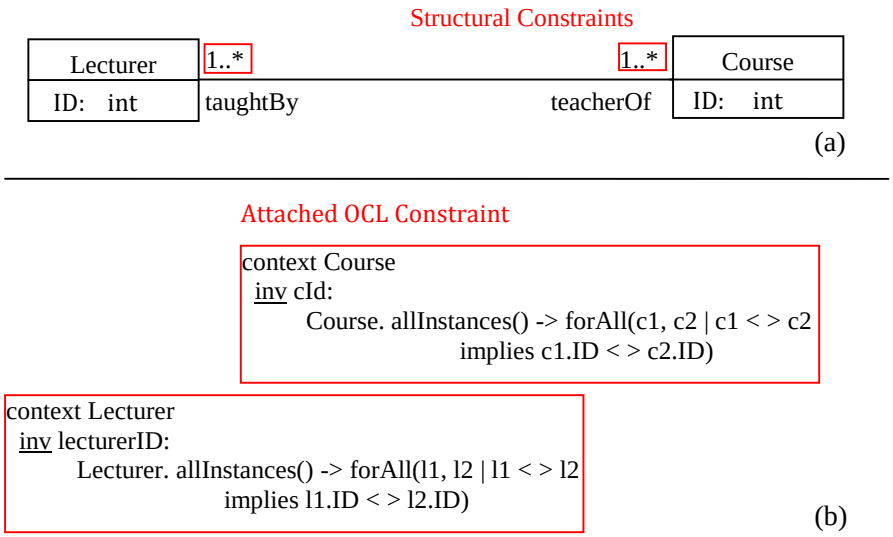


Figure 2.8: Constraints in MOF-based modelling languages: (a) structural constraints in UML (b) attached OCL constraint

For example, the requirement that a course must be `taughtBy` at least one `Lecturer` in the UML model in figure 2.8(a) was specified by the multiplicity constraint that uses the properties `lowerBound` and `upperBound` of the `Property` Class of UML metamodel. So the instances of the model must satisfy the multiplicity constraint in order for the model to be well-formed.

The structural constraints also includes typing constraints, which are defined by the metamodel of the modelling language [45]. These types of constraints specify which `type` of elements a model can have and how these elements are related to each other. For example, according to UML metamodel shown in figure 2.3, a UML class diagram may have `classs`, `association` and `properties` with `upperBound` and `lowerBound` values. However, these

structural constraints are not enough to specify all complex constraint's requirements. So UML models are extended with textual constraints by using an expression language like Object Constraint language [16]. Since these constraints are external to the UML structure, they are called `attached` constraints. In the UML class diagram, a constraint is added to the level of classes, but its semantic is applied on the level of objects. So the constraint which is specified in a modelling language should be satisfied by the models and constraints specified in a model should be satisfied by the instances of that model.

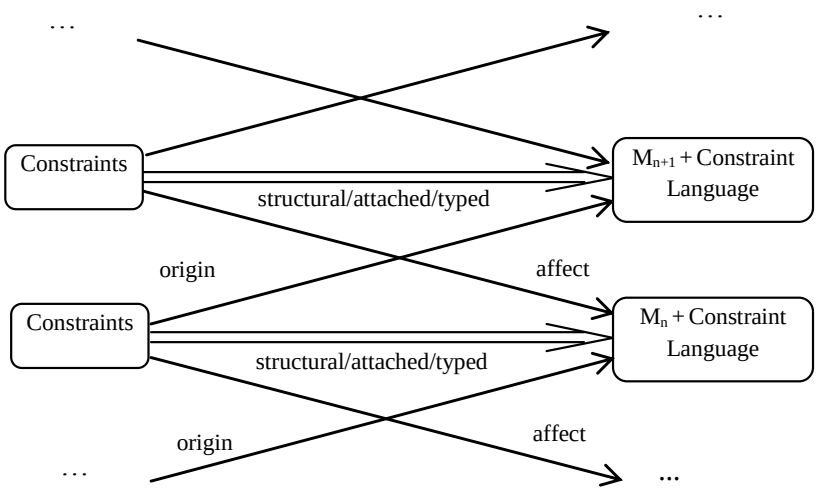


Figure 2.9: constraints in metamodelling

In metamodelling as shown in figure 2.9 there may be two kinds of constraints, including:

Structural Constraints

They came from the modelling language whose metamodel is at the level M_{n+1} and can be added to the model at the level M_n. So the model at the level M_{n-1} must satisfy these constraints.

Attached Constraints

These Constraints are written in a textual language like OCL and added to the Model at M_n level. These constraints should be satisfied by the model at M_{n-1} level.

In order to show the usage of Object Constraint Language (OCL) we add some more information to the UML class diagram in figure 2.8(b) including:

- **Requirement 1**: Each `Lecturer` is `teacherOf` at least one `Course`.
- **Requirement 2**: Each `Course` is `taughtBy` at least one `Lecturer`.

- **Requirement 3**: Each `Course` has a unique ID.
- **Requirement 4**: Each `Lecturer` has a unique ID.

The first two requirements can be ensured by using UML structural multiplicity constraint. However, the last two requirements cannot be ensured by the UML structural constraints. So these requirements can be achieved by using the attached OCL constraints as shown in figure 2.8(b). Listing 2.2 describes an OCL `invariant` constraint stating that "all the instances of `context` type `Course` can not have the same attribute `ID`". Listing 2.3 states that "all the instances of `context` type `Lecturer` can not have the same attribute `ID`".

```
context Course
  inv cId:
          Course. allInstances()->forAll(c1, c2 | c1 <> c2
                                  implies c1.ID <> c2.ID)
```

Listing 2.2: OCL Constraint on Course CLass

```
context Lecturer
  inv lecturerID:
             Lecturer. allInstances()->forAll(l1, l2 | l1 <> l2
                                  implies l1.ID <> l2.ID)
```

Listing 2.3: OCL Constraint on Lecturer Class

2.4 Object Constraint Language (OCL)

This section describes the Object Constraint Language (OCL). The primary references are the OMG standard [26] and the book by Jos Warmer and Anneke Kleppe [53].

The Eclipse infrastructure for modeling software systems is based on UML with support for OCL. It is available in eclipse as `EclipseOCL` [15] as part of a modelling component. Eclipse also provides a Console for the interactive evaluation of OCL expressions on models.

Object Constraint Language (OCL) has been part of the official OMG's standard for the Unified Modelling Language (UML) from version 1.1 onwards.

OCL was initially developed by Jos Warmer in IBM as a language for business modelling. During the last few years, OCL has been used for object oriented models as an expression language. OCL was adopted as a formal specification language within UML. Currently, it is used in semantics description to provide well-formedness rules to UML models. OCL not only facilitates UML users to define precise constraints on UML models, but it can also be used to define the semantics of the modelling language itself.

OCL is a formal specification language that is used to express constraints on MOF based models. All OCL constraints are side effect free. OCL does not contain commands that will change the state of the system, but it contains only operations that observe and check the state of the system.

Originally, OCL was developed for expressing constraints on UML models, but due to its ability to navigate through the model, and form the collection of an object has led it to be used also as a query language.

According to [53], Warmer and Kleppe defined constraints as follows:

"A constraint is a restriction on one or more values of (part of) an object-oriented model or system."

A UML diagram is generally not enough to define all the aspects of the specification. To design a model, there may be a need for additional constraints on the model elements. Initially, these constraints are often defined in natural languages but the problem was that they got ambiguous. To overcome this ambiguity, formal languages have been introduced. The problem with traditional formal languages was that they heavily rely on mathematical notation and are only usable by persons with mathematical background. They are hard to understand and use by the average system modeller. To fill this gap OCL has been developed. It is a formal language but is easy to read and write.

Every OCL expression evaluated on the types which are defined in UML class diagram, including classes or interfaces [53]. The OCL expressions add important information to the object-oriented models, e.g. UML class diagram that often cannot be expressed by using a UML diagram. In UML 1.1, this information is only limited to constraints but after UML 2.0, OCL expressions can be used to define querying, stating conditions or adding business rules to a model [53].

OCL is not a programming language so it is not possible to write a program in OCL.

2.4.1 OCL constraint

An OCL constraint is a sequence of identifiers, literals, white spaces, Special symbols, or comments.

Identifiers
They are case sensitive. Identifiers (i.e. variables, types) begin with a letter which can be followed by any number of letters or digits or underscores e.g. p, a1, Integer as in the ordinary programming languages.

Literals
They include numbers, floating point, single characters, strings and underscore. For example, 5, 9.5, 'a', and "element".

Keywords They are the OCL reserved words, e.g. `context`, `inv`, `post`.

Comments
They start with "- -" two dashes and lead to the end of line.

Special Symbols
They are used as punctuation symbols to create larger structures, e.g. ".", ",", "(", ")", "->".

Operators
For example "+", "-", "*"

2.4.2 Basic structure of OCL Constraint

The basic structure of OCL constraint has two parts(see figure 2.10) including:

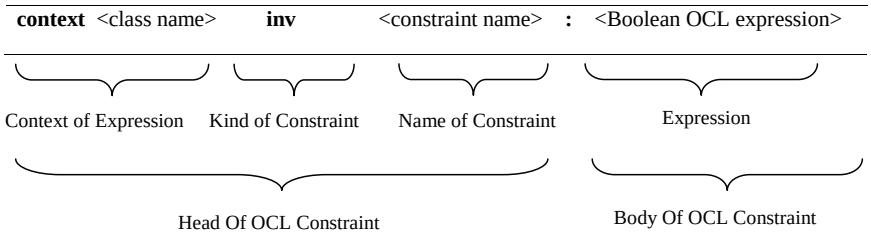


Figure 2.10: Basic Structure of OCL constraint

- **Head of Constraint**
  - **Context of Expression:** The `context` information contains the reserved word `context` and the name of the class or interface on which the OCL statements are parsed and evaluated. So the instances of this class satisfy this constraint. In OCL expression the reserve word `self` is used to refer to the object of the corresponding `context` of the constraint. For instance, if the `context`

is `Lecturer` in figure 2.8(b), then `self` refers to an instance of `Lecturer`.

- **Kind of Constraint:** Discussed in section 2.4.5.
- **Name of Constraint:** This part is used to assign a name to a constraint. This is optional. For instance, in figure 2.8(b) `lecturerID` is a name of the constraint.

- **Body of Constraint**
  - **Expression:** Discussed in section 2.4.4.

2.4.3 OCL Types

Basic Types

OCL is a strongly typed language. So every expression must have a type. It has a number of predefined types and operations related to these types. These types are part of the OCL definition.

Basic types: The OCL predefined basic types are `Boolean`, `String`, `Integer`, and `Real`. The operations on these basic types are logical operation such as and, or, not, string manipulation operations such as `concat()` and `substring()`, arithmetic operations such as $+, -, *$, respectively.

Collection types: All Collection types, `Set`, `Bag`, and `Sequence` are parameterised types.

Set : A `Set` is like a mathematical set with no duplicate values.

Bag: A `Bag` is like a multiSet i.e. allows duplicate Values.

Sequence: A `Sequence` is like a Bag in which elements are ordered. It is represented by a `list`.

```
Set { 1 , 2 , 6 , 88 }
Sequence { 1, 3, 45, 2, 3 }
Bag {1 , 3 , 5, 3, 5 }
```

Listing 2.4: Enumeration type syntax

OCL has a number of operations for collection type. Some of them are:

select(): It is used to select some values from a collection for which the Boolean expression evaluates to true. The result of this operation is a sub collection of the original collection.

forAll(): It is used to apply a Boolean expression on all of the elements of a collection. The result of this operation is true if the Boolean expression for all the elements of collection evaluates to be `true`, otherwise `false`.

exists(): This operation is used to check whether there is at least one element in a collection for which the Boolean expression evaluates to `true` or `false`. The result is either `true` or `false`.

Enumeration types: The syntax of Enumeration is shown in listing below.

```
enum { value1, value2, value3}
```

Listing 2.5: Enumeration type syntax

In OCL expressions, the individual values from an enumeration are referred to with a prefix symbol (#) , for example #value1.

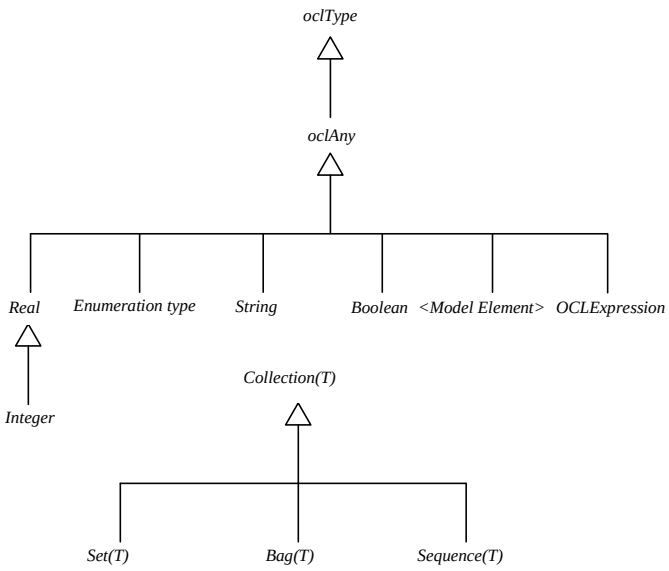


Figure 2.11: Overview of types in OCL

Class types

These types are defined by the UML Class diagram implicitly. The classes defined in a UML class diagram are also valid types in OCL

expressions related to that class diagram. The operations for the class type deal with the properties, i.e. attribute, operations and association of that class. These operations are defined as part of the UML model.

Special Types

The `OclType, OclAny, OclExpression,` and `OclState` are all special types in OCL. The `OclType` is the supertype of all the types in OCL either OCL predefined types or types define in UML model. `OclAny` is the supertype of all the types except for the collect types. `OclExpression` is the type of OCL expressions or the collection of OCL expressions. `OclState` is a type that is used in state machines to refer to a state name.

Type Conformance Rules in OCL

All the types are organised in a type hierarchy that creates a supertype and subtype relationship between them. The overview of OCL types is shown in figure 2.11.

The defined relationships between OCL types are:

- `Integer` is a subtype of `Real`.
- `Set(T)`, `Bag(T)` and `Sequence(T)` have a common supertype, a Collection(T).
- One special type `OclAny` is supertype of all the types except the `Collection(T)` type.
- The subtype relation is transitive, reflexive, and anti-symmetric.
- A `Collection(T1)` is a subtype of `Collection(T2)`, if T1 is a subtype of T2.

2.4.4 OCL Expressions and Queries

An expression is " a statement which will evaluate to a (possibly empty) set of instances when executed in a context" [26].

A simple OCL expression contains `Literals` and `variables`. The invariant expressions are written in the `context` of a `Classifier` and the instance of that `Classifier` are referred by using the reserved word `self`.

The complex OCL expressions can be made up of `if-then-else` constructs or operations calls. The first argument of an operation call is the object on which the operation is applied to. The notation of this expression is the target expression followed by a dot, the name of the operation and optionally a list of arguments in parentheses. The nesting of operations can be written as a sequence of operations with a dot in between them. The sequence of these operations is applied to the first argument of the sequence. An arrow is used instead of a dot if the target of an operation is a collection

of values.

```
self                  -- self expression
a.name()              -- operation call
a.name().charAt(1)    -- nested operation call
a.name() -> size      -- operation call on collection
self.age              -- attribute access
if Expression then    -- if-expression
     Expression
else
     Expression
endif
```

Listing 2.6: OCL Expression Syntax

OCL is a strongly typed language. Each expression written in OCL has a unique type. Each OCL expression must conform to the type conformance rules of the OCL language. For example, each class in a UML class diagram represents a separate type in OCL, an `Integer` type cannot compare with the `String` or `Boolean` type. The evaluation of OCL expression does not change the state of an object. The actual OCL expression comes after the colon in the structure of an OCL constraint as shown in figure 2.10. The OCL expressions are written by using ASCII characters.

OCL can also be used as a query language. A `Query` is also an expression, but it has an arbitrary result type and has no free variables. A `Query` is used to retrieve complex information from the system.

2.4.5 Kinds of OCL Constraints

OCL defines three different kinds of constraints called stereotypes for the UML class diagrams. Figure 2.7 shows a UML class diagram for flight system [21], which is used in this section to show OCL constraints.

Invariants

An `invariant` is a condition that always holds in order for the model to be well-formed. So it is a static constraint. An `invariant` is an expression that is specified for a `Classifier` in UML class diagram and must be true for all the instances of that type. All invariants are always of `Boolean` type. In an OCL expression, a `context`, which is part of an invariant, is written by using the keyword `context`, followed by the name of the type as shown in listing 2.7. The label `inv` shows that the kind of constraint is an `invariant`. The OCL syntax for an `invariant` is defined as follows:

```
context <class name> inv:
            <Boolean OCL expression>
```

Listing 2.7: Syntax for OCL Invariant

For example in the UML model shown in figure 2.7, if a `context` is `Flight` then the expression in listing 2.8 would specify an `invariant` that every instance of a `Flight` has a unique `flightNo` and that this `invariant` condition should always hold for every instance of type `Flight`.

```
context Flight inv :
                   Flight.allInstances()->forAll(f1, f2 |
                    f1 < > f2 implies
                    f1.flightNo < > f2.flightNo)
```

Listing 2.8: OCL invariant for flightNo

For example, in the context of `Flight`, the two expressions in listing 2.9and 2.10 would specify invariants that the `noOfPassengers` in a `Flight` should be less than the `noOfSeats`.

```
context Flight inv : self.noOfSeats > = self.noOfPassengers
```

Listing 2.9: OCL invariant to limit no of passengers

If the context is clear, we can omit the `self` keyword as shown in Listing 2.10. This invariant is equivalent to the invariant shown in listing above.

```
context Flight inv : noOfPassengers < = noOfSeats
```

Listing 2.10: OCL invariant to limit no of passengers

Pre Condition

The `pre` condition is a constraint that must hold before the execution

of an operation in a particular context type. It is used to define the semantic of an operation/method. The `pre` condition may be attached to any operation in a UML class diagram. The `pre` condition is a Boolean-valued expression on the objects, by using additional parameters that correspond to the parameters of an operation. The caller of the operation is responsible for satisfying the `pre` condition. The OCL syntax for `pre` condition is shown in listing 2.11.

```
context <class name> :: <operation> (<parameters>)
               pre <constraint name>: <Boolean OCL expression>
```

Listing 2.11: OCL syntax for Precondition

The `pre` condition stereotype of a constraint is defined by using the reserve keyword `pre`. The parameters are defined in the same way as in the UML class diagram, the name of the parameter with its type separated by a colon ":". The is optional for precondition and can be written after keyword `pre`, which allows the constraint to be referenced by name. One example of `pre` condition is shown in listing2.12.

```
context Peron :: book(f: Flight)
               pre : self.age >12
```

Listing 2.12: Example of Precondition

Post Condition

The `post` condition is a constraint stereotype that describes how the actual effect of an operation is described in OCL. This constraint must be true just after to the execution of an operation. The OCL syntax for `post` condition is shown in listing 2.13.

```
context <class name> :: <operation> (<parameters>)
               post <constraint name>: <Boolean OCL expression>
```

Listing 2.13: OCL syntax for Postcondition

The `Postcondition` stereotype reserve keyword `post` is used before the actual `post` condition expression. The operation is responsible for satisfying the `post` condition. The <constraint name> is optional.

In listing 2.14, an example constraint states that the result of du-

ration for an instance of type `Flight` is the difference between the `arrivalTime` and `DepartTime`.

```
context Flight :: duration(): Integer
            post:result = self.arrivalTime - self.departTime
```

Listing 2.14: OCL Postcondition example

2.4.6 Other concepts

Some of the important OCL concepts are as follows:

Navigation

An association between two classes can be used for navigation. A navigation expression can start with an object and navigate to a connected object by using the name on its association end separated by a dot. The result of navigation is a single object or a collection of objects, specified by the multiplicity constraint on the association end. Listing 2.15 shows an example of navigation stating that starting from the `Flight` we can navigate to `Airline` by using the name of an association end `airline` and then to `Passenger` by using `CEO` to get the name of the `CEO` of an `Airline`.

```
context Flight inv:
        self. airline. CEO. name
```

Listing 2.15: OCL Navigation Example

Iterate

OCL has an `iteration` expression to provide an iteration construct. The source of an iteration expression is always a collection of values, and the expression is evaluated for each value of a collection, and the result is accumulated in a result variable. Listing 2.16 shows an OCL constraint using iteration function `select()`. This OCL constraint return all the passengers on a flight who are less than two years old.

```
context Flight inv:
        self.Passengers->select(p:Passenger | p.age < 2 )
```

Listing 2.16: OCL Iterate Example

Shorthand Notation

The `let` construct is used to assign a variable to a complex expression which may occur again in a constraint. Example of an OCL shorthand notation is shown in listing 2.17 and it states that a shorthand name `dF` is assigned to the list of all the `departingflights` from an `Airport` and it is not possible that a single `Flight` from that list is also in the list of `arrivingFlight` to that `Airport`.

```
context <Class-name> inv :
        let <name> :TypeName ? = expression in
```

Listing 2.17: Syntax of OCL shorthand Notation

```
context Airport inv :
        let dF :Boolean = self.departingFlights in
       not(self.arrivingFlights->includeAll(dF))
```

Listing 2.18: Example of OCL shorthand Notation

2.4.7 Validation of Constraints

The quality of a software system depends on the verification and validation of a model. Verification and validation are techniques that are used to detect errors in software models early in the development life cycle. Verification of a software model is a technique that ensures that the software fully satisfy all the expected specifications whereas validation is used for the evaluation of models with respect to their semantic and syntax [51].

In Model Driven Engineering, model validation or consistency checking is the process of evaluating model with their metamodels. In addition to typing (metamodel), structural/attached constraints, which are defined in metamodel can be used for validating a model. Validation checks the model to make sure that all the defined constraints must be satisfied for the model to be well-formed.

The Unified Modelling Language (UML) [42] is an important and wide spread modelling language that has support for adding OCL constraints on models. The model and constraints should be validated before the actual coding is started [35]. The purpose of validation is to acquire a good model before the actual implementation starts.

2.5 The Diagram Predicate Framework (DPF)

Diagram Predicate Framework (DPF) [19] is a research-based project. It was started in early 2006 by Bergen University College and the University of Bergen, Norway, with the aim to formalize the important concepts in MDE, including (meta)modelling, model transformation and model management. This formalization is based on category theory and graph transformation.

DPF provides a complete diagrammatic approach to MDE. It is based on the multi-level (meta)modelling hierarchy where the model at each level are formalized as a diagrammatic specification. The DPF extends the Generalised Sketches formalism developed by Zinovy Diskin and Boris Kadish [11]. DPF provides a multi-layer diagrammatic metamodelling hierarchy. Because of its general nature; It can be used to model other modelling languages like UML, or coloured PetriNets [46].

Some of the important concepts in DPF are diagrammatic specification, diagrammatic signature, diagrammatic predicate, atomic constraint, graph and graph homomorphism. All the following definitions are taken from [45].

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 $trg(f) = Y$.

Graph Homomorphism

A graph homomorphism φ from a graph G to a graph H is a mapping $\varphi : G \to H$ which preserves source and target for each arrow, i.e. for each arrow $f : X \to Y$ in G we have $\varphi_1(f) : \varphi_0(X) \to \varphi_0(Y)$.

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, "="] & Y \arrow[d] \\ \varphi_0(X) \arrow[r, "\varphi_1(f)"'] & \varphi_0(Y) \end{tikzcd}$$

Diagrammatic Specification

A DPF `Specification` $\mathfrak{S}$ contains an underlying graph S along with a set of `atomic constraints` $C^{\mathfrak{S}}$.

Diagrammatic Signature

In DPF, a signature $\Sigma = (P^\Sigma, \alpha^\Sigma)$ is a collection of predicates symbols P^Σ with a map α^Σ that assigns a graph to each predicate symbol $p \in P^\Sigma$. α^Σ is the arity of the predicate symbol p.

Diagrammatic Predicate

A predicate from a predefined diagrammatic predicate signature is used to add constraints to a diagrammatic specification (model). A Diagrammatic Predicate has a name, a shape (graph), a visualization, and a semantic interpretation.

Atomic Constraint

An atomic constraint (p, δ) added to a graph S is given by a predicate symbol p and a graph homomorphism $\delta : \alpha^{\Sigma}(p) \rightarrow S$.

In DPF, Each predicate must have semantic, and is given by the set of its instances $\iota : O \rightarrow \alpha(p)$ where each ι is a graph homomorphism into the arity of the predicate [45]. Table 2.1 shows the semantic of predicates from a DPF predefined signature. The semantic interpretation of predicates is given by using the mathematical notation of set theory.

The arity of the `[exclusive-or]` predicate is shown in figure 2.12. Consider the graph homomorphisms shown in figure 2.13.

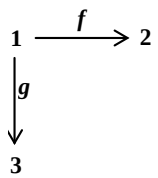


Figure 2.12: arity of [exclusive-or] predicate

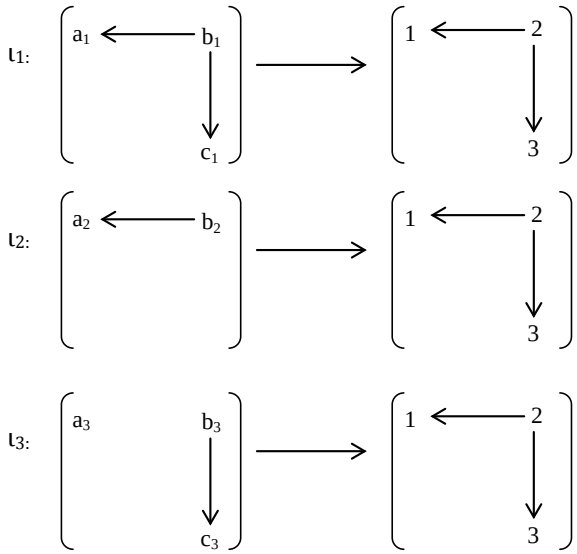


Figure 2.13: Graph homomorphism of [exlusive-or] predicate

p	$\alpha^{\Sigma}(p)$	Proposed vis.	Sem. Interpretation
[mult (n, m)]	1 —f→ 2	X —f [n.. m]→ Y	$\forall x \in X : m \leq \lvert f(x) \rvert \leq n$, with $0 \leq m \leq n$ and $n \geq 1$
[injective]	1 —f→ 2	X —f [inj]→ Y	$\forall x, x' \in X: f(x) = f(x')$ implies $x = x'$
[surjective]	1 —f→ 2	X —f [surj]→ Y	$f(X) = Y$
[irreflexive]	f: 1	[irref] f: X	$\forall x \in X: x \notin f(x)$
[non-overlapping]	1 —f→ 2	X —f [nov]→ Y	$\forall x, x' \in X: f(x) \cap f(x') \neq \emptyset$ implies $x = x'$
[nand]	1 —f→ 2, 1 —g→ 3	X —f→ Y, X —g [nand]→ Z	$\forall x \in X$: $f(x) = \emptyset \vee g(x) = \emptyset$
[inverse]	f: 1 → 2, g: 2 → 1	f: X → Y, g: Y → X [inv]	$\forall x \in X, \forall y \in Y: y \in f(x)$ iff $x \in g(y)$
[exclusive-or]	1 —f→ 2, 1 —g→ 3	X —f→ Y, X —g [xor]→ Z	$\forall x \in X: (f(x) = \emptyset \vee g(x) = \emptyset)$ and $(f(x) \neq \emptyset \vee g(x) \neq \emptyset)$
[jointly-surjective_2]	1 —f→ 2, 3 —g→ 2	X —f [js]→ Y, Z —g→ Y	$f(X) \cup g(Z) = Y$
[jointly-injective]	1 —f→ 2, 1 —g→ 3	X —f [ji]→ Y, X —g→ Z	$\forall x, x' \in X: f(x) = f(x')$ and $g(x) = g(x')$ implies $x = x'$

Table 2.1: Semantics of Predicates

The graph homomorphisms $\iota_1, \iota_2, \iota_3$ represents multi-valued functions. The homomorphisms $\iota_2, \iota_3 \in$ `[exclusive-or]` and the graph homomorphism $\iota_1 \notin$ `[exclusive-or]`.

The semantics of a specification is defined by the set of its instances (I, ι).

Instance of a Specification

An instance (I, ι) of a `specification` $\mathfrak{S}$ is a graph I with a graph homomorphism $\iota : I \to S$ that satisfies all the atomic constraints $C^{\mathfrak{S}}$ and is given by the following `pullback diagram`.

$$\begin{tikzcd} \alpha^{\Sigma}(p) \arrow[r, "\delta"] \arrow[d, "\iota^*"] \arrow[dr, phantom, "PB"] & S \arrow[d, "\iota"] \\ O^* \arrow[r, "\delta^*"'] & I \end{tikzcd}$$

DPF defines two types of conformance relations; `conformance` is a relation that lies between models at any two adjacent levels in a metamodelling hierarchy (i.e. a relation between a model and its metamodel). These conformance relations are `typed by` and `conforms to`.

Typed by

A `specification` $\mathfrak{S}_i$ at the level i is `typed by` a specification $\mathfrak{S}_{i+1}$ at the level $n+1$ if there exists a typing morphism $\iota[i] : S[i] \to S[i+1]$(A graph homomorphism) between the underlying graphs of the specifications [45].

Conforms to

A `specification` $\mathfrak{S}_i$ at the level i is said to conform to a specification $\mathfrak{S}_{i+1}$ at the level $n+1$ if there exists a typing morphism $\iota[i] : S[i] \to S[i+1]$ such that $(S[i], \iota[i])$ is a valid instance of $\mathfrak{S}_{i+1}$ i.e. $\iota[i]$ satisfies all the atomic constraints $C^{\mathfrak{S}_{i+1}}$ [45].

2.6 The Diagram Predicate Framework Workbench

The DPF Workbench is a prototype diagrammatic tool for the specification of metamodels, diagrammatic signatures and for generating specification editors from metamodels. It is the reference implementation of the Diagram Predicate Framework (DPF) concepts [19]. The DPF Workbench supports an arbitrary number of metamodelling hierarchy, i.e. each model at any

level can be used as a metamodel for the next level, and it also has a code generation facility.

The DPF Workbench has been developed in Java as an add-on for the Eclipse Modelling IDE [44]. Figure 2.14 shows architecture of the DPF Workbench. It consists of three main components, which are built on the top of three auxiliary components. The auxiliary component "DPF Core" provides the core functionality of the tool. The core functionality includes features such as creating, storing and validating the DPF models. It is based on Eclipse Modelling Framework (EMF) [13]. The "DPF Diagram" component extends the "DPF Core" component by an auxiliary component, which stores additional information about the visualization of models. It depends on the Graphical Editing Framework (GEF) [23]. GEF is used to built rich client editors and views for the Eclipse platform following a Model-View-Controller (MVC) [38]. The "DPF Xpand metamodel" auxiliary component implements an extension to the Xpand generator.

The three main components are "DPF Model Editor", the "DPF Signature Editor", and the "DPF Code Generator". The first two components provide visual editors; the "DPF Model Editor" component allows the creation and modification of DPF specifications. It uses "DPF Core" and "DPF Diagram" components and also the view part of GEF's MVC architecture. The "DPF Signature Editor" allows the creation of user defined, reusable predicate signatures [33]. It mainly relies on the functionality provided by the "DPF Core" and the "DPF Model Editor". The last component, "DPF Code Generator" builds on top of "DPF Core" and "DPF Xpand metamodel" [47].

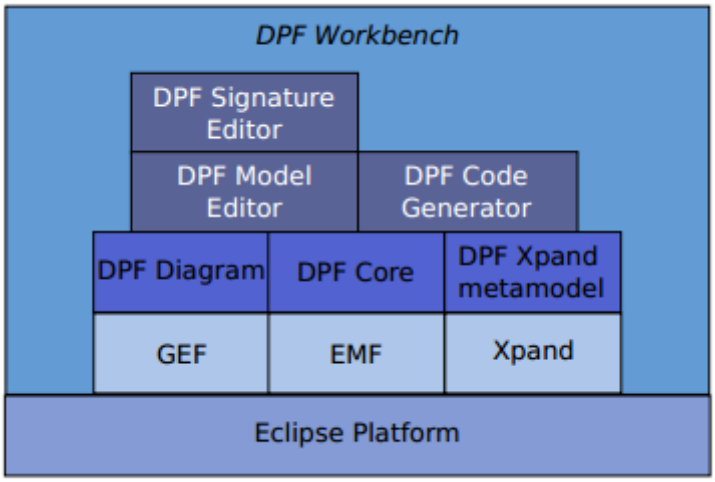


Figure 2.14: The main component architecture of the DPF Workbench

2.6.1 The DPF Model Editor

The DPF Model Editor is a prototype Model Editor tool for the diagram predicate framework. Since 2006 there has been made a lot of effort to develop the DPF Model Editor.

In 2006, the first attempt was carried out by Ørjan Hatland [28] The Sketcher .NET was developed using a Microsoft .Net framework. Unfortunately, this implementation was not completed, and some of the features left unimplemented. It was not considered as a good foundation for further development.

In 2008, Stian Skjerveggen [48] started work on a DPF editor based on eclipse framework [44]. This editor was developed using the Graphical Modelling Framework (GMF) [49]. GMF is a tool for the generation of editors and tools based on Eclipse Modelling Framework (EMF) [13] and Graphical Modelling Framework (GEF) [23]. But at the end of the project Skjerveggen concluded that GMF is not a suitable technology for the development of the DPF Workbench.

In 2010, Øyvind Bech and Dag Viggo Lokøen have started to work on the current implementation of the DPF Model Editor [4]. It is developed using the EMF and GEF technologies. This DPF editor supports most important features of metamodelling:

- It supports graph based creation of models.
- It supports an arbitrary number of metamodelling hierarchy.
- It supports graph homomorphism based type checking between two adjacent meta levels.
- It supports constraints checking between meta levels.
- It supports loading and storing of models in the form of XML Metadata Interchange (XMI) [41].

Figure 2.15 shows the editor view in the DPF editor. To create a model in the DPF Model Editor we start by using `Node` and `Arrow`. This model has two nodes `A, B` and an arrow `AB` in between them. There is also one constraint named `[surj]` on arrow `AB`. Figure 2.16 shows how two adjacent meta levels typed by each other. The dotted line shows the typing relation between two adjacent meta levels.

2.6.2 Metamodel based Code Generation

In 2011, Anders Sandven, in his Master thesis extended the DPF Editor functionality with `Metamodel based Code Generation` [47] feature using

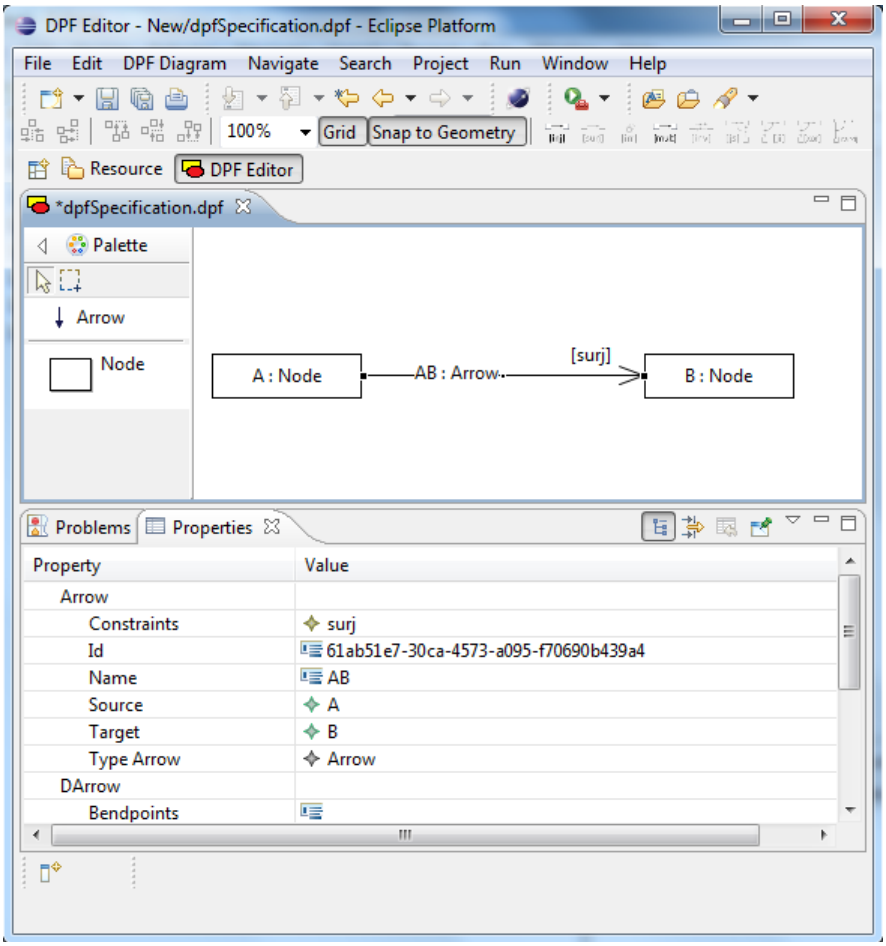


Figure 2.15: Creating a model in the DPF Editor

the Xpand Framework [54]. The code generator is based on eclipse technologies and is available as a plug-in inside eclipse. This metamodel based code generator provides a generalized solution that is suitable to all domain-specific modelling languages (DSML).

The main features of this metamodel based code generation facility are:

Xpand metamodel

The Xpand metamodel for the DPF, is the core functionality of this code generation facility. It is a mapping between the DPF types, and the project defined Xpand custom types.

Type system

The defined type system enables Xpand to understand modelling constructs of the DPF by defining new functionality for each DPF modelling construct such as DSML specific getters and setters or the core

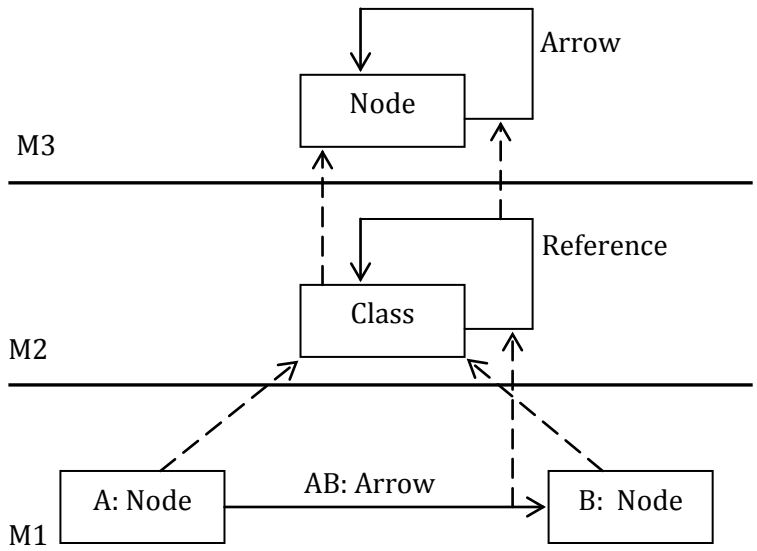


Figure 2.16: Typing relation between different layers in the DPF

functionality defined in the DPF Ecore metamodel.

Workflow integration

It is done by implementing the `DPF Reader component`, which integrates the DPF metamodel with the Modelling Workflow Engine (MWE). The Workflow integration allows the DPF metamodel to integrate with the different components offered by the Xpand Framework.

Eclipse integration

The metamodel contributor provides the support of using a template editor for Xpand, extension editing with Xtend and constraints checking with Check.

Project environment

A project environment defines a wizard for generating a project structure that is appropriate for code generation.

2.6.3 The DPF Signature Editor

The Signature Editor is an editor that extends the DPF Workbench with the functionality to define domain-specific predicates. It was implemented by PHD student Xialiang Wang.

A signature contains a set of predicates. Each predicate has a name, syntax, visualization, a semantic validator, properties and a graphical icon. Those predicates are used to add constraints on models. The syntax of a predicate is defined by a shape graph and a graphical icon. Their semantics are defined by a java validator. Figure 2.17 show the Signature Editor. The detail on how to define dynamic predicates using the Signature Editor can be found in appendix A.

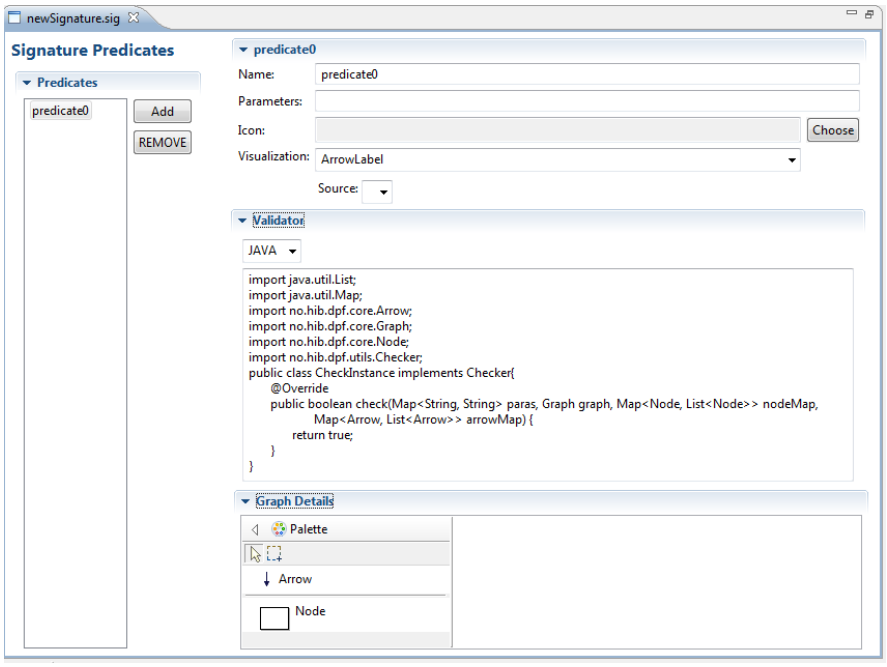


Figure 2.17: Definition of new predicate using Signature Editor

Name of Predicate

It is the symbol that is appeared when the predicate is added to a model in the DPF Model Editor.

Predicate Properties

The predicate properties define the minimum and maximum number of instances of one model element related to the number of instances of another model element. They are optional, but can be defined as `min` and `max` values separated by a semicolon.

The format of parameters is “min:0;max:1”

Graphical Predicate Icon

It is the image that will contribute to the DPF Model Editor’s signature

toolbar. If an icon is left empty the name of the predicate will appear on the Signature toolbar.

Predicate Visualization

It depends on the arity of the predicate. There are different types of predefined visualizations in the Signature Editor, including `ArrowLabel`, `ArrowToArrow`, `NodeToArrow`, `ArrowToNode` and `composed` depending on the syntax of the predicate. A user can select a Visualization type depending on the number of the model elements involved in a predicate, e.g. if a predicate is unary or binary you can select ArrowLabel, NodeToArrow or ArrowToNode and if the predicate is ternary you can select ArrowToArrow, etc.

Predicate Semantic Validator

It defines the semantics of a predicate. The semantics of a model describes what the effect is of executing a model. A user can define the semantics of a predicate by using the java programming language.

Predicate Syntax

It is an arity (shape graph) of a predicate.

Chapter 3

Constraints and their Validation in MDE

This chapter explains the use of constraints in MDE and why there is a need for adding constraints to the UML models by using a textual language like Object Constraint Language (OCL). It also discusses some of the existing approaches to the visualization of OCL. At the end of the chapter, the current DPF validation mechanism and Java validator is presented.

3.1 What are Constraints in MDE

A model is considered as a basic artefact in Model driven engineering (MDE) and from models, instances of models can be constructed. The Meta-Object Facility has classes, association between classes and structural features of classes. Usually models are specified in a graphical way which only allows rough specification of the system.

A number of possible instances can be constructed from model elements that are defined in a model. However, its not possible that all the instances are valid with respect to the semantics of the model. Constraints are used on models to express restrictions or details that are not possible to express in a diagrammatic way (i.e. only typing and multiplicity constraints can be expressed diagrammatically).

A constraint is a restriction on instances of an object-oriented model elements or system [53].

Constraints are used to add restrictions on UML model elements. It can be attached to all kinds of model elements. For example, a Class `invariant` can be specified by adding a constraint to a `Class`. The constraints defined a `Boolean` expression that must be satisfied by all the instances of that `Class`.

The constraints are specified on model elements but the instances of those model elements have to fulfill these constraints.

Constraints may have different sources including:

- Some legal restriction that a model/system should obey.
- Some grant privileges to certain types of customers according to the company policies.
- Some technical restriction on the system.
- There may be some constraints that cannot be expressed diagrammatically.

3.2 Constraints and OCL in UML

The Unified Modelling Language (UML) is used as a standard general purpose modelling language for modelling software systems.

A UML Class model only contains information about the structural aspects of the software system. The UML models specify software systems in a graphical way, but the problem is that the graphical specification languages like UML does not have enough expressive power to describe all the aspects of the software system. So there is a need to describe additional constraints about the objects in a model. Typically, these constraints are specified using a natural language but this may lead to ambiguities. In order to specify the unambiguous constraints on objects in the model, formal languages have been developed. However, the problem with these formal languages is that they are strongly based on mathematics and requires a good mathematical understanding by the users, so they are difficult for a normal system modeler. Formal languages are mostly applied in the academic world instead of the industry.

OCL can be used for a number of different purposes::

- to specify `invariants` on `Classes`.
- to specify `invariants` on `Stereotypes`.
- to describe `pre- and post-condition` on operations or methods.
- as a `navigation` language.
- to specify constraints on operations.
- as a `Query` language.
- to specify initial or derived values for `Attributes` and `Association` ends.

3.3 Visualization of OCL

Although UML is used as a standard modelling language for documenting and designing software systems, but there are still some differences due to the use of its sub-languages. For instance, the Object Constraint language (OCL), which is used to add constraints to the UML model, is, however, of limited use in organisations. The reasons for that is the integration of a purely textual language into the diagrammatic paradigm. The combination of UML and OCL needs the users to learn two separate languages to represent same model elements. Both languages are based on different metamodels. The Visualization of OCL is trying to overcome this gap.

To express logical constraints in object oriented modelling, two diagrammatic languages have been developed. These two visualizations of OCL are `VisualOCL` [52] and `Constraint Diagrams` [31]. These two visualizations were designed for the diagrammatic modelling paradigm.

3.3.1 VisualOCL

`VisualOCL`, which is a visualization of OCL was developed in [6, 9] as an alternative to the textual OCL. `VisualOCL` is the visualization of complete OCL 2.0 version 1.5. The tool for `VisualOCL` is available as an Eclipse plug-in.

Visual Object Constraint Language (`VisualOCL`) is a graphical representation of Object Constraint Language (OCL). It is based on OCL metamodel. `VisualOCL` follows the UML graphical representation and notation. It makes the integration of OCL in UML diagrams easier. `VisualOCL` is also formal, object oriented and typed language like OCL. The advantage of `VisualOCL` is that the user does not have to learn another textual language. The new data types such as collection and operations such as `forAll()`, `union()` and `select()` are represented by simple graphics. The logical expressions are denoted by using the different kind of boxes to express conjunctions and disjunctions.

Representation Of an OCL Constraint

An OCL constraint is visualized as a rounded rectangle as shown in figure 3.1. The rectangle is divided into two main sections including:

context section

The top-most section is a context section. It contains the keyword `context` followed by the type-name, e.g. class-name of the model followed by the kind of the model e.g. `inv, pre`.

body section

The body of a constraint is visualized in the body section. The

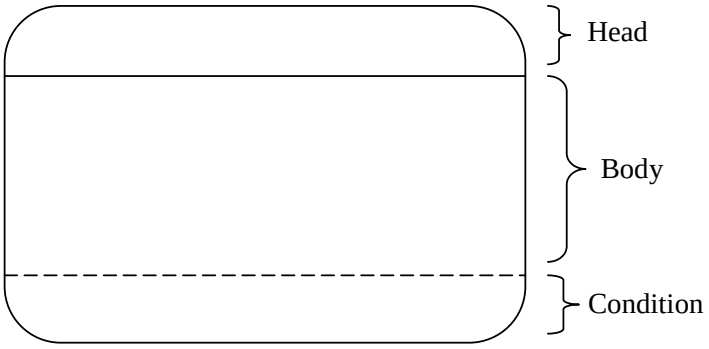


Figure 3.1: Principle representation of a constraint

body section can contain a condition.

condition section
The condition of the constraint is declared in the condition section by using the variables declared in the body section. The condition section is separated by a dashed line from the rest of the body.

Examples Of Visual OCL Constraint:
The OCL constraint shown in listing 3.1 can be visualized by a rounded rectangle in `VisualOCL` shown in figure 3.2.

```
context Person inv: self.age > 0
```

Listing 3.1: Textual OCL Constraint

In this example:

- `self` refers to the instances of class `Person`.
- the variable `x` refers to the `age` of the `Person`.
- The condition declared in the condition section states that the `age` of an individual `Person` is always greater than 0.

Another example shown in listing 3.2 specifies that the `duration` of a `Flight` is the difference between its `arrivalTime` and `departTime`, and the return type of the operation is of type `Integer`. This constraint is a `post` condition kind of constraint. The `post` condition is visualized by an arrow from the instance on which the operation is called to the instance which is being called. This post condition constraint can be visualized by the diagram shown in figure 3.3.

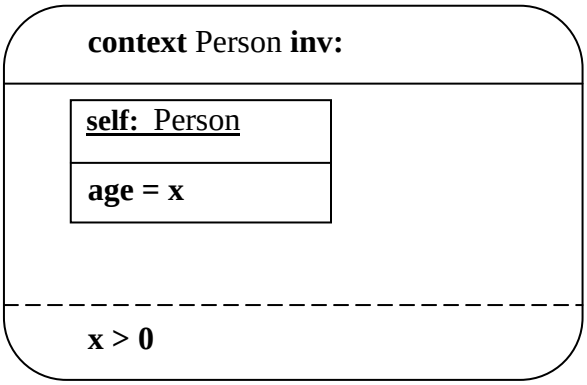


Figure 3.2: Example of Visual OCL Constraint

```
context Flight :: duration(): Integer
          post:result = self.arrivalTime - self.departTime
```

Listing 3.2: OCL Post condition example

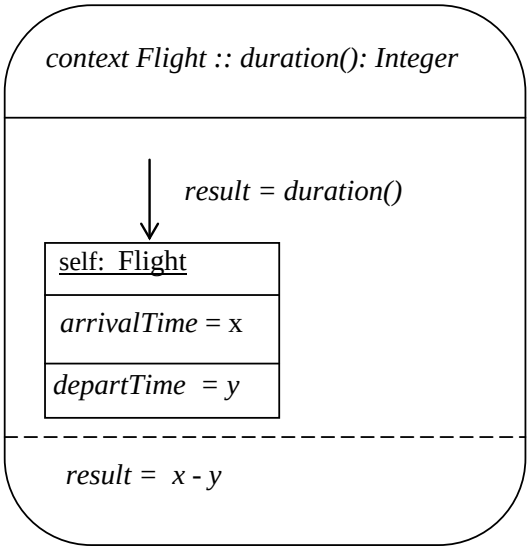


Figure 3.3: Example of Visual OCL Constraint for $Post$ condition

`VisualOCL` is a partially diagrammatic form of textual OCL. `VisualOCL` uses UML Collaboration Diagrams to visualize the OCL constraints, but they

are basically relied on text, and they also need extra text for the context declaration, collection operations and attributes. `VisualOCL` involves more syntactic elements for OCL thus closer to textual OCL. The tool available for `VisualOCL` allows users to transform VisualOCL constraints to textual OCL constraints, which can then be used with UML models, it does not have support for the model editor [52] by itself.

3.3.2 Constraint Diagrams

`Constraint Diagrams` is a visual notation to express logical constraints on object oriented models, it was introduced by Stuart Kent in [31]. The formalization of constraint diagrams has been completed in [17]. They can be used as a visualization of OCL in the context of UML, but a complete modelling framework for `Constraint Diagrams` is not fully developed [1], i.e. it does not have support for the modelling. However, they have a mechanism to translate constraint diagrams to other notations, e.g. to First-Order Predicate Logic (FOPL) or to OCL using Kent Modelling Framework (KMF) [20]. The example shown in listing 3.3 specifies that the `firstname` of a `Person` must be different from `lastname`.

```
context Person inv: self.firstname <> self.lastname
```

Listing 3.3: Textual OCL Invariant

A visualization of the above constraint in `Constraint Diagrams` is shown in figure 3.4. The class `Person` is modelled by a rectangle which represents a set. The attributes `firstname` and `lastname` are shown using arrows, which represent relations. The asterisk labelled p corresponds to $for\ all\ p$: `Person`. The keyword `self` can be used for the `context` instead of p. The two dots labeled s and t represents the existence of two distinct elements. The reading tree under the outer box specifies the order in which to read the diagram. Thus, the diagram is read as “every person has a firstname and a different lastname”.

`Constraint Diagrams` are fully diagrammatic with a minimum amount text. They are based on first-order predicate logic. This structural approach of `Constraint Diagrams` is not based on graphical representation of the UML diagrams. The semantics of OCL constraints are expressed by set theory. They are similar to `Euler diagrams` [2] in structure. Due to their diagrammatic nature, they are more difficult to read and understand for the normal system modeler, but can be easily understand by the persons with mathematical background or persons having knowledge of `Euler diagrams`.

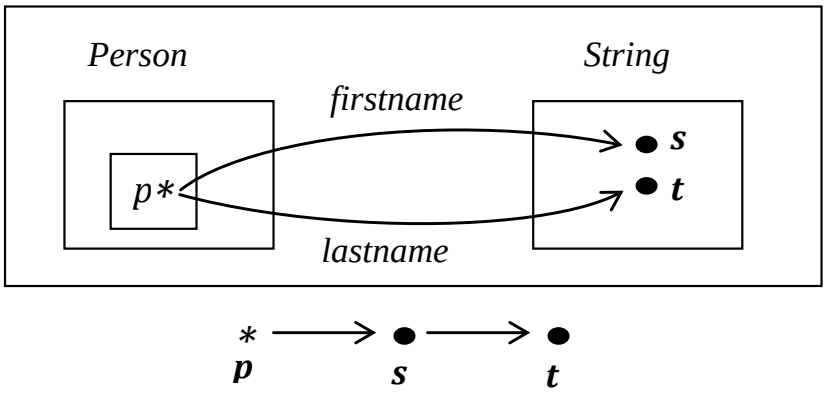


Figure 3.4: Each person has a different first name from last name

3.4 DPF Semantic Validator

The DPF Model Editor has a (hardcoded) java validator for all the DPF predicates. This is not a `general-purpose` validator for the DPF semantics and was implemented only for the prototyping purposes. In the DPF Editor, in order for the model (specification) to be `valid`, all the constraints that have been applied to the metamodel (Meta specification) should be valid. The semantic validator of the DPF Workbench is classical, meaning that it will validate a complete model every time a single model element has been changed.

Listing 3.4 specifies a simplified implementation of the DPF semantic validator, generally it implements a pullback diagram from figure 2.5. This method is triggered every time a model (graph) is changed i.e. when a node or/and an arrow is added or removed from a model.

```
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 3.4: A simplified listing of the semantics validator

The example shown in figure 3.5 is modelled by using the DPF Model Editor. The information system in this example models students working on projects at universities.

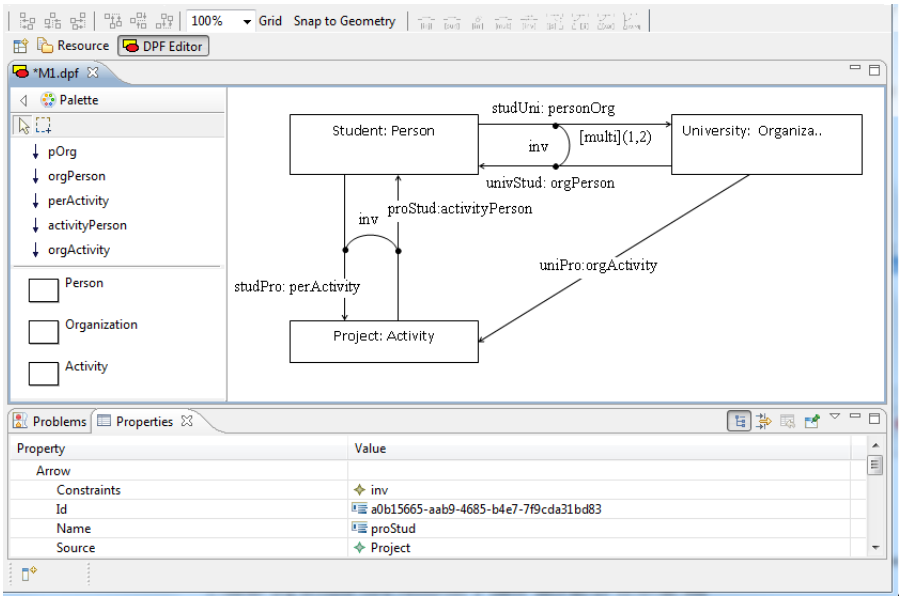


Figure 3.5: Model of different Classes (Student, University and Project) and association between them in the DPF Model Editor

The basic requirements to model this information system includes:

Nodes

A set of nodes is `Student, University` and `Project`.

Arrows

The arrows between Classes are:

- Universities can have arrows to `Students` and `Projects`.
- `Students` can have arrows to `Projects` and `University`.
- A `Project` can have arrow to `Students`.

constraints

The requirements are expressed by following constraints:

- A `Student` must be connected to at least one and at most two universities i.e. `[mult](1,2)`.
- Arrows between `Students` and `universities` must be `inverse` and arrows between `Students` and `Projects` must be `inverse`. These requirements are expressed by adding constraint `[inv]`.

Figure 3.6 shows an instance of a model at level `M0` that conforms to the

model at level `M1`, note that the DPF Model Editor shows message on the status bar stating that "`All constraints validated`".

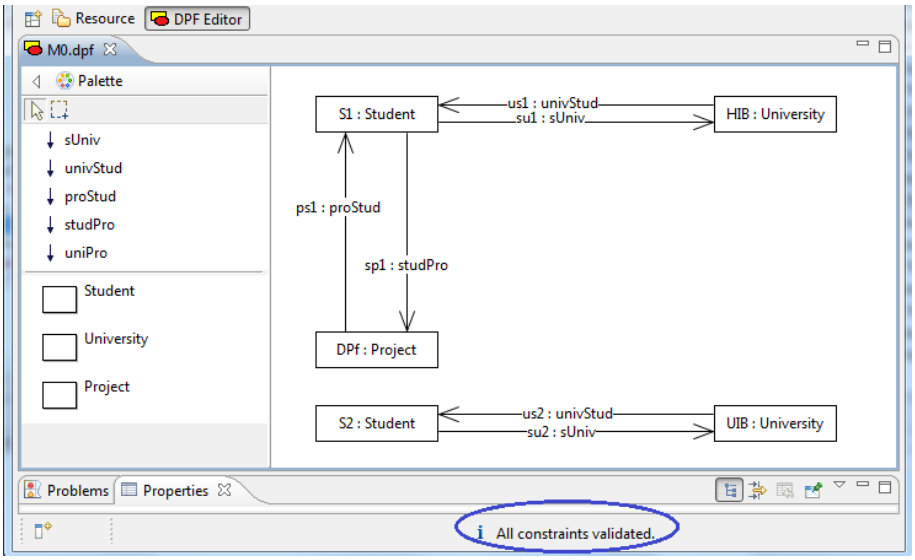


Figure 3.6: The DPf editor All constraint validated

If the well typed graph violates the constraint's semantics, the Editor will display a message on the status bar stating that the "`Validation Failed`" as shown in figure 3.7.

Figure 3.7 shows an instance at level M0 which has failed the validation. The reasons for validation failure are:

1. There is no arrow between a `Student` named `S1` and a `Project` named DPF which violates the `inverse` constraint.
2. There is no arrow between a `Student` named `S2` and a `University` named `UIB/HIB` which violates the `multiplicity` constraint stating that a `Student` must have an arrow to least one `University`.

The DPF Model Editor informs the user by showing an error message ' "`Validation Failed`" on the lower left corner of the status bar as shown in figure 3.7 (see blue oval on the left hand corner).

3.4.1 Problems with the DPF validation mechanism

This section explains some of the problems found in the DPF Workbench's predefined Java validator and the Signature Editor.

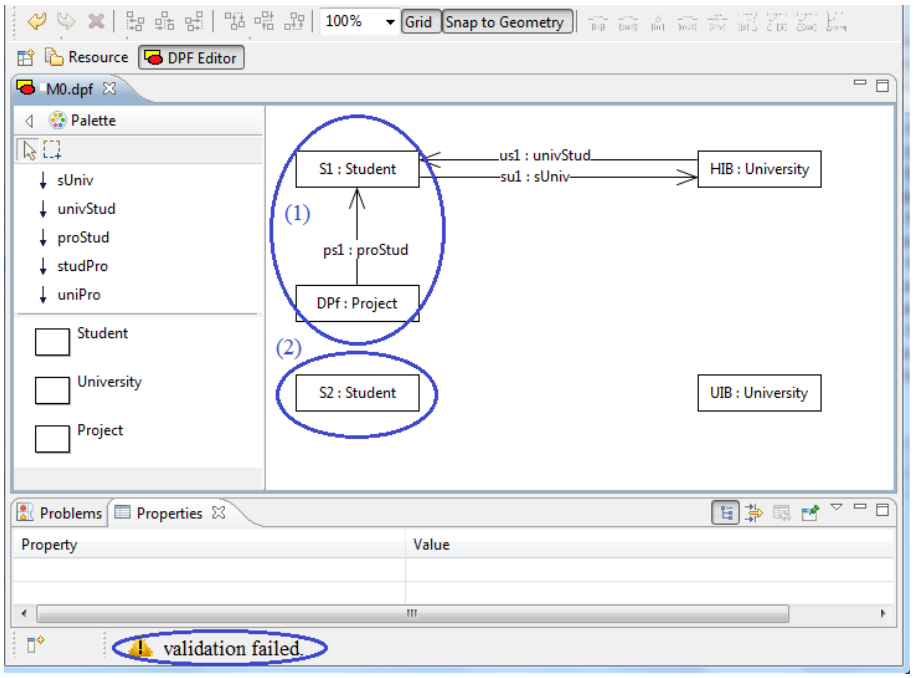


Figure 3.7: The DPf editor Validation Failure Error Message

3.4.1.1 Problems with the predefined DPF Signature

Initially, the DPF Model Editor contains a predefined signature that can be used to add constraints on DPF based models. It was constructed only for prototyping purposes. The DPF predefined signature was hard coded with a semantic validator for every predicate using the java programming language. It is not `general-purpose` for DPF semantics and there exists a lack of functionality for implementing new constraints.

A major problem was that when an instance of a model is created, upon any change in the model element, i.e. when a node/arrow is added or deleted from the model, the DPF Editor's validation mechanism is triggered. It checks the complete model on every change, which leads to be time consuming.

Another problem was that the validation failure message is only shown on the status bar of the DPF Model Editor window without any information about which predicate has violated and which model elements are involved in the violation of constraint.

A general solution to define a semantic validator for new predicates would be an obvious extension to the DPF Model Editor. The DPF Workbench extends the DPF Model Editor with the Signature Editor to implement new predicates. It was implemented by PHD student Xialiang Wang.

The further detail on the Signature Editor is presented in section 2.6.3 and in Appendix A.

3.4.1.2 Problems with DPF Workbench Signature Editor

The Signature Editor is an obvious extension to the DPF project. It allows the DPF Workbench users to create new predicates. It makes the DPF Model Editor more dynamic. It can be used to define the arity of the predicates, a graphical icon that illustrates the predicates in the DPF Workbench toolbar and the semantics of the predicates. The Signature Editor only supports a semantic validator for a predicate written in the java programming language. Figure 3.8 shows how the arity and icon of an injective predicate are defined.

Figure 3.9 shows how the semantics of the `[inj]` predicate is defined with use of a Java validator.

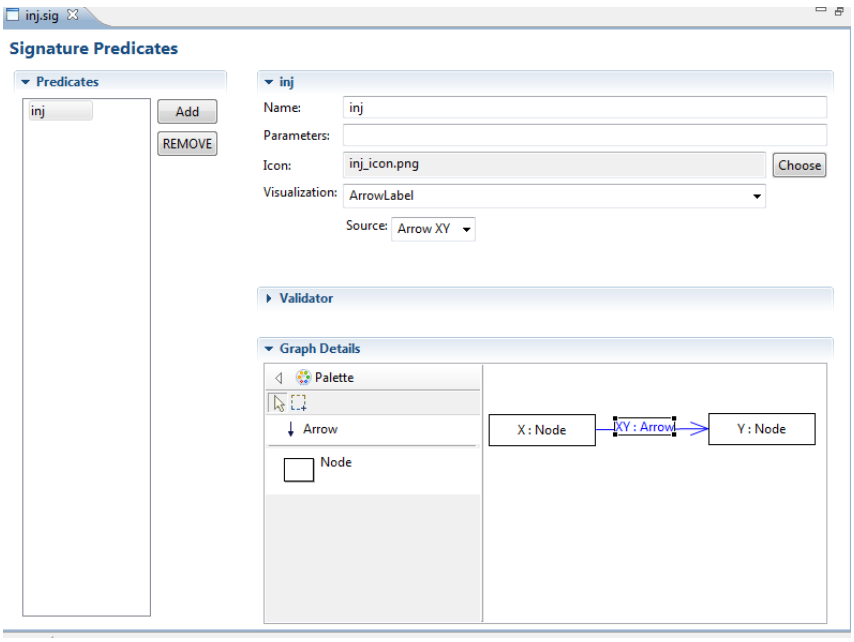


Figure 3.8: Definition of the arity and the graphical icon of the [inj] predicate

When this work started the semantic of predicates could only be defined using java. To make new predicates using the Signature Editor, a user needs to be a java programmer. It does not support writing of semantic validator in Object Constraint Language (OCL), which is a more precise, and an efficient way to write a validator for the predicates. However, with the implementation of an OCL validator, the semantic of predicates can be defined by writing OCL expressions.

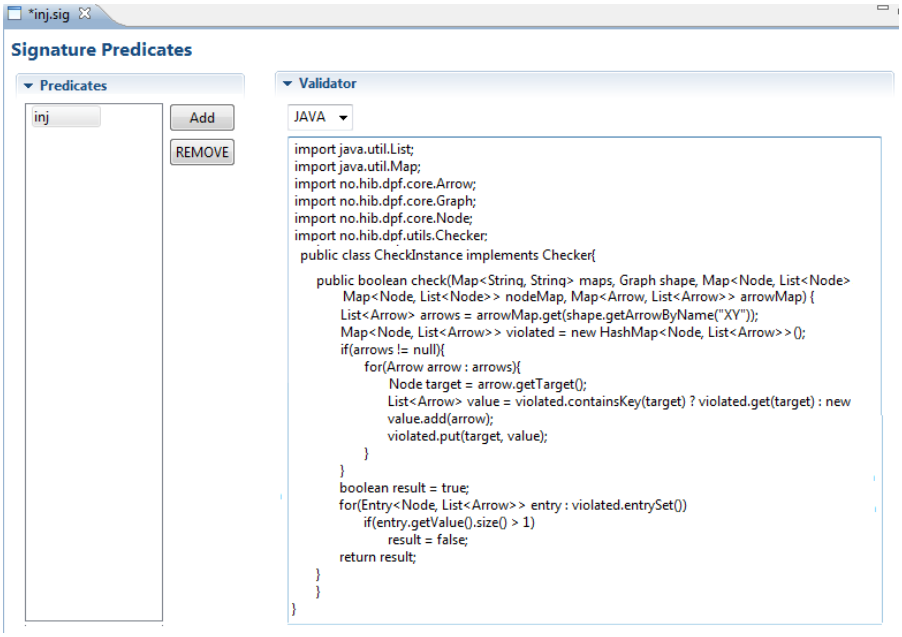


Figure 3.9: Definition of the arity and the graphical icon of the [inj] predicate

3.5 Summary

This chapter explains why we need additional constraints to the UML models due to the reason that not all the aspects of the software system can be expressed diagrammatically. For this reason usually OCL is used, which is a textual constraint language. By adding textual language to the pure modeling language leads to ambiguity. To reduce this ambiguity, different approaches to the visualization of OCL i.e. `VisualOCL` and `Constraint Diagrams` were introduced.

At the end of the chapter we discussed the DPF Workbench's semantics validator and identified some of the problems lies in the DPF validation mechanism which are the basis for this thesis.

Chapter 4

Design, Development and Solution

This chapter gives in-depth description of what are the main goals for this thesis and how these goals have been achieved. It also discusses how the OCL checker has been integrated with the DPF Workbench tool. It also explains some of the other frameworks that gave benefits during the development process and enhances the DPF Model Editor's validation error feedback. In section 4.4, a solution will be presented that explains how main goals have been achieved throughout the development.

4.1 Development Process

This section describes about which software development methodology we have chosen for this thesis and what coding standards and tools have used during the development process.

4.1.1 Development Methodology

We have chosen `Agile` development methodologies [3] for this project. Agile technologies are based on iterative and incremental development. Some of the important concepts of Agile methodologies and how these concepts apply in this project, are given below.

Fast cycle and Frequent delivery

One of the important concepts of Agile is, it schedules many releases of a software product with short time span between two to three weeks instead of delivering the complete software at the end of the development process. During the development phase of this project, we have

a biweekly project review meeting, and the aim was to deliver working software every two weeks in the meeting for review. The meeting gave a chance to discuss what had been achieved, and assign work for the next iteration.

Pair Programming

Pair Programming is a style of programming in which two programmers work side by side at one computer and program alternatively. As part of a Master degree a student has to work individually on a project, and everyone has been assigned a different problem, so there is no chance of pair programming.

Refactoring

Refactoring is the reconstruction of a software to remove duplicate code and make it simple and add flexibility to it without changing its behaviour. This concept is followed extensively during the coding phase of this project.

Working software over comprehensive documentation

In Agile concepts, presenting working software is more useful than presenting documents. We followed this tactic by presenting working software every two weeks.

Test Driven development

Agile development is a test-driven development, tests are written by developers/customer before and during the coding. In this project, we only practice of writing test after coding has been done.

The agile method chosen for this project is `eXtreme Programming (XP)`, but following it completely is not possible for this project. XP, originally described by Kent Beck [5], has emerged as one of the most popular agile methodologies. The main reason for choosing XP over other Agile methodologies is that, it is taught in MOD251 course at Bergen University College.

Some of the important practices of XP, which we have followed in this project are:

Short Iterations

As described earlier the DPF project arranges a meeting every two weeks for the review, for the delivery of working software and for assigning work for the next iteration.

Continuous Integration

In DPF, for continuous integration of software product, `subversion` is used to `checkin` and `checkout` project to/from the assigned repository on a server at Bergen University College. Whenever a task has been

completed and reviewed in a biweekly meeting, we update the repository so that the other members can keep up with the new version of a tool.

Pair Programming
Due to the assignment of one problem per member, it is not possible to practice pair programming in this project.

4.1.2 Coding Conventions

Naming conventions are the set of defined rules for choosing character sequences for variables, types and functions, etc. in the source code and documentation. The reason for using the coding conventions are to reduce the effort that is needed to read and understand the source code. The use of standard coding conventions also enhances the appearance of the source code. An important advantage of using coding convention is that code look familiar. It also supports collective ownership and it helps new developers to read and understand the code easily. The coding convention provides code consistency and avoids the conflicts for naming convention.

At the start of the DPF project, the developer has fixed a set of naming rules. The coding standard used in this project are the Eclipse Naming conventions [14]. The Eclipse naming convention defines how specific element (such as method name, variable name, package name and plug-ins) should be named. The reason for following this naming convention is that it was already fixed for the DPF project, and it is also used as a standard in the java ecosystem.

The naming convention used in this project for different elements are:

Projects: no.hib.dpf.signature.examples

Class: public class SemanticValidator

Method/function: private String getContextVariable()

Variables: String contextVar

Constants: public static final org.eclipse.swt.graphics.Image IMAGE

4.1.3 Tools

Different tools have been used during the development of this project. As the DPF Workbench is an Eclipse based project; we have used the `Indigo` version of Eclipse Modelling tool [44] because when we have started work

it was the latest version. The new version of eclipse is `Juno` [43] and was launched when we were half way in our project, and it was not worth shifting it to the `Juno` version.

An important tool which is available in eclipse is "Eclipse OCL" [16]. Eclipse OCL is an implementation of Object Constraint Language(OCL). It provides the API for parsing and evaluating OCL constraints and queries on the UML models.

The additional "OCL Examples and Editors" [16] component provides interactive support for OCL. It provides a console for interactive evaluation of OCL expression on models, an `Impact Analyzer` to support analysis and optimised re-evaluation, and an `Xtext editor` for the OCL expressions.

4.2 Problem Description

As discussed earlier in section 3.4.1, the current DPF Model Editor validator is hard coded in java. The Signature Editor only supports predicate's semantics in Java programming language, it can be extended with an OCL Validator. The OCL validator aims to make the DPF validation process more dynamic and efficient.

The first goal of this thesis is to extend the DPF Workbench with an Object Constraint Language (OCL) validator. The OCL validator will facilitate the DPF Workbench users, when defining domain-specific constraints, it could be possible to define the constraint's semantics in OCL syntax instead of writing a Java validator.

Currently, the DPF validation mechanism checks a complete DPF model every time a model element is changed e.g. a node or an arrow has been added or removed. This mechanism is not an efficient way of doing validation and affects the efficiency of the DPF Workbench and is time consuming. The second goal of the thesis is to change this mechanism to the incremental way which instead of validating the whole model every time, will find only the elements which are related to a constraint which is affected by the change in a model element. This will make the validation process more efficient and less time consuming.

The third and the last goal of the thesis is to change the way validation errors are reported to the user. As discussed earlier in section 3.4 when the validation of a model occurs it will only display a message on the left side of the status bar stating that "All Constraints Validated" if validation is successful otherwise displays "Validation failed" message. It does not state that which predicate has failed validation and which model elements are involved in validation failure. This project changes the validation feedback mechanism and displays an error image on the model elements which

caused the failure, directly on the DPF Model Editor window. The detail of the error messages will be shown in the `Problem View`.

4.3 Eclipse OCL

This section explains the OCL tool, `Eclipse OCL` [15] which is used to implement OCL Validator in the DPF Workbench. More detailed discussion on OCL as a language can be found in section 2.4.

Eclipse OCL is an implementation of Object Constraint Language (OCL). Eclipse OCL provides an interpreter and a parser for OCL constraints and expressions on any metamodel, which is based on Eclipse Modelling Framework (EMF) [13]. OCL can be used to parse constraints to UML and Ecore models and can be evaluated on instances generated from these models.

Figure 4.1 shows a part of the architecture of Eclipse OCL. An individual OCL expression or a complete OCL document may be parsed to produce a Concrete Syntax Tree (CST), which is then processed to produce `Abstract Syntax Tree (AST)`. The validator checks the well-formedness of AST, and the AST is used by the evaluator to evaluate OCL expression on the models.

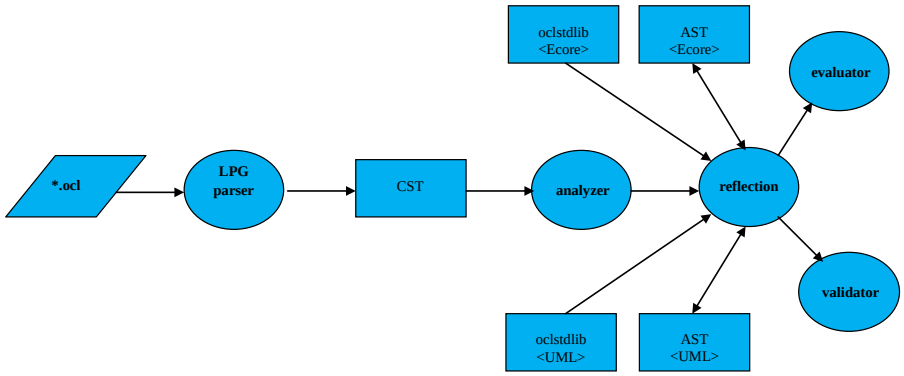


Figure 4.1: Eclipse OCL Architecture

4.3.1 Eclipse OCL CORE Component

The core OCL component provides a lot of capabilities to support the OCL integration.

- It provides APIs for evaluating and parsing of OCL constraints.
- It provides UML/Ecore implementation of the OCL abstract syntax model.

- It also supports serialization of parsed OCL constraints.
- It provides the extensibility API for the customization of parsing and evaluation environment, which is used by the parser.
- It also provides a `Visitor API` for analyzing/transforming the AST model of OCL expressions.

4.3.1.1 APIs for Parsing Constraints and Queries

Eclipse OCL provides two APIs to parse the OCL constraints and query expressions, the `OCL` and the `OCLHelper` shown in figure 4.2.

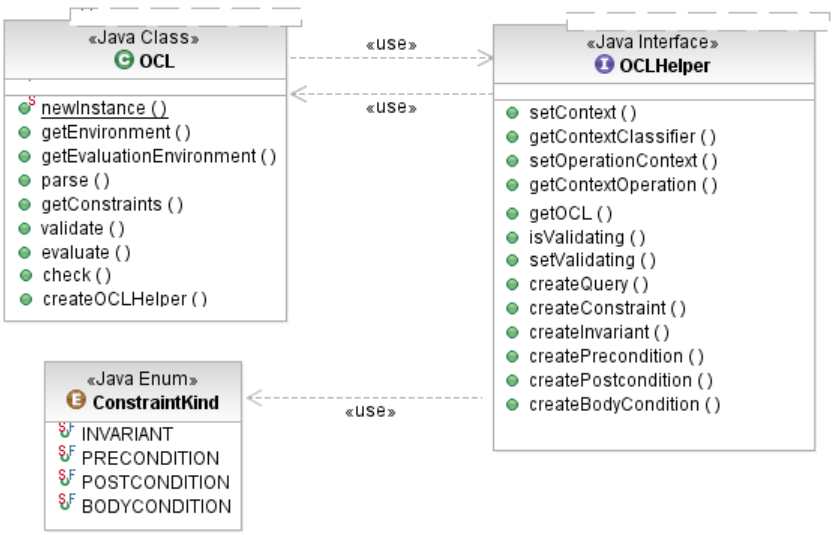


Figure 4.2: The OCL Class and the OCLHelper Class APIs

OCL Class

The `OCL` Class is a main entry point into the parsing API. It also implements the parsing of OCL documents, for example, from an external text file. It is presented in `org.eclipse.ocl` package. It is a generic type; its type parameter represents various meta-classes basically for the UML/MOF specifications. The instance of `OCL` Class can be created by using the static factory methods. Listing 4.1 shows an instance of `OCL` Class for ecore environment.

```
OCL ocl = OCL.newInstance(EcoreEnvironmentFactory.INSTANCE);
```

Listing 4.1: Instance Of OCL CLass for Ecore Environment

OCLHelper Interface

The `OCLHelper` interface provides a mechanism for parsing OCL constraints and query expressions, which are embedded in models, such as Ecore or UML models. It is presented in `org.eclipse ocl.helper` package. The OCL Class instance can be used to create an object of the `OCLHelper` Class which is used to parse OCL constraints. Listing 4.2 shows how an instance of `OCL Helper` Class can be created.

```
OCLHelper<EClassifier, EOperation, EStructuralFeature,
      Constraint> helper = ocl.createOCLHelper();
```

Listing 4.2: Instance Of OCLHelper CLass

OCL context is required on which an OCL constraint has been evaluated, e.g. `setContext()` is a method of the `OCLHelper` Class that is used to create an appropriate environment for the evaluation of OCL expressions.

The parsing process of OCL is wrapped in the creation of a constraint function. The `OCLHelper` provides an APIs for the creation of each type of OCL function expressions, e.g `invariant, pre, post, and query, etc.` including:

- **createQuery():** It parses a query expression.
- **createConstraint():** It parses a constraint of a given constraint kind such as `inv, pre, post`.
- **createInvariant():** It parses an invariant constraint.
- **createPrecondition():** It parses a pre-condition constraint.
- **createPostcondition():** It parses a post-condition constraint.

All these methods takes parameter expressions as `String`.

After parsing of constraints, expression has been validated, by invoking the validation visitor. The result of parsing a constraint returns the corresponding constraint object if it succeeds otherwise it throws ParserException. The result of parsing a query returns a `OCLExpression` which is instance of the Abstract Syntax Tree (AST). Once the constraint has been defined, parsed and validated, they can be evaluated on model objects in which it defines in order to get the results. All those tasks are provided by the `OCL Class`.

At the end, an OCL expression is represented by the AST, and can be serialized/ de-serialized to/from XMI format when required.

4.3.2 OCL Examples Component

The `OCL Examples component` provides `Interactive OCL Console`, which can be used for the evaluation of interactive OCL constraints on UML/Ecore based models and also for querying the models.

The `Eclipse OCL` indigo version provides an `Impact Analyzer` which supports incrementally re-evaluation of OCL Constraint. Since an OCL expression navigates from one resource to another resource freely, so changes to a model element in one resource can easily affect the model elements in other resource. In order to find all invalidated constraints, after a change it would become necessary to re-evaluate all constraints regardless of their resource. This would affect performance when done manually. An `Impact Analyzer` re-evaluates all the OCL expressions after a change has been made in a resource.

4.3.3 OCL Examples and Editors Component

The `OCL Example and Editor` component is an IDE that can be downloaded from the install modelling component option within eclipse. This additional component provides interactive support for OCL. It provides

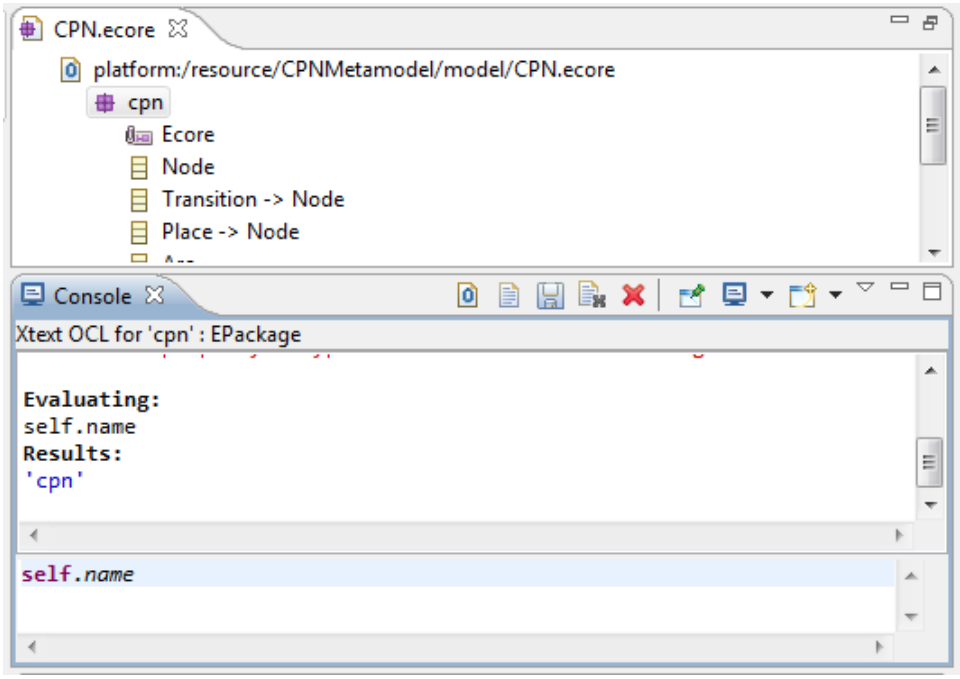


Figure 4.3: Interactive Xtext Console

- The "`Interactive OCL`" console which is especially useful for debugging OCL constraints on models. It also has the feature of content assistance.
- The "`Interactive Xtext OCL`" console is used to evaluate an OCL expression on a model. Figure 4.3 shows an `Interactive Xtext OCL` console that is used to evaluate OCL expressions on ecore.

4.4 DPF OCL Checker

This section outlined the design and development of OCL Checker in the DPF Workbench and which tools and plug-in dependencies have been used. At the end, it also describes how the DPF Workbench validation mechanism has been changed and how the validation feedback of the DPF Workbench has been changed.

4.4.1 The OCL Checker

This section describes how the OCL checker has been implemented in the DPF Workbench, which tool we have chosen and which plug-ins we have used for the implementation of this extension.

4.4.1.1 Tool Selection

As described in section 3.4.1 there exists some problems in the DPF Workbench's approach towards validation of constraints such as defining a semantic validator, i.e. the users can only define the semantics of predicates using Java. There is no OCL support, so it is required to implement an OCL checker in the DPF Workbench, which will allow users to define the semantics of predicates by OCL.

The DPF Workbench is available as an add-on in Eclipse. The tool that is used to implement OCL checker in the DPF Workbench is `Eclipse OCL` [16].

4.4.1.2 Plugin Dependencies

The plug-ins that are necessary to support parsing and evaluation of `OCL` bounds directly to the `EMF` Ecore meta-model. Since the metamodel of the DPF is an Ecore, so the plug-ins that we used to support and implement OCL checker in the DPF Workbench are:

- `org.eclipse.ocl`

- `org.eclipse.ocl.helper`
- `org.eclipse.ocl.ecore`

4.4.1.3 Architecture

The "DPF OCL Checker" is an extension to the "DPF Signature Editor" component. It is used both by the "DPF Signature Editor" component for defining new constraints and by the "DPF Model Editor" component for validating constraints. The core functionality of the "DPF OCL Checker" component is implemented in the "DPF Core" component, basically it extends the features provided by the "DPF Core" component by an OCL validator. It is built by using the `EclipseOCL` [15], which is an implementation of the Object Constraint Language (OCL) [16]. Figure 4.4 show an overview of the architecture of the "DPF OCL Checker". Detailed component architecture of the DPF Workbench has been discussed in section 2.6 and in [34].

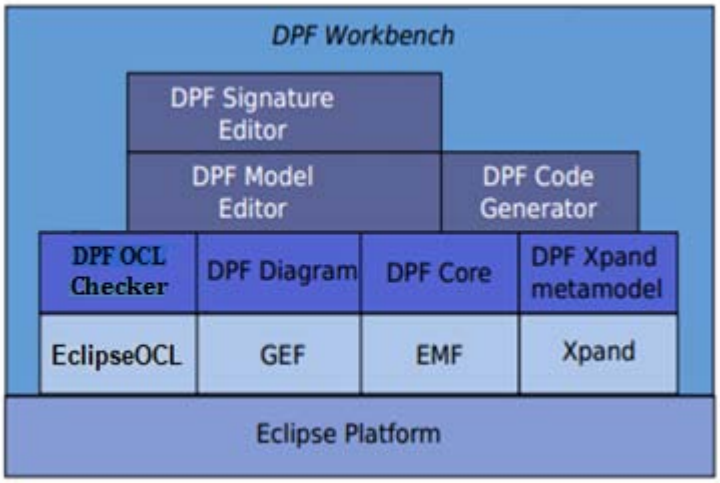


Figure 4.4: An overview of the architecture of the DPF OCL Checker

4.4.1.4 Implementation

When the OCL constraints are added to a model, it is not possible to evaluate these OCL constraints on model level, actual evaluation starts when an instance of a model is created. Figure 4.5 shows how the OCL checker in the DPF Workbench is working.

1. After an instance of a model is created; all the constraints which have been specified in a model are checked.

2. For each predicate, the tool does the following steps:
   (a) Each predicate has an arity which is typed-by `Graph`, the tool finds all the model elements, which are related to a predicate. All the related elements to a predicate give a sub graph containing a list of `Nodes` and a list of `Arrows`.
   (b) Each predicate has a semantics validator, which is an `Extended OCL Expression`.
   (c) An `Extended OCL Expression` is then changed into the `DPF OCL Expression` so that it can be evaluated on the `DPF` model elements, i.e. `Nodes` and `Arrows`.
   (d) A `context` variable has been specified, which is always the type of `Node` and the `DPF OCL Expression` is evaluated on the instances of the `context` type.
   (e) An `invariant` is then created from the `DPF OCL Expression`.
   (f) Then at the end, instances of a `context` type and an `invariant`, are sent to the OCL checker for parsing and evaluation.
   (g) The OCL checker always returns a `Boolean`.

4.4.2 Visualization of OCL

This section describes our approach to the visualization of OCL, conceptually and with the help of an example.

As described earlier, the DPF Workbench extends the DPF Model Editor with the functionality to create new domain-specific predicates. In order to define new predicates by using the Signature Editor (see section 2.6.3), a user specifies a name, parameters (if any), icon, visualization, semantic validator (Java/OCL) and a shape graph for the predicate. The Signature Editor is shown in figure 4.6. Further detail on how to define new predicates using the DPF Workbench's Signature Editor can be found in appendix A. The newly defined signature can be used in the DPF Model Editor to add constraint to a model instead of using a predefined java signature.

When a new predicate is created using the DPF Workbench Signature Editor, one of the important things which users have to specify, is to assign a name to a predicate. The name of the newly defined predicate (an OCL Constraint) is visualized or will appear in a set of square brackets as shown in figure 4.7, when it is added to the model elements in the DPF Model Editor window.

4.4.3 Relation to the Default DPF metamodel

In DPF, the default metamodel contains two elements, a `Node` and an `Arrow` self- referencing the node as shown in figure 5.4. This default metamodel

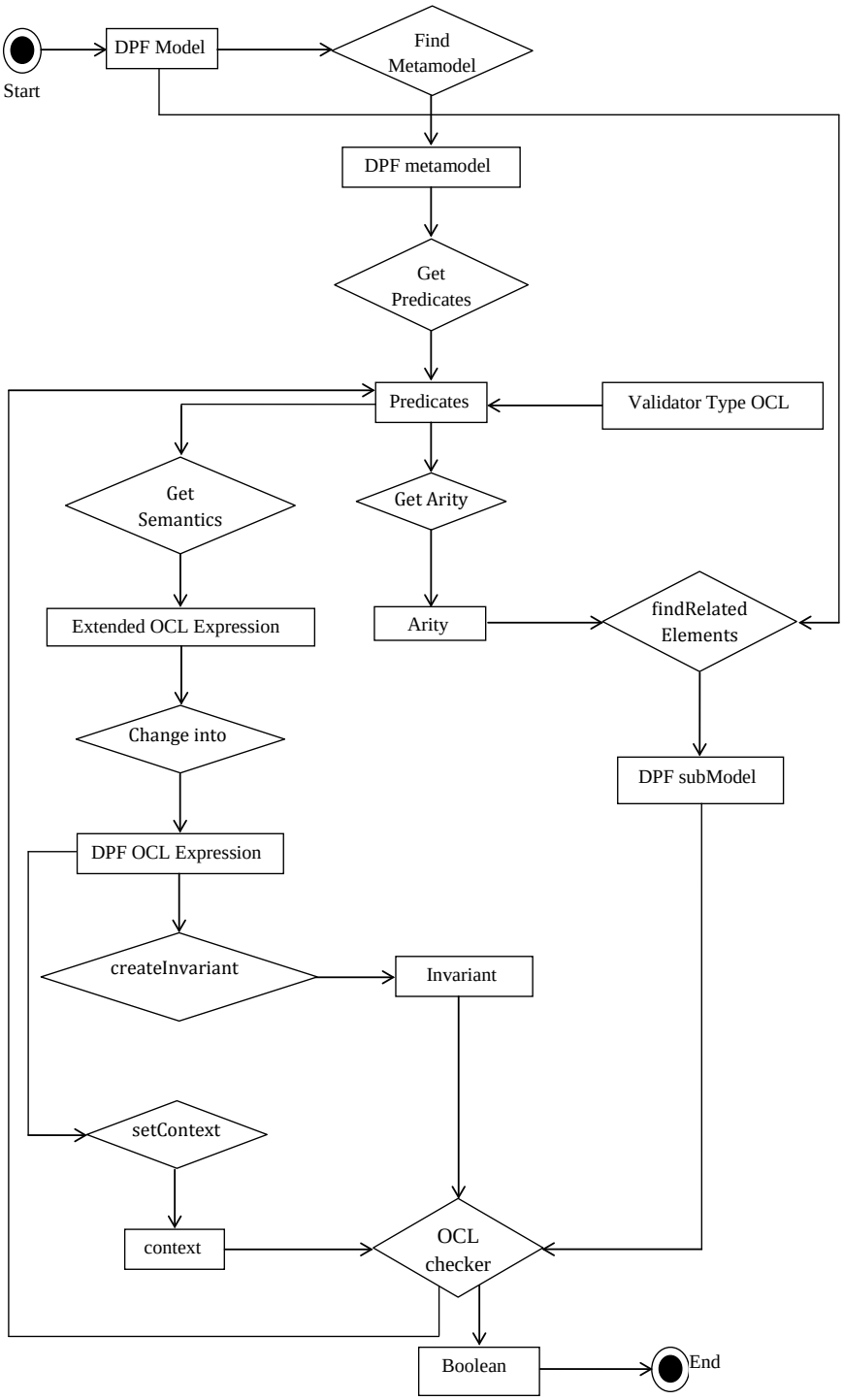


Figure 4.5: Diagram to show the working of OCL Validator in the DPF Workbench

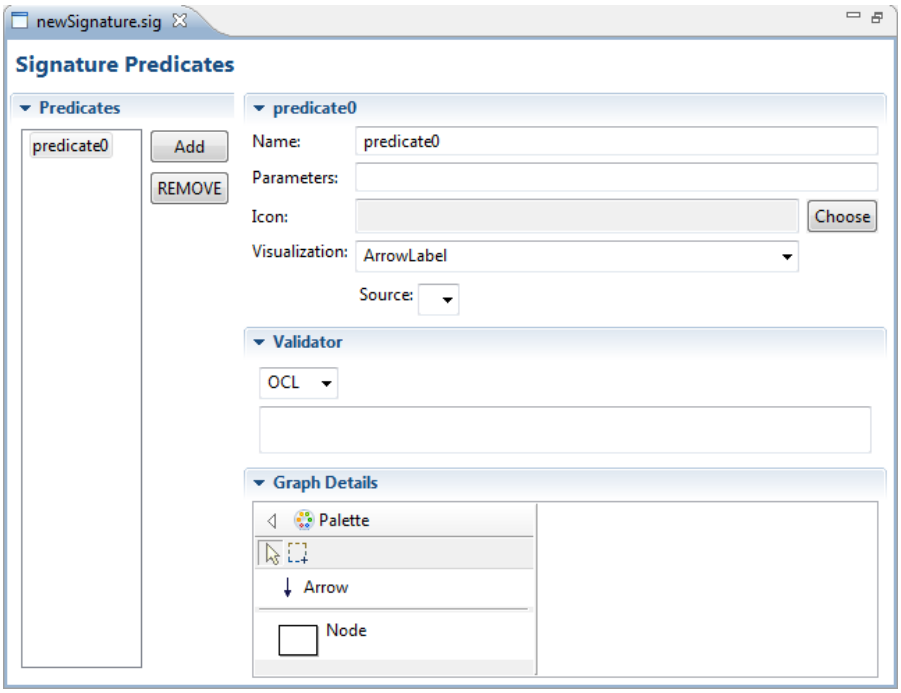


Figure 4.6: The DPF Workbench Signature Editor

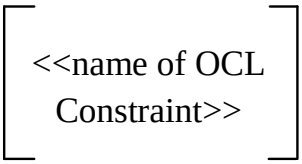


Figure 4.7: Representation of OCL Constraint in the DPF Model Editor

is the type graph of every model, which is created by using the DPF Model Editor.

Listing 4.3 shows a semantic of a predicate in set theory stating that any two instances of `Node X` cannot have an `Arrow` of type `f` to the same `Node` of type `Y`.

```
∀x1,x2 ∈ X: f(x1) = f(x2) implies x1 = x2
```

Listing 4.3: Semantic of predicate in set theory

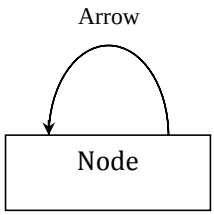


Figure 4.8: The graph of the default metamodel, consisting of a single node and a single arrow self-referencing the node.

Defining a Shape(Graph)

In order to define a predicate, whose semantic is equivalent to the semantic given in listing 4.3, we first have to define an arity (shape graph) of the predicate. Since the semantics defined in listing 4.3 involved two `Nodes` with an `Arrow` in between them, so we define an arity graph given in figure 4.9, which has two `Nodes` X and Y and an `Arrow` XY in between them.

Figure 4.9: Arity (shape graph) of a Predicate

Specifying the context

The DPF `context` can be specified like in OCL.

In the DPF default metamodel, there are two types of model elements, i.e. a `Node` and an `Arrow`, so the `context` of an OCL constraint can be either of type `Node` or `Arrow`. But in the context of OCL `invariants`, the `context` of an OCL constraint is of type `Node`. We have set the `context` for this predicate to `Node` of type Y as shown in listing 4.4.

```
context Y:  self. #XY#in -> forAll(a:Arrow,b:Arrow | a<>b
            and  a.target = b.target
         implies a.source = b.source)
```

Listing 4.4: Defining a semantic validator in DPF

self

The keyword `self` is used like in OCL. It always refers to an instance of the type of the `context`.

#XY#in

The keyword `#XY#in` is used to specify the name of an `Arrow` from the arity (shape graph) of the predicate, which should be affected by the constraint. Since `XY` is the type of incoming `Arrow` so we specify `in` followed by the name of the `Arrow #XY#`. Before parsing, this keyword is replaced by this string shown in listing below.

```
incomings->select (x:Arrow| x.typeArrow.name ='XY')
```

Listing 4.5: Replacement String for constraining incoming Arrow type

For example, to constrain the outgoings `Arrow` type, we have to specify the keyword `#XY#out` instead of `#XY#in`, if the specified `context` has an outgoing `Arrow XY`.

```
outgoings->select (x:Arrow| x.typeArrow.name ='XY')
```

Listing 4.6: Replacement String for constraining outgoing Arrow type

#p#min and #p#max

The keywords `#p#min and #p#max` are used to specify parameters in the `Extended OCL Expression`. These keywords would then be replaced by the minimum and maximum values as specified by the user in the parameter section when defining a new predicate using the Signature Editor. For example, if the user specified "min:0;max:5" in the parameter section, then the tool will replace the `Extended OCL Expression` as shown in listing 4.8.

```
context X:
          self.#XY#out->size()>= #p#min  and
          self.#XY#out->size()<= #p#max
```

Listing 4.7: Usage of keywords #p#min and #p#max

```
context X:
           self.#XY#out->size()>= 0  and
           self.#XY#out->size()<= 5
```

Listing 4.8: Extended OCL Expression after replacement of keywords #p#min and #p#max

invariant

All the constraints in the context of the DPF are always `invariant`, because we always want these constraints to be fulfilled by the instances of the `context` type. Listing 4.9 shows a semantic validator of the predicate defined in OCL after replacing the keyword `#XY#in`. This semantic validator is equivalent to the semantic defined in set theory in the listing 4.3.

```
context X:
         self.incomings->select(x:Arrow|x.typeArrow.name='XY')
             ->forAll(a : Arrow, b : Arrow | a<>b
             and a.target=b.target
             implies a.source=b.source)
```

Listing 4.9: An OCL invariant before parsing

4.4.4 Incremental Validation of constraints

The main goal of model validation is to achieve a good design before the actual implementation starts. There have been many different approaches to validation, the traditional approach of model validation always verifies the complete consistency of a model on every change in the model elements.

As described earlier in section 3.4.1, the DPF Workbench approach to the validation of constraints was classical, i.e it validates the whole model every time a model element is changed i.e a node/arrow is added or removed from the model. This increases the checking cost and reduces the efficiency of the tool. In this section, we introduce an incremental validation approach in the DPF Workbench, which will reduce the checking cost by just verifying the part of a model that is involved in the modification.

In order to change the DPF Workbench validation approach to incremental validation, we need to design an algorithm that will find the related model elements and validate them against the constraints specified in the metamodel. In an incremental approach, there are two issues.

Which constraints needs to be checked?

The solution to the first question is to consider that for any modification, a series of constraints must be verified. In this matter, we only validate those constraints from a (meta)model in which the modified model element is involved when a specific action occurs, and not validates the ones that are not involved in this action. When a constraint is defined in a model then an instance of that model needs to fulfil this constraint. So if there is no constraint defined in the (meta)model, then no validation checks take place.

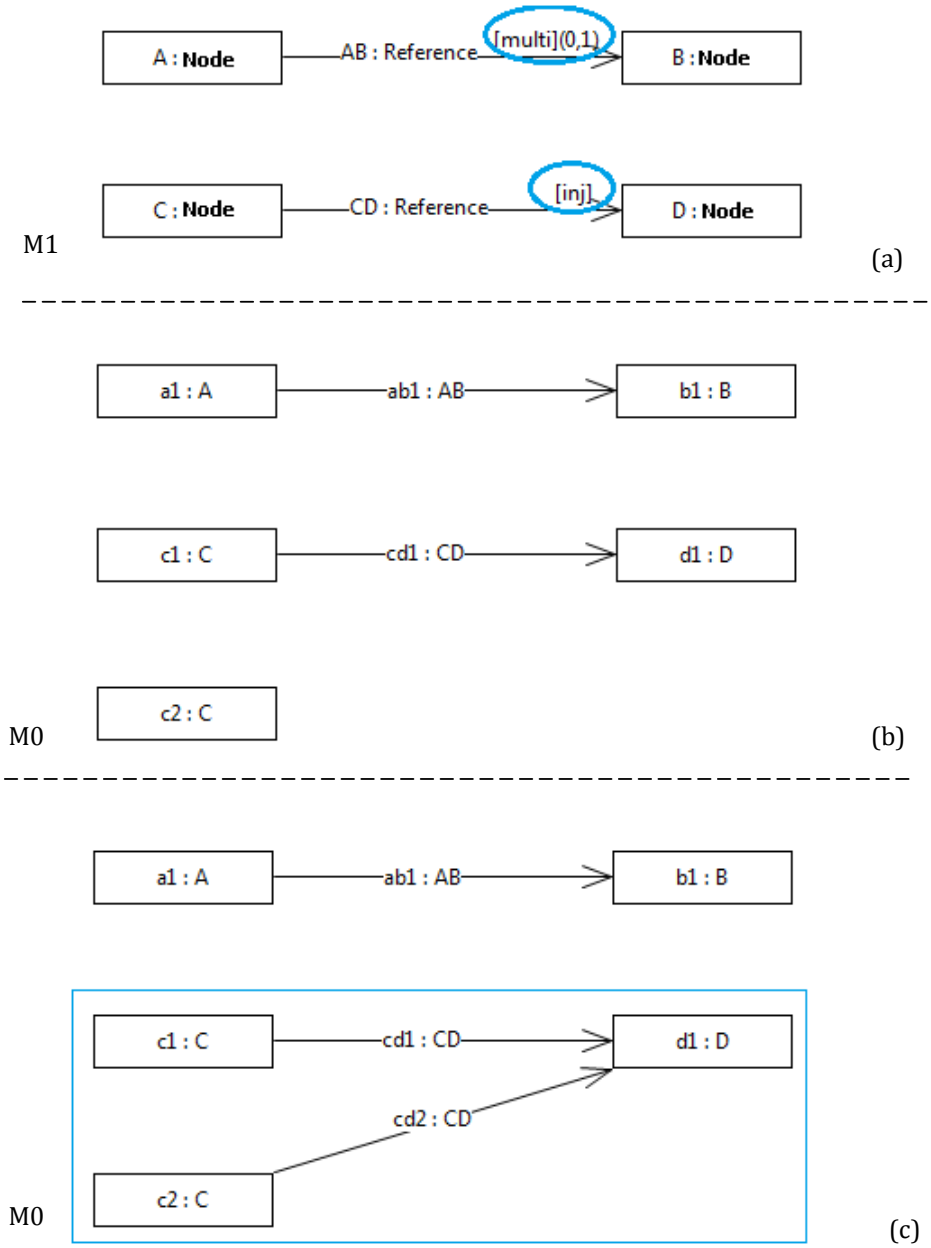


Figure 4.10: A DPF model with two Constraints

For example figure 4.10 shows a model with two constraints:

Multiplicity
This constraint is between nodes `A` and `B` on an arrow `AB`.

Injectivity
This constraint is between node `C` and `D` on arrow `CD`.

When an instance of this model is created as shown in figure 4.10 (b), if any change occurs in the instance, e.g. if we add a reference `cd2` between Classes `c2` and `d1` then the validator will only check and validate `[inj]` constraint and part of a model that is involved in this constraint as shown in a blue rectangle in figure 4.10(c). It will not check `[mult(m,n)]` constraint.

Which model elements needs to be checked?
In order to optimize the validation process, we only verify those parts of a model which are affected by the modification. Following is some of the possible types of modifications that will trigger validation.

1. When a node/arrow is added to a model.
2. When a node/arrow removed from a model.

Our aim is to find which part of a model must be checked after a modification has been made in a model. For this purpose, we have designed an algorithm which will find a part (sub graph) of a model to be checked by the semantic validator. When any modification occurs, the incremental validation algorithm runs as follows:

1. Finds model elements (a part of a model) that are associated in a predicate.
2. Send two lists of nodes and arrows to the semantic validator to be checked by the OCL/Java Validator.

In order to apply our approach, we develop a model, Figure 4.11 shows how our incremental validation algorithm works. The general algorithm is shown in listing 4.10. The meta(model) of this example is shown in figure 4.10 (a).

Adding a new Node/Arrow
When adding a new Node/Arrow, our incremental validation algorithm works as follows:

- When a node `a0` of type `A` is added it will only validate a node `a0` as shown in a blue coloured rectangle in figure 4.11 (a).
- When a node `b1` of type `B` is added it will only validate a node `b1` as shown in a blue coloured rectangle in figure 4.11 (b).

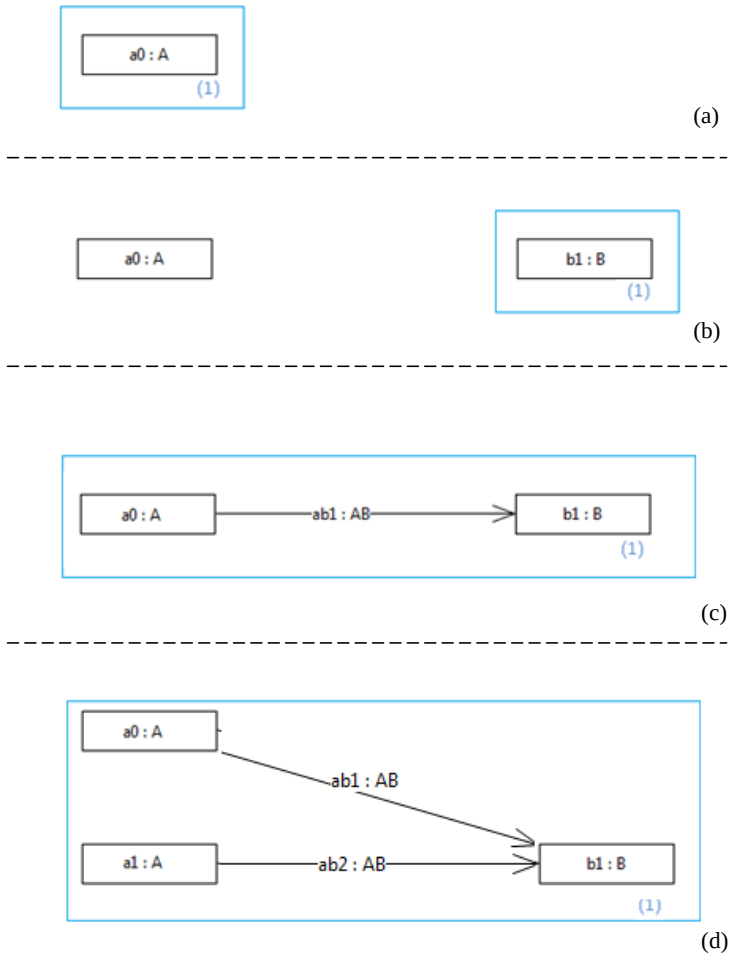


Figure 4.11: Example on how our incremental algorithm works when adding a node or arrow

- When an arrow `ab1` of a type arrow `AB` is added between nodes `a0` and node `b1` it will only validate the part shown in a blue coloured rectangle in figure 4.11 (c).
- When a new node `a1` of type `A` and an arrow `ab2` of type `AB` is added between nodes `a1` and node `b1` it will validate the graph shown in a blue coloured rectangle in figure 4.11 (d).

Removing a `Node`, an `Arrow` or both

When removing a `Node`, or an `Arrow` or both, our incremental validation algorithm works as follows:

- An instance shown in figure 4.12(a) has two nodes `a0`, `a1` of type node `A`, and one node `b1` of a type node `B`. The two arrows

ab1 and ab2 are typed by arrow AB. This is an instance of multiplicity constraint from figure 4.10(a).

- When an arrow ab2 of type AB is removed, the validator is called twice, first it validates node a1 alone, and second time it validates part of a graph shown in a blue coloured rectangle in figure 4.12 (b) marked as (1).
- When a node a0 of type A is removed, it will also delete an arrow ab1 and validator will only validate part shown in a blue coloured rectangle in figure 4.12 (c) marker as (1).
- When a node a1 of type A is removed from figure 4.12(c), no validation takes place because it was the standalone node(i.e. no incoming's or outgoing's arrows) as shown in figure 4.12 (d).

```
Input:
      a Node node, a Constraint constraint, a Graph graph,
      an empty List of Nodes nodes and
      an empty List of Arrows arrows

Output:
      add Nodes to nodes and
      add Arrows to arrows

Initialization:
      Create new ArrayList visit
      Add node to visit
Construction:
      while visit not empty
      do:
         select one element cur from  visit at position 0
         for all outgoing arrow from Node cur
         do:
            if typeArrow of arrow ∈ constraint arrow then
               get target Node of an arrow
               if target not null then
                  if arrow ∉ arrows then
                     put arrow to arrows
                  end if
                  if target ∉ nodes and target ∉ visit then
                     put target to visit
                  end if
               end if
            end if
         end for

         for all incoming arrow to Node cur
         do:
             if typeArrow of arrow ∈ constraint arrow then
                get source Node of an arrow
```

```
            if source not null then
               if arrow ∉ arrows then
                  put arrow to arrows
               end if
               if source ∉ nodes and source ∉ visit then
                  put source to visit
               end if
            end if
         end if
      end for
      put cur to nodes
      remove one element from visit at position 0
   end while
```

Listing 4.10: Algorithm for finding a constraint related model elements

4.4.5 Validation Feedback

As described earlier in section 4.2, the previous version of the DPF Workbench has lack in reporting validation errors to the user, it will only report "`Validation Failed`" message on the DPF's Model Editor status bar without specifying which constraint has failed the validation and which model elements (i.e. node or arrow) are involved in validation failure.

The DPF Workbench is an eclipse based plug-in and eclipse Workbench provides a central mechanism for managing resource and communicate problems and other information to the users, called `markers` which manages this information. The resource plug-in `org.eclipse.core.resources` provides a general mechanism in `interface IMarker` to define and manipulate markers.

A marker is like a sticky note attached to a resource. It is a general mechanism that is used to associate notes and metadata with the resources. There are some of the predefined marker types.

Problem Marker: The problem Markers are used for representing errors, warnings and other information in the `problem view`. It is defined in `org.eclipse.core.resources.problemmarker`

Task Marker: The task Markers are used for capturing user created reminders. These markers are shown in the `task view`. It is defined in `org.eclipse.core.resources.taskmmarker`.

Bookmark Marker: The bookmark markers are shown in the `bookmark view`. It is used for marking location in resources that the user can jump to. It is defined in `org.eclipse.core.resources.bookmark`.

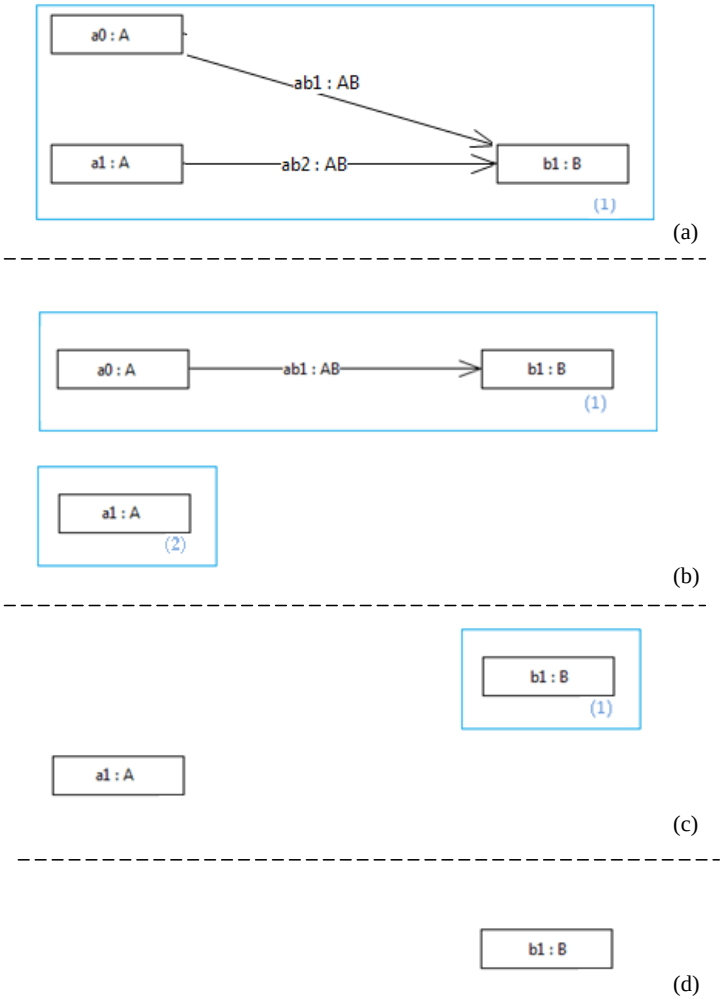


Figure 4.12: Example on how our incremental algorithm works when removing a node, an arrow or both

The platform resource `API` provides methods for marker's creation, deletion, setting their values, and to extend the platform with new marker types. Markers are managed by the platform. The resource plug-in provides an extension point `org.eclipse.core.resources.markers` in to which different marker types extension plug-ins can be installed. New marker types are derived from the existing marker types. The plug-in developer defines their own marker types by configuring extensions in the plug-in manifest.

To implement marker's extension in the DPF Workbench, we introduce a new marker type "validationmarker" by configuring an extension point `org.eclipse core.resources.markers` in the plug-in manifest of `no.hib.dpf.editor` project as shown in figure 4.13. We want our marker to be appear in the `Problem view`, so we specify `org.eclipse.core.problemmarker` as a super type of the "validationmarker" marker type. We don't want our markers to persist across multiple sessions, so we set persistence to be false, and we define a new attribute named "constraint" which we set every time we create a new marker.

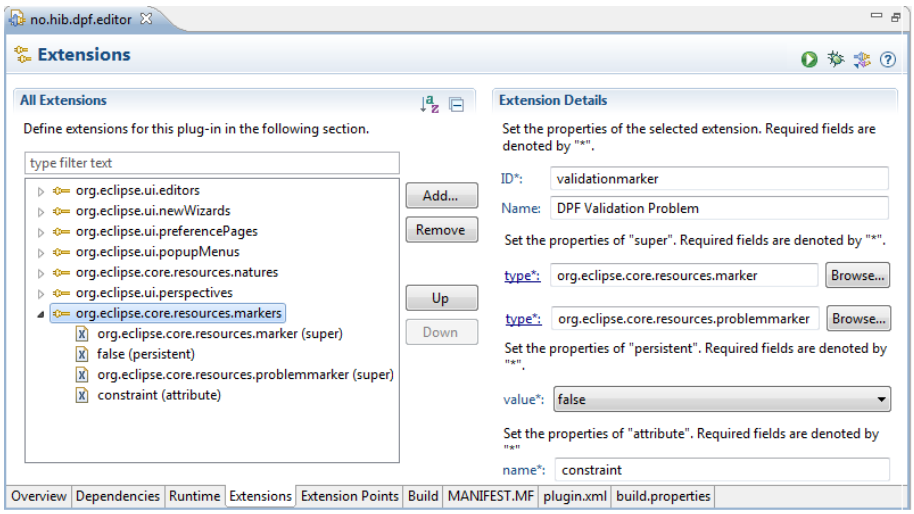


Figure 4.13: The New Extension wizard showing the markers extension point selected.

Listing 4.11 shows an `XML` form of an extension which we configure for the DPF Workbench.

```
<extension
        id="validationmarker"

        name="DPF␣Validation␣Problem"
        point="org.eclipse.core.resources.markers">

     <super
           type="org.eclipse.core.resources.marker">
     </super>

     <persistent
           value="false">
     </persistent>

     <super
```

```
        type="org.eclipse.core.resources.problemmarker">
    </super>

    <attribute
        name="constraint">
    </attribute>
    </extension>
```

Listing 4.11: Declaration of new marker extension

In addition to the validation errors shown in the Problem view as shown in figure 4.14, we want errors to be directly shown on the Model Editor window, eclipse provides a mechanism for this, `Annotations` which will underline the erroneous part in the editor. But the problem with annotations is that they work for the textual editor. Since the DPF model editor is a graphical editor, it is not possible to use annotations, so we adopt a different approach to show validation errors directly on the DPF Model Editor.

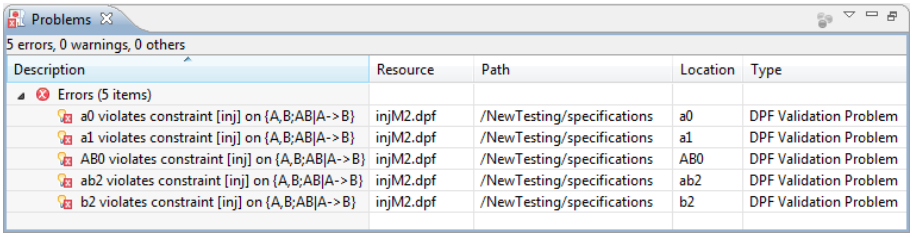


Figure 4.14: Validation Errors in the Problem View

The approach we adapted for the DPF Workbench model elements is:

- When a node is involved in validation failure, a small error image will appear in the top left side within the node figure as shown in figure 4.15.
- When an arrow is involved in validation failure, the colour of that arrow will change to the colour which the DPF Workbench user sets in the DPF preference page as shown in figure 4.16.

4.5 Summary

In this chapter, the development process, problem description and the complete solution for an OCL Checker in the DPF Workbench has been described. It described which tool we choose for the development process,

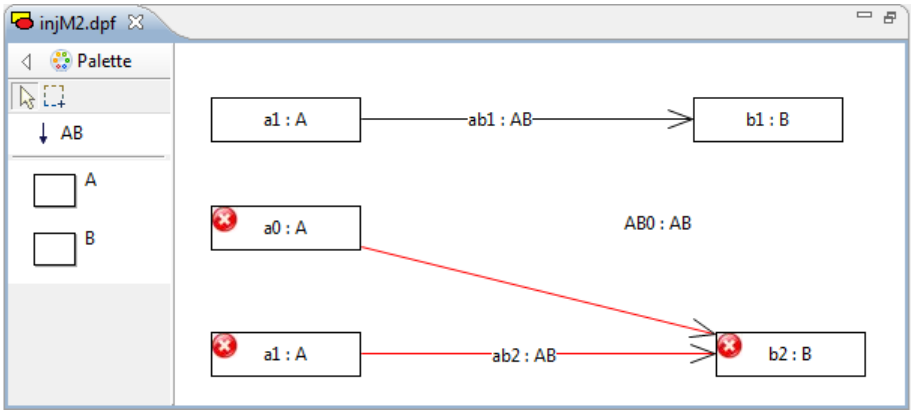


Figure 4.15: Validation Errors in the DPF Model Editor

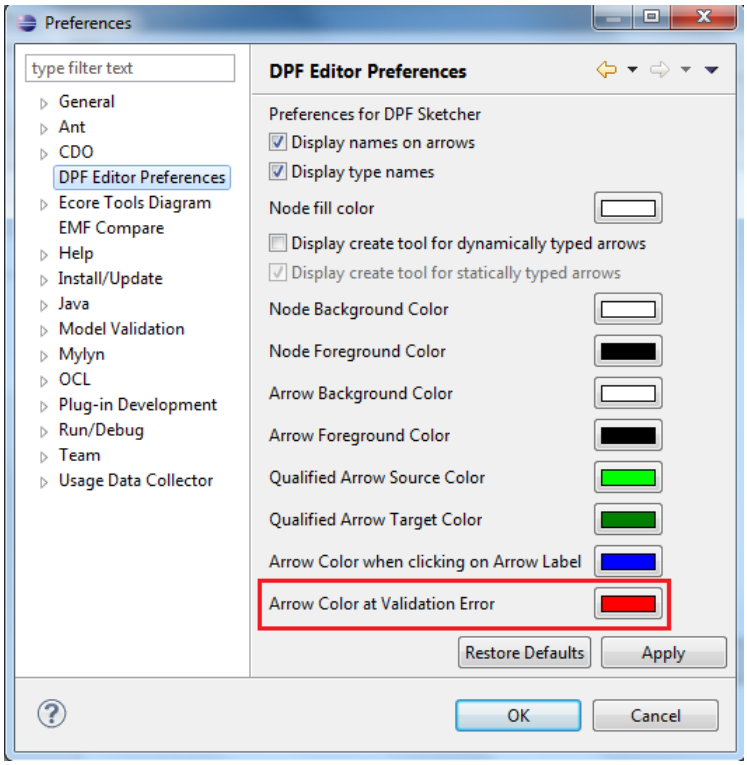


Figure 4.16: The DPF Preferences Page to set Validation error for Arrows

and also how to write a semantic validator in OCL for the DPF model elements. At the end of the chapter we described our incremental approach to the validation of a model in DPF Workbench and how we changed the DPF Workbench Validation feedback mechanism.

Chapter 5

Demonstration

This chapter will demonstrate how the DPF Workbench's Signature Editor is used to create a domain-specific signature, i.e. the predicates that will be available during the modelling process, with semantics of predicates defined in the Object Constraint Language (OCL). The use of the newly defined signature is demonstrated by the help of a small example. This will be a brief demonstration of the DPF Workbench's capabilities to validate models based on the constraints defined in the metamodel.

5.1 Demonstration Setup

This section introduces a simple example model that is created by using the DPF Workbench Model Editor, and a signature is defined by using the DPF Workbench Signature Editor. We will set up the demonstration by specifying an example that models person's working in companies. Our example system consists of:

Classes
A set of classes contain `Person`, `Company`, `Car`, and `Engine`.

References between Classes
The references between classes are:

- `Person` can have reference a `Company` and a `Car`.
- `Company` can have reference to a `Person`.
- `Person` can have reference to a `Person`.
- A `Car` can have reference to an `Engine` and vice versa.

Constraints
The requirement for constraints are:

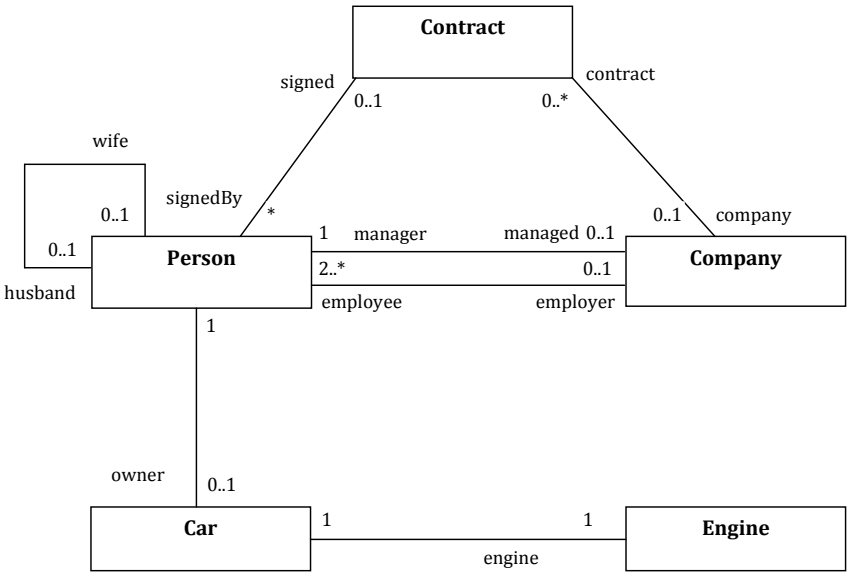


Figure 5.1: Example UML CLass Diagram

1. A `Person` can be an `employee` to zero or one `Company`.
2. A `Company` is an `employer` of at least 2 and at most many `employees`.
3. A `Company` has exactly one `manager`.
4. A `Person` `managed` at-least zero and at most one `Company`.
5. A `Person` either is/ isn't a `husband` or a `wife` of another `Person` but not to itself.
6. A `Person` is an owner of either `0` or `1` `Car`, a `Car` has exactly one owner.
7. Each `Car` must have one `engine`.
8. Each engine belongs to exactly one `Car`.
9. If a `Person` is `manager` of a `Company`, the `Company` must be `managed` by that `Person`.
10. If a `Person` is an `employee` of a `Company`, the `Company` must be the `employer` of that `Person`.
11. If a `Person` is an `employee` in a `Company`, he/she must `sign` exactly one `Contract`.

5.1.1 Creating Signature

Before we model the system described in figure 5.1, we create a new signature with predicates belonging to the DPF predefined java based signature. The semantics of individual predicates is defined by OCL. We could model this example with using the DPF Workbench predefined signature, but our main purpose is to demonstrate our new OCL validator extension in the DPF

Workbench.

The Signature Editor is used to define the arity (shape) of the predicates, a graphical icon that will contribute to the DPF Workbench toolbar, and the semantics of the predicates is defined with use of OCL syntax. Figure 5.2 shows how the semantics of `[exclusive-or]` predicates is defined by the OCL validator. The predicates that are part of the newly defined signature will be available throughout the modelling process.

Table 5.1 shows the semantics of the predicates from table 2.1 defined by the OCL validator, and some of them are used in our modelling example.

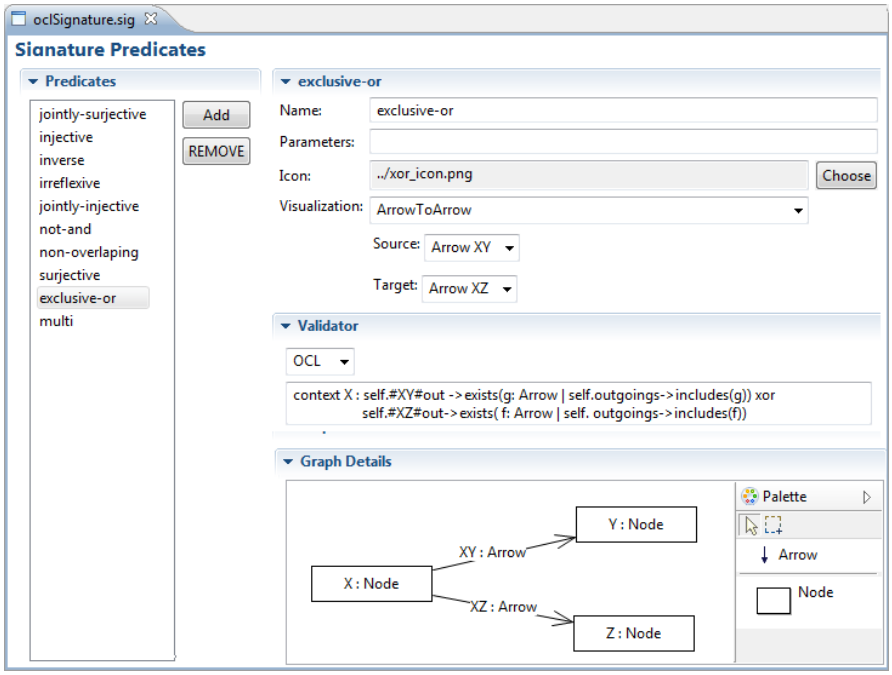


Figure 5.2: Definition of the arity, the graphical icon and the OCL Semantics of the [exclusive-or] predicate

In addition to the predicate defined in figure 5.2, we need four versions of a multiplicity predicate with different parameters i.e. “min:0;max:1”, “min:1;max:1”, “min:2;max:*”, and “min:0;max:*”. These additional predicates have the same arity as of a mult[(m,n)] predicate from table 5.1, but have different parameters. In order to fulfill the requirement no 11, we need an additional constraint stating that “if a person is an employee in a company, he should have signed a contract”. This constraint is represented by `[com]` predicate.

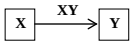
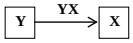
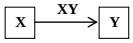
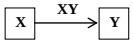
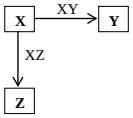
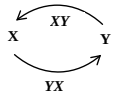
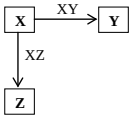
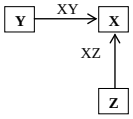
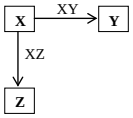
Predicate	Predicate Arity	OCL Syntax
[mult (n, m)]		**context X**: self. #XY#out -> size() >= #p#min **and** self. #XY#out -> size() <= #p#max
[injective]		**context X:** self. #YX#in -> forAll(a:Arrow, b:Arrow | a< >b and a.target = b.target **implies** a.source = b.source)
[surjective]		**context Y:** self. #XY#in ->exists(x: Arrow | self. incomings -> includes(x))
[irreflexive]		**context X**: self.#XX#out ->forAll(a: Arrow | a.source < > a.target)
[non-overlapping]		**context Y:** self. #XY#in->forAll(a: Arrow, b:Arrow | a<>b **and** a.target->asSet()-> intersection (b.target->asSet())) ->notEmpty() **implies** a.source=b.source)
[nand]		**context X:** **not** (self. #XY#out->exists(a:Arrow|self.outgoings ->includes(a)) **and** self. #XZ#out ->exists(b:Arrow| self.outgoings -> includes(b)))
[inverse]		**context Y:** self. #XY#in -> forAll(x: Arrow | x-> includes (x.target)) **implies** self. #YX#out->forAll (y:Arrow | self-> includes(y.target))
[exclusive-or]		**context X:** self. #XY#out -> exists(g: Arrow | self.outgoings -> includes(g)) **xor** self. #XZ#out -> exists(f: Arrow | self. outgoings-> includes(f))
[jointly-surjective_2]		**Context X:** self. #YX#in -> exists (g: Arrow | self. Incomings ->includes(g)) **or** self. #ZX#in -> exists (f: Arrow | self. Incomings -> includes(f))
[jointly-injective]		**context X:** self.outgoings.target.incomings.source -> forAll(x1:Node, x2:Node |x1.typeNode.name = x2.typeNode.name **and** x1.#XY#out -> forAll(c:Arrow, d:Arrow | c.target = d.target) **and** x2.#XZ#out -> forAll(e:Arrow, f:Arrow | e.target =f.target) **implies** x1=x2)

Table 5.1: The DPF predicates defined as OCL syntax

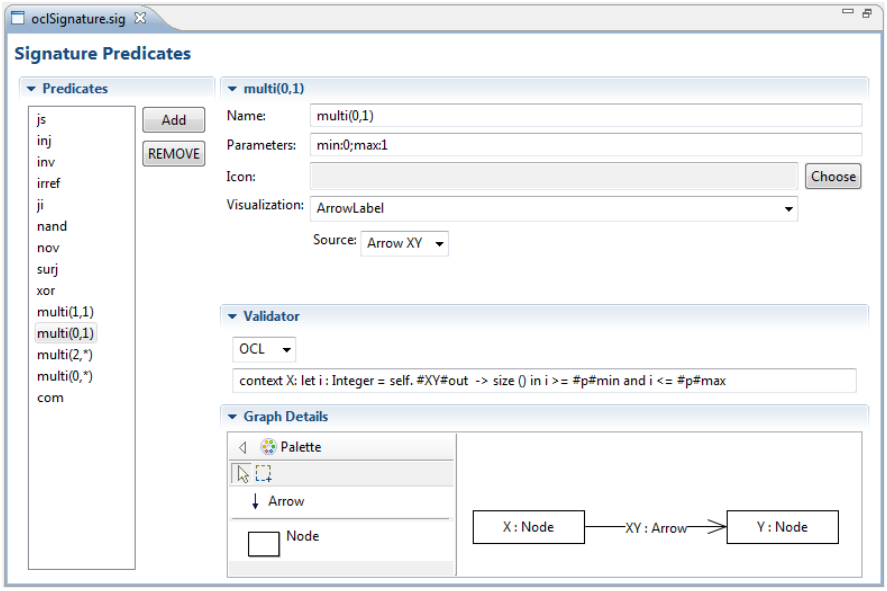


Figure 5.3: Definition of the arity, the graphical icon and the OCL Semantics of the [exclusive-or] predicate

5.1.2 Defining Metamodel

After defining all the necessary predicates in a signature, we load the DPF Workbench with the signature that corresponds to the signature shown in table 5.1.

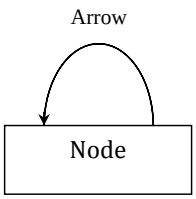


Figure 5.4: The graph of the default metamodel, consisting of a single `Node` and a single `Arrow` self-referencing the node.

We have started the metamodelling process by configuring the tool with the DPF Workbench default metamodel $\mathfrak{S}_3$ consisting of `Node` and `Arrow`, which is the starting point for metamodelling in the DPF Workbench. This default metamodel at level $\mathfrak{S}_3$ is used as the type graph for the metamodel at level $\mathfrak{S}_2$. In $\mathfrak{S}_2$, we introduce concepts of `Class` and `Reference` between `Classs`, `Class` is typed by `Node` and `Reference` is typed by `Arrow`. Figure 5.5 shows the DPF Workbench configured with the default metamodel consisting of `Node` and `Arrow` and the signature Σ from the table 5.1 indicated with a black bold rectangle; It, furthermore, shows metamodel at $\mathfrak{S}_2$ level con-

sisting of `Class` and a `Reference` between Classes. The new file, containing the metamodel, is given the name `m2.dpf`.

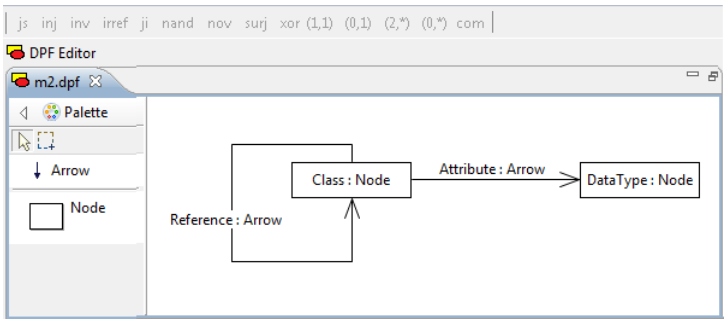


Figure 5.5: Metamodel $\mathfrak{S}_2$ at M2 level

5.1.3 Generating Model Editor from Metamodel

In this section, we illustrate that a Model Editor $\mathfrak{S}_1$ can be generated from the existing (meta) models $\mathfrak{S}_2$. This can be done by using the graph from the metamodel $\mathfrak{S}_2$ as a type graph for a new model $\mathfrak{S}_1$ by invoking the wizard to create a new `DPF Specification Diagram`. This time the new file, containing the specification, is given the name `m1.dpf`. We also specify that a metamodel $\mathfrak{S}_2$ from file `m2.dpf` is used as a metamodel for this new model file `m1.dpf`. We used the same signature from table 5.1 as we used with the metamodel `m2.dpf`.

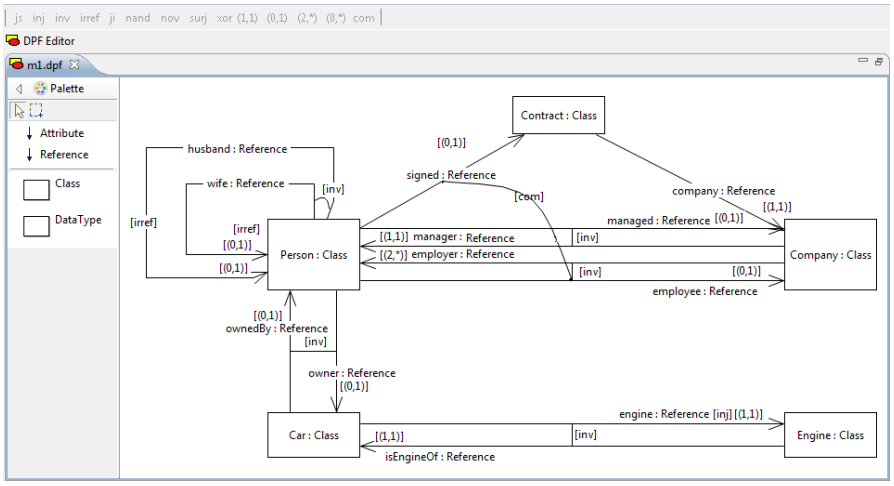


Figure 5.6: Model $\mathfrak{S}_1$ at M1 level

Figure 5.6 shows the DPF Model Editor generated from the metamodel

$\mathfrak{S}_2$ by using it as a type graph. We will explain how the requirements for constraints mentioned in section 5.1 are specified in $\mathfrak{S}_1$.

Requirement 1 and 4: The requirements 1 and 4 are specified by adding the constraint `[0,1]` on the arrow's `employee` and `managed`, which are typed by `Reference`.

Requirement 2: The requirement 2 is specified by adding the constraint `[2,*]` on the arrow `employer` which is typed by `Reference`.

Requirement 3: The requirement 3 is specified by adding the constraint `[1,1]` on the arrow `manager` which is typed by `Reference`.

Requirement 5: The requirement 5 is specified by adding the constraints `[irref]` and `[0,1]` and `[inv]` on the arrow's `wife` and `husband` which are typed by `Reference`.

Requirement 6: The requirement 6 is specified by adding the constraints `[0,1]` and `[inv]` on the arrow's `owner` and `ownedBy` which are typed by `Reference`.

Requirement 7: The requirement 7 is specified by adding the constraints `[1,1]` and `[inv]` on the arrow's `engine` and `is engineOf` which are typed by `Reference`.

Requirement 8: The requirement 8 is specified by adding the constraint `[inj]` on the arrow `engine` which is typed by `Reference`.

Requirement 9: The requirement 9 is specified by adding the constraint `[inv]` on the arrow's `manager` and `managed`.

Requirement 10: The requirement 10 is specified by adding the constraint `[inv]` on the arrow's `employee` and `employer`.

Requirement 11: The requirement 11 is specified by adding the constraint `[com]` between the arrow's `signed` and `employee`.

Since the metamodel $\mathfrak{S}_2$ contains no constraints on model elements so no validation takes place in model $\mathfrak{S}_1$, it only checks that the model $\mathfrak{S}_1$ is correctly typed by its metamodel $\mathfrak{S}_2$, if there exists a graph homomorphism from model to its metamodel. It has been done internally by the DPF tool.

5.1.4 Generating Instance Editor from Model

The instance editor can be generated by repeating the process of invoking the wizard to create a new `DPF Specification Diagram`, using the model $\mathfrak{S}_1$ as a type graph of the instance $\mathfrak{S}_0$. Figure 5.7 shows an instance editor generated by using model $\mathfrak{S}_1$ as a type graph.

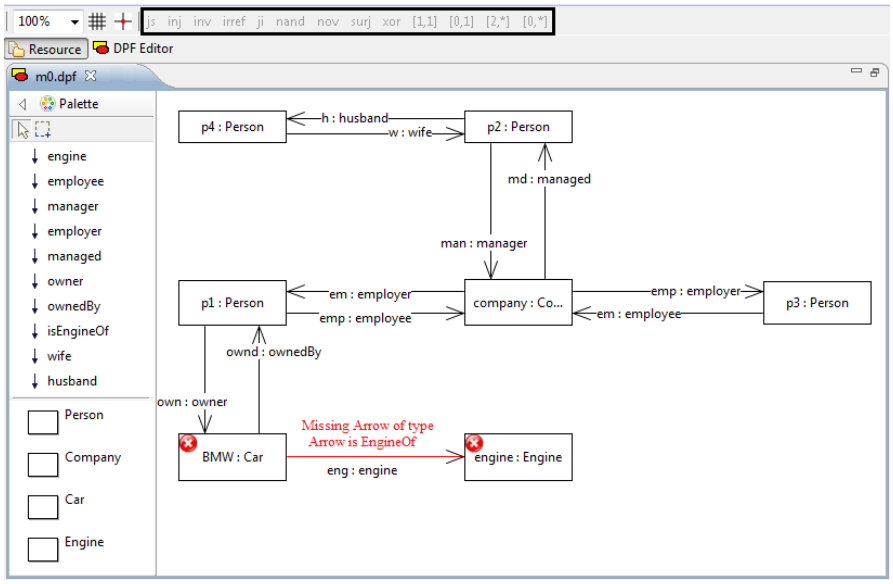


Figure 5.7: Instance $\mathfrak{S}_0$ at M0 level violating the `[inv]` between nodes `BMW` and `engine` because of missing arrow of type arrow `isEngineOf`.

5.1.4.1 Validating Constraints

In the DPF Workbench, when an instance of a model is created the constraints defined in the model are checked by their corresponding semantic validators. In figure 5.6, we see that there are no constraints specified in the metamodel $\mathfrak{S}_2$, hence no validation takes place in the model $\mathfrak{S}_1$. Figure 5.7 shows an instance $\mathfrak{S}_0$ which violates some of the constraints of $\mathfrak{S}_1$, i.e. the `[inv]` constraint on the arrows `engine` and `isEngineOf` in $\mathfrak{S}_1$ is violated because of missing arrow of a type arrow `isEngineOf` between nodes `BMW` and `engine`. Figure 5.8 shows a valid instance $\mathfrak{S}_0$ of the model $\mathfrak{S}_1$ which satisfies all the constraints specified in the model $\mathfrak{S}_1$.

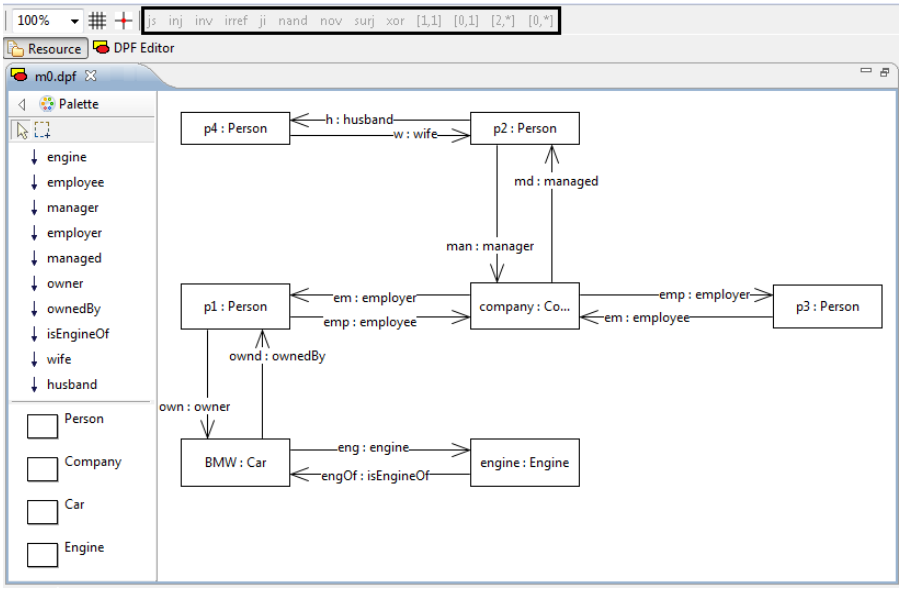
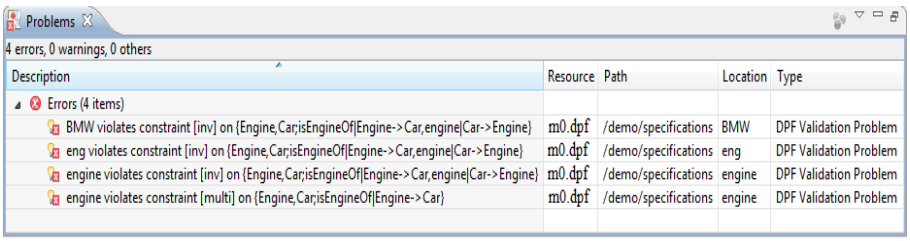


Figure 5.8: a Valid Instance $\mathfrak{S}_0$ of a model $\mathfrak{S}_1$.

5.1.4.2 Validation Feedback Reporting

Upon violation of any constraint on an instance $\mathfrak{S}_0$ level, by any model element a validation feedback is reported to the user. This feedback is given both in the DPF Model Editor window and in the `Problem View`.

Figure 5.7 shows how the validation feedback is reported to the user in the DPF Model Editor window. When a certain constraint has failed validation, the DPF tool will highlight only those model elements that are related to that constraint, by showing an error image on top right of the `Node` figure and changing the colour of the `Arrows` to as prescribed by the user in the `Preferences Page`. Figure 5.9 shows how the validation feedback is reported to the user on the `Problem View`.



Problems

4 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (4 items)				
BMW violates constraint [inv] on {Engine,Car;isEngineOf|Engine->Car,engine|Car->Engine}	m0.dpf	/demo/specifications	BMW	DPF Validation Problem
eng violates constraint [inv] on {Engine,Car;isEngineOf|Engine->Car,engine|Car->Engine}	m0.dpf	/demo/specifications	eng	DPF Validation Problem
engine violates constraint [inv] on {Engine,Car;isEngineOf|Engine->Car,engine|Car->Engine}	m0.dpf	/demo/specifications	engine	DPF Validation Problem
engine violates constraint [multi] on {Engine,Car;isEngineOf|Engine->Car}	m0.dpf	/demo/specifications	engine	DPF Validation Problem

Figure 5.9: Validation Feedback in the `Problem View`.

5.2 summary

In this chapter, we have demonstrated the extension that has been made to the DPF Workbench, i.e. the extension of the OCL checker, incremental validation approach and validation feedback to the user. We have demonstrated the tool with the help of a modelling example and creating new signature using the DPF Workbench's Signature Editor with predicate's semantics defined in OCL syntax. At the end, we have demonstrated how validation feedback is reported to the user upon violation of any defined constraint.

Chapter 6

Evaluation and Conclusion

6.1 Evaluation

In this section, we evaluate the new OCL validator of the DPF Workbench by considering some of the major attributes of a software product. This section also compares the new OCL validator with the previous DPF Workbench validation approach and with other related approaches to the visualizations of OCL.

6.1.1 Previous Approach

This section gives a brief description of the flaws in the previous DPF Workbench approach to the validation of the DPF Constraints. The previous predefined validator of the DPF Editor was hard coded with semantics validators for the DPF predicates. It was implemented in java. It validates the complete model upon any change in the model elements (i.e. a `Node` or an `Arrow` is added or removed from the model). The DPF Workbench extends DPF Editor with the Signature Editor. The Signature Editor allows users to define domain-specific constraints with their corresponding validators defined by using java. To use the Signature Editor, users must know the java programming language. In order to make the DPF Workbench validator more efficient and easier to use, it can be extended with an OCL validator. For more details on the previous DPF Workbench's validation mechanism see section 3.4.

Some of the flaws that lied in the DPF Workbench's java validator are:

- In order to define new constraints by using the Signature Editor, a user has to be a java programmer.

- There was no OCL support, which is an easy and efficient way. OCL is much used nowadays, and it is known by many people.
- The earlier version of the DPF Workbench validates the complete model every time a model element has been changed. This leads to be a time-consuming job and is not considered as an efficient way to check model validity.
- The DPF Workbench will not show any details for validation failure, it only shows "Validation Failed" message on the DPF Model Editor `Status bar`. This makes it difficult for users to change errors in the model.

6.1.2 Our Approach

In this thesis, we have extended the DPF Workbench with an OCL validator which enables users to define the semantics of predicates by OCL, but it is still possible to use the previous Java validator functionality. We have changed the actual syntax of OCL expressions so that it can be evaluated for the DPF Workbench model elements, i.e `Node` and `Arrow`. This extension lead to be a visual OCL approach, because every OCL expression is assigned a name, shape (arity), and name of the predicate will be shown directly on the DPF Model Editor window when added on model elements. The previous DPF Workbench validation approach has been changed to an incremental approach, i.e for this purpose, we have defined an algorithm (see listing 4.10) that finds all the elements that are related to a constraint in which the newly changed model element belongs to. Furthermore, we have changed the validation feedback reporting mechanism so that upon validation failure, the model elements that are related to a constraint which has failed validation will be highlighted directly in the DPF Model Editor window, and the detail of the message will be shown in the `Problem View`.

6.1.3 Evaluation of Our Approach

In this section, we have evaluated our extension to the DPF Workbench. In order to evaluate our extension, we have considered the following attributes:

Functionality

By using this new extension a user has the following possibilities:

- A user can define a new signature with predicate semantics defined by OCL; these predicates will be available throughout the modelling process. The OCL `invariant` constraints can be defined for the DPF Workbench model elements.

- In a signature, a user can define as many predicates as he wants, but it can only be used to create atomic constraints.
- A user can set `Node` as a `context` on which the OCL `invariant` constraint will be evaluated.

By using this new extension a user can not have the following possibilities:

- Once the user has created a signature and loads it into the DPF Workbench, it is not possible to extend that signature. If the user needs any new constraint in this signature, he first has to modify the signature file and then starts the modelling from scratch once again. Otherwise, the newly added constraint will not be available during the modelling process.
- The solution only supports the `invariant` part of OCL; it is not possible to define `pre` and `post` condition types of constraints, as these constraints are related to methods and behavioural models.

Cost

This project is an extension of the DPF Workbench tool which is an open source and is available free of cost. It can be downloaded from the DPF website [19] as an `Eclipse` update site. All the other plug-ins that are required to run this tool are also available free of charge.

Useability

One prerequisite for the user to use this extension is, the user should know the Object Constraint language (OCL) in order to be able to write the DPF predicates semantics by OCL, which is a formal language which is easy to read and write. Since the DPF Workbench validation feedback mechanism has been improved, so that upon validation failure, the model elements involved in failure will be highlighted directly on the DPF Model Editor Window. This makes it easier for the users to remove errors in the model. This improves useability of the DPF Workbench.

Reliability

If the semantics of predicates in a signature is defined as a valid OCL expression, then the DPF Workbench guarantee to work correctly. If there are some syntax errors in the OCL expression, the DPF Workbench does not work.

Support

To support users on how to use this new OCL validator extension has been written in a tutorial presented in appendix A. This tutorial explains how to create a new signature in the DPF Workbench with predicate semantics defined by OCL and how to use this new signature in

the modelling process.

Maintenance and Further Work
As this is the first implementation of an OCL validator in the DPF Workbench, it also needs maintenance and further work (will be described in section 6.3).

Speed and Performance
The performance of the new DPF Workbench incremental validation approach should be better than the previous one. In order to evaluate the performance some experiments should be run on the new validation approach. However, unfortunately due to time constraint, it was not possible to evaluate the performance of the new incremental validation algorithm, but this work will be done in the future.

6.1.4 Related Work

This section gives a brief comparison of the visualization of OCL constraints in the DPF Workbench with two other approaches. It also gives a comparison of the DPF Workbench with other modelling frameworks w.r.t the OCL constraints.

6.1.4.1 Comparison with other visualizations of OCL approaches

The eclipse infrastructure for modelling software systems is based on UML. However, the UML diagrams are generally not enough to capture all the interesting aspects of the software system. In order to design a software system, there may be a need to define additional constraints. In the beginning, these constraints were defined in natural languages, which may lead to ambiguity. To reduce this ambiguity a formal (textual) language Object Constraint Language (OCL) has been developed. The system modeler defines the additional textual constraints in order to be able to model a software system. But it is still of limited use in organisations. The reasons for that is the integration of a purely textual language into the diagrammatic paradigm. A lot of work has been done on visualization of OCL constraints. Some of the most known are `VisualOCL` [6] and `Constraint Diagrams` an `Spider Diagrams` [29]. For more details on `VisualOCL` and `Constraint Diagrams` [31] saw sections 3.3.1 and 3.3.2 respectively.

Table 6.1 shows that the `VisualOCL` supports the complete OCL 2.0 Version 1.5. The tool for `Constraint Diagrams` supports static (`invariants`) as well as dynamic (`pre` and `post` conditions) OCL constraints, while the DPF Workbench only supports some `invariant` part of the OCL. The `Constraint Diagrams` are fully diagrammatic, while the `VisualOCL` is more textual, and the DPF Workbench has both textual and visual OCL constraints.

Tool	Expressivity	Diagrammatic	Tool Support	Learnability	Readability	Model Editor
VisualOCL	Complete OCL 2.0 v 1.5	Basically textual	developed	similar to OCL + UML	Easy to read	No
Constraint Diagram	static + dynamic	Fully Visual/-Diagrammatic	not fully developed	similar to Venn/Euler Diagrams	Difficult / more diagrammatic	No
DPF	part of OCL	textual+visual constraints	Prototype Version	OCL + DPF concepts	easy to read for the DPF users	Yes

Table 6.1: Comparison of DPF Workbench's Visualization of OCL with two other OCL Visualizations, VisualOCL and Constraint Diagram.

The tools for the `Constraint Diagrams` and `Visual OCL` only allows users to define constraints visually and then transform them into textual OCL constraints or some other form to be used with the UML models. They does not provide the creation of models by providing the model editors by themselves. The first version of the DPF Workbench's OCL Visualization is available. The constraints defined by `VisualOCL` is easy to read and learn because of its nature related to UML and OCL, while the `Constraint Diagrams` are similar to Venn diagrams and Euler diagrams and are more difficult to read because of their diagrammatic nature. In order to define dynamic definition of constraints in the DPF Workbench, a user should be familiar with the DPF concepts and the OCL. `VisualOCL` and `Constraint Diagrams` does not have any Model Editor support, while the DPF Workbench facilitates users to create Domain-Specific modelling languages and generate code for these modelling languages. Table 6.1 summaries the comparison of some popular OCL visualization approaches with the OCL visualization of the DPF Workbench.

6.1.4.2 Comparison with the other modelling Frameworks

There are a lot of visual modelling tools available. Some of these tools possess the visual metamodelling features, allowing the users to specify metamodel and then create new specification editors from this metamodel. But to specify constraints on the model these tools make use of textual OCL. Eclipse Modelling Framework (EMF) [13] is the most popular tool available as an open source for modelling software systems. It has a support for OCL, but as a textual language. It does not support diagrammatic definition of OCL constraints. The DPF Workbench has the feature for the diagrammatic definition of constraints and their corresponding semantics by use of OCL validators. MetaEdit+ [37] is a commercial language workbench that enables users to create their own modelling and code generation tools. It comes in two main components: MetaEdit+ Workbench and MetaEdit+ Modeler. The MetaEdit+ Workbench allows metamodelers to build modeling languages (DSLs), generators and graphical notations. The MetaEdit+ Modeler provides customizable DSM functionality for developers which is defined by using the MetaEdit Workbench. The main limitation of using MetaEdit+ is that, it only has a set of predefined constraints, and it does

not have any support for customizable OCL constraints.

Table 6.2 summaries the comparison of EMF [13] with the DPF Workbench and MetaEdit+ with respect to the OCL constraints and their validation mechanism. More detailed comparison of these modelling tools as the language workbenches see [34]. Table shows that both the EMF and the DPF Workbench supports OCL, while MetaEdit+ has no support for OCL. EMF only supports OCL as a textual language, no visualization available, while the DPF Workbench supports diagrammatic (visual) OCL constraints. The validation mechanism of EMF is explicitly triggered by the user (i.e. batch), while in the DPF Workbench and MetaEdit+, it is done automatically (i.e. live) by the tool. The validation diagnostic in EMF is shown in a dialog box, and in the problem view if an instance failed validation, while in the DPF Workbench, the validation feedback is directly displayed in the Model Editor window (by showing error directly on the model elements) and also in the Problem View. MetaEdit+ shows validation diagnostic only in a dialog box.

Tool	**OCL support**	**Visual OCL**	**Constraint**	**Validation**	**Validation Diagnostic**
EMF	✔		Predefined/ textual OCL	triggered by user	problem view/ dialog box
DPF	✔	✔	Predefined/ Visual OCL	automatically	Editor window and Problem view
MetaEdit+			Predefined	automatically	Dialog box

Table 6.2: Comparison of DPF Workbench with Eclipse Modelling Framework (EMF) w.r.t the OCL.

6.2 Conclusion

In this section, we conclude our thesis by giving a summary, of the basis for this thesis and what has been achieved through this thesis. We also describe some of the possible future work related to this thesis and to the overall DPF Workbench.

6.2.1 Summary

In this section we summarize, the main goals of this project and how these goals have been achieved through this project. The DPF Editor [4] was the first tool support for the Diagram Predicate Framework, it has a predefined signature (with predicates and their corresponding semantics validators) to add constraints on models. In this signature, the semantics of predicates was hard coded as java validators. Sometimes a DPF user may wish to define their own signature. The DPF Workbench [34] extends the DPF Editor with a Signature Editor to create new predicates and their corresponding semantics validators. The Signature Editor is used to define the arity of the

predicates, a graphical icon that contributes in the DPF Workbench toolbar and the semantics of the predicates. But the semantics of a predicate can only be defined by a java validator. In order to write the semantics of predicates using the Signature Editor a user has to be a programmer, so there should be a possibility to define semantics as OCL syntax which we have done in this thesis. Moreover, the DPF Workbench has a validation mechanism that checks instances against all the constraints that are defined in the (meta) model. It implements the traditional approach, i.e. it checks the complete instance every time a model is changed. It was not an efficient way to do validation. The DPF Workbench has no proper mechanism for validation feedback, i.e. it only displayed "Validation Failed" or "All Constraints Validated" on the DPF Model Editor `status bar`, without specifying any information about which constraint has failed validation and which model elements are involved in failure. The functionality of this new extension has been demonstrated by the help of an example in chapter 5.

Through this thesis we have achieved the following:

Visual OCL Validator

This thesis extends the DPF Workbench with an OCL validator. By this extension to the DPF Workbench, the user can now define the semantics of predicates with OCL syntax. Each predicate is assigned; a name which will be shown directly on the DPF Model Editor Window when a corresponding constraint is added to the model elements, the arity of the predicates, a graphical icon that contributes in the DPF Workbench toolbar, and the semantics of the predicates defined in OCL.

Incremental Validation

We have changed the previous validation approach of the DPF Workbench to an incremental approach. According to this approach the DPF Workbench checks the constraint in an incremental way, i.e. if we assume that `M` is a valid model and then an element, a `Node` or an `Arrow`, is added (changed or deleted). Then the model becomes `M'`. To check if `M'` is valid, the DPF tool only checks the constraints related to this newly added model element instead of checking the complete model repeatedly. This approach reduced the cost of model validation and increase efficiency. For this purpose, we design a small algorithm that finds related model elements from a model and sends them to the semantic validator to be checked.

Validation Feedback

We have also changed the DPF Workbench Validation feedback mechanism (see section 3.4.1) so that the validation failure message will directly be shown in the DPF Model Editor main window. Upon validation failure of any constraints by the model elements, the model elements that are related to that constraint will display an error image on `Node` figure and changes the colour of `Arrow` as prescribed by the

user in the DPF Preferences Page. In addition to this, validation errors message detail will also appear in the `Problem View` for each model element that is related to the constraint which has been violated.

6.3 Suggestion For Further Work

This section contains a proposal for further work that can be done for the the validation of constraints in the DPF Workbench and the DPF Workbench overall.

6.3.1 Validation

The OCL validator for the DPF Workbench should be considered as a prototype, as it has not been tested properly. The following section explores some of the aspects on which further work is required.

Validation Failure Message on the DPF Model Editor Window

Our approach to the validation failure report was that we add an error image on `Nodes` and change the `Arrow` colour upon failure of any constraints by model elements. The details of the error messages will appear in the `Problem View`. One possible extension could be to directly show error messages on the DPF Editor window instead of in the `Problem View`.

Content Assistant for OCL Expression

While defining the semantics of predicates in OCL, the DPF Workbench does not provide any suggestion about OCL syntax, i.e. which methods are available for the defined context. So if the user accidentally missed any dot or the syntax of OCL expression is not correct, then the DPF Workbench's OCL validator does not work. The solution to this problem could be to include an OCL content assistant in the DPF Workbench.

Visual OCL Constraint

Our approach to visual OCL was that, each OCL constraint is given a name that will appear in the DPF Model Editor window when added on model elements. It works fine for constraints having arity `X`→`Y`, but as the constraint arity changed to ternary

$$\begin{tikzcd} 1 \arrow[r, "f"] \arrow[d, "g"] & 2 \\ 3 & \end{tikzcd}$$

(involving three nodes and two arrows), the name of the constraint hides under arrows and shows up by moving the arrows up and down. This Problem needs to be fixed.

Modification in a Signature after the modelling process starts

Once a signature has been loaded for modelling, it is not possible to add any new constraint in that signature. If the user needs any new constraint during the modelling process, first he has to modify the signature and then starts the modelling process again from scratch. This could be a basis for further work in the DPF Workbench.

Defining OCL Constraints for behavioural models

Currently, In the DPF Workbench, it is only possible to define static constraints such as `invariants` in OCL, so to define the dynamic constraints such as `pre` and `post` conditions to capture the dynamic behaviour of the model would be a nice extension to this project.

Signatures for different Domains

One possibility for further work in the DPF Workbench is to design different signatures for the modelling of different domains, e.g. a signature for database modelling includes predicates for primary key and foreign key, etc. A user can select any one signature from a list of available signatures, relative to their domain and can modify it if required. This will reduce the modelling time.

6.3.2 The DPF Workbench

This section specifies some of the ongoing and future work within the context of the DPF Workbench overall.

6.3.2.1 Visualization of Models

One of the important challenges the DPF Workbench is facing is the visualization of models. All visualizations (nodes and arrows) are hard coded in the DPF Workbench, so it would be desirable to decouple the visualizations from the `display Model`. In DPF, we only have nodes and arrow, and there is no support for attributes on nodes. This will lead for a Model to look more complex as the size of the model increases.

This problem can be tackled by introducing a single diagram representation for classes and their attributes. A proposal is to change the concrete graphical representation of the model to allow customisable visualization in the DPF Workbench tool. An initial work to tackle this problem has been started by Master student Ola Bråten.

6.3.2.2 Layout and routing

The DPF Model Editor has some issues of routing, currently it is based on GEFs, Draw2D `ShortestPathConectionRouter` class. As the size of model increases, an automated layout becomes an issue. The problem to find a routing algorithm that can produce easily readable output could be a new research task within the context of DPF Workbench.

Appendix A

Tutorial for Defining a New Signature in DPF Signature Editor

The Diagram Predicate Framework (DPF) workbench [**?**] mainly have two types of editors; the DPF model editor [33] and the Signature Editor. The DPF model editor is a graphical modelling tool for the specification of domain specific (meta)models, models and instances. The Signature editor provides the user with the functionality to define domain-specific constraints. This tutorial explains how to define a new signature for your (meta)models, models, or instances.

A.1 Installation

Before starting you should have these software installed on your computer.

- Eclipse Indigo Version.
- The DPF workbench.
- OCL Examples and Editors. This can be installed via Help → Installed Modelling Components → Select OCL Examples and Editors → Finish.

To get familiar with the DPF model editor you can find a good tutorial here `http://dpf.hib.no/wp-content/uploads/Introduction-to-using-the-DPF-editor-Google-Disk1.pdf`.

A.2 Define a new Signature File

In DPF workbench dynamic predicates are defined with the help of the Signature Editor. A signature contains a set of predicates. Each predicate has a name, a syntax, a visualization, a semantic validator, properties and an icon. The steps that are necessary to define new constraints are:

A.2.1 Create a new dpf project

Create a new Diagram Predicate FrameWork project called no.dpf.project via File → New → other → Diagram Predicate Framework → Diagram Predicate Framework Project and click Next as shown in Figure A.1.

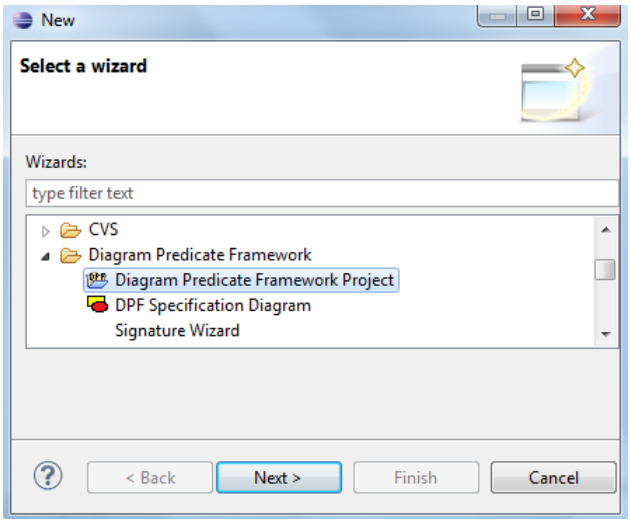


Figure A.1: Select DPF Project Wizard

Add no.dpf.project as the Project name and click Finish as shown in Figure A.2.

A.2.2 Create New Signature Wizard

Select the specification folder, right click on it and select New → other → Diagram Predicate Framework → Signature Wizard → Next as shown in Figure A.3.
Add a newSignature.sig as the File name and click Finish as shown in Figure A.4.
This should open a visual editor for creating Signature Predicates as shown in Figure A.5.

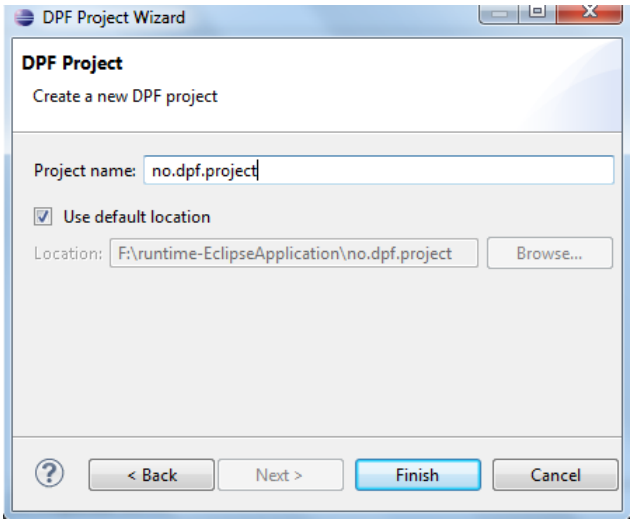


Figure A.2: Create a New DPF Project

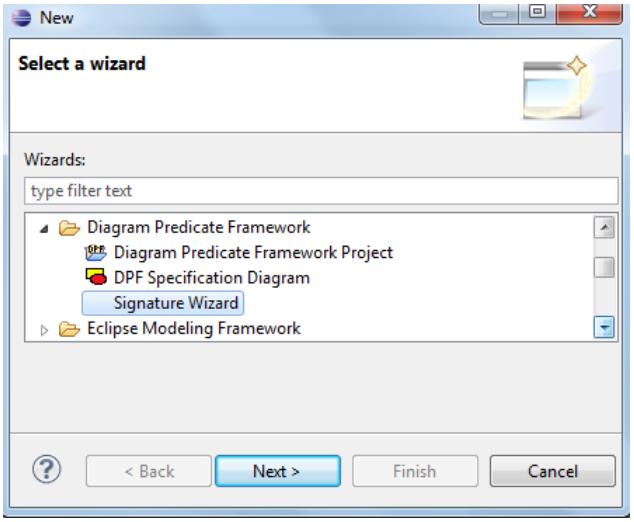


Figure A.3: Create New Signature Wizard

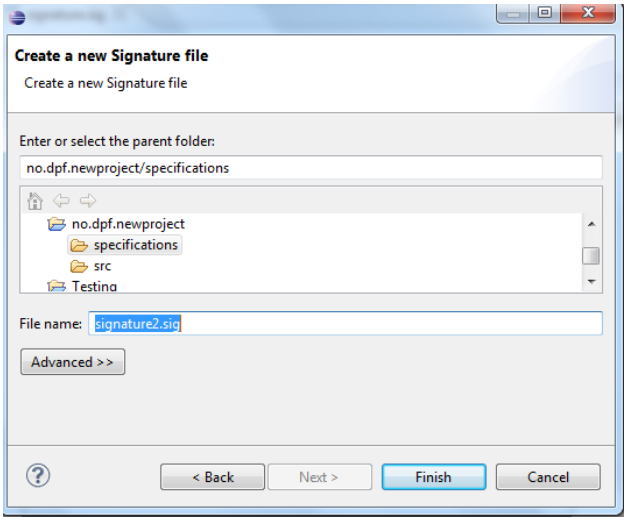


Figure A.4: Create a New Signature File

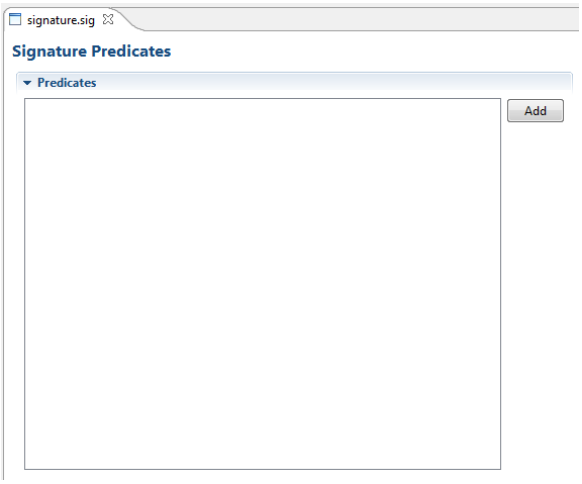


Figure A.5: Signature Visual Editor

Figure A.6: Signature Predicate Details Editor

Click on Add, this will add a new predicate in the Signature Editor and open the details for the new predicate0 as shown in Figure A.6

These details include:

A.2.2.1 Name of Predicate

It is the symbol that is appeared when a predicate is added to the model in the DPF model editor.

Add *mult* as the Name of the predicate in the box following *Name* as shown in Figure A.7.

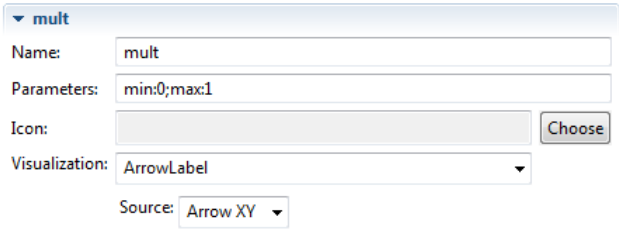


Figure A.7: mult Predicate details

A.2.2.2 Predicate Properties

The predicate properties define the minimum and maximum number of instances of one model element related to the number of instances of another model element. They are optional, but can be defined as min and max values separated by a semicolon.

The format of parameters is “min:0;max:1” as shown in Figure A.7.

A.2.2.3 Predicate Icon

It is the image that will contribute to the DPF model Editor’s signature toolbar. If an *Icon* is left empty the name of the predicate will appear on the Signature toolbar. An *Icon* can be added by clicking the choose button.

A.2.2.4 Predicate Visualization

It includes different types of visualization, i.e. *ArrowLabel, ArrowToArrow, NodeToArrow, ArrowToNode and composed* depending on the syntax of the predicate. Visualization type can be selected depending on the no of model

elements involved in the predicate, e.g. if the predicate is unary or binary you can select ArrowLabel, NodeToArrow or ArrowToNode and if the predicate is ternary you can select ArrowToArrow and etc.

Select ArrowLabel as visualization type and also select `Source` i.e. the name of the Arrow defined in the graph detail as shown in Figure A.7.

A.2.2.5 Predicate Semantic Validator

The validator can be a Java or OCL type. It defines the semantics of a predicate. The semantics of a model describe what will be the affect when a model will be executed. Select OCL as Validator type and add Validation String in the text box following the validator type as shown in listing below.

```
context X: self.#XY#out->size() >=#p#min
                    and
          self.#XY#out->size() <=#p#max
```

Listing A.1: Multiplicity Predicate OCL Expression

```
context X:self.#XY#out->exists(g: Arrow|self. outgoings->includes(g))
                    xor
         self.#XZ#out->exists(f:Arrow|self. Outgoings ->includes(f))
```

Listing A.2: XOR Predicate OCL Expression

Listing below shows a semantic validator for a predicate that is ternary predicate written as OCL expression.

A.2.2.6 Predicate Syntax

It contains the shape of the predicate. Define a graph with two nodes X and Y and one Arrow XY in between them as shown in Figure A.8.

Save the Signature File it is ready to load into the Specification. Figure A.9 shows an example of a predicate which involve three nodes and two arrows in between them.

A.3 Load newly defined Signature

The newly defined signature *newSignature.sig* can be loaded by Select the Signature File *newSignature.sig* right click on it and select New → other

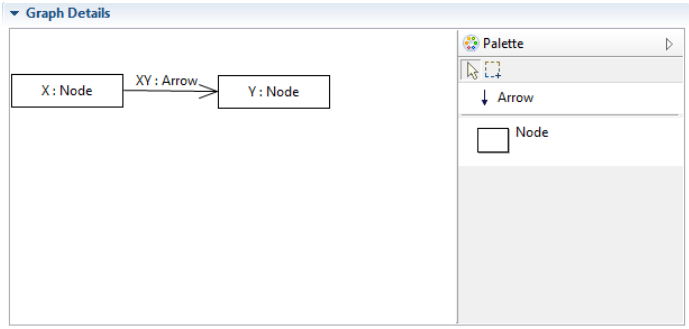


Figure A.8: Define Predicate Shape

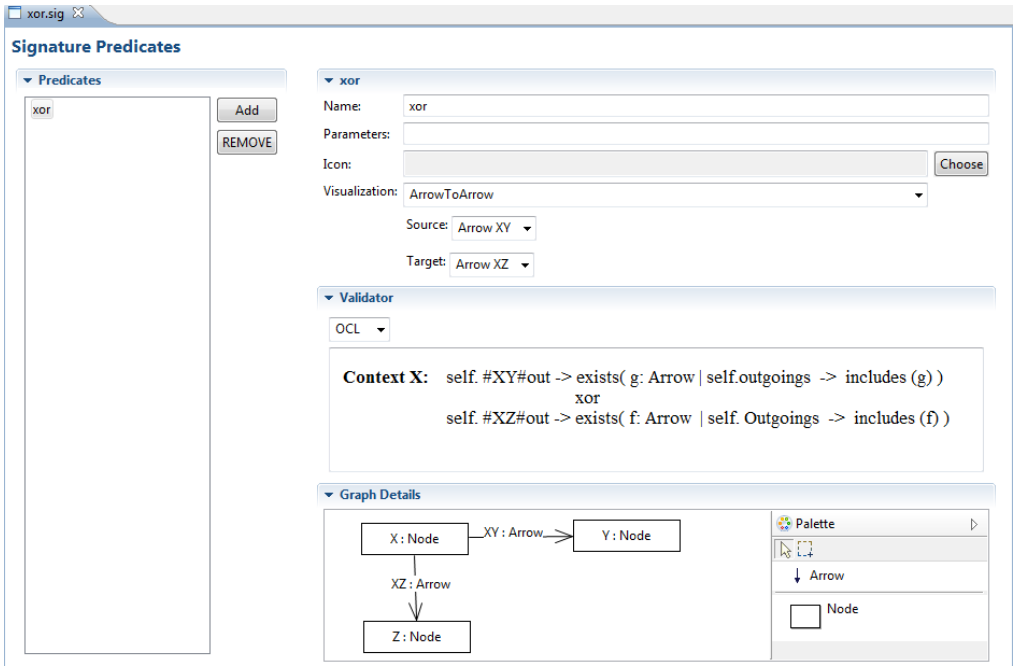


Figure A.9: Example of predicate with ternary no of model elements

→ Diagram Predicate Framework → DPF Specification Diagram → Next as shown in Figure A.10.

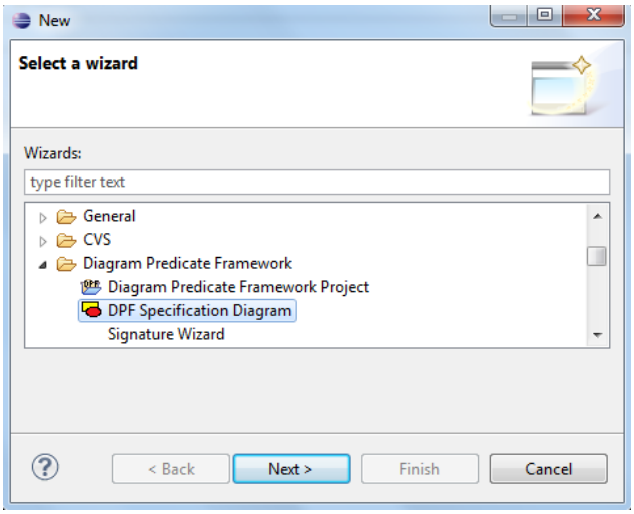


Figure A.10: Create New Specification

Add *Model.dpf* as File Name as shown in Figure A.11.

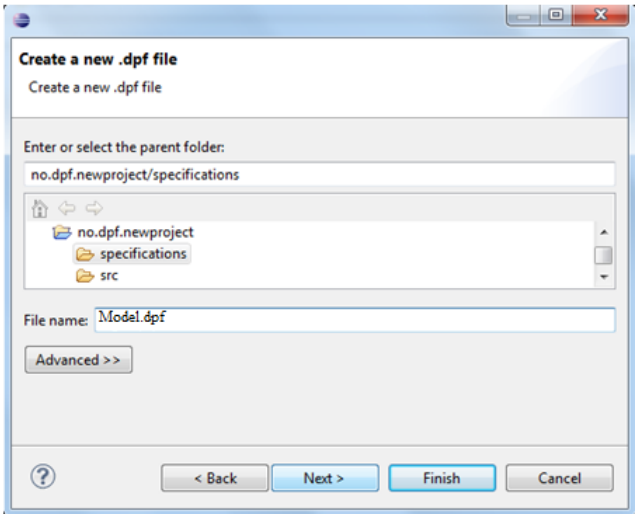


Figure A.11: Enter Specification Name

Click Next → Next → check Load file box to include signature file into the *Model.dpf* Specification file and click finish as shown in Figure A.12.

This will open Model.dpf specification file in DPF editor as shown in Figure A.13. This specification contains the signature with one constraint which we have just created.

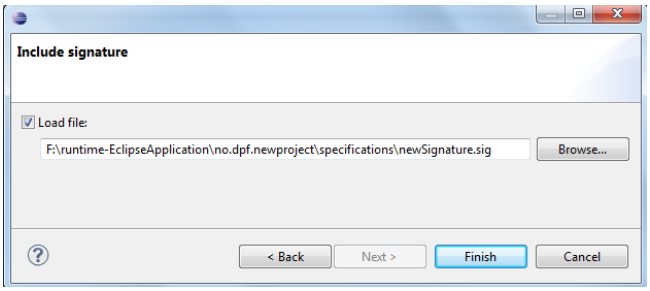


Figure A.12: Load New Signature

A.4 Add Constraint on Model

Edit the *Model.dpf*. Add two nodes named A, B and an Arrow AB as shown in Figure A.13. Select Arrow by mouse left click, the predicate mult in the toolbar get enabled click on it. It will be added to the model. Save *Model.dpf*.

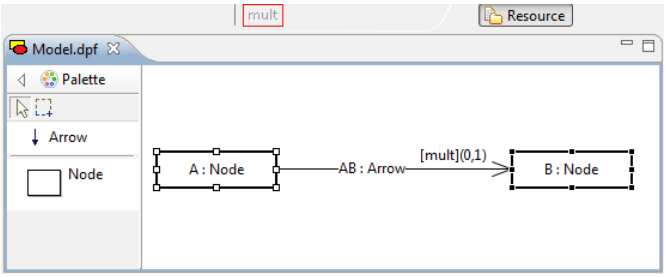


Figure A.13: Specification Editor

A.5 Creating Instance of Model

Create a new specification file named *instance.dpf* by loading *Model.dpf* as type Graph. The validation process starts when an instance of a model is created. Try to add some nodes of type A, B and an arrow in between them. The editor will show validation errors if the instance violates the constraint which we add to the model as shown in Figure A.14.

The validation errors are also shown in problem view as shown in Figure A.15. The*Problem View* is part of eclipse IDE and is used to display system generated errors, information associated with a resource and warnings.

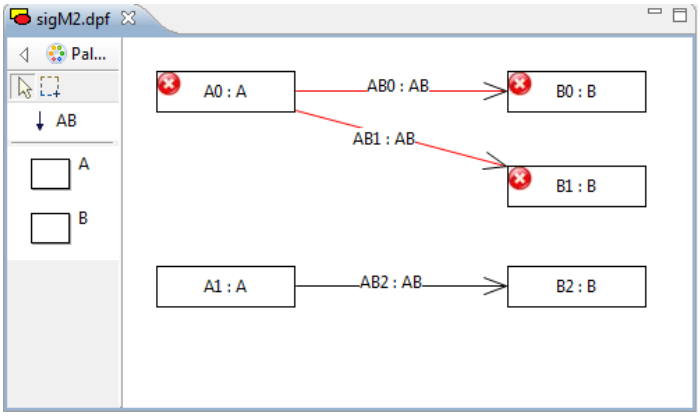


Figure A.14: Validation Error

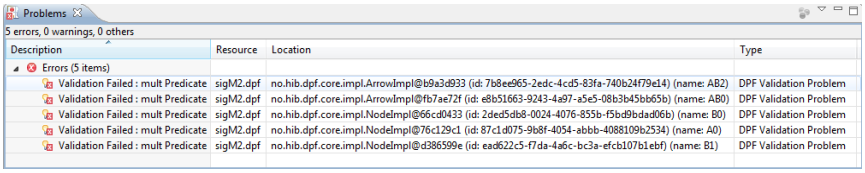


Figure A.15: Validation Error in Problem View

Bibliography

[1] *Project Web Site*. http://www.cs.kent.ac.uk/projects/rwd/index.html.

[2] *Project Web Site*. http://www.eulerdiagrams.com/.

[3] Agile Methodologies. *Project Web Site*. http://agilemethodology.org/.

[4] Øyvind Bech. DPF Editor – A Multi-Layer Modelling Environment for Diagram Predicate Framework in Eclipse. Master's thesis, Department of Informatics, University of Bergen, Norway, May 2011.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999. http://books.google.no/books/about/Extreme_Programming_Explained.html?id=G8EL4H4vf7UC&redir_esc=y/.

[6] Paolo Bottoni, Manuel Koch, Francesco Parisi, Presicce, and Gabriele Taentzer. Visualization of OCL using Collaborations. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185, pages 257–271. Springer Berlin Heidelberg, 2001.

[7] Maarco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model Driven Software Engineering In Practice*. Morgans & ClayPool Publishers, September 2012. http://www.morganclaypool.com/.

[8] Cambridge. Dictionaries Online.http://dictionary.cambridge.org/.

[9] Christiane Kiesner and Gebriele Taentzer and Jassica Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. Technical report, Technical University of Berlin, 2002.

[10] den Haan Johan. Mde-model driven engineering-reference guide, January 2009. http://www.theenterprisearchitect.eu/.

[11] Zinovy Diskin and Boris Kadish. Variable set semantics for keyed generalized sketches: formal semantics for object identity and abstract syntax for conceptual modeling. *Data Knowl. Eng.*, 47(1):1–59, 2003.

[12] Douglas C. Schmidt. *Model Driven Engineering*, February 2006. http://www.computer.org/.

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

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

[15] Eclipse OCL. *Project Web Site*. http://www.eclipse.org/projects/project.php?id=modeling.mdt.ocl.

[16] OCL Examples and Editor. *Project Web Site*. http://wiki.eclipse.org/MDT-OCL.

[17] Andrew Fish, Jean Flower, and John Howse. The semantics of augmented constraint diagrams. Journal of Visual Languages and Computing, February 2005. http://dl.acm.org/citation.cfm?id=1747411.

[18] Martin Fowler and Rebecca Parsons. *Domain-specific languages*. Addison Wesley Signature Series. Addison-Wesley, September 2010. http://books.google.no/books?id=ri1muolw_YwC.

[19] Diagram Predicate Framework. *Web site*. http://dpf.hib.no/.

[20] Kent Modelling Framework. *Project Web Site*. http://www.cs.kent.ac.uk/projects/kmf/.

[21] Ing. Norbert Fuhr. Object Constraint Language (OCL), 2006.

[22] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Architecture and Ontology Development*. Springer, Germany, 2006. http://eprints.mdx.ac.uk/6060/.

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

[24] Object Management Group. *MDA Guide*, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[25] Object Management Group. *OMG Unified Modeling LanguageTM (OMG UML),Infrastructure*, May 2010. http://www.omg.org/spec/UML/2.3/Infrastructure.

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

[27] Object Mangement Group. MDA Guide Version 1.0 OMG Document omg/2003-05-01, May 2003.

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

[29] JOHN HOWSE, FERNANDO MOLINA, JOHN TAYLOR, STUART KENT, , and JOSEPH (YOSSI) GIL. Spider Diagrams: A Diagrammatic Reasoning System, December 2001.

[30] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley and sons, New Jersey, 1 edition, March 2008. http://dsmbook.com/.

[31] Stuart Kent. Constraint diagrams: Visualizing invariants in object oriented modelling, October 1997.

[32] Thomas Kuhne. *Matters of (meta-)modeling*, 5(4):369–385 edition, December 2006. http://link.springer.com/article/10.1007%2Fs10270-006-0017-9/.

[33] Yngve Lamo, Xiaoliang Wang, Florian Mantz, Wendy MacCaull, and Adrian Rutle. *DPF Workbench: A Diagrammatic Multi-Layer Domain Specific (Meta-)Modelling Environment*, volume 429 of *Studies in Computational Intelligence*. Springer Berlin Heidelberg, 1 edition, 2012. http://link.springer.com/chapter/10.1007%2F978-3-642-30454-5_3#.

[34] Yngve Lamo, Xiaoling Wang, Florian Mantz, Øyvind Bech, Anders Sandven, and Adrian Rutle. DPF Workbench: a multi-level language workbench for MDE. Proceedings of the Estonian Academy of Sciences, February 2013. http://www.kirj.ee/public/proceedings_pdf/2013/issue_1/Proc-2013-1-3-15.pdf.

[35] Mark Richters and Martin Gogolla. *Validating UML Models and OCL Constraints*. http://www.db.informatik.uni-bremen.de.

[36] Model Driven Architecture (MDA). OMG Document ormsc/2001-07-01, July 2001.

[37] MetaCase. *Project Web Site*. http://www.metacase.com/.

[38] GEF (MVC). *Project Web Site*. http://www.eclipse.org/gef/gef_mvc/index.php/.

[39] OMG. The Meta Object Facility (MOF) [Web-Page]. http://www.omg.org/mof/.

[40] OMG. The Object Management Group (OMG). http://omg.org/.

[41] OMG. MOF 2.0/XMI Mapping Specification, August 2011. http://www.omg.org/spec/XMI/2.4.1/.

[42] The Object Management Group (OMG). The Unified Modeling Language (UML). [Web-Page], 2006. http://www.uml.org/.

[43] Eclipse Plateform. Project Web Site. http://www.eclipse.org/juno/.

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

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

[46] 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.

[47] Anders Sandven. Metamodel based Code Generation in DPF Editor. Master's thesis, Department of Informatics, University of Bergen, Norway, March 2012.

[48] 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.

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

[50] Clark Tony, Sammut Paul, and Willans James. *Applied metamodelling: a foundation for language driven development*. Number 6060. Xactium, Ceteva Sheffield, 2004. `http://eprints.mdx.ac.uk/6060/`.

[51] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models* . Number 978-0471727835. Wiley-Interscience, ist edition, July 2005.

[52] Visual OCL. *Project Web Site*. `http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/`.

[53] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for Mda*. Addison-Wesley Professional, 2003.

[54] Xpand. *Project Web Site*. `http://wiki.eclipse.org/Xpand`.